

# GnuTLS

---

Transport Layer Security Library for the GNU system  
for version 3.3.20, 23 March 2015



Nikos Mavrogiannopoulos  
Simon Josefsson ([bugs@gnutls.org](mailto:bugs@gnutls.org))

---

This manual is last updated 23 March 2015 for version 3.3.20 of GnuTLS.

Copyright © 2001-2013 Free Software Foundation, Inc.\\ Copyright © 2001-2013 Nikos Mavrogiannopoulos

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

# Table of Contents

<b>1</b>	<b>Preface</b>	<b>1</b>
<b>2</b>	<b>Introduction to GnuTLS</b>	<b>2</b>
2.1	Downloading and installing	2
2.2	Overview	3
<b>3</b>	<b>Introduction to TLS and DTLS</b>	<b>4</b>
3.1	TLS layers	4
3.2	The transport layer	4
3.3	The TLS record protocol	5
3.3.1	Encryption algorithms used in the record layer	5
3.3.2	Compression algorithms used in the record layer	6
3.3.3	Weaknesses and countermeasures	7
3.3.4	On record padding	7
3.4	The TLS alert protocol	8
3.5	The TLS handshake protocol	9
3.5.1	TLS ciphersuites	9
3.5.2	Authentication	9
3.5.3	Client authentication	10
3.5.4	Resuming sessions	10
3.6	TLS extensions	10
3.6.1	Maximum fragment length negotiation	10
3.6.2	Server name indication	10
3.6.3	Session tickets	11
3.6.4	HeartBeat	11
3.6.5	Safe renegotiation	11
3.6.6	OCSP status request	13
3.6.7	SRTP	13
3.6.8	Application Layer Protocol Negotiation (ALPN)	15
3.7	How to use TLS in application protocols	15
3.7.1	Separate ports	15
3.7.2	Upward negotiation	15
3.8	On SSL 2 and older protocols	17
<b>4</b>	<b>Authentication methods</b>	<b>18</b>
4.1	Certificate authentication	18
4.1.1	X.509 certificates	19
4.1.1.1	X.509 certificate structure	20
4.1.1.2	Importing an X.509 certificate	21
4.1.1.3	X.509 distinguished names	22
4.1.1.4	X.509 extensions	23
4.1.1.5	Accessing public and private keys	27

4.1.1.6	Verifying X.509 certificate paths.....	28
4.1.1.7	Verifying a certificate in the context of TLS session ..	34
4.1.1.8	Verifying a certificate using PKCS #11 .....	35
4.1.2	OpenPGP certificates .....	37
4.1.2.1	OpenPGP certificate structure.....	38
4.1.2.2	Verifying an OpenPGP certificate .....	39
4.1.2.3	Verifying a certificate in the context of a TLS session .....	39
4.1.3	Advanced certificate verification .....	40
4.1.3.1	Verifying a certificate using trust on first use authentication .....	40
4.1.3.2	Verifying a certificate using DANE (DNSSEC) .....	40
4.1.4	Digital signatures .....	41
4.1.4.1	Trading security for interoperability .....	42
4.2	More on certificate authentication .....	42
4.2.1	PKCS #10 certificate requests .....	42
4.2.2	PKIX certificate revocation lists .....	46
4.2.3	OCSP certificate status checking .....	48
4.2.4	Managing encrypted keys.....	53
4.2.5	Invoking certtool.....	58
4.2.6	Invoking ocsptool .....	68
4.2.7	Invoking danetool .....	70
4.3	Shared-key and anonymous authentication.....	74
4.3.1	SRP authentication .....	74
4.3.1.1	Authentication using SRP .....	74
4.3.1.2	Invoking srptool.....	75
4.3.2	PSK authentication .....	77
4.3.2.1	Authentication using PSK .....	77
4.3.2.2	Invoking psktool.....	77
4.3.3	Anonymous authentication .....	78
4.4	Selecting an appropriate authentication method .....	79
4.4.1	Two peers with an out-of-band channel.....	79
4.4.2	Two peers without an out-of-band channel.....	79
4.4.3	Two peers and a trusted third party .....	79
<b>5</b>	<b>Hardware security modules and abstract key types .....</b>	<b>81</b>
5.1	Abstract key types .....	81
5.1.1	Public keys .....	82
5.1.2	Private keys.....	84
5.1.3	Operations .....	85
5.2	Smart cards and HSMs.....	88
5.2.1	Initialization .....	89
5.2.2	Accessing objects that require a PIN .....	90
5.2.3	Reading objects .....	91
5.2.4	Writing objects.....	94
5.2.5	Using a PKCS #11 token with TLS.....	95
5.2.6	Invoking p11tool .....	96

5.2.7	p11tool help/usage ( <b>--help</b> )	96
5.2.8	debug option (-d)	96
5.2.9	export-chain option	96
5.2.10	list-all-privkeys option	96
5.2.11	list-privkeys option	96
5.2.12	list-keys option	96
5.2.13	write option	96
5.2.14	generate-random option	97
5.2.15	generate-rsa option	97
5.2.16	generate-dsa option	97
5.2.17	generate-ecc option	97
5.2.18	export-pubkey option	97
5.2.19	mark-wrap option	97
5.2.20	mark-trusted option	97
5.2.21	mark-ca option	97
5.2.22	mark-private option	98
5.2.23	trusted option	98
5.2.24	ca option	98
5.2.25	private option	98
5.2.26	so-login option	98
5.2.27	admin-login option	98
5.2.28	curve option	98
5.2.29	sec-param option	98
5.2.30	inder option	98
5.2.31	inraw option	99
5.2.32	outder option	99
5.2.33	outraw option	99
5.2.34	set-pin option	99
5.2.35	set-so-pin option	99
5.2.36	provider option	99
5.2.37	p11tool exit status	99
5.2.38	p11tool See Also	99
5.2.39	p11tool Examples	99
5.3	Trusted Platform Module (TPM)	100
5.3.1	Keys in TPM	100
5.3.2	Key generation	101
5.3.3	Using keys	102
5.3.4	Invoking tpmtool	103
5.3.5	tpmtool help/usage ( <b>--help</b> )	103
5.3.6	debug option (-d)	103
5.3.7	generate-rsa option	104
5.3.8	user option	104
5.3.9	system option	104
5.3.10	sec-param option	104
5.3.11	inder option	104
5.3.12	outder option	104
5.3.13	tpmtool exit status	105
5.3.14	tpmtool See Also	105

5.3.15	tpmtool Examples .....	105
<b>6</b>	<b>How to use GnuTLS in applications .....</b>	<b>106</b>
6.1	Introduction .....	106
6.1.1	General idea .....	106
6.1.2	Error handling .....	107
6.1.3	Common types .....	107
6.1.4	Debugging and auditing .....	108
6.1.5	Thread safety .....	109
6.1.6	Sessions and fork .....	110
6.1.7	Callback functions .....	110
6.2	Preparation .....	110
6.2.1	Headers .....	110
6.2.2	Initialization .....	110
6.2.3	Version check .....	111
6.2.4	Building the source .....	111
6.3	Session initialization .....	112
6.4	Associating the credentials .....	113
6.4.1	Certificates .....	113
6.4.2	SRP .....	118
6.4.3	PSK .....	120
6.4.4	Anonymous .....	121
6.5	Setting up the transport layer .....	122
6.5.1	Asynchronous operation .....	125
6.5.2	DTLS sessions .....	126
6.6	TLS handshake .....	126
6.7	Data transfer and termination .....	127
6.8	Buffered data transfer .....	130
6.9	Handling alerts .....	131
6.10	Priority strings .....	132
6.11	Selecting cryptographic key sizes .....	138
6.12	Advanced topics .....	140
6.12.1	Session resumption .....	140
6.12.2	Certificate verification .....	142
6.12.2.1	Trust on first use .....	142
6.12.2.2	DANE verification .....	144
6.12.3	Parameter generation .....	146
6.12.4	Deriving keys for other applications/protocols .....	146
6.12.5	Channel bindings .....	147
6.12.6	Interoperability .....	148
6.12.7	Compatibility with the OpenSSL library .....	148

<b>7</b>	<b>GnuTLS application examples</b>	<b>149</b>
7.1	Client examples	149
7.1.1	Simple client example with X.509 certificate support	149
7.1.2	Simple client example with SSH-style certificate verification	153
7.1.3	Simple client example with anonymous authentication	156
7.1.4	Simple datagram TLS client example	158
7.1.5	Obtaining session information	161
7.1.6	Using a callback to select the certificate to use	164
7.1.7	Verifying a certificate	170
7.1.8	Using a smart card with TLS	173
7.1.9	Client with resume capability example	177
7.1.10	Simple client example with SRP authentication	180
7.1.11	Simple client example using the C++ API	183
7.1.12	Helper functions for TCP connections	185
7.1.13	Helper functions for UDP connections	186
7.2	Server examples	188
7.2.1	Echo server with X.509 authentication	188
7.2.2	Echo server with OpenPGP authentication	192
7.2.3	Echo server with SRP authentication	196
7.2.4	Echo server with anonymous authentication	200
7.2.5	DTLS echo server with X.509 authentication	204
7.3	OCSP example	213
7.4	Miscellaneous examples	220
7.4.1	Checking for an alert	220
7.4.2	X.509 certificate parsing example	221
7.4.3	Listing the ciphersuites in a priority string	223
7.4.4	PKCS #12 structure generation example	225
<b>8</b>	<b>Using GnuTLS as a cryptographic library</b>	<b>229</b>
8.1	Symmetric algorithms	229
8.2	Public key algorithms	229
8.3	Hash and HMAC functions	229
8.4	Random number generation	230
<b>9</b>	<b>Other included programs</b>	<b>231</b>
9.1	Invoking gnutls-cli	231
9.2	Invoking gnutls-serv	236
9.3	Invoking gnutls-cli-debug	239
<b>10</b>	<b>Internal Architecture of GnuTLS</b>	<b>242</b>
10.1	The TLS Protocol	242
10.2	TLS Handshake Protocol	242
10.3	TLS Authentication Methods	243
10.4	TLS Extension Handling	244
10.5	Cryptographic Backend	250

<b>Appendix A</b>	<b>Upgrading from previous versions</b>	<b>253</b>
<b>Appendix B</b>	<b>Support</b>	<b>256</b>
B.1	Getting Help	256
B.2	Commercial Support	256
B.3	Bug Reports	256
B.4	Contributing	257
B.5	Certification	257
<b>Appendix C</b>	<b>Error Codes and Descriptions</b>	<b>259</b>
<b>Appendix D</b>	<b>Supported Ciphersuites</b>	<b>266</b>
<b>Appendix E</b>	<b>API reference</b>	<b>273</b>
E.1	Core TLS API	273
E.2	Datagram TLS API	362
E.3	X.509 certificate API	365
E.4	OCSP API	467
E.5	OpenPGP API	477
E.6	PKCS 12 API	497
E.7	Hardware token via PKCS 11 API	504
E.8	TPM API	518
E.9	Abstract key API	520
E.10	DANE API	548
E.11	Cryptographic API	553
E.12	Compatibility API	560
<b>Appendix F</b>	<b>Copying Information</b>	<b>571</b>
<b>Bibliography</b>		<b>579</b>
<b>Function and Data Index</b>		<b>583</b>
<b>Concept Index</b>		<b>593</b>



# 1 Preface

This document demonstrates and explains the GnuTLS library API. A brief introduction to the protocols and the technology involved is also included so that an application programmer can better understand the GnuTLS purpose and actual offerings. Even if GnuTLS is a typical library software, it operates over several security and cryptographic protocols which require the programmer to make careful and correct usage of them. Otherwise it is likely to only obtain a false sense of security. The term of security is very broad even if restricted to computer software, and cannot be confined to a single cryptographic library. For that reason, do not consider any program secure just because it uses GnuTLS; there are several ways to compromise a program or a communication line and GnuTLS only helps with some of them.

Although this document tries to be self contained, basic network programming and public key infrastructure (PKI) knowledge is assumed in most of it. A good introduction to networking can be found in [*STEVENS*], to public key infrastructure in [*GUTPKI*] and to security engineering in [*ANDERSON*].

Updated versions of the GnuTLS software and this document will be available from <http://www.gnutls.org/>.

## 2 Introduction to GnuTLS

In brief GnuTLS can be described as a library which offers an API to access secure communication protocols. These protocols provide privacy over insecure lines, and were designed to prevent eavesdropping, tampering, or message forgery.

Technically GnuTLS is a portable ANSI C based library which implements the protocols ranging from SSL 3.0 to TLS 1.2 (see [Chapter 3 \[Introduction to TLS\]](#), page 4, for a detailed description of the protocols), accompanied with the required framework for authentication and public key infrastructure. Important features of the GnuTLS library include:

- Support for TLS 1.2, TLS 1.1, TLS 1.0 and SSL 3.0 protocols.
- Support for Datagram TLS 1.0 and 1.2.
- Support for handling and verification of X.509 and OpenPGP certificates.
- Support for password authentication using TLS-SRP.
- Support for keyed authentication using TLS-PSK.
- Support for TPM, PKCS #11 tokens and smart-cards.

The GnuTLS library consists of three independent parts, namely the “TLS protocol part”, the “Certificate part”, and the “Cryptographic back-end” part. The “TLS protocol part” is the actual protocol implementation, and is entirely implemented within the GnuTLS library. The “Certificate part” consists of the certificate parsing, and verification functions and it uses functionality from the `libtasn1` library. The “Cryptographic back-end” is provided by the `nettle` and `gmplib` libraries.

### 2.1 Downloading and installing

GnuTLS is available for download at: <http://www.gnutls.org/download.html>

GnuTLS uses a development cycle where even minor version numbers indicate a stable release and a odd minor version number indicate a development release. For example, GnuTLS 1.6.3 denote a stable release since 6 is even, and GnuTLS 1.7.11 denote a development release since 7 is odd.

GnuTLS depends on `nettle` and `gmplib`, and you will need to install it before installing GnuTLS. The `nettle` library is available from <http://www.lysator.liu.se/~nisse/nettle/>, while `gmplib` is available from <http://www.gmp1ib.org/>. Don't forget to verify the cryptographic signature after downloading source code packages.

The package is then extracted, configured and built like many other packages that use Autoconf. For detailed information on configuring and building it, refer to the `INSTALL` file that is part of the distribution archive. Typically you invoke `./configure` and then `make check install`. There are a number of compile-time parameters, as discussed below.

Several parts of GnuTLS require ASN.1 functionality, which is provided by a library called `libtasn1`. A copy of `libtasn1` is included in GnuTLS. If you want to install it separately (e.g., to make it possibly to use `libtasn1` in other programs), you can get it from <http://www.gnu.org/software/libtasn1/>.

The compression library, `libz`, the PKCS #11 helper library `p11-kit`, as well as the TPM library `trousers`, are optional dependencies. You may get `libz` from <http://www.zlib.org>.

`net/`, `p11-kit` from <http://p11-glue.freedesktop.org/> and `trousers` from <http://trousers.sourceforge.net/>.

A few `configure` options may be relevant, summarized below. They disable or enable particular features, to create a smaller library with only the required features. Note however, that although a smaller library is generated, the included programs are not guaranteed to compile if some of these options are given.

```
--disable-srp-authentication
--disable-psk-authentication
--disable-anon-authentication
--disable-openpgp-authentication
--disable-dhe
--disable-ecdh
--disable-openssl-compatibility
--disable-dtls-srtp-support
--disable-alpn-support
--disable-heartbeat-support
--disable-libdane
--without-p11-kit
--without-tpm
--without-zlib
```

For the complete list, refer to the output from `configure --help`.

## 2.2 Overview

In this document we present an overview of the supported security protocols in [Chapter 3 \[Introduction to TLS\]](#), [page 4](#), and continue by providing more information on the certificate authentication in [Section 4.1 \[Certificate authentication\]](#), [page 18](#), and shared-key as well anonymous authentication in [Section 4.3 \[Shared-key and anonymous authentication\]](#), [page 74](#). We elaborate on certificate authentication by demonstrating advanced usage of the API in [Section 4.2 \[More on certificate authentication\]](#), [page 42](#). The core of the TLS library is presented in [Chapter 6 \[How to use GnuTLS in applications\]](#), [page 106](#) and example applications are listed in [Chapter 7 \[GnuTLS application examples\]](#), [page 149](#). In [Chapter 9 \[Other included programs\]](#), [page 231](#) the usage of few included programs that may assist debugging is presented. The last chapter is [Chapter 10 \[Internal architecture of GnuTLS\]](#), [page 242](#) that provides a short introduction to GnuTLS' internal architecture.

## 3 Introduction to TLS and DTLS

TLS stands for “Transport Layer Security” and is the successor of SSL, the Secure Sockets Layer protocol [SSL3] designed by Netscape. TLS is an Internet protocol, defined by IETF<sup>1</sup>, described in [RFC5246]. The protocol provides confidentiality, and authentication layers over any reliable transport layer. The description, above, refers to TLS 1.0 but applies to all other TLS versions as the differences between the protocols are not major.

The DTLS protocol, or “Datagram TLS” [RFC4347] is a protocol with identical goals as TLS, but can operate under unreliable transport layers such as UDP. The discussions below apply to this protocol as well, except when noted otherwise.

### 3.1 TLS layers

TLS is a layered protocol, and consists of the record protocol, the handshake protocol and the alert protocol. The record protocol is to serve all other protocols and is above the transport layer. The record protocol offers symmetric encryption, data authenticity, and optionally compression. The alert protocol offers some signaling to the other protocols. It can help informing the peer for the cause of failures and other error conditions. See [The Alert Protocol], page 8, for more information. The alert protocol is above the record protocol.

The handshake protocol is responsible for the security parameters’ negotiation, the initial key exchange and authentication. See [The Handshake Protocol], page 9, for more information about the handshake protocol. The protocol layering in TLS is shown in Figure 3.1.

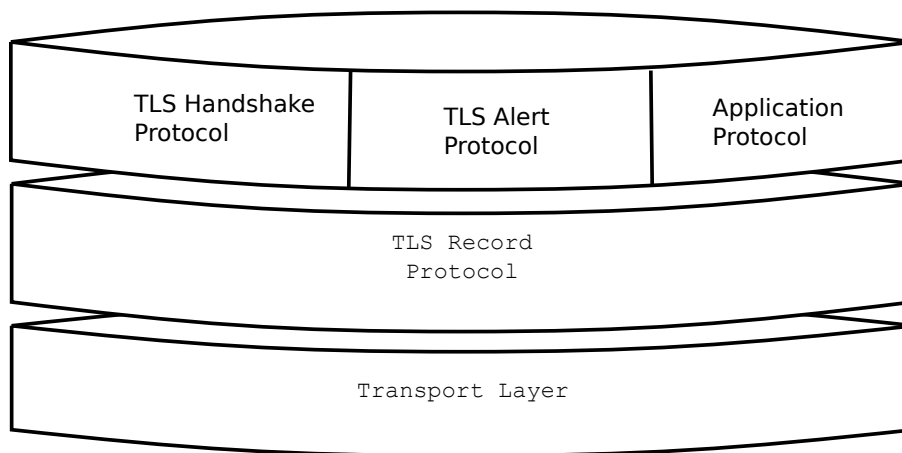


Figure 3.1: The TLS protocol layers.

### 3.2 The transport layer

TLS is not limited to any transport layer and can be used above any transport layer, as long as it is a reliable one. DTLS can be used over reliable and unreliable transport

<sup>1</sup> IETF, or Internet Engineering Task Force, is a large open international community of network designers, operators, vendors, and researchers concerned with the evolution of the Internet architecture and the smooth operation of the Internet. It is open to any interested individual.

layers. GnuTLS supports TCP and UDP layers transparently using the Berkeley sockets API. However, any transport layer can be used by providing callbacks for GnuTLS to access the transport layer (for details see [Section 6.5 \[Setting up the transport layer\]](#), page 122).

### 3.3 The TLS record protocol

The record protocol is the secure communications provider. Its purpose is to encrypt, authenticate and —optionally— compress packets. The record layer functions can be called at any time after the handshake process is finished, when there is need to receive or send data. In DTLS however, due to re-transmission timers used in the handshake out-of-order handshake data might be received for some time (maximum 60 seconds) after the handshake process is finished.

The functions to access the record protocol are limited to send and receive functions, which might, given the importance of this protocol in TLS, seem awkward. This is because the record protocol's parameters are all set by the handshake protocol. The record protocol initially starts with NULL parameters, which means no encryption, and no MAC is used. Encryption and authentication begin just after the handshake protocol has finished.

#### 3.3.1 Encryption algorithms used in the record layer

Confidentiality in the record layer is achieved by using symmetric block encryption algorithms like 3DES, AES or stream algorithms like ARCFOUR\_128. Ciphers are encryption algorithms that use a single, secret, key to encrypt and decrypt data. Block algorithms in CBC mode also provide protection against statistical analysis of the data. Thus, if you're using the TLS protocol, a random number of blocks will be appended to data, to prevent eavesdroppers from guessing the actual data size.

The supported in GnuTLS ciphers and MAC algorithms are shown in [Table 3.1](#) and [Table 3.2](#).

<b>Algorithm</b>	<b>Description</b>
3DES_CBC	This is the DES block cipher algorithm used with triple encryption (EDE). Has 64 bits block size and is used in CBC mode.
ARCFOUR_128	ARCFOUR_128 is a compatible algorithm with RSA's RC4 algorithm, which is considered to be a trade secret. It is a fast cipher but considered weak today.
AES_CBC	AES or RIJNDAEL is the block cipher algorithm that replaces the old DES algorithm. Has 128 bits block size and is used in CBC mode.
AES_GCM	This is the AES algorithm in the authenticated encryption GCM mode. This mode combines message authentication and encryption and can be extremely fast on CPUs that support hardware acceleration.
CAMELLIA_-CBC	This is an 128-bit block cipher developed by Mitsubishi and NTT. It is one of the approved ciphers of the European NESSIE and Japanese CRYPTREC projects.

Table 3.1: Supported ciphers.

<b>Algorithm</b>	<b>Description</b>
MAC_MD5	This is an HMAC based on MD5 a cryptographic hash algorithm designed by Ron Rivest. Outputs 128 bits of data.
MAC_SHA1	An HMAC based on the SHA1 cryptographic hash algorithm designed by NSA. Outputs 160 bits of data.
MAC_SHA256	An HMAC based on SHA256. Outputs 256 bits of data.
MAC_AEAD	This indicates that an authenticated encryption algorithm, such as GCM, is in use.

Table 3.2: Supported MAC algorithms.

### 3.3.2 Compression algorithms used in the record layer

The TLS record layer also supports compression. The algorithms implemented in GnuTLS can be found in the table below. The included algorithms perform really good when text, or other compressible data are to be transferred, but offer nothing on already compressed data, such as compressed images, zipped archives etc. These compression algorithms, may

be useful in high bandwidth TLS tunnels, and in cases where network usage has to be minimized. It should be noted however that compression increases latency.

The record layer compression in GnuTLS is implemented based on [RFC3749]. The supported algorithms are shown below.

GNUTLS\_COMP\_UNKNOWN

Unknown compression method.

GNUTLS\_COMP\_NULL

The NULL compression method (no compression).

GNUTLS\_COMP\_DEFLATE

The DEFLATE compression method from zlib.

GNUTLS\_COMP\_ZLIB

Same as GNUTLS\_COMP\_DEFLATE .

Figure 3.2: Supported compression algorithms

Note that compression enables attacks such as traffic analysis, or even plaintext recovery under certain circumstances. To avoid some of these attacks GnuTLS allows each record to be compressed independently (i.e., stateless compression), by using the "%STATELESS-COMPRESSION" priority string, in order to be used in cases where the attacker controlled data are pt in separate records.

### 3.3.3 Weaknesses and countermeasures

Some weaknesses that may affect the security of the record layer have been found in TLS 1.0 protocol. These weaknesses can be exploited by active attackers, and exploit the facts that

1. TLS has separate alerts for “decryption\_failed” and “bad\_record\_mac”
2. The decryption failure reason can be detected by timing the response time.
3. The IV for CBC encrypted packets is the last block of the previous encrypted packet.

Those weaknesses were solved in TLS 1.1 [RFC4346] which is implemented in GnuTLS. For this reason we suggest to always negotiate the highest supported TLS version with the peer<sup>2</sup>. For a detailed discussion of the issues see the archives of the TLS Working Group mailing list and [CBCATT].

### 3.3.4 On record padding

The TLS protocol allows for extra padding of records in CBC ciphers, to prevent statistical analysis based on the length of exchanged messages (see [RFC5246] section 6.2.3.2). GnuTLS appears to be one of few implementations that take advantage of this feature: the user can provide some plaintext data with a range of lengths she wishes to hide, and GnuTLS adds extra padding to make sure the attacker cannot tell the real plaintext length is in a range smaller than the user-provided one. Use [gnutls\_record\_send\_range], page 335 to send length-hidden messages and [gnutls\_record\_can\_use\_length\_hiding], page 332 to check whether the current session supports length hiding. Using the standard [gnutls\_record\_send], page 335 will only add minimal padding.

<sup>2</sup> If this is not possible then please consult Section 6.12.6 [Interoperability], page 148.

The TLS implementation in the Symbian operating system, frequently used by Nokia and Sony-Ericsson mobile phones, cannot handle non-minimal record padding. What happens when one of these clients handshake with a GnuTLS server is that the client will fail to compute the correct MAC for the record. The client sends a TLS alert (`bad_record_mac`) and disconnects. Typically this will result in error messages such as 'A TLS fatal alert has been received', 'Bad record MAC', or both, on the GnuTLS server side.

If compatibility with such devices is a concern, not sending length-hidden messages solves the problem by using minimal padding.

If you implement an application that has a configuration file, we recommend that you make it possible for users or administrators to specify a GnuTLS protocol priority string, which is used by your application via `[gnutls_priority_set]`, page 326. To allow the best flexibility, make it possible to have a different priority string for different incoming IP addresses.

### 3.4 The TLS alert protocol

The alert protocol is there to allow signals to be sent between peers. These signals are mostly used to inform the peer about the cause of a protocol failure. Some of these signals are used internally by the protocol and the application protocol does not have to cope with them (e.g. `GNUTLS_A_CLOSE_NOTIFY`), and others refer to the application protocol solely (e.g. `GNUTLS_A_USER_CANCELLED`). An alert signal includes a level indication which may be either fatal or warning. Fatal alerts always terminate the current connection, and prevent future re-negotiations using the current session ID. All alert messages are summarized in the table below.

The alert messages are protected by the record protocol, thus the information that is included does not leak. You must take extreme care for the alert information not to leak to a possible attacker, via public log files etc.

Alert	ID	Description
<code>GNUTLS_A_CLOSE_NOTIFY</code>	0	Close notify
<code>GNUTLS_A_UNEXPECTED_MESSAGE</code>	10	Unexpected message
<code>GNUTLS_A_BAD_RECORD_MAC</code>	20	Bad record MAC
<code>GNUTLS_A_DECRYPTION_FAILED</code>	21	Decryption failed
<code>GNUTLS_A_RECORD_OVERFLOW</code>	22	Record overflow
<code>GNUTLS_A_DECOMPRESSION_FAILURE</code>	30	Decompression failed
<code>GNUTLS_A_HANDSHAKE_FAILURE</code>	40	Handshake failed
<code>GNUTLS_A_SSL3_NO_CERTIFICATE</code>	41	No certificate (SSL 3.0)
<code>GNUTLS_A_BAD_CERTIFICATE</code>	42	Certificate is bad
<code>GNUTLS_A_UNSUPPORTED_CERTIFICATE</code>	43	Certificate is not supported
<code>GNUTLS_A_CERTIFICATE_REVOKED</code>	44	Certificate was revoked
<code>GNUTLS_A_CERTIFICATE_EXPIRED</code>	45	Certificate is expired
<code>GNUTLS_A_CERTIFICATE_UNKNOWN</code>	46	Unknown certificate
<code>GNUTLS_A_ILLEGAL_PARAMETER</code>	47	Illegal parameter
<code>GNUTLS_A_UNKNOWN_CA</code>	48	CA is unknown
<code>GNUTLS_A_ACCESS_DENIED</code>	49	Access was denied
<code>GNUTLS_A_DECODE_ERROR</code>	50	Decode error



GNUTLS_A_DECRYPT_ERROR	51	Decrypt error
GNUTLS_A_EXPORT_RESTRICTION	60	Export restriction
GNUTLS_A_PROTOCOL_VERSION	70	Error in protocol version
GNUTLS_A_INSUFFICIENT_SECURITY	71	Insufficient security
GNUTLS_A_INTERNAL_ERROR	80	Internal error
GNUTLS_A_USER_CANCELED	90	User canceled
GNUTLS_A_NO_RENEGOTIATION	100	No renegotiation is allowed
GNUTLS_A_UNSUPPORTED_EXTENSION	110	An unsupported extension was sent
GNUTLS_A_CERTIFICATE_UNOBTAINABLE	111	Could not retrieve the specified certificate
GNUTLS_A_UNRECOGNIZED_NAME	112	The server name sent was not recognized
GNUTLS_A_UNKNOWN_PSK_IDENTITY	115	The SRP/PSK username is missing or not known
GNUTLS_A_NO_APPLICATION_PROTOCOL	120	No supported application protocol could be negotiated

## 3.5 The TLS handshake protocol

The handshake protocol is responsible for the ciphersuite negotiation, the initial key exchange, and the authentication of the two peers. This is fully controlled by the application layer, thus your program has to set up the required parameters. The main handshake function is [\[gnutls\\_handshake\]](#), page 309. In the next paragraphs we elaborate on the handshake protocol, i.e., the ciphersuite negotiation.

### 3.5.1 TLS ciphersuites

The handshake protocol of TLS negotiates cipher suites of a special form illustrated by the `TLS_DHE_RSA_WITH_3DES_CBC_SHA` cipher suite name. A typical cipher suite contains these parameters:

- The key exchange algorithm. `DHE_RSA` in the example.
- The Symmetric encryption algorithm and mode `3DES_CBC` in this example.
- The MAC<sup>3</sup> algorithm used for authentication. `MAC_SHA` is used in the above example.

The cipher suite negotiated in the handshake protocol will affect the record protocol, by enabling encryption and data authentication. Note that you should not over rely on TLS to negotiate the strongest available cipher suite. Do not enable ciphers and algorithms that you consider weak.

All the supported ciphersuites are listed in [\[ciphersuites\]](#), page 266.

### 3.5.2 Authentication

The key exchange algorithms of the TLS protocol offer authentication, which is a prerequisite for a secure connection. The available authentication methods in GnuTLS follow.

<sup>3</sup> MAC stands for Message Authentication Code. It can be described as a keyed hash algorithm. See RFC2104.

- Certificate authentication: Authenticated key exchange using public key infrastructure and certificates (X.509 or OpenPGP).
- SRP authentication: Authenticated key exchange using a password.
- PSK authentication: Authenticated key exchange using a pre-shared key.
- Anonymous authentication: Key exchange without peer authentication.

### 3.5.3 Client authentication

In the case of ciphersuites that use certificate authentication, the authentication of the client is optional in TLS. A server may request a certificate from the client using the `[gnutls_certificate_server_set_request]`, page 281 function. We elaborate in [Section 6.4.1 \[Certificate credentials\]](#), page 113.

### 3.5.4 Resuming sessions

The TLS handshake process performs expensive calculations and a busy server might easily be put under load. To reduce the load, session resumption may be used. This is a feature of the TLS protocol which allows a client to connect to a server after a successful handshake, without the expensive calculations. This is achieved by re-using the previously established keys, meaning the server needs to store the state of established connections (unless session tickets are used – [Section 3.6.3 \[Session tickets\]](#), page 11).

Session resumption is an integral part of GnuTLS, and [Section 6.12.1 \[Session resumption\]](#), page 140, [\[ex-resume-client\]](#), page 177 illustrate typical uses of it.

## 3.6 TLS extensions

A number of extensions to the TLS protocol have been proposed mainly in *[TLSEXT]*. The extensions supported in GnuTLS are discussed in the subsections that follow.

### 3.6.1 Maximum fragment length negotiation

This extension allows a TLS implementation to negotiate a smaller value for record packet maximum length. This extension may be useful to clients with constrained capabilities. The functions shown below can be used to control this extension.

```
size_t [gnutls_record_get_max_size], page 333 (gnutls_session_t session)
ssize_t [gnutls_record_set_max_size], page 336 (gnutls_session_t session,
size_t size)
```

### 3.6.2 Server name indication

A common problem in HTTPS servers is the fact that the TLS protocol is not aware of the hostname that a client connects to, when the handshake procedure begins. For that reason the TLS server has no way to know which certificate to send.

This extension solves that problem within the TLS protocol, and allows a client to send the HTTP hostname before the handshake begins within the first handshake packet. The functions [\[gnutls\\_server\\_name\\_set\]](#), page 339 and [\[gnutls\\_server\\_name\\_get\]](#), page 339 can be used to enable this extension, or to retrieve the name sent by a client.

```
int [gnutls_server_name_set], page 339 (gnutls_session_t session,
gnutls_server_name_type_t type, const void * name, size_t name_length)
int [gnutls_server_name_get], page 339 (gnutls_session_t session, void *
data, size_t * data_length, unsigned int * type, unsigned int indx)
```

### 3.6.3 Session tickets

To resume a TLS session, the server normally stores session parameters. This complicates deployment, and can be avoided by delegating the storage to the client. Because session parameters are sensitive they are encrypted and authenticated with a key only known to the server and then sent to the client. The Session Tickets extension is described in RFC 5077 [TLSTKT].

A disadvantage of session tickets is that they eliminate the effects of forward secrecy when a server uses the same key for long time. That is, the secrecy of all sessions on a server using tickets depends on the ticket key being kept secret. For that reason server keys should be rotated and discarded regularly.

Since version 3.1.3 GnuTLS clients transparently support session tickets.

### 3.6.4 HeartBeat

This is a TLS extension that allows to ping and receive confirmation from the peer, and is described in [RFC6520]. The extension is disabled by default and [gnutls\_heartbeat\_enable], page 312 can be used to enable it. A policy may be negotiated to only allow sending heartbeat messages or sending and receiving. The current session policy can be checked with [gnutls\_heartbeat\_allowed], page 312. The requests coming from the peer result to GNUTLS\_E\_HEARTBEAT\_PING\_RECEIVED being returned from the receive function. Ping requests to peer can be send via [gnutls\_heartbeat\_ping], page 313.

```
int [gnutls_heartbeat_allowed], page 312 (gnutls_session_t session, unsigned
int type)
void [gnutls_heartbeat_enable], page 312 (gnutls_session_t session, unsigned
int type)
int [gnutls_heartbeat_ping], page 313 (gnutls_session_t session, size_t
data_size, unsigned int max_tries, unsigned int flags)
int [gnutls_heartbeat_pong], page 313 (gnutls_session_t session, unsigned int
flags)
void [gnutls_heartbeat_set_timeouts], page 314 (gnutls_session_t session,
unsigned int retrans_timeout, unsigned int total_timeout)
unsigned int [gnutls_heartbeat_get_timeout], page 313 (gnutls_session_t
session)
```

### 3.6.5 Safe renegotiation

TLS gives the option to two communicating parties to renegotiate and update their security parameters. One useful example of this feature was for a client to initially connect using anonymous negotiation to a server, and the renegotiate using some authenticated ciphersuite. This occurred to avoid having the client sending its credentials in the clear.

However this renegotiation, as initially designed would not ensure that the party one is renegotiating is the same as the one in the initial negotiation. For example one server could

forward all renegotiation traffic to an other server who will see this traffic as an initial negotiation attempt.

This might be seen as a valid design decision, but it seems it was not widely known or understood, thus today some application protocols use the TLS renegotiation feature in a manner that enables a malicious server to insert content of his choice in the beginning of a TLS session.

The most prominent vulnerability was with HTTPS. There servers request a renegotiation to enforce an anonymous user to use a certificate in order to access certain parts of a web site. The attack works by having the attacker simulate a client and connect to a server, with server-only authentication, and send some data intended to cause harm. The server will then require renegotiation from him in order to perform the request. When the proper client attempts to contact the server, the attacker hijacks that connection and forwards traffic to the initial server that requested renegotiation. The attacker will not be able to read the data exchanged between the client and the server. However, the server will (incorrectly) assume that the initial request sent by the attacker was sent by the now authenticated client. The result is a prefix plain-text injection attack.

The above is just one example. Other vulnerabilities exists that do not rely on the TLS renegotiation to change the client's authenticated status (either TLS or application layer).

While fixing these application protocols and implementations would be one natural reaction, an extension to TLS has been designed that cryptographically binds together any renegotiated handshakes with the initial negotiation. When the extension is used, the attack is detected and the session can be terminated. The extension is specified in [RFC5746].

GnuTLS supports the safe renegotiation extension. The default behavior is as follows. Clients will attempt to negotiate the safe renegotiation extension when talking to servers. Servers will accept the extension when presented by clients. Clients and servers will permit an initial handshake to complete even when the other side does not support the safe renegotiation extension. Clients and servers will refuse renegotiation attempts when the extension has not been negotiated.

Note that permitting clients to connect to servers when the safe renegotiation extension is not enabled, is open up for attacks. Changing this default behavior would prevent interoperability against the majority of deployed servers out there. We will reconsider this default behavior in the future when more servers have been upgraded. Note that it is easy to configure clients to always require the safe renegotiation extension from servers.

To modify the default behavior, we have introduced some new priority strings (see [Section 6.10 \[Priority Strings\]](#), page 132). The `%UNSAFE_RENEGOTIATION` priority string permits (re-)handshakes even when the safe renegotiation extension was not negotiated. The default behavior is `%PARTIAL_RENEGOTIATION` that will prevent renegotiation with clients and servers not supporting the extension. This is secure for servers but leaves clients vulnerable to some attacks, but this is a trade-off between security and compatibility with old servers. The `%SAFE_RENEGOTIATION` priority string makes clients and servers require the extension for every handshake. The latter is the most secure option for clients, at the cost of not being able to connect to legacy servers. Servers will also deny clients that do not support the extension from connecting.

It is possible to disable use of the extension completely, in both clients and servers, by using the `%DISABLE_SAFE_RENEGOTIATION` priority string however we strongly recommend you to only do this for debugging and test purposes.

The default values if the flags above are not specified are:

**Server:**    `%PARTIAL_RENEGOTIATION`

**Client:**    `%PARTIAL_RENEGOTIATION`

For applications we have introduced a new API related to safe renegotiation. The `[gnutls_safe_renegotiation_status]`, page 338 function is used to check if the extension has been negotiated on a session, and can be used both by clients and servers.

### 3.6.6 OCSP status request

The Online Certificate Status Protocol (OCSP) is a protocol that allows the client to verify the server certificate for revocation without messing with certificate revocation lists. Its drawback is that it requires the client to connect to the server's CA OCSP server and request the status of the certificate. This extension however, enables a TLS server to include its CA OCSP server response in the handshake. That is an HTTPS server may periodically run `ocsptool` (see [Section 4.2.6 \[ocsptool Invocation\]](#), page 68) to obtain its certificate revocation status and serve it to the clients. That way a client avoids an additional connection to the OCSP server.

```
void [gnutls_certificate_set_ocsp_status_request_function], page 282
(gnutls_certificate_credentials_t sc, gnutls_status_request_ocsp_func
ocsp_func, void * ptr)
int [gnutls_certificate_set_ocsp_status_request_file], page 281
(gnutls_certificate_credentials_t sc, const char * response_file, unsigned
int flags)
int [gnutls_ocsp_status_request_enable_client], page 317 (gnutls_session_t
session, gnutls_datum_t * responder_id, size_t responder_id_size,
gnutls_datum_t * extensions)
int [gnutls_ocsp_status_request_is_checked], page 318 (gnutls_session_t
session, unsigned int flags)
```

A server is required to provide the OCSP server's response using the `[gnutls_certificate_set_ocsp_status_request_file]`, page 281. The response may be obtained periodically using the following command.

```
ocsptool --ask --load-cert server_cert.pem --load-issuer the_issuer.pem
--load-signer the_issuer.pem --outfile ocsf.response
```

Since version 3.1.3 GnuTLS clients transparently support the certificate status request.

### 3.6.7 SRTP

The TLS protocol was extended in [\[RFC5764\]](#) to provide keying material to the Secure RTP (SRTP) protocol. The SRTP protocol provides an encapsulation of encrypted data that is optimized for voice data. With the SRTP TLS extension two peers can negotiate keys using TLS or DTLS and obtain keying material for use with SRTP. The available SRTP profiles are listed below.

```

GNUTLS_SRTP_AES128_CM_HMAC_SHA1_80
    128 bit AES with a 80 bit HMAC-SHA1

GNUTLS_SRTP_AES128_CM_HMAC_SHA1_32
    128 bit AES with a 32 bit HMAC-SHA1

GNUTLS_SRTP_NULL_HMAC_SHA1_80
    NULL cipher with a 80 bit HMAC-SHA1

GNUTLS_SRTP_NULL_HMAC_SHA1_32
    NULL cipher with a 32 bit HMAC-SHA1

```

Figure 3.3: Supported SRTP profiles

To enable use the following functions.

```

int [gnutls_srtp_set_profile], page 354 (gnutls_session_t session,
gnutls_srtp_profile_t profile)
int [gnutls_srtp_set_profile_direct], page 354 (gnutls_session_t session,
const char * profiles, const char ** err_pos)

```

To obtain the negotiated keys use the function below.

```

int gnutls_srtp_get_keys (gnutls_session_t session, void * [Function]
    key_material, unsigned int key_material_size, gnutls_datum_t *
    client_key, gnutls_datum_t * client_salt, gnutls_datum_t *
    server_key, gnutls_datum_t * server_salt)

```

*session*: is a `gnutls_session_t` structure.

*key\_material*: Space to hold the generated key material

*key\_material\_size*: The maximum size of the key material

*client\_key*: The master client write key, pointing inside the key material

*client\_salt*: The master client write salt, pointing inside the key material

*server\_key*: The master server write key, pointing inside the key material

*server\_salt*: The master server write salt, pointing inside the key material

This is a helper function to generate the keying material for SRTP. It requires the space of the key material to be pre-allocated (should be at least 2x the maximum key size and salt size). The `client_key`, `client_salt`, `server_key` and `server_salt` are convenience datums that point inside the key material. They may be NULL.

**Returns:** On success the size of the key material is returned, otherwise, `GNUTLS_E_SHORT_MEMORY_BUFFER` if the buffer given is not sufficient, or a negative error code.

Since 3.1.4

Other helper functions are listed below.

```

int [gnutls_srtp_get_selected_profile], page 353 (gnutls_session_t session,
gnutls_srtp_profile_t * profile)
const char * [gnutls_srtp_get_profile_name], page 353 (gnutls_srtp_profile_t
profile)
int [gnutls_srtp_get_profile_id], page 353 (const char * name,
gnutls_srtp_profile_t * profile)

```

### 3.6.8 Application Layer Protocol Negotiation (ALPN)

The TLS protocol was extended in `draft-ietf-tls-applayerprotoneg-00` to provide the application layer a method of negotiating the application protocol version. This allows for negotiation of the application protocol during the TLS handshake, thus reducing round-trips. The application protocol is described by an opaque string. To enable, use the following functions.

```
int [gnutls_alpn_set_protocols], page 274 (gnutls_session_t session, const
gnutls_datum_t * protocols, unsigned protocols_size, unsigned int flags)
int [gnutls_alpn_get_selected_protocol], page 274 (gnutls_session_t session,
gnutls_datum_t * protocol)
```

Note that these functions are intended to be used with protocols that are registered in the Application Layer Protocol Negotiation IANA registry. While you can use them for other protocols (at the risk of collisions), it is preferable to register them.

## 3.7 How to use TLS in application protocols

This chapter is intended to provide some hints on how to use TLS over simple custom made application protocols. The discussion below mainly refers to the TCP/IP transport layer but may be extended to other ones too.

### 3.7.1 Separate ports

Traditionally SSL was used in application protocols by assigning a new port number for the secure services. By doing this two separate ports were assigned, one for the non-secure sessions, and one for the secure sessions. This method ensures that if a user requests a secure session then the client will attempt to connect to the secure port and fail otherwise. The only possible attack with this method is to perform a denial of service attack. The most famous example of this method is “HTTP over TLS” or HTTPS protocol [*RFC2818*].

Despite its wide use, this method has several issues. This approach starts the TLS Handshake procedure just after the client connects on the —so called— secure port. That way the TLS protocol does not know anything about the client, and popular methods like the host advertising in HTTP do not work<sup>4</sup>. There is no way for the client to say “I connected to YYY server” before the Handshake starts, so the server cannot possibly know which certificate to use.

Other than that it requires two separate ports to run a single service, which is unnecessary complication. Due to the fact that there is a limitation on the available privileged ports, this approach was soon deprecated in favor of upward negotiation.

### 3.7.2 Upward negotiation

Other application protocols<sup>5</sup> use a different approach to enable the secure layer. They use something often called as the “TLS upgrade” method. This method is quite tricky but it is more flexible. The idea is to extend the application protocol to have a “STARTTLS” request, whose purpose it to start the TLS protocols just after the client requests it. This

<sup>4</sup> See also the Server Name Indication extension on [*serverind*], page 10.

<sup>5</sup> See LDAP, IMAP etc.



approach does not require any extra port to be reserved. There is even an extension to HTTP protocol to support this method [RFC2817].

The tricky part, in this method, is that the “STARTTLS” request is sent in the clear, thus is vulnerable to modifications. A typical attack is to modify the messages in a way that the client is fooled and thinks that the server does not have the “STARTTLS” capability. See a typical conversation of a hypothetical protocol:

```
(client connects to the server)
CLIENT: HELLO I'M MR. XXX
SERVER: NICE TO MEET YOU XXX
CLIENT: PLEASE START TLS
SERVER: OK
*** TLS STARTS
CLIENT: HERE ARE SOME CONFIDENTIAL DATA
```

And an example of a conversation where someone is acting in between:

```
(client connects to the server)
CLIENT: HELLO I'M MR. XXX
SERVER: NICE TO MEET YOU XXX
CLIENT: PLEASE START TLS
(here someone inserts this message)
SERVER: SORRY I DON'T HAVE THIS CAPABILITY
CLIENT: HERE ARE SOME CONFIDENTIAL DATA
```

As you can see above the client was fooled, and was naïve enough to send the confidential data in the clear, despite the server telling the client that it does not support “STARTTLS”.

How do we avoid the above attack? As you may have already noticed this situation is easy to avoid. The client has to ask the user before it connects whether the user requests TLS or not. If the user answered that he certainly wants the secure layer the last conversation should be:

```
(client connects to the server)
CLIENT: HELLO I'M MR. XXX
SERVER: NICE TO MEET YOU XXX
CLIENT: PLEASE START TLS
(here someone inserts this message)
SERVER: SORRY I DON'T HAVE THIS CAPABILITY
CLIENT: BYE
```

(the client notifies the user that the secure connection was not possible)

This method, if implemented properly, is far better than the traditional method, and the security properties remain the same, since only denial of service is possible. The benefit is that the server may request additional data before the TLS Handshake protocol starts, in order to send the correct certificate, use the correct password file, or anything else!



### 3.8 On SSL 2 and older protocols

One of the initial decisions in the GnuTLS development was to implement the known security protocols for the transport layer. Initially TLS 1.0 was implemented since it was the latest at that time, and was considered to be the most advanced in security properties. Later the SSL 3.0 protocol was implemented since it is still the only protocol supported by several servers and there are no serious security vulnerabilities known.

One question that may arise is why we didn't implement SSL 2.0 in the library. There are several reasons, most important being that it has serious security flaws, unacceptable for a modern security library. Other than that, this protocol is barely used by anyone these days since it has been deprecated since 1996. The security problems in SSL 2.0 include:

- Message integrity compromised. The SSLv2 message authentication uses the MD5 function, and is insecure.
- Man-in-the-middle attack. There is no protection of the handshake in SSLv2, which permits a man-in-the-middle attack.
- Truncation attack. SSLv2 relies on TCP FIN to close the session, so the attacker can forge a TCP FIN, and the peer cannot tell if it was a legitimate end of data or not.
- Weak message integrity for export ciphers. The cryptographic keys in SSLv2 are used for both message authentication and encryption, so if weak encryption schemes are negotiated (say 40-bit keys) the message authentication code uses the same weak key, which isn't necessary.

Other protocols such as Microsoft's PCT 1 and PCT 2 were not implemented because they were also abandoned and deprecated by SSL 3.0 and later TLS 1.0.

## 4 Authentication methods

The initial key exchange of the TLS protocol performs authentication of the peers. In typical scenarios the server is authenticated to the client, and optionally the client to the server.

While many associate TLS with X.509 certificates and public key authentication, the protocol supports various authentication methods, including pre-shared keys, and passwords. In this chapter a description of the existing authentication methods is provided, as well as some guidance on which use-cases each method can be used at.

### 4.1 Certificate authentication

The most known authentication method of TLS are certificates. The PKIX [*PKIX*] public key infrastructure is daily used by anyone using a browser today. GnuTLS supports both X.509 certificates [*PKIX*] and OpenPGP certificates using a common API.

The key exchange algorithms supported by certificate authentication are shown in [Table 4.1](#).

Key exchange	Description
RSA	The RSA algorithm is used to encrypt a key and send it to the peer. The certificate must allow the key to be used for encryption.
DHE_RSA	The RSA algorithm is used to sign ephemeral Diffie-Hellman parameters which are sent to the peer. The key in the certificate must allow the key to be used for signing. Note that key exchange algorithms which use ephemeral Diffie-Hellman parameters, offer perfect forward secrecy. That means that even if the private key used for signing is compromised, it cannot be used to reveal past session data.
ECDHE_RSA	The RSA algorithm is used to sign ephemeral elliptic curve Diffie-Hellman parameters which are sent to the peer. The key in the certificate must allow the key to be used for signing. It also offers perfect forward secrecy. That means that even if the private key used for signing is compromised, it cannot be used to reveal past session data.
DHE_DSS	The DSA algorithm is used to sign ephemeral Diffie-Hellman parameters which are sent to the peer. The certificate must contain DSA parameters to use this key exchange algorithm. DSA is the algorithm of the Digital Signature Standard (DSS).
ECDHE_ECDSA	The Elliptic curve DSA algorithm is used to sign ephemeral elliptic curve Diffie-Hellman parameters which are sent to the peer. The certificate must contain ECDSA parameters (i.e., EC and marked for signing) to use this key exchange algorithm.

Table 4.1: Supported key exchange algorithms.

### 4.1.1 X.509 certificates

The X.509 protocols rely on a hierarchical trust model. In this trust model Certification Authorities (CAs) are used to certify entities. Usually more than one certification authorities exist, and certification authorities may certify other authorities to issue certificates as well, following a hierarchical model.



Figure 4.1: An example of the X.509 hierarchical trust model.

One needs to trust one or more CAs for his secure communications. In that case only the certificates issued by the trusted authorities are acceptable. The framework is illustrated on [Figure 4.1](#).

#### 4.1.1.1 X.509 certificate structure

An X.509 certificate usually contains information about the certificate holder, the signer, a unique serial number, expiration dates and some other fields [*PKIX*] as shown in [Table 4.2](#).

Field	Description
version	The field that indicates the version of the certificate.
serialNumber	This field holds a unique serial number per certificate.
signature	The issuing authority's signature.
issuer	Holds the issuer's distinguished name.
validity	The activation and expiration dates.
subject	The subject's distinguished name of the certificate.
extensions	The extensions are fields only present in version 3 certificates.

Table 4.2: X.509 certificate fields.

The certificate's *subject or issuer name* is not just a single string. It is a Distinguished name and in the ASN.1 notation is a sequence of several object identifiers with their corresponding values. Some of available OIDs to be used in an X.509 distinguished name are defined in `gnutls/x509.h`.

The *Version* field in a certificate has values either 1 or 3 for version 3 certificates. Version 1 certificates do not support the extensions field so it is not possible to distinguish a CA from a person, thus their usage should be avoided.

The *validity* dates are there to indicate the date that the specific certificate was activated and the date the certificate's key would be considered invalid.

In GnuTLS the X.509 certificate structures are handled using the `gnutls_x509_cert_t` type and the corresponding private keys with the `gnutls_x509_privkey_t` type. All the available functions for X.509 certificate handling have their prototypes in `gnutls/x509.h`. An example program to demonstrate the X.509 parsing capabilities can be found in [\[ex-x509-info\]](#), page 221.

#### 4.1.1.2 Importing an X.509 certificate

The certificate structure should be initialized using [\[gnutls\\_x509\\_cert\\_init\]](#), page 423, and a certificate structure can be imported using [\[gnutls\\_x509\\_cert\\_import\]](#), page 422.

```
int [gnutls_x509_cert_init], page 423 (gnutls_x509_cert_t * cert)
int [gnutls_x509_cert_import], page 422 (gnutls_x509_cert_t cert, const
gnutls_datum_t * data, gnutls_x509_cert_fmt_t format)
void [gnutls_x509_cert_deinit], page 402 (gnutls_x509_cert_t cert)
```

In several functions an array of certificates is required. To assist in initialization and import the following two functions are provided.

```
int [gnutls_x509_cert_list_import], page 423 (gnutls_x509_cert_t * certs,
unsigned int * cert_max, const gnutls_datum_t * data, gnutls_x509_cert_fmt_t
format, unsigned int flags)
int [gnutls_x509_cert_list_import2], page 423 (gnutls_x509_cert_t ** certs,
unsigned int * size, const gnutls_datum_t * data, gnutls_x509_cert_fmt_t
format, unsigned int flags)
```

In all cases after use a certificate must be deinitialized using [\[gnutls\\_x509\\_cert\\_deinit\], page 402](#). Note that although the functions above apply to `gnutls_x509_cert_t` structure, similar functions exist for the CRL structure `gnutls_x509_crl_t`.

### 4.1.1.3 X.509 distinguished names

The “subject” of an X.509 certificate is not described by a single name, but rather with a distinguished name. This in X.509 terminology is a list of strings each associated an object identifier. To make things simple GnuTLS provides [\[gnutls\\_x509\\_cert\\_get\\_dn2\], page 407](#) which follows the rules in [\[RFC4514\]](#) and returns a single string. Access to each string by individual object identifiers can be accessed using [\[gnutls\\_x509\\_cert\\_get\\_dn\\_by\\_oid\], page 407](#).

```
int gnutls_x509_cert_get_dn2 (gnutls_x509_cert_t cert, [Function]
gnutls_datum_t * dn)
cert: should contain a gnutls_x509_cert_t structure
```

dn: a pointer to a structure to hold the name

This function will allocate buffer and copy the name of the Certificate. The name will be in the form "C=xxxx,O=yyyy,CN=zzzz" as described in RFC4514. The output string will be ASCII or UTF-8 encoded, depending on the certificate data.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value. and a negative error code on error.

**Since:** 3.1.10

```
int [gnutls_x509_cert_get_dn], page 406 (gnutls_x509_cert_t cert, char * buf,
size_t * buf_size)
int [gnutls_x509_cert_get_dn_by_oid], page 407 (gnutls_x509_cert_t cert, const
char * oid, int indx, unsigned int raw_flag, void * buf, size_t * buf_size)
int [gnutls_x509_cert_get_dn_oid], page 407 (gnutls_x509_cert_t cert, int indx,
void * oid, size_t * oid_size)
```

Similar functions exist to access the distinguished name of the issuer of the certificate.

```

int [gnutls_x509_cert_get_issuer_dn], page 413 (gnutls_x509_cert_t cert, char *
buf, size_t * buf_size)
int [gnutls_x509_cert_get_issuer_dn2], page 413 (gnutls_x509_cert_t cert,
gnutls_datum_t * dn)
int [gnutls_x509_cert_get_issuer_dn_by_oid], page 413 (gnutls_x509_cert_t
cert, const char * oid, int indx, unsigned int raw_flag, void * buf, size_t *
buf_size)
int [gnutls_x509_cert_get_issuer_dn_oid], page 414 (gnutls_x509_cert_t cert,
int indx, void * oid, size_t * oid_size)
int [gnutls_x509_cert_get_issuer], page 411 (gnutls_x509_cert_t cert,
gnutls_x509_dn_t * dn)

```

The more powerful `[gnutls_x509_cert_get_subject]`, page 420 and `[gnutls_x509_dn_get_rdn_ava]`, page 436 provide efficient but low-level access to the contents of the distinguished name structure.

```

int [gnutls_x509_cert_get_subject], page 420 (gnutls_x509_cert_t cert,
gnutls_x509_dn_t * dn)
int [gnutls_x509_cert_get_issuer], page 411 (gnutls_x509_cert_t cert,
gnutls_x509_dn_t * dn)

int gnutls_x509_dn_get_rdn_ava (gnutls_x509_dn_t dn, int irdn, int iava, gnutls_x509_ava_st * ava) [Function]

```

*dn*: a pointer to DN

*irdn*: index of RDN

*iava*: index of AVA.

*ava*: Pointer to structure which will hold output information.

Get pointers to data within the DN. The format of the `ava` structure is shown below.

```

struct gnutls_x509_ava_st { gnutls_datum_t oid; gnutls_datum_t value; unsigned long
value_tag; };

```

The X.509 distinguished name is a sequence of sequences of strings and this is what the `irdn` and `iava` indexes model.

Note that `ava` will contain pointers into the `dn` structure which in turns points to the original certificate. Thus you should not modify any data or deallocate any of those.

This is a low-level function that requires the caller to do the value conversions when necessary (e.g. from UCS-2).

**Returns:** Returns 0 on success, or an error code.

#### 4.1.1.4 X.509 extensions

X.509 version 3 certificates include a list of extensions that can be used to obtain additional information on the subject or the issuer of the certificate. Those may be e-mail addresses, flags that indicate whether the belongs to a CA etc. All the supported X.509 version 3 extensions are shown in [Table 4.3](#).

The certificate extensions access is split into two parts. The first requires to retrieve the extension, and the second is the parsing part.

To enumerate and retrieve the DER-encoded extension data available in a certificate the following two functions are available.

```

int [gnutls_x509_cert_get_extension_info], page 410 (gnutls_x509_cert_t cert,
int indx, void * oid, size_t * oid_size, unsigned int * critical)
int [gnutls_x509_cert_get_extension_data2], page 409 (gnutls_x509_cert_t cert,
unsigned indx, gnutls_datum_t * data)
int [gnutls_x509_cert_get_extension_by_oid2], page 408 (gnutls_x509_cert_t
cert, const char * oid, int indx, gnutls_datum_t * output, unsigned int *
critical)

```

After a supported DER-encoded extension is retrieved it can be parsed using the APIs in `x509-ext.h`. Complex extensions may require initializing an intermediate structure that holds the parsed extension data. Examples of simple parsing functions are shown below.

```

int [gnutls_x509_ext_import_basic_constraints], page 442 (const
gnutls_datum_t * ext, unsigned int * ca, int * pathlen)
int [gnutls_x509_ext_export_basic_constraints], page 438 (unsigned int ca,
int pathlen, gnutls_datum_t * ext)
int [gnutls_x509_ext_import_key_usage], page 442 (const gnutls_datum_t * ext,
unsigned int * key_usage)
int [gnutls_x509_ext_export_key_usage], page 439 (unsigned int usage,
gnutls_datum_t * ext)

```

More complex extensions, such as Name Constraints, require an intermediate structure, in that case `gnutls_x509_name_constraints_t` to be initialized in order to store the parsed extension data.

```

int [gnutls_x509_ext_import_name_constraints], page 443 (const
gnutls_datum_t * ext, gnutls_x509_name_constraints_t nc, unsigned int flags)
int [gnutls_x509_ext_export_name_constraints], page 439
(gnutls_x509_name_constraints_t nc, gnutls_datum_t * ext)

```

After the name constraints are extracted in the structure, the following functions can be used to access them.



```
int [gnutls_x509_name_constraints_get_permitted], page 448
(gnutls_x509_name_constraints_t nc, unsigned idx, unsigned * type,
 gnutls_datum_t * name)
int [gnutls_x509_name_constraints_get_excluded], page 447
(gnutls_x509_name_constraints_t nc, unsigned idx, unsigned * type,
 gnutls_datum_t * name)
int [gnutls_x509_name_constraints_add_permitted], page 446
(gnutls_x509_name_constraints_t nc, gnutls_x509_subject_alt_name_t type,
 const gnutls_datum_t * name)
int [gnutls_x509_name_constraints_add_excluded], page 446
(gnutls_x509_name_constraints_t nc, gnutls_x509_subject_alt_name_t type,
 const gnutls_datum_t * name)

unsigned [gnutls_x509_name_constraints_check], page 447
(gnutls_x509_name_constraints_t nc, gnutls_x509_subject_alt_name_t type,
 const gnutls_datum_t * name)
unsigned [gnutls_x509_name_constraints_check_cert], page 447
(gnutls_x509_name_constraints_t nc, gnutls_x509_subject_alt_name_t type,
 gnutls_x509_cert_t cert)
```

Other utility functions are listed below.

```
int [gnutls_x509_name_constraints_init], page 448
(gnutls_x509_name_constraints_t * nc)
void [gnutls_x509_name_constraints_deinit], page 447
(gnutls_x509_name_constraints_t nc)
```

Similar functions exist for all of the other supported extensions, listed in [Table 4.3](#).

<b>Extension</b>	<b>OID</b>	<b>Description</b>
Subject key id	2.5.29.14	An identifier of the key of the subject.
Key usage	2.5.29.15	Constraints the key's usage of the certificate.
Private key usage period	2.5.29.16	Constraints the validity time of the private key.
Subject alternative name	2.5.29.17	Alternative names to subject's distinguished name.
Issuer alternative name	2.5.29.18	Alternative names to the issuer's distinguished name.
Basic constraints	2.5.29.19	Indicates whether this is a CA certificate or not, and specify the maximum path lengths of certificate chains.
Name constraints	2.5.29.30	A field in CA certificates that restricts the scope of the name of issued certificates.
CRL distribution points	2.5.29.31	This extension is set by the CA, in order to inform about the issued CRLs.
Certificate policy	2.5.29.32	This extension is set to indicate the certificate policy as object identifier and may contain a descriptive string or URL.
Authority key identifier	2.5.29.35	An identifier of the key of the issuer of the certificate. That is used to distinguish between different keys of the same issuer.
Extended key usage	2.5.29.37	Constraints the purpose of the certificate.
Authority information access	1.3.6.1.5.5.7.1.1	Information on services by the issuer of the certificate.
Proxy Certification Information	1.3.6.1.5.5.7.1.14	Proxy Certificates includes this extension that contains the OID of the proxy policy language used, and can specify limits on the maximum lengths of proxy chains.

Note, that there are also direct APIs to access extensions that may be simpler to use for non-complex extensions. They are available in `x509.h` and some examples are listed below.

```
int [gnutls_x509_cert_get_basic_constraints], page 405 (gnutls_x509_cert_t
cert, unsigned int * critical, unsigned int * ca, int * pathlen)
int [gnutls_x509_cert_set_basic_constraints], page 425 (gnutls_x509_cert_t
cert, unsigned int ca, int pathLenConstraint)
int [gnutls_x509_cert_get_key_usage], page 415 (gnutls_x509_cert_t cert,
unsigned int * key_usage, unsigned int * critical)
int [gnutls_x509_cert_set_key_usage], page 430 (gnutls_x509_cert_t cert,
unsigned int usage)
```

#### 4.1.1.5 Accessing public and private keys

Each X.509 certificate contains a public key that corresponds to a private key. To get a unique identifier of the public key the `[gnutls_x509_cert_get_key_id]`, page 415 function is provided. To export the public key or its parameters you may need to convert the X.509 structure to a `gnutls_pubkey_t`. See [Section 5.1.1 \[Abstract public keys\]](#), page 82 for more information.

```
int gnutls_x509_cert_get_key_id (gnutls_x509_cert_t cert, unsigned int [Function]
                                int flags, unsigned char * output_data, size_t * output_data_size)
    cert: Holds the certificate
```

*flags*: should be 0 for now

*output\_data*: will contain the key ID

*output\_data\_size*: holds the size of *output\_data* (and will be replaced by the actual size of parameters)

This function will return a unique ID that depends on the public key parameters. This ID can be used in checking whether a certificate corresponds to the given private key.

If the buffer provided is not long enough to hold the output, then *\*output\_data\_size* is updated and `GNUTLS_E_SHORT_MEMORY_BUFFER` will be returned. The output will normally be a SHA-1 hash output, which is 20 bytes.

**Returns:** In case of failure a negative error code will be returned, and 0 on success.

The private key parameters may be directly accessed by using one of the following functions.

```
int [gnutls_x509_privkey_get_pk_algorithm2], page 455 (gnutls_x509_privkey_t
key, unsigned int * bits)
int [gnutls_x509_privkey_export_rsa_raw2], page 453 (gnutls_x509_privkey_t
key, gnutls_datum_t * m, gnutls_datum_t * e, gnutls_datum_t * d, gnutls_datum_t
```

```

* p, gnutls_datum_t * q, gnutls_datum_t * u, gnutls_datum_t * e1,
  gnutls_datum_t * e2)
int [gnutls_x509_privkey_export_ecc_raw], page 452 (gnutls_x509_privkey_t
key, gnutls_ecc_curve_t * curve, gnutls_datum_t * x, gnutls_datum_t * y,
  gnutls_datum_t * k)
int [gnutls_x509_privkey_export_dsa_raw], page 452 (gnutls_x509_privkey_t
key, gnutls_datum_t * p, gnutls_datum_t * q, gnutls_datum_t * g, gnutls_datum_t
* y, gnutls_datum_t * x)
int [gnutls_x509_privkey_get_key_id], page 454 (gnutls_x509_privkey_t key,
  unsigned int flags, unsigned char * output_data, size_t * output_data_size)

```

#### 4.1.1.6 Verifying X.509 certificate paths

Verifying certificate paths is important in X.509 authentication. For this purpose the following functions are provided.

```

int gnutls_x509_trust_list_add_cas (gnutls_x509_trust_list_t [Function]
  list, const gnutls_x509_crt_t * clist, unsigned clist_size, unsigned int
  flags)

```

*list*: The structure of the list

*clist*: A list of CAs

*clist\_size*: The length of the CA list

*flags*: should be 0 or an or'ed sequence of GNUTLS\_TL options.

This function will add the given certificate authorities to the trusted list. The list of CAs must not be deinitialized during this structure's lifetime.

If the flag GNUTLS\_TL\_NO\_DUPLICATES is specified, then the provided *clist* entries that are duplicates will not be added to the list and will be deinitialized.

**Returns:** The number of added elements is returned.

**Since:** 3.0.0

```

int gnutls_x509_trust_list_add_named_crt [Function]
  (gnutls_x509_trust_list_t list, gnutls_x509_crt_t cert, const void * name,
  size_t name_size, unsigned int flags)

```

*list*: The structure of the list

*cert*: A certificate

*name*: An identifier for the certificate

*name\_size*: The size of the identifier

*flags*: should be 0.

This function will add the given certificate to the trusted list and associate it with a name. The certificate will not be used for verification with `gnutls_x509_trust_list_verify_crt()` but only with `gnutls_x509_trust_list_verify_named_crt()`.

In principle this function can be used to set individual "server" certificates that are trusted by the user for that specific server but for no other purposes.

The certificate must not be deinitialized during the lifetime of the trusted list.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.0.0

```
int gnutls_x509_trust_list_add_crls (gnutls_x509_trust_list_t [Function]
    list, const gnutls_x509_crl_t *crl_list, int crl_size, unsigned int flags,
    unsigned int verification_flags)
```

*list*: The structure of the list

*crl\_list*: A list of CRLs

*crl\_size*: The length of the CRL list

*flags*: if GNUTLS\_TL\_VERIFY\_CRL is given the CRLs will be verified before being added.

*verification\_flags*: gnutls\_certificate\_verify\_flags if flags specifies GNUTLS\_TL\_VERIFY\_CRL

This function will add the given certificate revocation lists to the trusted list. The list of CRLs must not be deinitialized during this structure's lifetime.

This function must be called after `gnutls_x509_trust_list_add_cas()` to allow verifying the CRLs for validity. If the flag GNUTLS\_TL\_NO\_DUPLICATES is given, then any provided CRLs that are a duplicate, will be deinitialized and not added to the list (that assumes that `gnutls_x509_trust_list_deinit()` will be called with `all=1`).

**Returns:** The number of added elements is returned.

**Since:** 3.0

```
int gnutls_x509_trust_list_verify_cert (gnutls_x509_trust_list_t [Function]
    list, gnutls_x509_cert_t *cert_list, unsigned int cert_list_size,
    unsigned int flags, unsigned int *voutput, gnutls_verify_output_function
    func)
```

*list*: The structure of the list

*cert\_list*: is the certificate list to be verified

*cert\_list\_size*: is the certificate list size

*flags*: Flags that may be used to change the verification algorithm. Use OR of the gnutls\_certificate\_verify\_flags enumerations.

*voutput*: will hold the certificate verification output.

*func*: If non-null will be called on each chain element verification with the output.

This function will try to verify the given certificate and return its status. The `verify` parameter will hold an OR'ed sequence of `gnutls_certificate_status_t` flags.

Additionally a certificate verification profile can be specified from the ones in `gnutls_certificate_verification_profiles_t` by ORing the result of `GNUTLS_PROFILE_TO_VFLAGS()` to the verification flags.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.0

```
int gnutls_x509_trust_list_verify_cert2 (gnutls_x509_trust_list_t [Function]
    list, gnutls_x509_cert_t * cert_list, unsigned int cert_list_size,
    gnutls_typed_vdata_st * data, unsigned int elements, unsigned int flags,
    unsigned int * voutput, gnutls_verify_output_function func)
```

*list*: The structure of the list

*cert\_list*: is the certificate list to be verified

*cert\_list\_size*: is the certificate list size

*data*: an array of typed data

*elements*: the number of data elements

*flags*: Flags that may be used to change the verification algorithm. Use OR of the `gnutls_certificate_verify_flags` enumerations.

*voutput*: will hold the certificate verification output.

*func*: If non-null will be called on each chain element verification with the output.

This function will try to verify the given certificate and return its status. The `verify` parameter will hold an OR'ed sequence of `gnutls_certificate_status_t` flags.

Additionally a certificate verification profile can be specified from the ones in `gnutls_certificate_verification_profiles_t` by ORing the result of `GNUTLS_PROFILE_TO_VFLAGS()` to the verification flags.

The acceptable `data` types are `GNUTLS_DT_DNS_HOSTNAME` and `GNUTLS_DT_KEY_PURPOSE_OID`. The former accepts as data a null-terminated hostname, and the latter a null-terminated object identifier (e.g., `GNUTLS_KP_TLS_WWW_SERVER`). If a DNS hostname is provided then this function will compare the hostname in the certificate against the given. If names do not match the `GNUTLS_CERT_UNEXPECTED_OWNER` status flag will be set. If a key purpose OID is provided and the end-certificate contains the extended key usage PKIX extension, it will be required to be have the provided key purpose or be marked for any purpose, otherwise verification will fail with `GNUTLS_CERT_SIGNER_CONSTRAINTS_FAILURE` status.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value. Note that verification failure will not result to an error code, only `voutput` will be updated.

**Since:** 3.3.8

```
int gnutls_x509_trust_list_verify_named_cert [Function]
    (gnutls_x509_trust_list_t list, gnutls_x509_cert_t cert, const void * name,
    size_t name_size, unsigned int flags, unsigned int * voutput,
    gnutls_verify_output_function func)
```

*list*: The structure of the list

*cert*: is the certificate to be verified

*name*: is the certificate's name

*name\_size*: is the certificate's name size

*flags*: Flags that may be used to change the verification algorithm. Use OR of the `gnutls_certificate_verify_flags` enumerations.

*voutput*: will hold the certificate verification output.

*func*: If non-null will be called on each chain element verification with the output.

This function will try to find a certificate that is associated with the provided name – see `gnutls_x509_trust_list_add_named_cert()`. If a match is found the certificate is considered valid. In addition to that this function will also check CRLs. The *voutput* parameter will hold an OR'ed sequence of `gnutls_certificate_status_t` flags.

Additionally a certificate verification profile can be specified from the ones in `gnutls_certificate_verification_profiles_t` by ORing the result of `GNUTLS_PROFILE_TO_VFLAGS()` to the verification flags.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.0.0

`int gnutls_x509_trust_list_add_trust_file` [Function]

(*gnutls\_x509\_trust\_list\_t list*, *const char \* ca\_file*, *const char \* crl\_file*,  
*gnutls\_x509\_cert\_fmt\_t type*, *unsigned int tl\_flags*, *unsigned int tl\_vflags*)

*list*: The structure of the list

*ca\_file*: A file containing a list of CAs (optional)

*crl\_file*: A file containing a list of CRLs (optional)

*type*: The format of the certificates

*tl\_flags*: `GNUTLS_TL_*`

*tl\_vflags*: `gnutls_certificate_verify_flags` if flags specifies `GNUTLS_TL_VERIFY_CRL`

This function will add the given certificate authorities to the trusted list. PKCS 11 URLs are also accepted, instead of files, by this function. A PKCS 11 URL implies a trust database (a specially marked module in p11-kit); the URL "pkcs11:" implies all trust databases in the system. Only a single URL specifying trust databases can be set; they cannot be stacked with multiple calls.

**Returns:** The number of added elements is returned.

**Since:** 3.1

`int gnutls_x509_trust_list_add_trust_mem` [Function]

(*gnutls\_x509\_trust\_list\_t list*, *const gnutls\_datum\_t \* cas*, *const gnutls\_datum\_t \* crls*,  
*gnutls\_x509\_cert\_fmt\_t type*, *unsigned int tl\_flags*, *unsigned int tl\_vflags*)

*list*: The structure of the list

*cas*: A buffer containing a list of CAs (optional)

*crls*: A buffer containing a list of CRLs (optional)

*type*: The format of the certificates

*tl\_flags*: `GNUTLS_TL_*`

*tl\_vflags*: `gnutls_certificate_verify_flags` if flags specifies `GNUTLS_TL_VERIFY_CRL`

This function will add the given certificate authorities to the trusted list.

**Returns:** The number of added elements is returned.

**Since:** 3.1

```
int gnutls_x509_trust_list_add_system_trust [Function]
    (gnutls_x509_trust_list_t list, unsigned int tl_flags, unsigned int
     tl_vflags)
```

*list*: The structure of the list

*tl\_flags*: GNUTLS\_TL\_\*

*tl\_vflags*: gnutls\_certificate\_verify\_flags if flags specifies GNUTLS\_TL\_VERIFY\_CRL

This function adds the system's default trusted certificate authorities to the trusted list. Note that on unsupported systems this function returns GNUTLS\_E\_UNIMPLEMENTED\_FEATURE .

This function implies the flag GNUTLS\_TL\_NO\_DUPLICATES .

**Returns:** The number of added elements or a negative error code on error.

**Since:** 3.1

The verification function will verify a given certificate chain against a list of certificate authorities and certificate revocation lists, and output a bit-wise OR of elements of the `gnutls_certificate_status_t` enumeration shown in [Figure 4.2](#). The GNUTLS\_CERT\_INVALID flag is always set on a verification error and more detailed flags will also be set when appropriate.



**GNUTLS\_CERT\_INVALID**

The certificate is not signed by one of the known authorities or the signature is invalid (deprecated by the flags **GNUTLS\_CERT\_SIGNATURE\_FAILURE** and **GNUTLS\_CERT\_SIGNER\_NOT\_FOUND** ).

**GNUTLS\_CERT\_REVOKED**

Certificate is revoked by its authority. In X.509 this will be set only if CRLs are checked.

**GNUTLS\_CERT\_SIGNER\_NOT\_FOUND**

The certificate's issuer is not known. This is the case if the issuer is not included in the trusted certificate list.

**GNUTLS\_CERT\_SIGNER\_NOT\_CA**

The certificate's signer was not a CA. This may happen if this was a version 1 certificate, which is common with some CAs, or a version 3 certificate without the basic constraints extension.

**GNUTLS\_CERT\_INSECURE\_ALGORITHM**

The certificate was signed using an insecure algorithm such as MD2 or MD5. These algorithms have been broken and should not be trusted.

**GNUTLS\_CERT\_NOT\_ACTIVATED**

The certificate is not yet activated.

**GNUTLS\_CERT\_EXPIRED**

The certificate has expired.

**GNUTLS\_CERT\_SIGNATURE\_FAILURE**

The signature verification failed.

**GNUTLS\_CERT\_REVOCATION\_DATA\_SUPERSEDED**

The revocation data are old and have been superseded.

**GNUTLS\_CERT\_UNEXPECTED\_OWNER**

The owner is not the expected one.

**GNUTLS\_CERT\_REVOCATION\_DATA\_ISSUED\_IN\_FUTURE**

The revocation data have a future issue date.

**GNUTLS\_CERT\_SIGNER\_CONSTRAINTS\_FAILURE**

The certificate's signer constraints were violated.

**GNUTLS\_CERT\_MISMATCH**

The certificate presented isn't the expected one (TOFU)

Figure 4.2: The `gnutls_certificate_status_t` enumeration.

An example of certificate verification is shown in [\[ex-verify2\]](#), page 170. It is also possible to have a set of certificates that are trusted for a particular server but not to authorize other certificates. This purpose is served by the functions [\[gnutls\\_x509\\_trust\\_list\\_add\\_named\\_cert\]](#), page 461 and [\[gnutls\\_x509\\_trust\\_list\\_verify\\_named\\_cert\]](#), page 466.

#### 4.1.1.7 Verifying a certificate in the context of TLS session

When operating in the context of a TLS session, the trusted certificate authority list may also be set using:

```
int [gnutls_certificate_set_x509_trust_file], page 290
(gnutls_certificate_credentials_t cred, const char * cafile,
gnutls_x509_crt_fmt_t type)
int [gnutls_certificate_set_x509_trust_dir], page 290
(gnutls_certificate_credentials_t cred, const char * ca_dir,
gnutls_x509_crt_fmt_t type)
int [gnutls_certificate_set_x509_crl_file], page 285
(gnutls_certificate_credentials_t res, const char * crlfile,
gnutls_x509_crt_fmt_t type)
int [gnutls_certificate_set_x509_system_trust], page 289
(gnutls_certificate_credentials_t cred)
```

These functions allow the specification of the trusted certificate authorities, either via a file, a directory or use the system-specified certificate authorities. Unless the authorities are application specific, it is generally recommended to use the system trust storage (see [\[gnutls\\_certificate\\_set\\_x509\\_system\\_trust\]](#), page 289).

Unlike the previous section it is not required to setup a trusted list, and the function [\[gnutls\\_certificate\\_verify\\_peers3\]](#), page 293 is used to verify the peer's certificate chain and identity. The reported verification status is identical to the verification functions described in the previous section. Note that in certain cases it is required to check the marked purpose of the end certificate (e.g. GNUTLS\_KP\_TLS\_WWW\_SERVER); in these cases the more advanced [\[gnutls\\_certificate\\_verify\\_peers\]](#), page 292 should be used instead.

There is also the possibility to pass some input to the verification functions in the form of flags. For [\[gnutls\\_x509\\_trust\\_list\\_verify\\_cert2\]](#), page 465 the flags are passed directly, but for [\[gnutls\\_certificate\\_verify\\_peers3\]](#), page 293, the flags are set using [\[gnutls\\_certificate\\_set\\_verify\\_flags\]](#), page 284. All the available flags are part of the enumeration `gnutls_certificate_verify_flags` shown in [Figure 4.3](#).

<code>GNUTLS_VERIFY_DISABLE_CA_SIGN</code>	If set a signer does not have to be a certificate authority. This flag should normally be disabled, unless you know what this means.
<code>GNUTLS_VERIFY_DO_NOT_ALLOW_SAME</code>	If a certificate is not signed by anyone trusted but exists in the trusted CA list do not treat it as trusted.
<code>GNUTLS_VERIFY_ALLOW_ANY_X509_V1_CA_CRT</code>	Allow CA certificates that have version 1 (both root and intermediate). This might be dangerous since those haven't the basicConstraints extension.
<code>GNUTLS_VERIFY_ALLOW_SIGN_RSA_MD2</code>	Allow certificates to be signed using the broken MD2 algorithm.
<code>GNUTLS_VERIFY_ALLOW_SIGN_RSA_MD5</code>	Allow certificates to be signed using the broken MD5 algorithm.
<code>GNUTLS_VERIFY_DISABLE_TIME_CHECKS</code>	Disable checking of activation and expiration validity periods of certificate chains. Don't set this unless you understand the security implications.
<code>GNUTLS_VERIFY_DISABLE_TRUSTED_TIME_CHECKS</code>	If set a signer in the trusted list is never checked for expiration or activation.
<code>GNUTLS_VERIFY_DO_NOT_ALLOW_X509_V1_CA_CRT</code>	Do not allow trusted CA certificates that have version 1. This option is to be used to deprecate all certificates of version 1.
<code>GNUTLS_VERIFY_DISABLE_CRL_CHECKS</code>	Disable checking for validity using certificate revocation lists or the available OCSP data.
<code>GNUTLS_VERIFY_ALLOW_UNSORTED_CHAIN</code>	A certificate chain is tolerated if unsorted (the case with many TLS servers out there). This is the default since GnuTLS 3.1.4.
<code>GNUTLS_VERIFY_DO_NOT_ALLOW_UNSORTED_CHAIN</code>	Do not tolerate an unsorted certificate chain.
<code>GNUTLS_VERIFY_DO_NOT_ALLOW_WILDCARDS</code>	When including a hostname check in the verification, do not consider any wildcards.

Figure 4.3: The `gnutls_certificate_verify_flags` enumeration.

#### 4.1.1.8 Verifying a certificate using PKCS #11

Some systems provide a system wide trusted certificate storage accessible using the PKCS #11 API. That is, the trusted certificates are queried and accessed using the PKCS #11

API, and trusted certificate properties, such as purpose, are marked using attached extensions. One example is the p11-kit trust module<sup>1</sup>.

These special PKCS #11 modules can be used for GnuTLS certificate verification if marked as trust policy modules, i.e., with `trust-policy: yes` in the p11-kit module file. The way to use them is by specifying to the file verification function (e.g., `[gnutls_certificate_set_x509_trust_file]`, page 290), a pkcs11 URL, or simply `pkcs11:` to use all the marked with trust policy modules.

The trust modules of p11-kit assign a purpose to trusted authorities using the extended key usage object identifiers. The common purposes are shown in Table 4.4. Note that typically according to [RFC5280] the extended key usage object identifiers apply to end certificates. Their application to CA certificates is an extension used by the trust modules.

Purpose	OID	Description
GNUTLS_KP_TLS_WWW_SERVER	2.5.29.32	The certificate is to be used for TLS WWW authentication. When in a CA certificate, it indicates that the CA is allowed to sign certificates for TLS WWW authentication.
GNUTLS_KP_TLS_WWW_CLIENT	2.5.29.33	The certificate is to be used for TLS WWW client authentication. When in a CA certificate, it indicates that the CA is allowed to sign certificates for TLS WWW client authentication.
GNUTLS_KP_CODE_SIGNING	2.5.29.36	The certificate is to be used for code signing. When in a CA certificate, it indicates that the CA is allowed to sign certificates for code signing.
GNUTLS_KP_EMAIL_PROTECTION	2.5.29.37	The certificate is to be used for email protection. When in a CA certificate, it indicates that the CA is allowed to sign certificates for email users.
GNUTLS_KP_OCSP_SIGNING	2.5.29.39	The certificate is to be used for signing OCSP responses. When in a CA certificate, it indicates that the CA is allowed to sign certificates which sign OCSP responses.
GNUTLS_KP_ANY	2.5.29.37.0	The certificate is to be used for any purpose. When in a CA certificate, it indicates that the CA is allowed to sign any kind of certificates.

Table 4.4: Key purpose object identifiers.

<sup>1</sup> see <http://p11-glue.freedesktop.org/trust-module.html>.

With such modules, it is recommended to use the verification functions `[gnutls_x509_trust_list_verify_cert2]`, page 465, or `[gnutls_certificate_verify_peers]`, page 292, which allow to explicitly specify the key purpose. The other verification functions which do not allow setting a purpose, would operate as if `GNUTLS_KP_TLS_WWW_SERVER` was requested from the trusted authorities.

### 4.1.2 OpenPGP certificates

The OpenPGP key authentication relies on a distributed trust model, called the “web of trust”. The “web of trust” uses a decentralized system of trusted introducers, which are the same as a CA. OpenPGP allows anyone to sign anyone else’s public key. When Alice signs Bob’s key, she is introducing Bob’s key to anyone who trusts Alice. If someone trusts Alice to introduce keys, then Alice is a trusted introducer in the mind of that observer. For example in Figure 4.4, David trusts Alice to be an introducer and Alice signed Bob’s key thus Dave trusts Bob’s key to be the real one.

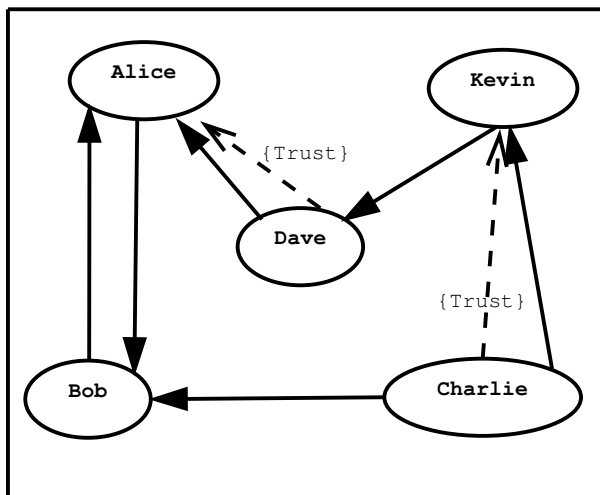


Figure 4.4: An example of the OpenPGP trust model.

There are some key points that are important in that model. In the example Alice has to sign Bob’s key, only if she is sure that the key belongs to Bob. Otherwise she may also make Dave falsely believe that this is Bob’s key. Dave has also the responsibility to know who to trust. This model is similar to real life relations.

Just see how Charlie behaves in the previous example. Although he has signed Bob’s key - because he knows, somehow, that it belongs to Bob - he does not trust Bob to be an introducer. Charlie decided to trust only Kevin, for some reason. A reason could be that Bob is lazy enough, and signs other people’s keys without being sure that they belong to the actual owner.

Field	Description
version	The field that indicates the version of the OpenPGP structure.
user ID	An RFC 2822 string that identifies the owner of the key. There may be multiple user identifiers in a key.
public key	The main public key of the certificate.
expiration	The expiration time of the main public key.
public subkey	An additional public key of the certificate. There may be multiple subkeys in a certificate.
public subkey expiration	The expiration time of the subkey.

Table 4.5: OpenPGP certificate fields.

#### 4.1.2.1 OpenPGP certificate structure

In GnuTLS the OpenPGP certificate structures [RFC2440] are handled using the `gnutls_openpgp_cert_t` type. A typical certificate contains the user ID, which is an RFC 2822 mail and name address, a public key, possibly a number of additional public keys (called subkeys), and a number of signatures. The various fields are shown in Table 4.5.

The additional subkeys may provide key for various different purposes, e.g. one key to encrypt mail, and another to sign a TLS key exchange. Each subkey is identified by a unique key ID. The keys that are to be used in a TLS key exchange that requires signatures are called authentication keys in the OpenPGP jargon. The mapping of TLS key exchange methods to public keys is shown in Table 4.6.

Key exchange	Public key requirements
RSA	An RSA public key that allows encryption.
DHE_RSA	An RSA public key that is marked for authentication.
ECDHE_RSA	An RSA public key that is marked for authentication.
DHE_DSS	A DSA public key that is marked for authentication.

Table 4.6: The types of (sub)keys required for the various TLS key exchange methods.

The corresponding private keys are stored in the `gnutls_openpgp_privkey_t` type. All the prototypes for the key handling functions can be found in `gnutls/openpgp.h`.

### 4.1.2.2 Verifying an OpenPGP certificate

The verification functions of OpenPGP keys, included in GnuTLS, are simple ones, and do not use the features of the “web of trust”. For that reason, if the verification needs are complex, the assistance of external tools like GnuPG and GPGME<sup>2</sup> is recommended.

In GnuTLS there is a verification function for OpenPGP certificates, the `[gnutls_openpgp_cert_verify_ring]`, page 488. This checks an OpenPGP key against a given set of public keys (keyring) and returns the key status. The key verification status is the same as in X.509 certificates, although the meaning and interpretation are different. For example an OpenPGP key may be valid, if the self signature is ok, even if no signers were found. The meaning of verification status flags is the same as in the X.509 certificates (see Figure 4.3).

```
int gnutls_openpgp_cert_verify_ring (gnutls_openpgp_cert_t key,      [Function]
                                     gnutls_openpgp_keyring_t keyring, unsigned int flags, unsigned int *
                                     verify)
```

*key*: the structure that holds the key.

*keyring*: holds the keyring to check against

*flags*: unused (should be 0)

*verify*: will hold the certificate verification output.

Verify all signatures in the key, using the given set of keys (keyring).

The key verification output will be put in `verify` and will be one or more of the `gnutls_certificate_status_t` enumerated elements bitwise or'd.

Note that this function does not verify using any “web of trust”. You may use GnuPG for that purpose, or any other external PGP application.

**Returns:** GNUTLS\_E\_SUCCESS on success, or an error code.

```
int gnutls_openpgp_cert_verify_self (gnutls_openpgp_cert_t key,      [Function]
                                     unsigned int flags, unsigned int * verify)
```

*key*: the structure that holds the key.

*flags*: unused (should be 0)

*verify*: will hold the key verification output.

Verifies the self signature in the key. The key verification output will be put in `verify` and will be one or more of the `gnutls_certificate_status_t` enumerated elements bitwise or'd.

**Returns:** GNUTLS\_E\_SUCCESS on success, or an error code.

### 4.1.2.3 Verifying a certificate in the context of a TLS session

Similarly with X.509 certificates, one needs to specify the OpenPGP keyring file in the credentials structure. The certificates in this file will be used by `[gnutls_certificate_verify_peers3]`, page 293 to verify the signatures in the certificate sent by the peer.

<sup>2</sup> [http://www.gnupg.org/related\\_software/gpgme/](http://www.gnupg.org/related_software/gpgme/)

```
int gnutls_certificate_set_openpgp_keyring_file           [Function]
    (gnutls_certificate_credentials_t c, const char * file, gnutls_openpgp_cert_fmt_t
    format)
```

*c*: A certificate credentials structure

*file*: filename of the keyring.

*format*: format of keyring.

The function is used to set keyrings that will be used internally by various OpenPGP functions. For example to find a key when it is needed for an operations. The keyring will also be used at the verification functions.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

### 4.1.3 Advanced certificate verification

The verification of X.509 certificates in the HTTPS and other Internet protocols is typically done by loading a trusted list of commercial Certificate Authorities (see [\[gnutls\\_certificate\\_set\\_x509\\_system\\_trust\]](#), page 289), and using them as trusted anchors. However, there are several examples (eg. the Diginotar incident) where one of these authorities was compromised. This risk can be mitigated by using in addition to CA certificate verification, other verification methods. In this section we list the available in GnuTLS methods.

#### 4.1.3.1 Verifying a certificate using trust on first use authentication

It is possible to use a trust on first use (TOFU) authentication method in GnuTLS. That is the concept used by the SSH programs, where the public key of the peer is not verified, or verified in an out-of-bound way, but subsequent connections to the same peer require the public key to remain the same. Such a system in combination with the typical CA verification of a certificate, and OCSP revocation checks, can help to provide multiple factor verification, where a single point of failure is not enough to compromise the system. For example a server compromise may be detected using OCSP, and a CA compromise can be detected using the trust on first use method. Such a hybrid system with X.509 and trust on first use authentication is shown in [Section 7.1.2 \[Simple client example with SSH-style certificate verification\]](#), page 153.

See [Section 6.12.2 \[Certificate verification\]](#), page 142 on how to use the available functionality.

#### 4.1.3.2 Verifying a certificate using DANE (DNSSEC)

The DANE protocol is a protocol that can be used to verify TLS certificates using the DNS (or better DNSSEC) protocols. The DNS security extensions (DNSSEC) provide an alternative public key infrastructure to the commercial CAs that are typically used to sign TLS certificates. The DANE protocol takes advantage of the DNSSEC infrastructure to verify TLS certificates. This can be in addition to the verification by CA infrastructure or may even replace it where DNSSEC is fully deployed. Note however, that DNSSEC deployment is fairly new and it would be better to use it as an additional verification method rather than the only one.



The DANE functionality is provided by the `libgnutls-dane` library that is shipped with GnuTLS and the function prototypes are in `gnutls/dane.h`. See [Section 6.12.2 \[Certificate verification\]](#), page 142 for information on how to use the library.

Note however, that the DANE RFC mandates the verification methods one should use in addition to the validation via DNSSEC TLSA entries. GnuTLS doesn't follow that RFC requirement, and the term DANE verification in this manual refers to the TLSA entry verification. In GnuTLS any other verification methods can be used (e.g., PKIX or TOFU) on top of DANE.

#### 4.1.4 Digital signatures

In this section we will provide some information about digital signatures, how they work, and give the rationale for disabling some of the algorithms used.

Digital signatures work by using somebody's secret key to sign some arbitrary data. Then anybody else could use the public key of that person to verify the signature. Since the data may be arbitrary it is not suitable input to a cryptographic digital signature algorithm. For this reason and also for performance cryptographic hash algorithms are used to preprocess the input to the signature algorithm. This works as long as it is difficult enough to generate two different messages with the same hash algorithm output. In that case the same signature could be used as a proof for both messages. Nobody wants to sign an innocent message of donating 1 euro to Greenpeace and find out that they donated 1.000.000 euros to Bad Inc.

For a hash algorithm to be called cryptographic the following three requirements must hold:

1. Preimage resistance. That means the algorithm must be one way and given the output of the hash function  $H(x)$ , it is impossible to calculate  $x$ .
2. 2nd preimage resistance. That means that given a pair  $x, y$  with  $y = H(x)$  it is impossible to calculate an  $x'$  such that  $y = H(x')$ .
3. Collision resistance. That means that it is impossible to calculate random  $x$  and  $x'$  such  $H(x') = H(x)$ .

The last two requirements in the list are the most important in digital signatures. These protect against somebody who would like to generate two messages with the same hash output. When an algorithm is considered broken usually it means that the Collision resistance of the algorithm is less than brute force. Using the birthday paradox the brute force attack takes  $2^{(\text{hash size})/2}$  operations. Today colliding certificates using the MD5 hash algorithm have been generated as shown in [WEGER].

There has been cryptographic results for the SHA-1 hash algorithms as well, although they are not yet critical. Before 2004, MD5 had a presumed collision strength of  $2^{64}$ , but it has been showed to have a collision strength well under  $2^{50}$ . As of November 2005, it is believed that SHA-1's collision strength is around  $2^{63}$ . We consider this sufficiently hard so that we still support SHA-1. We anticipate that SHA-256/386/512 will be used in publicly-distributed certificates in the future. When  $2^{63}$  can be considered too weak compared to the computer power available sometime in the future, SHA-1 will be disabled as well. The collision attacks on SHA-1 may also get better, given the new interest in tools for creating them.

#### 4.1.4.1 Trading security for interoperability

If you connect to a server and use GnuTLS' functions to verify the certificate chain, and get a `GNUTLS_CERT_INSECURE_ALGORITHM` validation error (see [Section 4.1.1.6 \[Verifying X.509 certificate paths\]](#), page 28), it means that somewhere in the certificate chain there is a certificate signed using RSA-MD2 or RSA-MD5. These two digital signature algorithms are considered broken, so GnuTLS fails verifying the certificate. In some situations, it may be useful to be able to verify the certificate chain anyway, assuming an attacker did not utilize the fact that these signatures algorithms are broken. This section will give help on how to achieve that.

It is important to know that you do not have to enable any of the flags discussed here to be able to use trusted root CA certificates self-signed using RSA-MD2 or RSA-MD5. The certificates in the trusted list are considered trusted irrespective of the signature.

If you are using [\[gnutls\\_certificate\\_verify\\_peers3\]](#), page 293 to verify the certificate chain, you can call [\[gnutls\\_certificate\\_set\\_verify\\_flags\]](#), page 284 with the flags:

- `GNUTLS_VERIFY_ALLOW_SIGN_RSA_MD2`
- `GNUTLS_VERIFY_ALLOW_SIGN_RSA_MD5`

as in the following example:

```
gnutls_certificate_set_verify_flags (x509cred,  
                                     GNUTLS_VERIFY_ALLOW_SIGN_RSA_MD5);
```

This will signal the verifier algorithm to enable RSA-MD5 when verifying the certificates.

If you are using [\[gnutls\\_x509\\_crt\\_verify\]](#), page 435 or [\[gnutls\\_x509\\_crt\\_list\\_verify\]](#), page 424, you can pass the `GNUTLS_VERIFY_ALLOW_SIGN_RSA_MD5` parameter directly in the `flags` parameter.

If you are using these flags, it may also be a good idea to warn the user when verification failure occur for this reason. The simplest is to not use the flags by default, and only fall back to using them after warning the user. If you wish to inspect the certificate chain yourself, you can use [\[gnutls\\_certificate\\_get\\_peers\]](#), page 280 to extract the raw server's certificate chain, [\[gnutls\\_x509\\_crt\\_list\\_import\]](#), page 423 to parse each of the certificates, and then [\[gnutls\\_x509\\_crt\\_get\\_signature\\_algorithm\]](#), page 419 to find out the signing algorithm used for each certificate. If any of the intermediary certificates are using `GNUTLS_SIGN_RSA_MD2` or `GNUTLS_SIGN_RSA_MD5`, you could present a warning.

## 4.2 More on certificate authentication

Certificates are not the only structures involved in a public key infrastructure. Several other structures that are used for certificate requests, encrypted private keys, revocation lists, GnuTLS abstract key structures, etc., are discussed in this chapter.

### 4.2.1 PKCS #10 certificate requests

A certificate request is a structure, which contain information about an applicant of a certificate service. It usually contains a private key, a distinguished name and secondary data such as a challenge password. GnuTLS supports the requests defined in PKCS #10 [\[RFC2986\]](#). Other formats of certificate requests are not currently supported.

A certificate request can be generated by associating it with a private key, setting the subject's information and finally self signing it. The last step ensures that the requester is in possession of the private key.

```
int [gnutls_x509_crq_set_version], page 399 (gnutls_x509_crq_t crq, unsigned
int version)
int [gnutls_x509_crq_set_dn], page 397 (gnutls_x509_crq_t crq, const char *
dn, const char ** err)
int [gnutls_x509_crq_set_dn_by_oid], page 397 (gnutls_x509_crq_t crq, const
char * oid, unsigned int raw_flag, const void * data, unsigned int sizeof_data)
int [gnutls_x509_crq_set_key_usage], page 398 (gnutls_x509_crq_t crq,
unsigned int usage)
int [gnutls_x509_crq_set_key_purpose_oid], page 398 (gnutls_x509_crq_t crq,
const void * oid, unsigned int critical)
int [gnutls_x509_crq_set_basic_constraints], page 396 (gnutls_x509_crq_t
crq, unsigned int ca, int pathLenConstraint)
```

The `[gnutls_x509_crq_set_key]`, page 398 and `[gnutls_x509_crq_sign2]`, page 400 functions associate the request with a private key and sign it. If a request is to be signed with a key residing in a PKCS #11 token it is recommended to use the signing functions shown in Section 5.1 [Abstract key types], page 81.

```
int gnutls_x509_crq_set_key (gnutls_x509_crq_t crq, [Function]
                           gnutls_privkey_t key)
    crq: should contain a gnutls_x509_crq_t structure
    key: holds a private key
    This function will set the public parameters from the given private key to the request.
Returns: On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error
value.
```

```
int gnutls_x509_crq_sign2 (gnutls_x509_crq_t crq, [Function]
                           gnutls_privkey_t key, gnutls_digest_algorithm_t dig, unsigned int flags)
    crq: should contain a gnutls_x509_crq_t structure
    key: holds a private key
    dig: The message digest to use, i.e., GNUTLS_DIG_SHA1
    flags: must be 0
    This function will sign the certificate request with a private key. This must be the
    same key as the one used in gnutls_x509_crq_set_key() since a certificate request
    is self signed.
    This must be the last step in a certificate request generation since all the previously
    set parameters are now signed.
Returns: GNUTLS_E_SUCCESS on success, otherwise a negative error code. GNUTLS_E_
ASN1_VALUE_NOT_FOUND is returned if you didn't set all information in the certificate
request (e.g., the version using gnutls_x509_crq_set_version() ).
```

The following example is about generating a certificate request, and a private key. A certificate request can be later be processed by a CA which should return a signed certificate.

```
/* This example code is placed in the public domain. */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <gnutls/gnutls.h>
#include <gnutls/x509.h>
#include <gnutls/abstract.h>
#include <time.h>

/* This example will generate a private key and a certificate
 * request.
 */

int main(void)
{
    gnutls_x509_crq_t crq;
    gnutls_x509_privkey_t key;
    unsigned char buffer[10 * 1024];
    size_t buffer_size = sizeof(buffer);
    unsigned int bits;

    gnutls_global_init();

    /* Initialize an empty certificate request, and
     * an empty private key.
     */
    gnutls_x509_crq_init(&crq);

    gnutls_x509_privkey_init(&key);

    /* Generate an RSA key of moderate security.
     */
    bits =
        gnutls_sec_param_to_pk_bits(GNUTLS_PK_RSA,
                                    GNUTLS_SEC_PARAM_MEDIUM);
    gnutls_x509_privkey_generate(key, GNUTLS_PK_RSA, bits, 0);

    /* Add stuff to the distinguished name
     */
    gnutls_x509_crq_set_dn_by_oid(crq, GNUTLS_OID_X520_COUNTRY_NAME,
                                  0, "GR", 2);
}
```

```
gnutls_x509_crq_set_dn_by_oid(crq, GNUTLS_OID_X520_COMMON_NAME,
                              0, "Nikos", strlen("Nikos"));

/* Set the request version.
 */
gnutls_x509_crq_set_version(crq, 1);

/* Set a challenge password.
 */
gnutls_x509_crq_set_challenge_password(crq,
                                       "something to remember here");

/* Associate the request with the private key
 */
gnutls_x509_crq_set_key(crq, key);

/* Self sign the certificate request.
 */
gnutls_x509_crq_sign2(crq, key, GNUTLS_DIG_SHA1, 0);

/* Export the PEM encoded certificate request, and
 * display it.
 */
gnutls_x509_crq_export(crq, GNUTLS_X509_FMT_PEM, buffer,
                      &buffer_size);

printf("Certificate Request:  \n%s", buffer);

/* Export the PEM encoded private key, and
 * display it.
 */
buffer_size = sizeof(buffer);
gnutls_x509_privkey_export(key, GNUTLS_X509_FMT_PEM, buffer,
                          &buffer_size);

printf("\n\nPrivate key:  \n%s", buffer);

gnutls_x509_crq_deinit(crq);
gnutls_x509_privkey_deinit(key);

return 0;

}
```

### 4.2.2 PKIX certificate revocation lists

A certificate revocation list (CRL) is a structure issued by an authority periodically containing a list of revoked certificates serial numbers. The CRL structure is signed with the issuing authorities' keys. A typical CRL contains the fields as shown in [Table 4.7](#). Certificate revocation lists are used to complement the expiration date of a certificate, in order to account for other reasons of revocation, such as compromised keys, etc.

Each CRL is valid for limited amount of time and is required to provide, except for the current issuing time, also the issuing time of the next update.

Field	Description
version	The field that indicates the version of the CRL structure.
signature	A signature by the issuing authority.
issuer	Holds the issuer's distinguished name.
thisUpdate	The issuing time of the revocation list.
nextUpdate	The issuing time of the revocation list that will update that one.
revokedCertificates	List of revoked certificates serial numbers.
extensions	Optional CRL structure extensions.

Table 4.7: Certificate revocation list fields.

The basic CRL structure functions follow.

```
int [gnutls_x509_crl_init], page 381 (gnutls_x509_crl_t *crl)
int [gnutls_x509_crl_import], page 381 (gnutls_x509_crl_t crl, const
gnutls_datum_t *data, gnutls_x509_crt_fmt_t format)
int [gnutls_x509_crl_export], page 374 (gnutls_x509_crl_t crl,
gnutls_x509_crt_fmt_t format, void *output_data, size_t *output_data_size)
int [gnutls_x509_crl_export], page 374 (gnutls_x509_crl_t crl,
gnutls_x509_crt_fmt_t format, void *output_data, size_t *output_data_size)
```

#### Reading a CRL

The most important function that extracts the certificate revocation information from a CRL is [\[gnutls\\_x509\\_crl\\_get\\_crt\\_serial\], page 376](#). Other functions that return other fields of the CRL structure are also provided.

```
int gnutls_x509_crl_get_crt_serial (gnutls_x509_crl_t crl, int [Function]
    indx, unsigned char *serial, size_t *serial_size, time_t *t)
    crl: should contain a gnutls_x509_crl_t structure
    indx: the index of the certificate to extract (starting from 0)
```

*serial*: where the serial number will be copied

*serial\_size*: initially holds the size of *serial*

*t*: if non null, will hold the time this certificate was revoked

This function will retrieve the serial number of the specified, by the index, revoked certificate.

Note that this function will have performance issues in large sequences of revoked certificates. In that case use `gnutls_x509_crl_iter_cert_serial()`.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

```
int [gnutls_x509_crl_get_version], page 381 (gnutls_x509_crl_t crl)
int [gnutls_x509_crl_get_issuer_dn], page 378 (const gnutls_x509_crl_t crl,
char * buf, size_t * sizeof_buf)
int [gnutls_x509_crl_get_issuer_dn2], page 378 (gnutls_x509_crl_t crl,
gnutls_datum_t * dn)
time_t [gnutls_x509_crl_get_this_update], page 380 (gnutls_x509_crl_t crl)
time_t [gnutls_x509_crl_get_next_update], page 379 (gnutls_x509_crl_t crl)
int [gnutls_x509_crl_get_crt_count], page 376 (gnutls_x509_crl_t crl)
```

## Generation of a CRL

The following functions can be used to generate a CRL.

```
int [gnutls_x509_crl_set_version], page 385 (gnutls_x509_crl_t crl, unsigned
int version)
int [gnutls_x509_crl_set_cert_serial], page 384 (gnutls_x509_crl_t crl, const
void * serial, size_t serial_size, time_t revocation_time)
int [gnutls_x509_crl_set_crt], page 383 (gnutls_x509_crl_t crl,
gnutls_x509_crt_t crt, time_t revocation_time)
int [gnutls_x509_crl_set_next_update], page 384 (gnutls_x509_crl_t crl,
time_t exp_time)
int [gnutls_x509_crl_set_this_update], page 384 (gnutls_x509_crl_t crl,
time_t act_time)
```

The `[gnutls_x509_crl_sign2]`, page 385 and `[gnutls_x509_crl_privkey_sign]`, page 546 functions sign the revocation list with a private key. The latter function can be used to sign with a key residing in a PKCS #11 token.

```
int gnutls_x509_crl_sign2 (gnutls_x509_crl_t crl, gnutls_x509_crt_t issuer, gnutls_x509_privkey_t issuer_key, gnutls_digest_algorithm_t dig,
unsigned int flags) [Function]
```

*crl*: should contain a `gnutls_x509_crl_t` structure

*issuer*: is the certificate of the certificate issuer

*issuer\_key*: holds the issuer's private key

*dig*: The message digest to use. `GNUTLS_DIG_SHA1` is the safe choice unless you know what you're doing.

*flags*: must be 0

This function will sign the CRL with the issuer's private key, and will copy the issuer's information into the CRL.

This must be the last step in a certificate CRL since all the previously set parameters are now signed.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

```
int gnutls_x509_crl_privkey_sign (gnutls_x509_crl_t crl,           [Function]
                                gnutls_x509_cert_t issuer, gnutls_privkey_t issuer_key,
                                gnutls_digest_algorithm_t dig, unsigned int flags)
```

*crl*: should contain a `gnutls_x509_crl_t` structure

*issuer*: is the certificate of the certificate issuer

*issuer\_key*: holds the issuer's private key

*dig*: The message digest to use. `GNUTLS_DIG_SHA1` is the safe choice unless you know what you're doing.

*flags*: must be 0

This function will sign the CRL with the issuer's private key, and will copy the issuer's information into the CRL.

This must be the last step in a certificate CRL since all the previously set parameters are now signed.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since 2.12.0

Few extensions on the CRL structure are supported, including the CRL number extension and the authority key identifier.

```
int [gnutls_x509_crl_set_number], page 384 (gnutls_x509_crl_t crl, const void
* nr, size_t nr_size)
int [gnutls_x509_crl_set_authority_key_id], page 383 (gnutls_x509_crl_t crl,
const void * id, size_t id_size)
```

### 4.2.3 OCSP certificate status checking

Certificates may be revoked before their expiration time has been reached. There are several reasons for revoking certificates, but a typical situation is when the private key associated with a certificate has been compromised. Traditionally, Certificate Revocation Lists (CRLs) have been used by application to implement revocation checking, however, several problems with CRLs have been identified [*RIVESTCRL*].

The Online Certificate Status Protocol, or OCSP [*RFC2560*], is a widely implemented protocol which performs certificate revocation status checking. An application that wish to verify the identity of a peer will verify the certificate against a set of trusted certificates and then check whether the certificate is listed in a CRL and/or perform an OCSP check for the certificate.

Note that in the context of a TLS session the server may provide an OCSP response that will be used during the TLS certificate verification (see [*gnutls\_certificate\_verify\_peers2*], page 293). You may obtain this response using [*gnutls\_ocsp\_status\_request\_get*], page 318.



Before performing the OCSF query, the application will need to figure out the address of the OCSF server. The OCSF server address can be provided by the local user in manual configuration or may be stored in the certificate that is being checked. When stored in a certificate the OCSF server is in the extension field called the Authority Information Access (AIA). The following function extracts this information from a certificate.

```
int [gnutls_x509_cert_get_authority_info_access], page 403 (gnutls_x509_cert_t
cert, unsigned int seq, int what, gnutls_datum_t * data, unsigned int *
critical)
```

There are several functions in GnuTLS for creating and manipulating OCSF requests and responses. The general idea is that a client application creates an OCSF request object, stores some information about the certificate to check in the request, and then exports the request in DER format. The request will then need to be sent to the OCSF responder, which needs to be done by the application (GnuTLS does not send and receive OCSF packets). Normally an OCSF response is received that the application will need to import into an OCSF response object. The digital signature in the OCSF response needs to be verified against a set of trust anchors before the information in the response can be trusted.

The ASN.1 structure of OCSF requests are briefly as follows. It is useful to review the structures to get an understanding of which fields are modified by GnuTLS functions.

```
OCSPRequest      ::=      SEQUENCE {
    tbsRequest          TBSRequest,
    optionalSignature    [0]    EXPLICIT Signature OPTIONAL }

TBSRequest       ::=      SEQUENCE {
    version              [0]    EXPLICIT Version DEFAULT v1,
    requestorName        [1]    EXPLICIT GeneralName OPTIONAL,
    requestList          SEQUENCE OF Request,
    requestExtensions    [2]    EXPLICIT Extensions OPTIONAL }

Request          ::=      SEQUENCE {
    reqCert              CertID,
    singleRequestExtensions [0] EXPLICIT Extensions OPTIONAL }

CertID           ::=      SEQUENCE {
    hashAlgorithm        AlgorithmIdentifier,
    issuerNameHash       OCTET STRING, -- Hash of Issuer's DN
    issuerKeyHash        OCTET STRING, -- Hash of Issuers public key
    serialNumber         CertificateSerialNumber }
```

The basic functions to initialize, import, export and deallocate OCSF requests are the following.

```

int [gnutls_ocsp_req_init], page 470 (gnutls_ocsp_req_t * req)
void [gnutls_ocsp_req_deinit], page 468 (gnutls_ocsp_req_t req)
int [gnutls_ocsp_req_import], page 470 (gnutls_ocsp_req_t req, const
gnutls_datum_t * data)
int [gnutls_ocsp_req_export], page 468 (gnutls_ocsp_req_t req, gnutls_datum_t
* data)
int [gnutls_ocsp_req_print], page 470 (gnutls_ocsp_req_t req,
gnutls_ocsp_print_formats_t format, gnutls_datum_t * out)

```

To generate an OCSF request the issuer name hash, issuer key hash, and the checked certificate's serial number are required. There are two interfaces available for setting those in an OCSF request. The first is a low-level function when you have the issuer name hash, issuer key hash, and certificate serial number in binary form. The second is more useful if you have the certificate (and its issuer) in a `gnutls_x509_cert_t` type. There is also a function to extract this information from existing an OCSF request.

```

int [gnutls_ocsp_req_add_cert_id], page 467 (gnutls_ocsp_req_t req,
gnutls_digest_algorithm_t digest, const gnutls_datum_t * issuer_name_hash,
const gnutls_datum_t * issuer_key_hash, const gnutls_datum_t * serial_number)
int [gnutls_ocsp_req_add_cert], page 467 (gnutls_ocsp_req_t req,
gnutls_digest_algorithm_t digest, gnutls_x509_cert_t issuer,
gnutls_x509_cert_t cert)
int [gnutls_ocsp_req_get_cert_id], page 468 (gnutls_ocsp_req_t req, unsigned
indx, gnutls_digest_algorithm_t * digest, gnutls_datum_t * issuer_name_hash,
gnutls_datum_t * issuer_key_hash, gnutls_datum_t * serial_number)

```

Each OCSF request may contain a number of extensions. Extensions are identified by an Object Identifier (OID) and an opaque data buffer whose syntax and semantics is implied by the OID. You can extract or set those extensions using the following functions.

```

int [gnutls_ocsp_req_get_extension], page 469 (gnutls_ocsp_req_t req,
unsigned indx, gnutls_datum_t * oid, unsigned int * critical, gnutls_datum_t *
data)
int [gnutls_ocsp_req_set_extension], page 471 (gnutls_ocsp_req_t req, const
char * oid, unsigned int critical, const gnutls_datum_t * data)

```

A common OCSF Request extension is the nonce extension (OID 1.3.6.1.5.5.7.48.1.2), which is used to avoid replay attacks of earlier recorded OCSF responses. The nonce extension carries a value that is intended to be sufficiently random and unique so that an attacker will not be able to give a stale response for the same nonce.

```

int [gnutls_ocsp_req_get_nonce], page 469 (gnutls_ocsp_req_t req, unsigned
int * critical, gnutls_datum_t * nonce)
int [gnutls_ocsp_req_set_nonce], page 471 (gnutls_ocsp_req_t req, unsigned
int critical, const gnutls_datum_t * nonce)
int [gnutls_ocsp_req_randomize_nonce], page 470 (gnutls_ocsp_req_t req)

```

The OCSF response structure is a complex structure. A simplified overview of it is in [Table 4.8](#). Note that a response may contain information on multiple certificates.

Field	Description
version	The OCSP response version number (typically 1).
responder ID	An identifier of the responder (DN name or a hash of its key).
issue time	The time the response was generated.
thisUpdate	The issuing time of the revocation information.
nextUpdate	The issuing time of the revocation information that will update that one.
	Revoked certificates
certificate status	The status of the certificate.
certificate serial	The certificate's serial number.
revocationTime	The time the certificate was revoked.
revocationReason	The reason the certificate was revoked.

Table 4.8: The most important OCSP response fields.

We provide basic functions for initialization, importing, exporting and deallocating OCSP responses.

```
int [gnutls_ocsp_resp_init], page 475 (gnutls_ocsp_resp_t * resp)
void [gnutls_ocsp_resp_deinit], page 471 (gnutls_ocsp_resp_t resp)
int [gnutls_ocsp_resp_import], page 475 (gnutls_ocsp_resp_t resp, const
gnutls_datum_t * data)
int [gnutls_ocsp_resp_export], page 472 (gnutls_ocsp_resp_t resp,
gnutls_datum_t * data)
int [gnutls_ocsp_resp_print], page 476 (gnutls_ocsp_resp_t resp,
gnutls_ocsp_print_formats_t format, gnutls_datum_t * out)
```

The utility function that extracts the revocation as well as other information from a response is shown below.

```
int gnutls_ocsp_resp_get_single (gnutls_ocsp_resp_t resp, [Function]
    unsigned indx, gnutls_digest_algorithm_t * digest, gnutls_datum_t *
    issuer_name_hash, gnutls_datum_t * issuer_key_hash, gnutls_datum_t *
    serial_number, unsigned int * cert_status, time_t * this_update,
    time_t * next_update, time_t * revocation_time, unsigned int *
    revocation_reason)
resp: should contain a gnutls_ocsp_resp_t structure
```

*indx*: Specifies response number to get. Use (0) to get the first one.

*digest*: output variable with `gnutls_digest_algorithm_t` hash algorithm

*issuer\_name\_hash*: output buffer with hash of issuer's DN

*issuer\_key\_hash*: output buffer with hash of issuer's public key

*serial\_number*: output buffer with serial number of certificate to check

*cert\_status*: a certificate status, a `gnutls_ocsp_cert_status_t` enum.

*this\_update*: time at which the status is known to be correct.

*next\_update*: when newer information will be available, or (time\_t)-1 if unspecified

*revocation\_time*: when `cert_status` is `GNUTLS_OCSP_CERT_REVOKED` , holds time of revocation.

*revocation\_reason*: revocation reason, a `gnutls_x509_crl_reason_t` enum.

This function will return the certificate information of the `indx` 'ed response in the Basic OCSP Response `resp` . The information returned corresponds to the OCSP SingleResponse structure except the final singleExtensions.

Each of the pointers to output variables may be NULL to indicate that the caller is not interested in that value.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned. If you have reached the last CertID available `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

The possible revocation reasons available in an OCSP response are shown below.

GNUTLS_X509_CRLREASON_UNSPECIFIED	Unspecified reason.
GNUTLS_X509_CRLREASON_KEYCOMPROMISE	Private key compromised.
GNUTLS_X509_CRLREASON_CACOMPROMISE	CA compromised.
GNUTLS_X509_CRLREASON_AFFILIATIONCHANGED	Affiliation has changed.
GNUTLS_X509_CRLREASON_SUPERSEDED	Certificate superseded.
GNUTLS_X509_CRLREASON_CESSATIONOFOPERATION	Operation has ceased.
GNUTLS_X509_CRLREASON_CERTIFICATEHOLD	Certificate is on hold.
GNUTLS_X509_CRLREASON_REMOVEFROMCRL	Will be removed from delta CRL.
GNUTLS_X509_CRLREASON_PRIVILEGEWITHDRAWN	Privilege withdrawn.
GNUTLS_X509_CRLREASON_AACOMPROMISE	AA compromised.

Figure 4.5: The revocation reasons

Note, that the OCSP response needs to be verified against some set of trust anchors before it can be relied upon. It is also important to check whether the received OCSP response corresponds to the certificate being checked.

```
int [gnutls_ocsp_resp_verify], page 476 (gnutls_ocsp_resp_t resp,
gnutls_x509_trust_list_t trustlist, unsigned int * verify, unsigned int flags)
int [gnutls_ocsp_resp_verify_direct], page 477 (gnutls_ocsp_resp_t resp,
gnutls_x509_cert_t issuer, unsigned int * verify, unsigned int flags)
int [gnutls_ocsp_resp_check_cert], page 471 (gnutls_ocsp_resp_t resp, unsigned
int indx, gnutls_x509_cert_t crt)
```

#### 4.2.4 Managing encrypted keys

Transferring or storing private keys in plain may not be a good idea, since any compromise is irreparable. Storing the keys in hardware security modules (see [Section 5.2 \[Smart cards and HSMs\]](#), page 88) could solve the storage problem but it is not always practical or efficient enough. This section describes ways to store and transfer encrypted private keys.

There are methods for key encryption, namely the PKCS #8, PKCS #12 and OpenSSL's custom encrypted private key formats. The PKCS #8 and the OpenSSL's method allow encryption of the private key, while the PKCS #12 method allows, in addition, the bundling

of accompanying data into the structure. That is typically the corresponding certificate, as well as a trusted CA certificate.

## High level functionality

Generic and higher level private key import functions are available, that import plain or encrypted keys and will auto-detect the encrypted key format.

```
int gnutls_privkey_import_x509_raw (gnutls_privkey_t pkey, const [Function]
    gnutls_datum_t * data, gnutls_x509_crt_fmt_t format, const char *
    password, unsigned int flags)
```

*pkey*: The private key

*data*: The private key data to be imported

*format*: The format of the private key

*password*: A password (optional)

*flags*: an ORed sequence of gnutls\_pkcs\_encrypt\_flags\_t

This function will import the given private key to the abstract gnutls\_privkey\_t structure.

The supported formats are basic unencrypted key, PKCS8, PKCS12, and the openssl format.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.1.0

```
int gnutls_x509_privkey_import2 (gnutls_x509_privkey_t key, const [Function]
    gnutls_datum_t * data, gnutls_x509_crt_fmt_t format, const char *
    password, unsigned int flags)
```

*key*: The structure to store the parsed key

*data*: The DER or PEM encoded key.

*format*: One of DER or PEM

*password*: A password (optional)

*flags*: an ORed sequence of gnutls\_pkcs\_encrypt\_flags\_t

This function will import the given DER or PEM encoded key, to the native gnutls\_x509\_privkey\_t format, irrespective of the input format. The input format is auto-detected.

The supported formats are basic unencrypted key, PKCS8, PKCS12, and the openssl format.

If the provided key is encrypted but no password was given, then GNUTLS\_E\_DECRYPTION\_FAILED is returned.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

Any keys imported using those functions can be imported to a certificate credentials structure using [\[gnutls\\_certificate\\_set\\_key\]](#), [page 520](#), or alternatively they can be directly imported using [\[gnutls\\_certificate\\_set\\_x509\\_key\\_file2\]](#), [page 286](#).

## PKCS #8 structures

PKCS #8 keys can be imported and exported as normal private keys using the functions below. An addition to the normal import functions, are a password and a flags argument. The flags can be any element of the `gnutls_pkcs_encrypt_flags_t` enumeration. Note however, that GnuTLS only supports the PKCS #5 PBES2 encryption scheme. Keys encrypted with the obsolete PBES1 scheme cannot be decrypted.

```
int [gnutls_x509_privkey_import_pkcs8], page 457 (gnutls_x509_privkey_t key,
const gnutls_datum_t * data, gnutls_x509_crt_fmt_t format, const char *
password, unsigned int flags)
int [gnutls_x509_privkey_export_pkcs8], page 452 (gnutls_x509_privkey_t key,
gnutls_x509_crt_fmt_t format, const char * password, unsigned int flags, void *
output_data, size_t * output_data_size)
int [gnutls_x509_privkey_export2_pkcs8], page 451 (gnutls_x509_privkey_t
key, gnutls_x509_crt_fmt_t format, const char * password, unsigned int flags,
gnutls_datum_t * out)
```

GNUTLS\_PKCS\_PLAIN

Unencrypted private key.

GNUTLS\_PKCS\_USE\_PKCS12\_3DES

PKCS-12 3DES.

GNUTLS\_PKCS\_USE\_PKCS12\_ARCFOUR

PKCS-12 ARCFOUR.

GNUTLS\_PKCS\_USE\_PKCS12\_RC2\_40

PKCS-12 RC2-40.

GNUTLS\_PKCS\_USE\_PBES2\_3DES

PBES2 3DES.

GNUTLS\_PKCS\_USE\_PBES2\_AES\_128

PBES2 AES-128.

GNUTLS\_PKCS\_USE\_PBES2\_AES\_192

PBES2 AES-192.

GNUTLS\_PKCS\_USE\_PBES2\_AES\_256

PBES2 AES-256.

GNUTLS\_PKCS\_NULL\_PASSWORD

Some schemas distinguish between an empty and a NULL password.

Figure 4.6: Encryption flags

## PKCS #12 structures

A PKCS #12 structure [*PKCS12*] usually contains a user's private keys and certificates. It is commonly used in browsers to export and import the user's identities. A file containing such a key can be directly imported to a certificate credentials structure by using `[gnutls_certificate_set_x509_simple_pkcs12_file]`, page 288.

In GnuTLS the PKCS #12 structures are handled using the `gnutls_pkcs12_t` type. This is an abstract type that may hold several `gnutls_pkcs12_bag_t` types. The bag types are the holders of the actual data, which may be certificates, private keys or encrypted data. A bag of type encrypted should be decrypted in order for its data to be accessed.

To reduce the complexity in parsing the structures the simple helper function `[gnutls_pkcs12_simple_parse]`, page 503 is provided. For more advanced uses, manual parsing of the structure is required using the functions below.

```
int [gnutls_pkcs12_get_bag], page 501 (gnutls_pkcs12_t pkcs12, int indx,
gnutls_pkcs12_bag_t bag)
int [gnutls_pkcs12_verify_mac], page 503 (gnutls_pkcs12_t pkcs12, const char
* pass)
int [gnutls_pkcs12_bag_decrypt], page 497 (gnutls_pkcs12_bag_t bag, const
char * pass)
int [gnutls_pkcs12_bag_get_count], page 498 (gnutls_pkcs12_bag_t bag)
int gnutls_pkcs12_simple_parse (gnutls_pkcs12_t p12, const char * [Function]
password, gnutls_x509_privkey_t * key, gnutls_x509_crt_t ** chain, unsigned
int * chain_len, gnutls_x509_crt_t ** extra_certs, unsigned int *
extra_certs_len, gnutls_x509_crl_t * crl, unsigned int flags)
```

*p12*: should contain a `gnutls_pkcs12_t` structure

*password*: optional password used to decrypt the structure, bags and keys.

*key*: a structure to store the parsed private key.

*chain*: the corresponding to key certificate chain (may be NULL )

*chain\_len*: will be updated with the number of additional (may be NULL )

*extra\_certs*: optional pointer to receive an array of additional certificates found in the PKCS12 structure (may be NULL ).

*extra\_certs\_len*: will be updated with the number of additional certs (may be NULL ).

*crl*: an optional structure to store the parsed CRL (may be NULL ).

*flags*: should be zero or one of `GNUTLS_PKCS12_SP_*`

This function parses a PKCS12 structure in `pkcs12` and extracts the private key, the corresponding certificate chain, any additional certificates and a CRL.

The `extra_certs` and `extra_certs_len` parameters are optional and both may be set to NULL . If either is non-NULL , then both must be set. The value for `extra_certs` is allocated using `gnutls_malloc()` .

Encrypted PKCS12 bags and PKCS8 private keys are supported, but only with password based security and the same password for all operations.

Note that a PKCS12 structure may contain many keys and/or certificates, and there is no way to identify which key/certificate pair you want. For this reason this function is useful for PKCS12 files that contain only one key/certificate pair and/or one CRL.

If the provided structure has encrypted fields but no password is provided then this function returns `GNUTLS_E_DECRYPTION_FAILED` .

Note that normally the chain constructed does not include self signed certificates, to comply with TLS' requirements. If, however, the flag `GNUTLS_PKCS12_SP_INCLUDE_SELF_SIGNED` is specified then self signed certificates will be included in the chain.



Prior to using this function the PKCS 12 structure integrity must be verified using `gnutls_pkcs12_verify_mac()` .

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.1.0

```
int [gnutls_pkcs12_bag_get_data], page 498 (gnutls_pkcs12_bag_t bag, int
indx, gnutls_datum_t * data)
int [gnutls_pkcs12_bag_get_key_id], page 499 (gnutls_pkcs12_bag_t bag, int
indx, gnutls_datum_t * id)
int [gnutls_pkcs12_bag_get_friendly_name], page 498 (gnutls_pkcs12_bag_t
bag, int indx, char ** name)
```

The functions below are used to generate a PKCS #12 structure. An example of their usage is shown at [Section 7.4.4 \[PKCS12 structure generation example\], page 225](#).

```
int [gnutls_pkcs12_set_bag], page 502 (gnutls_pkcs12_t pkcs12,
gnutls_pkcs12_bag_t bag)
int [gnutls_pkcs12_bag_encrypt], page 498 (gnutls_pkcs12_bag_t bag, const
char * pass, unsigned int flags)
int [gnutls_pkcs12_generate_mac], page 501 (gnutls_pkcs12_t pkcs12, const
char * pass)
int [gnutls_pkcs12_bag_set_data], page 500 (gnutls_pkcs12_bag_t bag,
gnutls_pkcs12_bag_type_t type, const gnutls_datum_t * data)
int [gnutls_pkcs12_bag_set_crl], page 499 (gnutls_pkcs12_bag_t bag,
gnutls_x509_crl_t crl)
int [gnutls_pkcs12_bag_set_cert], page 499 (gnutls_pkcs12_bag_t bag,
gnutls_x509_cert_t crt)
int [gnutls_pkcs12_bag_set_key_id], page 500 (gnutls_pkcs12_bag_t bag, int
indx, const gnutls_datum_t * id)
int [gnutls_pkcs12_bag_set_friendly_name], page 500 (gnutls_pkcs12_bag_t
bag, int indx, const char * name)
```

## OpenSSL encrypted keys

Unfortunately the structures discussed in the previous sections are not the only structures that may hold an encrypted private key. For example the OpenSSL library offers a custom key encryption method. Those structures are also supported in GnuTLS with [\[gnutls\\_x509\\_privkey\\_import\\_openssl\], page 457](#).

```
int gnutls_x509_privkey_import_openssl (gnutls_x509_privkey_t [Function]
key, const gnutls_datum_t * data, const char * password)
```

*key*: The structure to store the parsed key

*data*: The DER or PEM encoded key.

*password*: the password to decrypt the key (if it is encrypted).

This function will convert the given PEM encrypted to the native `gnutls_x509_privkey_t` format. The output will be stored in *key* .

The *password* should be in ASCII. If the password is not provided or wrong then `GNUTLS_E_DECRYPTION_FAILED` will be returned.

If the Certificate is PEM encoded it should have a header of "PRIVATE KEY" and the "DEK-Info" header.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

### 4.2.5 Invoking certtool

Tool to parse and generate X.509 certificates, requests and private keys. It can be used interactively or non interactively by specifying the template command line option.

The tool accepts files or URLs supported by GnuTLS. In case PIN is required for the URL access you can provide it using the environment variables GNUTLS\_PIN and GNUTLS\_SO\_PIN.

This section was generated by **AutoGen**, using the **agtexi-cmd** template and the option descriptions for the **certtool** program. This software is released under the GNU General Public License, version 3 or later.

#### **certtool help/usage (--help)**

This is the automatically generated usage text for certtool.

The text printed is the same whether selected with the **help** option (**--help**) or the **more-help** option (**--more-help**). **more-help** will print the usage text by passing it through a pager program. **more-help** is disabled on platforms without a working **fork(2)** function. The **PAGER** environment variable is used to select the program, defaulting to **more**. Both will exit with a status code of 0.

**certtool is unavailable - no --help**

#### **debug option (-d)**

This is the “enable debugging” option. This option takes a number argument. Specifies the debug level.

#### **generate-request option (-q)**

This is the “generate a pkcs #10 certificate request” option.

This option has some usage constraints. It:

- must not appear in combination with any of the following options: **infile**.

Will generate a PKCS #10 certificate request. To specify a private key use **-load-privkey**.

#### **verify-chain option (-e)**

This is the “verify a pem encoded certificate chain” option. The last certificate in the chain must be a self signed one.

#### **verify option**

This is the “verify a pem encoded certificate chain using a trusted list” option. The trusted certificate list can be loaded with **-load-ca-certificate**. If no certificate list is provided, then the system’s certificate list is used.

**verify-crl option**

This is the “verify a crl using a trusted list” option.

This option has some usage constraints. It:

- must appear in combination with the following options: load-ca-certificate.

The trusted certificate list must be loaded with `–load-ca-certificate`.

**get-dh-params option**

This is the “get the included pkcs #3 encoded diffie-hellman parameters” option. Returns stored DH parameters in GnuTLS. Those parameters are used in the SRP protocol. The parameters returned by fresh generation are more efficient since GnuTLS 3.0.9.

**load-privkey option**

This is the “loads a private key file” option. This option takes a string argument. This can be either a file or a PKCS #11 URL

**load-pubkey option**

This is the “loads a public key file” option. This option takes a string argument. This can be either a file or a PKCS #11 URL

**load-certificate option**

This is the “loads a certificate file” option. This option takes a string argument. This can be either a file or a PKCS #11 URL

**load-ca-privkey option**

This is the “loads the certificate authority’s private key file” option. This option takes a string argument. This can be either a file or a PKCS #11 URL

**load-ca-certificate option**

This is the “loads the certificate authority’s certificate file” option. This option takes a string argument. This can be either a file or a PKCS #11 URL

**password option**

This is the “password to use” option. This option takes a string argument. You can use this option to specify the password in the command line instead of reading it from the tty. Note, that the command line arguments are available for view in others in the system. Specifying password as ” is the same as specifying no password.

**null-password option**

This is the “enforce a null password” option. This option enforces a NULL password. This is different than the empty or no password in schemas like PKCS #8.

**empty-password option**

This is the “enforce an empty password” option. This option enforces an empty password. This is different than the NULL or no password in schemas like PKCS #8.

**cprint option**

This is the “in certain operations it prints the information in c-friendly format” option. In certain operations it prints the information in C-friendly format, suitable for including into C programs.

**p12-name option**

This is the “the pkcs #12 friendly name to use” option. This option takes a string argument. The name to be used for the primary certificate and private key in a PKCS #12 file.

**pubkey-info option**

This is the “print information on a public key” option. The option combined with `-load-request`, `-load-pubkey`, `-load-privkey` and `-load-certificate` will extract the public key of the object in question.

**to-p12 option**

This is the “generate a pkcs #12 structure” option.

This option has some usage constraints. It:

- must appear in combination with the following options: `load-certificate`.

It requires a certificate, a private key and possibly a CA certificate to be specified.

**rsa option**

This is the “generate rsa key” option. When combined with `-generate-privkey` generates an RSA private key.

**dsa option**

This is the “generate dsa key” option. When combined with `-generate-privkey` generates a DSA private key.

**ecc option**

This is the “generate ecc (ecdsa) key” option. When combined with `-generate-privkey` generates an elliptic curve private key to be used with ECDSA.

**ecdsa option**

This is an alias for the `ecc` option, see [\[certtool ecc\]](#), page 60.

**hash option**

This is the “hash algorithm to use for signing” option. This option takes a string argument. Available hash functions are SHA1, RMD160, SHA256, SHA384, SHA512.

### **inder option**

This is the “use der format for input certificates, private keys, and dh parameters ” option. This option has some usage constraints. It:

- can be disabled with `-no-inder`.

The input files will be assumed to be in DER or RAW format. Unlike options that in PEM input would allow multiple input data (e.g. multiple certificates), when reading in DER format a single data structure is read.

### **inraw option**

This is an alias for the `inder` option, see [\[certtool inder\]](#), page 60.

### **outder option**

This is the “use der format for output certificates, private keys, and dh parameters” option. This option has some usage constraints. It:

- can be disabled with `-no-outder`.

The output will be in DER or RAW format.

### **outraw option**

This is an alias for the `outder` option, see [\[certtool outder\]](#), page 61.

### **curve option**

This is the “specify the curve used for ec key generation” option. This option takes a string argument. Supported values are `secp192r1`, `secp224r1`, `secp256r1`, `secp384r1` and `secp521r1`.

### **sec-param option**

This is the “specify the security level [low, legacy, medium, high, ultra]” option. This option takes a string argument `Security parameter`. This is alternative to the bits option.

### **ask-pass option**

This is the “enable interaction for entering password when in batch mode.” option. This option will enable interaction to enter password when in batch mode. That is useful when the template option has been specified.

### **pkcs-cipher option**

This is the “cipher to use for pkcs #8 and #12 operations” option. This option takes a string argument `Cipher`. Cipher may be one of `3des`, `3des-pkcs12`, `aes-128`, `aes-192`, `aes-256`, `rc2-40`, `arcfour`.

### **provider option**

This is the “specify the pkcs #11 provider library” option. This option takes a string argument. This will override the default options in `/etc/gnutls/pkcs11.conf`

**certtool exit status**

One of the following exit values will be returned:

'0 (EXIT\_SUCCESS)'

Successful program execution.

'1 (EXIT\_FAILURE)'

The operation failed or the command syntax was not valid.

**certtool See Also**

pl1tool (1)

**certtool Examples****Generating private keys**

To create an RSA private key, run:

```
$ certtool --generate-privkey --outfile key.pem --rsa
```

To create a DSA or elliptic curves (ECDSA) private key use the above command combined with 'dsa' or 'ecc' options.

**Generating certificate requests**

To create a certificate request (needed when the certificate is issued by another party), run:

```
certtool --generate-request --load-privkey key.pem \
--outfile request.pem
```

If the private key is stored in a smart card you can generate a request by specifying the private key object URL.

```
$ ./certtool --generate-request --load-privkey "pkcs11:..." \
--load-pubkey "pkcs11:..." --outfile request.pem
```

**Generating a self-signed certificate**

To create a self signed certificate, use the command:

```
$ certtool --generate-privkey --outfile ca-key.pem
$ certtool --generate-self-signed --load-privkey ca-key.pem \
--outfile ca-cert.pem
```

Note that a self-signed certificate usually belongs to a certificate authority, that signs other certificates.

**Generating a certificate**

To generate a certificate using the previous request, use the command:

```
$ certtool --generate-certificate --load-request request.pem \
--outfile cert.pem --load-ca-certificate ca-cert.pem \
--load-ca-privkey ca-key.pem
```

To generate a certificate using the private key only, use the command:

```
$ certtool --generate-certificate --load-privkey key.pem \
--outfile cert.pem --load-ca-certificate ca-cert.pem \
--load-ca-privkey ca-key.pem
```

## Certificate information

To view the certificate information, use:

```
$ certtool --certificate-info --infile cert.pem
```

## PKCS #12 structure generation

To generate a PKCS #12 structure using the previous key and certificate, use the command:

```
$ certtool --load-certificate cert.pem --load-privkey key.pem \
  --to-p12 --outder --outfile key.p12
```

Some tools (reportedly web browsers) have problems with that file because it does not contain the CA certificate for the certificate. To work around that problem in the tool, you can use the `--load-ca-certificate` parameter as follows:

```
$ certtool --load-ca-certificate ca.pem \
  --load-certificate cert.pem --load-privkey key.pem \
  --to-p12 --outder --outfile key.p12
```

## Diffie-Hellman parameter generation

To generate parameters for Diffie-Hellman key exchange, use the command:

```
$ certtool --generate-dh-params --outfile dh.pem --sec-param medium
```

## Proxy certificate generation

Proxy certificate can be used to delegate your credential to a temporary, typically short-lived, certificate. To create one from the previously created certificate, first create a temporary key and then generate a proxy certificate for it, using the commands:

```
$ certtool --generate-privkey > proxy-key.pem
$ certtool --generate-proxy --load-ca-privkey key.pem \
  --load-privkey proxy-key.pem --load-certificate cert.pem \
  --outfile proxy-cert.pem
```

## Certificate revocation list generation

To create an empty Certificate Revocation List (CRL) do:

```
$ certtool --generate-crl --load-ca-privkey x509-ca-key.pem \
  --load-ca-certificate x509-ca.pem
```

To create a CRL that contains some revoked certificates, place the certificates in a file and use `--load-certificate` as follows:

```
$ certtool --generate-crl --load-ca-privkey x509-ca-key.pem \
  --load-ca-certificate x509-ca.pem --load-certificate revoked-certs.pem
```

To verify a Certificate Revocation List (CRL) do:

```
$ certtool --verify-crl --load-ca-certificate x509-ca.pem < crl.pem
```

## certtool Files

## Certtool's template file format

A template file can be used to avoid the interactive questions of certtool. Initially create a file named 'cert.cfg' that contains the information about the certificate. The template can be used as below:

```
$ certtool --generate-certificate --load-privkey key.pem \
  --template cert.cfg --outfile cert.pem \
  --load-ca-certificate ca-cert.pem --load-ca-privkey ca-key.pem
```

An example certtool template file that can be used to generate a certificate request or a self signed certificate follows.

```
# X.509 Certificate options
#
# DN options

# The organization of the subject.
organization = "Koko inc."

# The organizational unit of the subject.
unit = "sleeping dept."

# The locality of the subject.
# locality =

# The state of the certificate owner.
state = "Attiki"

# The country of the subject. Two letter code.
country = GR

# The common name of the certificate owner.
cn = "Cindy Lauper"

# A user id of the certificate owner.
#uid = "clauper"

# Set domain components
#dc = "name"
#dc = "domain"

# If the supported DN OIDs are not adequate you can set
# any OID here.
# For example set the X.520 Title and the X.520 Pseudonym
# by using OID and string pairs.
#dn_oid = 2.5.4.12 Dr.
#dn_oid = 2.5.4.65 jackal

# This is deprecated and should not be used in new
```



```
# certificates.
# pkcs9_email = "none@none.org"

# An alternative way to set the certificate's distinguished name directly
# is with the "dn" option. The attribute names allowed are:
# C (country), street, O (organization), OU (unit), title, CN (common name),
# L (locality), ST (state), placeOfBirth, gender, countryOfCitizenship,
# countryOfResidence, serialNumber, telephoneNumber, surName, initials,
# generationQualifier, givenName, pseudonym, dnQualifier, postalCode, name,
# businessCategory, DC, UID, jurisdictionOfIncorporationLocalityName,
# jurisdictionOfIncorporationStateOrProvinceName,
# jurisdictionOfIncorporationCountryName, XmppAddr, and numeric OIDs.

#dn = "cn=Nik,st=Attiki,C=GR,surName=Mavrogiannopoulos,2.5.4.9=Arkadias"

# The serial number of the certificate
# Comment the field for a time-based serial number.
serial = 007

# In how many days, counting from today, this certificate will expire.
# Use -1 if there is no expiration date.
expiration_days = 700

# Alternatively you may set concrete dates and time. The GNU date string
# formats are accepted. See:
# http://www.gnu.org/software/tar/manual/html\_node/Date-input-formats.html

#activation_date = "2004-02-29 16:21:42"
#expiration_date = "2025-02-29 16:24:41"

# X.509 v3 extensions

# A dnsname in case of a WWW server.
#dns_name = "www.none.org"
#dns_name = "www.morethanone.org"

# A subject alternative name URI
#uri = "http://www.example.com"

# An IP address in case of a server.
#ip_address = "192.168.1.1"

# An email in case of a person
email = "none@none.org"

# Challenge password used in certificate requests
challenge_password = 123456
```

```
# Password when encrypting a private key
#password = secret

# An URL that has CRLs (certificate revocation lists)
# available. Needed in CA certificates.
#crl_dist_points = "http://www.getcrl.crl/getcrl/"

# Whether this is a CA certificate or not
#ca

# for microsoft smart card logon
# key_purpose_oid = 1.3.6.1.4.1.311.20.2.2

### Other predefined key purpose OIDs

# Whether this certificate will be used for a TLS client
#tls_www_client

# Whether this certificate will be used for a TLS server
#tls_www_server

# Whether this certificate will be used to sign data (needed
# in TLS DHE ciphersuites).
signing_key

# Whether this certificate will be used to encrypt data (needed
# in TLS RSA ciphersuites). Note that it is preferred to use different
# keys for encryption and signing.
encryption_key

# Whether this key will be used to sign other certificates.
#cert_signing_key

# Whether this key will be used to sign CRLs.
#crl_signing_key

# Whether this key will be used to sign code.
#code_signing_key

# Whether this key will be used to sign OCSP data.
#ocsp_signing_key

# Whether this key will be used for time stamping.
#time_stamping_key

# Whether this key will be used for IPsec IKE operations.
```

```
#ipsec_ike_key

### end of key purpose OIDs

# When generating a certificate from a certificate
# request, then honor the extensions stored in the request
# and store them in the real certificate.
#honor_crq_extensions

# Path length constraint. Sets the maximum number of
# certificates that can be used to certify this certificate.
# (i.e. the certificate chain length)
#path_len = -1
#path_len = 2

# OCSP URI
# ocsp_uri = http://my.ocsp.server/ocsp

# CA issuers URI
# ca_issuers_uri = http://my.ca.issuer

# Certificate policies
#policy1 = 1.3.6.1.4.1.5484.1.10.99.1.0
#policy1_txt = "This is a long policy to summarize"
#policy1_url = http://www.example.com/a-policy-to-read

#policy2 = 1.3.6.1.4.1.5484.1.10.99.1.1
#policy2_txt = "This is a short policy"
#policy2_url = http://www.example.com/another-policy-to-read

# Name constraints

# DNS
#nc_permit_dns = example.com
#nc_exclude_dns = test.example.com

# EMAIL
#nc_permit_email = "nmav@ex.net"

# Exclude subdomains of example.com
#nc_exclude_email = .example.com

# Exclude all e-mail addresses of example.com
#nc_exclude_email = example.com

# Options for proxy certificates
```

```
#proxy_policy_language = 1.3.6.1.5.5.7.21.1

# Options for generating a CRL

# The number of days the next CRL update will be due.
# next CRL update will be in 43 days
#crl_next_update = 43

# this is the 5th CRL by this CA
# Comment the field for a time-based number.
#crl_number = 5
```

### 4.2.6 Invoking ocsptool

Ocsptool is a program that can parse and print information about OCSP requests/responses, generate requests and verify responses.

This section was generated by **AutoGen**, using the **agtexi-cmd** template and the option descriptions for the **ocsptool** program. This software is released under the GNU General Public License, version 3 or later.

#### ocsptool help/usage (--help)

This is the automatically generated usage text for ocsptool.

The text printed is the same whether selected with the **help** option (**--help**) or the **more-help** option (**--more-help**). **more-help** will print the usage text by passing it through a pager program. **more-help** is disabled on platforms without a working **fork(2)** function. The **PAGER** environment variable is used to select the program, defaulting to **more**. Both will exit with a status code of 0.

```
ocsptool is unavailable - no --help
```

#### debug option (-d)

This is the “enable debugging” option. This option takes a number argument. Specifies the debug level.

#### ask option

This is the “ask an ocsf/http server on a certificate validity” option. This option takes an optional string argument **server name|url**.

This option has some usage constraints. It:

- must appear in combination with the following options: **load-cert**, **load-issuer**.

Connects to the specified HTTP OCSP server and queries on the validity of the loaded certificate.

#### ocsptool exit status

One of the following exit values will be returned:

`'0 (EXIT_SUCCESS)'`

Successful program execution.

`'1 (EXIT_FAILURE)'`

The operation failed or the command syntax was not valid.

## **ocsptool See Also**

`certtool (1)`

## **ocsptool Examples**

### **Print information about an OCSP request**

To parse an OCSP request and print information about the content, the `-i` or `--request-info` parameter may be used as follows. The `-Q` parameter specifies the name of the file containing the OCSP request, and it should contain the OCSP request in binary DER format.

```
$ ocsptool -i -Q ocsptool-request.der
```

The input file may also be sent to standard input like this:

```
$ cat ocsptool-request.der | ocsptool --request-info
```

### **Print information about an OCSP response**

Similar to parsing OCSP requests, OCSP responses can be parsed using the `-j` or `--response-info` as follows.

```
$ ocsptool -j -Q ocsptool-response.der
```

```
$ cat ocsptool-response.der | ocsptool --response-info
```

### **Generate an OCSP request**

The `-q` or `--generate-request` parameters are used to generate an OCSP request. By default the OCSP request is written to standard output in binary DER format, but can be stored in a file using `--outfile`. To generate an OCSP request the issuer of the certificate to check needs to be specified with `--load-issuer` and the certificate to check with `--load-cert`. By default PEM format is used for these files, although `--inder` can be used to specify that the input files are in DER format.

```
$ ocsptool -q --load-issuer issuer.pem --load-cert client.pem \
  --outfile ocsptool-request.der
```

When generating OCSP requests, the tool will add an OCSP extension containing a nonce. This behaviour can be disabled by specifying `--no-nonce`.

### **Verify signature in OCSP response**

To verify the signature in an OCSP response the `-e` or `--verify-response` parameter is used. The tool will read an OCSP response in DER format from standard input, or from the file specified by `--load-response`. The OCSP response is verified against a set of trust anchors, which are specified using `--load-trust`. The trust anchors are concatenated certificates in PEM format. The certificate that signed the OCSP response needs to be in the set of trust anchors, or the issuer of the signer certificate needs to be in the set of trust anchors and the OCSP Extended Key Usage bit has to be asserted in the signer certificate.

```
$ ocsptool -e --load-trust issuer.pem \
    --load-response ocsdp-response.der
```

The tool will print status of verification.

### Verify signature in OCSF response against given certificate

It is possible to override the normal trust logic if you know that a certain certificate is supposed to have signed the OCSF response, and you want to use it to check the signature. This is achieved using `--load-signer` instead of `--load-trust`. This will load one certificate and it will be used to verify the signature in the OCSF response. It will not check the Extended Key Usage bit.

```
$ ocsptool -e --load-signer ocsdp-signer.pem \
    --load-response ocsdp-response.der
```

This approach is normally only relevant in two situations. The first is when the OCSF response does not contain a copy of the signer certificate, so the `--load-trust` code would fail. The second is if you want to avoid the indirect mode where the OCSF response signer certificate is signed by a trust anchor.

### Real-world example

Here is an example of how to generate an OCSF request for a certificate and to verify the response. For illustration we'll use the `blog.josefsson.org` host, which (as of writing) uses a certificate from CACert. First we'll use `gnutls-cli` to get a copy of the server certificate chain. The server is not required to send this information, but this particular one is configured to do so.

```
$ echo | gnutls-cli -p 443 blog.josefsson.org --print-cert > chain.pem
```

Use a text editor on `chain.pem` to create three files for each separate certificates, called `cert.pem` for the first certificate for the domain itself, secondly `issuer.pem` for the intermediate certificate and `root.pem` for the final root certificate.

The domain certificate normally contains a pointer to where the OCSF responder is located, in the Authority Information Access Information extension. For example, from `certtool -i < cert.pem` there is this information:

```
Authority Information Access Information (not critical):
Access Method: 1.3.6.1.5.5.7.48.1 (id-ad-ocsp)
Access Location URI: http://ocsp.CAcert.org/
```

This means the CA support OCSF queries over HTTP. We are now ready to create a OCSF request for the certificate.

```
$ ocsptool --ask ocsp.CAcert.org --load-issuer issuer.pem \
    --load-cert cert.pem --outfile ocsdp-response.der
```

The request is sent via HTTP to the OCSF server address specified. If the address is omitted ocsptool will use the address stored in the certificate.

#### 4.2.7 Invoking danetool

Tool to generate and check DNS resource records for the DANE protocol.

This section was generated by **AutoGen**, using the `agtexi-cmd` template and the option descriptions for the `danetool` program. This software is released under the GNU General Public License, version 3 or later.

**danetool help/usage (--help)**

This is the automatically generated usage text for danetool.

The text printed is the same whether selected with the **help** option (**--help**) or the **more-help** option (**--more-help**). **more-help** will print the usage text by passing it through a pager program. **more-help** is disabled on platforms without a working **fork(2)** function. The **PAGER** environment variable is used to select the program, defaulting to **more**. Both will exit with a status code of 0.

danetool is unavailable - no --help

**debug option (-d)**

This is the “enable debugging” option. This option takes a number argument. Specifies the debug level.

**load-pubkey option**

This is the “loads a public key file” option. This option takes a string argument. This can be either a file or a PKCS #11 URL

**load-certificate option**

This is the “loads a certificate file” option. This option takes a string argument. This can be either a file or a PKCS #11 URL

**dlv option**

This is the “sets a dlv file” option. This option takes a string argument. This sets a DLV file to be used for DNSSEC verification.

**hash option**

This is the “hash algorithm to use for signing” option. This option takes a string argument. Available hash functions are SHA1, RMD160, SHA256, SHA384, SHA512.

**check option**

This is the “check a host’s dane tlsa entry” option. This option takes a string argument. Obtains the DANE TLSA entry from the given hostname and prints information. Note that the actual certificate of the host can be provided using **-load-certificate**, otherwise danetool will connect to the server to obtain it. The exit code on verification success will be zero.

**check-ee option**

This is the “check only the end-entity’s certificate” option. Checks the end-entity’s certificate only. Trust anchors or CAs are not considered.

**check-ca option**

This is the “check only the ca’s certificate” option. Checks the trust anchor’s and CA’s certificate only. End-entities are not considered.

### **tlsa-rr option**

This is the “print the dane rr data on a certificate or public key” option.

This option has some usage constraints. It:

- must appear in combination with the following options: host.

This command prints the DANE RR data needed to enable DANE on a DNS server.

### **host option**

This is the “specify the hostname to be used in the dane rr” option. This option takes a string argument **Hostname**. This command sets the hostname for the DANE RR.

### **proto option**

This is the “the protocol set for dane data (tcp, udp etc.)” option. This option takes a string argument **Protocol**. This command specifies the protocol for the service set in the DANE data.

### **app-proto option**

This is the “the application protocol to be used to obtain the server’s certificate (https, ftp, smtp, imap)” option. This option takes a string argument. When the server’s certificate isn’t provided danetool will connect to the server to obtain the certificate. In that case it is required to know the protocol to talk with the server prior to initiating the TLS handshake.

### **ca option**

This is the “whether the provided certificate or public key is a certificate authority” option. Marks the DANE RR as a CA certificate if specified.

### **x509 option**

This is the “use the hash of the x.509 certificate, rather than the public key” option. This option forces the generated record to contain the hash of the full X.509 certificate. By default only the hash of the public key is used.

### **local option**

This is an alias for the **domain** option, see [\[danetool domain\]](#), page 72.

### **domain option**

This is the “the provided certificate or public key is issued by the local domain” option.

This option has some usage constraints. It:

- can be disabled with `-no-domain`.
- It is enabled by default.

DANE distinguishes certificates and public keys offered via the DNSSEC to trusted and local entities. This flag indicates that this is a domain-issued certificate, meaning that there could be no CA involved.



### local-dns option

This is the “use the local dns server for dnssec resolving” option.

This option has some usage constraints. It:

- can be disabled with `-no-local-dns`.

This option will use the local DNS server for DNSSEC. This is disabled by default due to many servers not allowing DNSSEC.

### insecure option

This is the “do not verify any dnssec signature” option. Ignores any DNSSEC signature verification results.

### inder option

This is the “use der format for input certificates and private keys” option.

This option has some usage constraints. It:

- can be disabled with `-no-inder`.

The input files will be assumed to be in DER or RAW format. Unlike options that in PEM input would allow multiple input data (e.g. multiple certificates), when reading in DER format a single data structure is read.

### inraw option

This is an alias for the `inder` option, see [\[danetool inder\]](#), page 73.

### print-raw option

This is the “print the received dane data in raw format” option.

This option has some usage constraints. It:

- can be disabled with `-no-print-raw`.

This option will print the received DANE data.

### quiet option

This is the “suppress several informational messages” option. In that case on the exit code can be used as an indication of verification success

### danetool exit status

One of the following exit values will be returned:

`'0 (EXIT_SUCCESS)'`

Successful program execution.

`'1 (EXIT_FAILURE)'`

The operation failed or the command syntax was not valid.

### danetool See Also

`certtool` (1)

## danetool Examples

### DANE TLSA RR generation

To create a DANE TLSA resource record for a certificate (or public key) that was issued locally and may or may not be signed by a CA use the following command.

```
$ danetool --tlsa-rr --host www.example.com --load-certificate cert.pem
```

To create a DANE TLSA resource record for a CA signed certificate, which will be marked as such use the following command.

```
$ danetool --tlsa-rr --host www.example.com --load-certificate cert.pem \
--no-domain
```

The former is useful to add in your DNS entry even if your certificate is signed by a CA. That way even users who do not trust your CA will be able to verify your certificate using DANE.

In order to create a record for the CA signer of your certificate use the following.

```
$ danetool --tlsa-rr --host www.example.com --load-certificate cert.pem \
--ca --no-domain
```

To read a server's DANE TLSA entry, use:

```
$ danetool --check www.example.com --proto tcp --port 443
```

To verify a server's DANE TLSA entry, use:

```
$ danetool --check www.example.com --proto tcp --port 443 --load-certificate chain.p
```

## 4.3 Shared-key and anonymous authentication

In addition to certificate authentication, the TLS protocol may be used with password, shared-key and anonymous authentication methods. The rest of this chapter discusses details of these methods.

### 4.3.1 SRP authentication

#### 4.3.1.1 Authentication using SRP

GnuTLS supports authentication via the Secure Remote Password or SRP protocol (see [RFC2945,TOMSRP] for a description). The SRP key exchange is an extension to the TLS protocol, and it provides an authenticated with a password key exchange. The peers can be identified using a single password, or there can be combinations where the client is authenticated using SRP and the server using a certificate.

The advantage of SRP authentication, over other proposed secure password authentication schemes, is that SRP is not susceptible to off-line dictionary attacks. Moreover, SRP does not require the server to hold the user's password. This kind of protection is similar to the one used traditionally in the UNIX `/etc/passwd` file, where the contents of this file did not cause harm to the system security if they were revealed. The SRP needs instead of the plain password something called a verifier, which is calculated using the user's password, and if stolen cannot be used to impersonate the user.

Typical conventions in SRP are a password file, called `tpasswd` that holds the SRP verifiers (encoded passwords) and another file, `tpasswd.conf`, which holds the allowed SRP pa-

rameters. The included in GnuTLS helper follow those conventions. The `srptool` program, discussed in the next section is a tool to manipulate the SRP parameters.

The implementation in GnuTLS is based on [TLSSRP]. The supported key exchange methods are shown below.

- SRP:** Authentication using the SRP protocol.
- SRP\_DSS:** Client authentication using the SRP protocol. Server is authenticated using a certificate with DSA parameters.
- SRP\_RSA:** Client authentication using the SRP protocol. Server is authenticated using a certificate with RSA parameters.

```
int gnutls_srp_verifier (const char * username, const char * password, const gnutls_datum_t * salt, const gnutls_datum_t * generator,
                        const gnutls_datum_t * prime, gnutls_datum_t * res)
```

[Function]

*username*: is the user's name

*password*: is the user's password

*salt*: should be some randomly generated bytes

*generator*: is the generator of the group

*prime*: is the group's prime

*res*: where the verifier will be stored.

This function will create an SRP verifier, as specified in RFC2945. The **prime** and **generator** should be one of the static parameters defined in `gnutls/gnutls.h` or may be generated.

The verifier will be allocated with `gnutls_malloc ()` and will be stored in **res** using binary format.

**Returns:** On success, `GNUTLS_E_SUCCESS (0)` is returned, or an error code.

```
int [gnutls_srp_base64_encode_alloc], page 348 (const gnutls_datum_t * data,
gnutls_datum_t * result)
```

```
int [gnutls_srp_base64_decode_alloc], page 347 (const gnutls_datum_t *
b64_data, gnutls_datum_t * result)
```

#### 4.3.1.2 Invoking `srptool`

Simple program that emulates the programs in the Stanford SRP (Secure Remote Password) libraries using GnuTLS. It is intended for use in places where you don't expect SRP authentication to be the used for system users.

In brief, to use SRP you need to create two files. These are the password file that holds the users and the verifiers associated with them and the configuration file to hold the group parameters (called `tpasswd.conf`).

This section was generated by **AutoGen**, using the `agtexi-cmd` template and the option descriptions for the `srptool` program. This software is released under the GNU General Public License, version 3 or later.

**srptool help/usage (--help)**

This is the automatically generated usage text for srptool.

The text printed is the same whether selected with the **help** option (**--help**) or the **more-help** option (**--more-help**). **more-help** will print the usage text by passing it through a pager program. **more-help** is disabled on platforms without a working **fork(2)** function. The **PAGER** environment variable is used to select the program, defaulting to **more**. Both will exit with a status code of 0.

```
srptool is unavailable - no --help
```

**debug option (-d)**

This is the “enable debugging” option. This option takes a number argument. Specifies the debug level.

**verify option**

This is the “just verify the password.” option. Verifies the password provided against the password file.

**passwd-conf option (-v)**

This is the “specify a password conf file.” option. This option takes a string argument. Specify a filename or a PKCS #11 URL to read the CAs from.

**create-conf option**

This is the “generate a password configuration file.” option. This option takes a string argument. This generates a password configuration file (tpasswd.conf) containing the required for TLS parameters.

**srptool exit status**

One of the following exit values will be returned:

```
'0 (EXIT_SUCCESS)'
```

Successful program execution.

```
'1 (EXIT_FAILURE)'
```

The operation failed or the command syntax was not valid.

**srptool See Also**

gnutls-cli-debug (1), gnutls-serv (1), srptool (1), psktool (1), certtool (1)

**srptool Examples**

To create **tpasswd.conf** which holds the **g** and **n** values for SRP protocol (generator and a large prime), run:

```
$ srptool --create-conf /etc/tpasswd.conf
```

This command will create **/etc/tpasswd** and will add user 'test' (you will also be prompted for a password). Verifiers are stored by default in the way libsrp expects.

```
$ srptool --passwd /etc/tpasswd --passwd-conf /etc/tpasswd.conf -u test
```

This command will check against a password. If the password matches the one in `/etc/tpasswd` you will get an ok.

```
$ srptool --passwd /etc/tpasswd --passwd\-conf /etc/tpasswd.conf --verify -u test
```

## 4.3.2 PSK authentication

### 4.3.2.1 Authentication using PSK

Authentication using Pre-shared keys is a method to authenticate using usernames and binary keys. This protocol avoids making use of public key infrastructure and expensive calculations, thus it is suitable for constraint clients.

The implementation in GnuTLS is based on [TLSPSK]. The supported PSK key exchange methods are:

**PSK:** Authentication using the PSK protocol.

**DHE-PSK:** Authentication using the PSK protocol and Diffie-Hellman key exchange. This method offers perfect forward secrecy.

**ECDHE-PSK:** Authentication using the PSK protocol and Elliptic curve Diffie-Hellman key exchange. This method offers perfect forward secrecy.

**RSA-PSK:** Authentication using the PSK protocol for the client and an RSA certificate for the server.

Helper functions to generate and maintain PSK keys are also included in GnuTLS.

```
int [gnutls_key_generate], page 315 (gnutls_datum_t * key, unsigned int
key_size)
int [gnutls_hex_encode], page 315 (const gnutls_datum_t * data, char * result,
size_t * result_size)
int [gnutls_hex_decode], page 314 (const gnutls_datum_t * hex_data, void *
result, size_t * result_size)
```

### 4.3.2.2 Invoking psktool

Program that generates random keys for use with TLS-PSK. The keys are stored in hexadecimal format in a key file.

This section was generated by **AutoGen**, using the `agtexi-cmd` template and the option descriptions for the `psktool` program. This software is released under the GNU General Public License, version 3 or later.

#### **psktool help/usage (--help)**

This is the automatically generated usage text for `psktool`.

The text printed is the same whether selected with the `help` option (`--help`) or the `more-help` option (`--more-help`). `more-help` will print the usage text by passing it through a pager program. `more-help` is disabled on platforms without a working `fork(2)` function. The `PAGER` environment variable is used to select the program, defaulting to `more`. Both will exit with a status code of 0.

```
psktool is unavailable - no --help
```

### debug option (-d)

This is the “enable debugging” option. This option takes a number argument. Specifies the debug level.

### psktool exit status

One of the following exit values will be returned:

‘0 (EXIT\_SUCCESS)’

Successful program execution.

‘1 (EXIT\_FAILURE)’

The operation failed or the command syntax was not valid.

### psktool See Also

gnutls-cli-debug (1), gnutls-serv (1), srptool (1), certtool (1)

### psktool Examples

To add a user ‘psk\_identity’ in `passwd.psk` for use with GnuTLS run:

```
$ ./psktool -u psk_identity -p passwd.psk
Generating a random key for user 'psk_identity'
Key stored to passwd.psk
$ cat psks.txt
psk_identity:88f3824b3e5659f52d00e959bacab954b6540344
$
```

This command will create `passwd.psk` if it does not exist and will add user ‘psk\_identity’ (you will also be prompted for a password).

### 4.3.3 Anonymous authentication

The anonymous key exchange offers encryption without any indication of the peer’s identity. This kind of authentication is vulnerable to a man in the middle attack, but can be used even if there is no prior communication or shared trusted parties with the peer. It is useful to establish a session over which certificate authentication will occur in order to hide the identities of the participants from passive eavesdroppers.

Unless in the above case, it is not recommended to use anonymous authentication. In the cases where there is no prior communication with the peers, an alternative with better properties, such as key continuity, is trust on first use (see [Section 4.1.3.1 \[Verifying a certificate using trust on first use authentication\]](#), page 40).

The available key exchange algorithms for anonymous authentication are shown below, but note that few public servers support them, and they have to be explicitly enabled.

**ANON\_DH:** This algorithm exchanges Diffie-Hellman parameters.

**ANON\_ECDH:**

This algorithm exchanges elliptic curve Diffie-Hellman parameters. It is more efficient than ANON\_DH on equivalent security levels.

## 4.4 Selecting an appropriate authentication method

This section provides some guidance on how to use the available authentication methods in GnuTLS in various scenarios.

### 4.4.1 Two peers with an out-of-band channel

Let's consider two peers who need to communicate over an untrusted channel (the Internet), but have an out-of-band channel available. The latter channel is considered safe from eavesdropping and message modification and thus can be used for an initial bootstrapping of the protocol. The options available are:

- Pre-shared keys (see [Section 4.3.2 \[PSK authentication\]](#), page 77). The server and a client communicate a shared randomly generated key over the trusted channel and use it to negotiate further sessions over the untrusted channel.
- Passwords (see [Section 4.3.1 \[SRP authentication\]](#), page 74). The client communicates to the server its username and password of choice and uses it to negotiate further sessions over the untrusted channel.
- Public keys (see [Section 4.1 \[Certificate authentication\]](#), page 18). The client and the server exchange their public keys (or fingerprints of them) over the trusted channel. On future sessions over the untrusted channel they verify the key being the same (similar to [Section 4.1.3.1 \[Verifying a certificate using trust on first use authentication\]](#), page 40).

Provided that the out-of-band channel is trusted all of the above provide a similar level of protection. An out-of-band channel may be the initial bootstrapping of a user's PC in a corporate environment, in-person communication, communication over an alternative network (e.g. the phone network), etc.

### 4.4.2 Two peers without an out-of-band channel

When an out-of-band channel is not available a peer cannot be reliably authenticated. What can be done, however, is to allow some form of registration of users connecting for the first time and ensure that their keys remain the same after that initial connection. This is termed key continuity or trust on first use (TOFU).

The available option is to use public key authentication (see [Section 4.1 \[Certificate authentication\]](#), page 18). The client and the server store each other's public keys (or fingerprints of them) and associate them with their identity. On future sessions over the untrusted channel they verify the keys being the same (see [Section 4.1.3.1 \[Verifying a certificate using trust on first use authentication\]](#), page 40).

To mitigate the uncertainty of the information exchanged in the first connection other channels over the Internet may be used, e.g., DNSSEC (see [Section 4.1.3.2 \[Verifying a certificate using DANE\]](#), page 40).

### 4.4.3 Two peers and a trusted third party

When a trusted third party is available (or a certificate authority) the most suitable option is to use certificate authentication (see [Section 4.1 \[Certificate authentication\]](#), page 18). The client and the server obtain certificates that associate their identity and public keys using a digital signature by the trusted party and use them to on the subsequent communications with each other. Each party verifies the peer's certificate using the trusted third party's

signature. The parameters of the third party's signature are present in its certificate which must be available to all communicating parties.

While the above is the typical authentication method for servers in the Internet by using the commercial CAs, the users that act as clients in the protocol rarely possess such certificates. In that case a hybrid method can be used where the server is authenticated by the client using the commercial CAs and the client is authenticated based on some information the client provided over the initial server-authenticated channel. The available options are:

- Passwords (see [Section 4.3.1 \[SRP authentication\]](#), page 74). The client communicates to the server its username and password of choice on the initial server-authenticated connection and uses it to negotiate further sessions. This is possible because the SRP protocol allows for the server to be authenticated using a certificate and the client using the password.
- Public keys (see [Section 4.1 \[Certificate authentication\]](#), page 18). The client sends its public key to the server (or a fingerprint of it) over the initial server-authenticated connection. On future sessions the client verifies the server using the third party certificate and the server verifies that the client's public key remained the same (see [Section 4.1.3.1 \[Verifying a certificate using trust on first use authentication\]](#), page 40).



## 5 Hardware security modules and abstract key types

In several cases storing the long term cryptographic keys in a hard disk or even in memory poses a significant risk. Once the system they are stored is compromised the keys must be replaced as the secrecy of future sessions is no longer guaranteed. Moreover, past sessions that were not protected by a perfect forward secrecy offering ciphersuite are also to be assumed compromised.

If such threats need to be addressed, then it may be wise storing the keys in a security module such as a smart card, an HSM or the TPM chip. Those modules ensure the protection of the cryptographic keys by only allowing operations on them and preventing their extraction. The purpose of the abstract key API is to provide an API that will allow the handle of keys in memory and files, as well as keys stored in such modules.

In GnuTLS the approach is to handle all keys transparently by the high level API, e.g., the API that loads a key or certificate from a file. The high-level API will accept URIs in addition to files that specify keys on an HSM or in TPM, and a callback function will be used to obtain any required keys. The URI format is defined in *[TPMURI]* and *[PKCS11URI]*, and is in the process of being standardized across systems.

More information on the API is provided in the next sections. Examples of a URI of a certificate stored in an HSM, as well as a key stored in the TPM chip are shown below. To discover the URIs of the objects the `p11tool` (see [Section 5.2.6 \[p11tool Invocation\]](#), page 96), or `tpmtool` (see [Section 5.3.4 \[tpmtool Invocation\]](#), page 103) may be used.

```
pkcs11:token=Nikos;serial=307521161601031;model=PKCS%2315; \
manufacturer=EnterSafe;object=test1;objecttype=cert
```

```
tpmkey:uuid=42309df8-d101-11e1-a89a-97bb33c23ad1;storage=user
```

### 5.1 Abstract key types

Since there are many forms of a public or private keys supported by GnuTLS such as X.509, OpenPGP, PKCS #11 or TPM it is desirable to allow common operations on them. For these reasons the abstract `gnutls_privkey_t` and `gnutls_pubkey_t` were introduced in `gnutls/abstract.h` header. Those types are initialized using a specific type of key and then can be used to perform operations in an abstract way. For example in order to sign an X.509 certificate with a key that resides in a token the following steps can be used.

```
#include <gnutls/abstract.h>

void sign_cert( gnutls_x509_crt_t to_be_signed)
{
    gnutls_x509_crt_t ca_cert;
    gnutls_privkey_t abs_key;

    /* initialize the abstract key */
    gnutls_privkey_init(&abs_key);

    /* keys stored in tokens are identified by URLs */
    gnutls_privkey_import_url(abs_key, key_url);
```

```

    gnutls_x509_cert_init(&ca_cert);
    gnutls_x509_cert_import_pkcs11_url(&ca_cert, cert_url);

    /* sign the certificate to be signed */
    gnutls_x509_cert_privkey_sign(to_be_signed, ca_cert, abs_key,
                                  GNUTLS_DIG_SHA256, 0);
}

```

### 5.1.1 Public keys

An abstract `gnutls_pubkey_t` can be initialized using the functions below. It can be imported through an existing structure like `gnutls_x509_cert_t`, or through an ASN.1 encoding of the X.509 SubjectPublicKeyInfo sequence.

```

int [gnutls_pubkey_import_x509], page 542 (gnutls_pubkey_t key,
gnutls_x509_cert_t crt, unsigned int flags)
int [gnutls_pubkey_import_openpgp], page 539 (gnutls_pubkey_t key,
gnutls_openpgp_cert_t crt, unsigned int flags)
int [gnutls_pubkey_import_pkcs11], page 540 (gnutls_pubkey_t key,
gnutls_pkcs11_obj_t obj, unsigned int flags)
int [gnutls_pubkey_import_url], page 542 (gnutls_pubkey_t key, const char *
url, unsigned int flags)
int [gnutls_pubkey_import_privkey], page 541 (gnutls_pubkey_t key,
gnutls_privkey_t pkey, unsigned int usage, unsigned int flags)
int [gnutls_pubkey_import], page 538 (gnutls_pubkey_t key, const
gnutls_datum_t * data, gnutls_x509_cert_fmt_t format)
int [gnutls_pubkey_export], page 534 (gnutls_pubkey_t key,
gnutls_x509_cert_fmt_t format, void * output_data, size_t * output_data_size)

int gnutls_pubkey_export2 (gnutls_pubkey_t key,                                [Function]
    gnutls_x509_cert_fmt_t format, gnutls_datum_t * out)

```

*key*: Holds the certificate

*format*: the format of output params. One of PEM or DER.

*out*: will contain a certificate PEM or DER encoded

This function will export the public key to DER or PEM format. The contents of the exported data is the SubjectPublicKeyInfo X.509 structure.

The output buffer will be allocated using `gnutls_malloc()`.

If the structure is PEM encoded, it will have a header of "BEGIN CERTIFICATE".

**Returns:** In case of failure a negative error code will be returned, and 0 on success.

**Since:** 3.1.3

Other helper functions that allow directly importing from raw X.509 or OpenPGP structures are shown below.

```
int [gnutls_pubkey_import_x509_raw], page 543 (gnutls_pubkey_t pkey, const
gnutls_datum_t * data, gnutls_x509_crt_fmt_t format, unsigned int flags)
int [gnutls_pubkey_import_opengpg_raw], page 540 (gnutls_pubkey_t pkey, const
gnutls_datum_t * data, gnutls_opengpg_crt_fmt_t format, const
gnutls_opengpg_keyid_t keyid, unsigned int flags)
```

An important function is [\[gnutls\\_pubkey\\_import\\_url\], page 542](#) which will import public keys from URLs that identify objects stored in tokens (see [Section 5.2 \[Smart cards and HSMs\], page 88](#) and [Section 5.3 \[Trusted Platform Module\], page 100](#)). A function to check for a supported by GnuTLS URL is [\[gnutls\\_url\\_is\\_supported\], page 361](#).

```
int gnutls_url_is_supported (const char * url) [Function]
url: A PKCS 11 url
```

Check whether url is supported. Depending on the system libraries GnuTLS may support pkcs11 or tpmkey URLs.

**Returns:** return non-zero if the given URL is supported, and zero if it is not known.

**Since:** 3.1.0

Additional functions are available that will return information over a public key, such as a unique key ID, as well as a function that given a public key fingerprint would provide a memorable sketch.

Note that [\[gnutls\\_pubkey\\_get\\_key\\_id\], page 536](#) calculates a SHA1 digest of the public key as a DER-formatted, subjectPublicKeyInfo object. Other implementations use different approaches, e.g., some use the “common method” described in section 4.2.1.2 of [\[RFC5280\]](#) which calculates a digest on a part of the subjectPublicKeyInfo object.

```
int [gnutls_pubkey_get_pk_algorithm], page 537 (gnutls_pubkey_t key, unsigned
int * bits)
int [gnutls_pubkey_get_preferred_hash_algorithm], page 537 (gnutls_pubkey_t
key, gnutls_digest_algorithm_t * hash, unsigned int * mand)
int [gnutls_pubkey_get_key_id], page 536 (gnutls_pubkey_t key, unsigned int
flags, unsigned char * output_data, size_t * output_data_size)
int [gnutls_random_art], page 331 (gnutls_random_art_t type, const char *
key_type, unsigned int key_size, void * fpr, size_t fpr_size, gnutls_datum_t *
art)
```

To export the key-specific parameters, or obtain a unique key ID the following functions are provided.

```
int [gnutls_pubkey_export_rsa_raw], page 536 (gnutls_pubkey_t key,
gnutls_datum_t * m, gnutls_datum_t * e)
int [gnutls_pubkey_export_dsa_raw], page 535 (gnutls_pubkey_t key,
gnutls_datum_t * p, gnutls_datum_t * q, gnutls_datum_t * g, gnutls_datum_t * y)
int [gnutls_pubkey_export_ecc_raw], page 535 (gnutls_pubkey_t key,
gnutls_ecc_curve_t * curve, gnutls_datum_t * x, gnutls_datum_t * y)
int [gnutls_pubkey_export_ecc_x962], page 535 (gnutls_pubkey_t key,
gnutls_datum_t * parameters, gnutls_datum_t * ecpoint)
```

### 5.1.2 Private keys

An abstract `gnutls_privkey_t` can be initialized using the functions below. It can be imported through an existing structure like `gnutls_x509_privkey_t`, but unlike public keys it cannot be exported. That is to allow abstraction over keys stored in hardware that makes available only operations.

```
int [gnutls_privkey_import_x509], page 531 (gnutls_privkey_t pkey,
gnutls_x509_privkey_t key, unsigned int flags)
int [gnutls_privkey_import_openpgp], page 528 (gnutls_privkey_t pkey,
gnutls_openpgp_privkey_t key, unsigned int flags)
int [gnutls_privkey_import_pkcs11], page 529 (gnutls_privkey_t pkey,
gnutls_pkcs11_privkey_t key, unsigned int flags)
```

Other helper functions that allow directly importing from raw X.509 or OpenPGP structures are shown below. Again, as with public keys, private keys can be imported from a hardware module using URLs.

```
int [gnutls_privkey_import_x509_raw], page 531 (gnutls_privkey_t pkey, const
gnutls_datum_t * data, gnutls_x509_crt_fmt_t format, const char * password,
unsigned int flags)
int [gnutls_privkey_import_openpgp_raw], page 528 (gnutls_privkey_t pkey,
const gnutls_datum_t * data, gnutls_openpgp_crt_fmt_t format, const
gnutls_openpgp_keyid_t keyid, const char * password)
```

```
int gnutls_privkey_import_url (gnutls_privkey_t key, const char *      [Function]
url, unsigned int flags)
```

*key*: A key of type `gnutls_privkey_t`

*url*: A PKCS 11 url

*flags*: should be zero

This function will import a PKCS11 or TPM URL as a private key. The supported URL types can be checked using `gnutls_url_is_supported()`.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.1.0

```
int [gnutls_privkey_get_pk_algorithm], page 526 (gnutls_privkey_t key,
unsigned int * bits)
gnutls_privkey_type_t [gnutls_privkey_get_type], page 526 (gnutls_privkey_t
key)
int [gnutls_privkey_status], page 533 (gnutls_privkey_t key)
```

In order to support cryptographic operations using an external API, the following function is provided. This allows for a simple extensibility API without resorting to PKCS #11.

```
int gnutls_privkey_import_ext2 (gnutls_privkey_t pkey,                  [Function]
gnutls_pk_algorithm_t pk, void * userdata, gnutls_privkey_sign_func
sign_func, gnutls_privkey_decrypt_func decrypt_func,
gnutls_privkey_deinit_func deinit_func, unsigned int flags)
pkey: The private key
```

*pk*: The public key algorithm

*userdata*: private data to be provided to the callbacks

*sign\_func*: callback for signature operations

*decrypt\_func*: callback for decryption operations

*deinit\_func*: a deinitialization function

*flags*: Flags for the import

This function will associate the given callbacks with the `gnutls_privkey_t` structure. At least one of the two callbacks must be non-null. If a deinitialization function is provided then flags is assumed to contain `GNUTLS_PRIVKEY_IMPORT_AUTO_RELEASE`.

Note that the signing function is supposed to "raw" sign data, i.e., without any hashing or preprocessing. In case of RSA the `DigestInfo` will be provided, and the signing function is expected to do the PKCS 1 1.5 padding and the exponentiation.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.1

### 5.1.3 Operations

The abstract key types can be used to access signing and signature verification operations with the underlying keys.

```
int gnutls_pubkey_verify_data2 (gnutls_pubkey_t pubkey,           [Function]
                               gnutls_sign_algorithm_t algo, unsigned int flags, const gnutls_datum_t *
                               data, const gnutls_datum_t * signature)
```

*pubkey*: Holds the public key

*algo*: The signature algorithm used

*flags*: Zero or one of `gnutls_pubkey_flags_t`

*data*: holds the signed data

*signature*: contains the signature

This function will verify the given signed data, using the parameters from the certificate.

**Returns:** In case of a verification failure `GNUTLS_E_PK_SIG_VERIFY_FAILED` is returned, and zero or positive code on success.

**Since:** 3.0

```
int gnutls_pubkey_verify_hash2 (gnutls_pubkey_t key,           [Function]
                                gnutls_sign_algorithm_t algo, unsigned int flags, const gnutls_datum_t *
                                hash, const gnutls_datum_t * signature)
```

*key*: Holds the public key

*algo*: The signature algorithm used

*flags*: Zero or one of `gnutls_pubkey_flags_t`

*hash*: holds the hash digest to be verified

*signature*: contains the signature

This function will verify the given signed digest, using the parameters from the public key. Note that unlike `gnutls_privkey_sign_hash()`, this function accepts a signature algorithm instead of a digest algorithm. You can use `gnutls_pk_to_sign()` to get the appropriate value.

**Returns:** In case of a verification failure `GNUTLS_E_PK_SIG_VERIFY_FAILED` is returned, and zero or positive code on success.

**Since:** 3.0

```
int gnutls_pubkey_encrypt_data (gnutls_pubkey_t key, unsigned int flags, const gnutls_datum_t * plaintext, gnutls_datum_t * ciphertext) [Function]
```

*key*: Holds the public key

*flags*: should be 0 for now

*plaintext*: The data to be encrypted

*ciphertext*: contains the encrypted data

This function will encrypt the given data, using the public key.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.0

```
int gnutls_privkey_sign_data (gnutls_privkey_t signer, gnutls_digest_algorithm_t hash, unsigned int flags, const gnutls_datum_t * data, gnutls_datum_t * signature) [Function]
```

*signer*: Holds the key

*hash*: should be a digest algorithm

*flags*: Zero or one of `gnutls_privkey_flags_t`

*data*: holds the data to be signed

*signature*: will contain the signature allocate with `gnutls_malloc()`

This function will sign the given data using a signature algorithm supported by the private key. Signature algorithms are always used together with a hash functions. Different hash functions may be used for the RSA algorithm, but only the SHA family for the DSA keys.

You may use `gnutls_pubkey_get_preferred_hash_algorithm()` to determine the hash algorithm.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

```
int gnutls_privkey_sign_hash (gnutls_privkey_t signer, gnutls_digest_algorithm_t hash_algo, unsigned int flags, const gnutls_datum_t * hash_data, gnutls_datum_t * signature) [Function]
```

*signer*: Holds the signer's key

*hash\_algo*: The hash algorithm used

*flags*: Zero or one of `gnutls_privkey_flags_t`

*hash\_data*: holds the data to be signed

*signature*: will contain newly allocated signature

This function will sign the given hashed data using a signature algorithm supported by the private key. Signature algorithms are always used together with a hash functions. Different hash functions may be used for the RSA algorithm, but only SHA-XXX for the DSA keys.

You may use `gnutls_pubkey_get_preferred_hash_algorithm()` to determine the hash algorithm.

Note that if `GNUTLS_PRIVKEY_SIGN_FLAG_TLS1_RSA` flag is specified this function will ignore `hash_algo` and perform a raw PKCS1 signature.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

```
int gnutls_privkey_decrypt_data (gnutls_privkey_t key, unsigned [Function]
                                int flags, const gnutls_datum_t * ciphertext, gnutls_datum_t *
                                plaintext)
```

*key*: Holds the key

*flags*: zero for now

*ciphertext*: holds the data to be decrypted

*plaintext*: will contain the decrypted data, allocated with `gnutls_malloc()`

This function will decrypt the given data using the algorithm supported by the private key.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

Signing existing structures, such as certificates, CRLs, or certificate requests, as well as associating public keys with structures is also possible using the key abstractions.

```
int gnutls_x509_crq_set_pubkey (gnutls_x509_crq_t crq, [Function]
                                gnutls_pubkey_t key)
```

*crq*: should contain a `gnutls_x509_crq_t` structure

*key*: holds a public key

This function will set the public parameters from the given public key to the request.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

```
int gnutls_x509_cert_set_pubkey (gnutls_x509_cert_t crt, [Function]
                                 gnutls_pubkey_t key)
```

*crt*: should contain a `gnutls_x509_cert_t` structure

*key*: holds a public key

This function will set the public parameters from the given public key to the request.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

```
int [gnutls_x509_cert_privkey_sign], page 547 (gnutls_x509_cert_t crt,
gnutls_x509_cert_t issuer, gnutls_privkey_t issuer_key,
gnutls_digest_algorithm_t dig, unsigned int flags)
int [gnutls_x509_crl_privkey_sign], page 546 (gnutls_x509_crl_t crl,
gnutls_x509_cert_t issuer, gnutls_privkey_t issuer_key,
gnutls_digest_algorithm_t dig, unsigned int flags)
int [gnutls_x509_crq_privkey_sign], page 546 (gnutls_x509_crq_t crq,
gnutls_privkey_t key, gnutls_digest_algorithm_t dig, unsigned int flags)
```

## 5.2 Smart cards and HSMs

In this section we present the smart-card and hardware security module (HSM) support in GnuTLS using PKCS #11 [PKCS11]. Hardware security modules and smart cards provide a way to store private keys and perform operations on them without exposing them. This decouples cryptographic keys from the applications that use them and provide an additional security layer against cryptographic key extraction. Since this can also be achieved in software components such as in Gnome keyring, we will use the term security module to describe any cryptographic key separation subsystem.

PKCS #11 is plugin API allowing applications to access cryptographic operations on a security module, as well as to objects residing on it. PKCS #11 modules exist for hardware tokens such as smart cards<sup>1</sup>, cryptographic tokens, as well as for software modules like Gnome Keyring. The objects residing on a security module may be certificates, public keys, private keys or secret keys. Of those certificates and public/private key pairs can be used with GnuTLS. PKCS #11's main advantage is that it allows operations on private key objects such as decryption and signing without exposing the key. In GnuTLS the PKCS #11 functionality is available in `gnutls/pkcs11.h`.

Moreover PKCS #11 can be (ab)used to allow all applications in the same operating system to access shared cryptographic keys and certificates in a uniform way, as in [Figure 5.1](#). That way applications could load their trusted certificate list, as well as user certificates from a common PKCS #11 module. Such a provider is the p11-kit trust storage module<sup>2</sup>.

<sup>1</sup> <http://www.opensc-project.org>

<sup>2</sup> <http://p11-glue.freedesktop.org/trust-module.html>



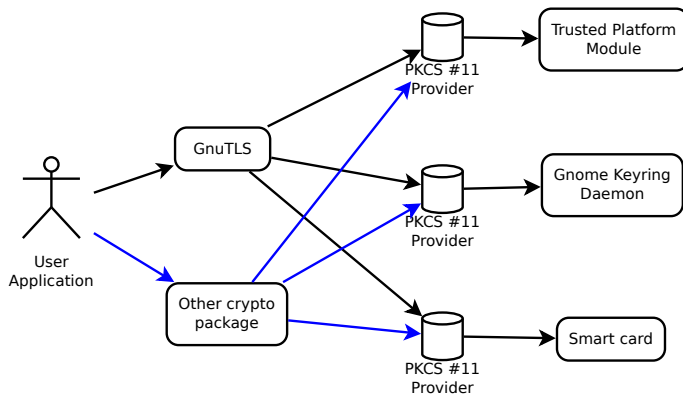


Figure 5.1: PKCS #11 module usage.

### 5.2.1 Initialization

To allow all GnuTLS applications to transparently access smart cards and tokens, PKCS #11 is automatically initialized during the global initialization (see [\[gnutls-global-init\]](#), page 307). The initialization function, to select which modules to load reads certain module configuration files. Those are stored in `/etc/pkcs11/modules/` and are the configuration files of `p11-kit`<sup>3</sup>. For example a file that will load the OpenSC module, could be named `/etc/pkcs11/modules/opensc.module` and contain the following:

```
module: /usr/lib/opensc-pkcs11.so
```

If you use these configuration files, then there is no need for other initialization in GnuTLS, except for the PIN and token functions (see next section). In several cases, however, it is desirable to limit badly behaving modules (e.g., modules that add an unacceptable delay on initialization) to single applications. That can be done using the “enable-in:” option followed by the base name of applications that this module should be used.

In all cases, you can also manually initialize the PKCS #11 subsystem if the default settings are not desirable. To completely disable PKCS #11 support you need to call [\[gnutls-pkcs11-init\]](#), page 507 with the flag `GNUTLS_PKCS11_FLAG_MANUAL` prior to [\[gnutls-global-init\]](#), page 307.

```
int gnutls_pkcs11_init (unsigned int flags, const char *
    deprecated_config_file) [Function]
```

*flags*: An ORed sequence of `GNUTLS_PKCS11_FLAG_*`

*deprecated\_config\_file*: either NULL or the location of a deprecated configuration file

This function will initialize the PKCS 11 subsystem in gnutls. It will read configuration files if `GNUTLS_PKCS11_FLAG_AUTO` is used or allow you to independently load PKCS 11 modules using `gnutls_pkcs11_add_provider()` if `GNUTLS_PKCS11_FLAG_MANUAL` is specified.

Normally you don’t need to call this function since it is being called when the first PKCS 11 operation is requested using the `GNUTLS_PKCS11_FLAG_AUTO` flag. If an-

<sup>3</sup> <http://p11-glue.freedesktop.org/>

other flags are required then it must be called independently prior to any PKCS 11 operation.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

Note that PKCS #11 modules must be reinitialized on the child processes after a `fork`. In older versions of GnuTLS it was required to call `[gnutls_pkcs11_reinit]`, page 515; since 3.3.0 this is no longer required, as reinitialization occurs automatically.

### 5.2.2 Accessing objects that require a PIN

Objects stored in token such as a private keys are typically protected from access by a PIN or password. This PIN may be required to either read the object (if allowed) or to perform operations with it. To allow obtaining the PIN when accessing a protected object, as well as probe the user to insert the token the following functions allow to set a callback.

```
void [gnutls_pkcs11_set_token_function], page 515
(gnutls_pkcs11_token_callback_t fn, void * userdata)
void [gnutls_pkcs11_set_pin_function], page 515 (gnutls_pin_callback_t fn,
void * userdata)
int [gnutls_pkcs11_add_provider], page 504 (const char * name, const char *
params)
gnutls_pin_callback_t [gnutls_pkcs11_get_pin_function], page 506 (void **
userdata)
```

The callback is of type `gnutls_pin_callback_t` and will have as input the provided user-data, the PIN attempt number, a URL describing the token, a label describing the object and flags. The PIN must be at most of `pin_max` size and must be copied to pin variable. The function must return 0 on success or a negative error code otherwise.

```
typedef int (*gnutls_pin_callback_t) (void *userdata, int attempt,
                                     const char *token_url,
                                     const char *token_label,
                                     unsigned int flags,
                                     char *pin, size_t pin_max);
```

The flags are of `gnutls_pin_flag_t` type and are explained below.

GNUTLS_PIN_USER	The PIN for the user.
GNUTLS_PIN_SO	The PIN for the security officer (admin).
GNUTLS_PIN_FINAL_TRY	This is the final try before blocking.
GNUTLS_PIN_COUNT_LOW	Few tries remain before token blocks.
GNUTLS_PIN_CONTEXT_SPECIFIC	The PIN is for a specific action and key like signing.
GNUTLS_PIN_WRONG	Last given PIN was not correct.

Figure 5.2: The `gnutls_pin_flag_t` enumeration.

Note that due to limitations of PKCS #11 there are issues when multiple libraries are sharing a module. To avoid this problem GnuTLS uses p11-kit that provides a middleware to control access to resources over the multiple users.

To avoid conflicts with multiple registered callbacks for PIN functions, [\[gnutls\\_pkcs11\\_get\\_pin\\_function\]](#), page 506 may be used to check for any previously set functions. In addition context specific PIN functions are allowed, e.g., by using functions below.

```
void [gnutls_certificate_set_pin_function], page 283
(gnutls_certificate_credentials_t cred, gnutls_pin_callback_t fn, void *
userdata)
void [gnutls_pubkey_set_pin_function], page 544 (gnutls_pubkey_t key,
gnutls_pin_callback_t fn, void * userdata)
void [gnutls_privkey_set_pin_function], page 532 (gnutls_privkey_t key,
gnutls_pin_callback_t fn, void * userdata)
void [gnutls_pkcs11_obj_set_pin_function], page 511 (gnutls_pkcs11_obj_t
obj, gnutls_pin_callback_t fn, void * userdata)
void [gnutls_x509_cert_set_pin_function], page 431 (gnutls_x509_cert_t crt,
gnutls_pin_callback_t fn, void * userdata)
```

### 5.2.3 Reading objects

All PKCS #11 objects are referenced by GnuTLS functions by URLs as described in [\[PKCS11URI\]](#). This allows for a consistent naming of objects across systems and applications in the same system. For example a public key on a smart card may be referenced as:

```
pkcs11:token=Nikos;serial=307521161601031;model=PKCS%2315; \
manufacturer=EnterSafe;object=test1;objecttype=public;\
id=32f153f3e37990b08624141077ca5dec2d15faed
```

while the smart card itself can be referenced as:

```
pkcs11:token=Nikos;serial=307521161601031;model=PKCS%2315;manufacturer=EnterSafe
```

Objects stored in a PKCS #11 token can be extracted if they are not marked as sensitive. Usually only private keys are marked as sensitive and cannot be extracted, while certificates and other data can be retrieved. The functions that can be used to access objects are shown below.

```
int [gnutls_pkcs11_obj_import_url], page 510 (gnutls_pkcs11_obj_t obj, const
char * url, unsigned int flags)
int [gnutls_pkcs11_obj_export_url], page 508 (gnutls_pkcs11_obj_t obj,
gnutls_pkcs11_url_type_t detailed, char ** url)
int gnutls_pkcs11_obj_get_info (gnutls_pkcs11_obj_t obj, [Function]
gnutls_pkcs11_obj_info_t itype, void * output, size_t * output_size)
obj: should contain a gnutls_pkcs11_obj_t structure
itype: Denotes the type of information requested
output: where output will be stored
output_size: contains the maximum size of the output and will be overwritten with
actual
```

This function will return information about the PKCS11 certificate such as the label, id as well as token information where the key is stored. When output is text it returns null terminated string although `output_size` contains the size of the actual data only.

**Returns:** GNUTLS\_E\_SUCCESS (0) on success or a negative error code on error.

**Since:** 2.12.0

```
int [gnutls_x509_cert_import_pkcs11], page 518 (gnutls_x509_cert_t crt,
gnutls_pkcs11_obj_t pkcs11_crt)
int [gnutls_x509_cert_import_pkcs11_url], page 518 (gnutls_x509_cert_t crt,
const char * url, unsigned int flags)
int [gnutls_x509_cert_list_import_pkcs11], page 518 (gnutls_x509_cert_t *
certs, unsigned int cert_max, gnutls_pkcs11_obj_t * const objs, unsigned int
flags)
```

Properties of the physical token can also be accessed and altered with GnuTLS. For example data in a token can be erased (initialized), PIN can be altered, etc.

```
int [gnutls_pkcs11_token_init], page 517 (const char * token_url, const char *
so_pin, const char * label)
int [gnutls_pkcs11_token_get_url], page 516 (unsigned int seq,
gnutls_pkcs11_url_type_t detailed, char ** url)
int [gnutls_pkcs11_token_get_info], page 516 (const char * url,
gnutls_pkcs11_token_info_t ttype, void * output, size_t * output_size)
int [gnutls_pkcs11_token_get_flags], page 515 (const char * url, unsigned int
* flags)
int [gnutls_pkcs11_token_set_pin], page 517 (const char * token_url, const
char * oldpin, const char * newpin, unsigned int flags)
```

The following examples demonstrate the usage of the API. The first example will list all available PKCS #11 tokens in a system and the latter will list all certificates in a token that have a corresponding private key.

```

    int i;
    char* url;

    gnutls_global_init();

    for (i=0;;i++)
    {
        ret = gnutls_pkcs11_token_get_url(i, &url);
        if (ret == GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE)
            break;

        if (ret < 0)
            exit(1);

        fprintf(stdout, "Token[%d]: URL: %s\n", i, url);
        gnutls_free(url);
    }
    gnutls_global_deinit();

/* This example code is placed in the public domain. */

#include <config.h>
#include <gnutls/gnutls.h>
#include <gnutls/pkcs11.h>
#include <stdio.h>
#include <stdlib.h>

#define URL "pkcs11:URL"

int main(int argc, char **argv)
{
    gnutls_pkcs11_obj_t *obj_list;
    gnutls_x509_crt_t xcrt;
    unsigned int obj_list_size = 0;
    gnutls_datum_t cinfo;
    int ret;
    unsigned int i;

    obj_list_size = 0;
    ret = gnutls_pkcs11_obj_list_import_url(NULL, &obj_list_size, URL,
                                           GNUTLS_PKCS11_OBJ_ATTR_CERT_WITH_PRIVATE_KEY,
                                           0);

    if (ret < 0 && ret != GNUTLS_E_SHORT_MEMORY_BUFFER)
        return -1;

/* no error checking from now on */
    obj_list = malloc(sizeof(*obj_list) * obj_list_size);

```

```

        gnutls_pkcs11_obj_list_import_url(obj_list, &obj_list_size, URL,
                                          GNUTLS_PKCS11_OBJ_ATTR_CERT_WITH_PRIVKEY,
                                          0);

/* now all certificates are in obj_list */
for (i = 0; i < obj_list_size; i++) {

    gnutls_x509_crt_init(&xcrt);

    gnutls_x509_crt_import_pkcs11(xcrt, obj_list[i]);

    gnutls_x509_crt_print(xcrt, GNUTLS_CERT_PRINT_FULL, &cinfo);

    fprintf(stdout, "cert[%d]:\n %s\n\n", i, cinfo.data);

    gnutls_free(cinfo.data);
    gnutls_x509_crt_deinit(xcrt);
}

return 0;
}

```

### 5.2.4 Writing objects

With GnuTLS you can copy existing private keys and certificates to a token. Note that when copying private keys it is recommended to mark them as sensitive using the `GNUTLS_PKCS11_OBJ_FLAG_MARK_SENSITIVE` to prevent its extraction. An object can be marked as private using the flag `GNUTLS_PKCS11_OBJ_FLAG_MARK_PRIVATE`, to require PIN to be entered before accessing the object (for operations or otherwise).

```

int gnutls_pkcs11_copy_x509_privkey (const char * token_url,      [Function]
                                     gnutls_x509_privkey_t key, const char * label, unsigned int key_usage,
                                     unsigned int flags)

```

*token\_url*: A PKCS 11 URL specifying a token

*key*: A private key

*label*: A name to be used for the stored data

*key\_usage*: One of `GNUTLS_KEY_*`

*flags*: One of `GNUTLS_PKCS11_OBJ_*` flags

This function will copy a private key into a PKCS 11 token specified by a URL. It is highly recommended flags to contain `GNUTLS_PKCS11_OBJ_FLAG_MARK_SENSITIVE` unless there is a strong reason not to.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

```
int gnutls_pkcs11_copy_x509_cert (const char * token_url,          [Function]
                                gnutls_x509_cert_t crt, const char * label, unsigned int flags)
```

*token\_url*: A PKCS 11 URL specifying a token

*crt*: A certificate

*label*: A name to be used for the stored data

*flags*: One of GNUTLS\_PKCS11\_OBJ\_FLAG\_\*

This function will copy a certificate into a PKCS 11 token specified by a URL. The certificate can be marked as trusted or not.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

```
int gnutls_pkcs11_delete_url (const char * object_url, unsigned    [Function]
                             int flags)
```

*object\_url*: The URL of the object to delete.

*flags*: One of GNUTLS\_PKCS11\_OBJ\_\* flags

This function will delete objects matching the given URL. Note that not all tokens support the delete operation.

**Returns:** On success, the number of objects deleted is returned, otherwise a negative error value.

**Since:** 2.12.0

### 5.2.5 Using a PKCS #11 token with TLS

It is possible to use a PKCS #11 token to a TLS session, as shown in [\[ex-pkcs11-client\]](#), [page 173](#). In addition the following functions can be used to load PKCS #11 key and certificates by specifying a PKCS #11 URL instead of a filename.

```
int [gnutls_certificate_set_x509_trust_file], page 290
```

```
(gnutls_certificate_credentials_t cred, const char * cafile,
 gnutls_x509_cert_fmt_t type)
```

```
int [gnutls_certificate_set_x509_key_file2], page 286
```

```
(gnutls_certificate_credentials_t res, const char * certfile, const char *
 keyfile, gnutls_x509_cert_fmt_t type, const char * pass, unsigned int flags)
```

```
int gnutls_certificate_set_x509_system_trust                      [Function]
    (gnutls_certificate_credentials_t cred)
```

*cred*: is a gnutls\_certificate\_credentials\_t structure.

This function adds the system's default trusted CAs in order to verify client or server certificates.

In the case the system is currently unsupported GNUTLS\_E\_UNIMPLEMENTED\_FEATURE is returned.

**Returns:** the number of certificates processed or a negative error code on error.

**Since:** 3.0.20

### 5.2.6 Invoking p11tool

Program that allows operations on PKCS #11 smart cards and security modules.

To use PKCS #11 tokens with GnuTLS the p11-kit configuration files need to be setup. That is create a .module file in /etc/pkcs11/modules with the contents 'module: /path/to/pkcs11.so'. Alternatively the configuration file /etc/gnutls/pkcs11.conf has to exist and contain a number of lines of the form 'load=/usr/lib/opensc-pkcs11.so'.

You can provide the PIN to be used for the PKCS #11 operations with the environment variables GNUTLS\_PIN and GNUTLS\_SO\_PIN.

This section was generated by **AutoGen**, using the **agtexi-cmd** template and the option descriptions for the **p11tool** program. This software is released under the GNU General Public License, version 3 or later.

### 5.2.7 p11tool help/usage (--help)

This is the automatically generated usage text for p11tool.

The text printed is the same whether selected with the **help** option (**--help**) or the **more-help** option (**--more-help**). **more-help** will print the usage text by passing it through a pager program. **more-help** is disabled on platforms without a working **fork(2)** function. The **PAGER** environment variable is used to select the program, defaulting to **more**. Both will exit with a status code of 0.

p11tool is unavailable - no --help

### 5.2.8 debug option (-d)

This is the “enable debugging” option. This option takes a number argument. Specifies the debug level.

### 5.2.9 export-chain option

This is the “export the certificate specified by the url and its chain of trust” option. Exports the certificate specified by the URL and generates its chain of trust based on the stored certificates in the module.

### 5.2.10 list-all-privkeys option

This is the “list all available private keys in a token” option. Lists all the private keys in a token that match the specified URL.

### 5.2.11 list-privkeys option

This is an alias for the **list-all-privkeys** option, see [\[p11tool list-all-privkeys\]](#), page 96.

### 5.2.12 list-keys option

This is an alias for the **list-all-privkeys** option, see [\[p11tool list-all-privkeys\]](#), page 96.

### 5.2.13 write option

This is the “writes the loaded objects to a pkcs #11 token” option. It can be used to write private keys, certificates or secret keys to a token.



#### 5.2.14 generate-random option

This is the “generate random data” option. This option takes a number argument. Asks the token to generate a number of bytes of random bytes.

#### 5.2.15 generate-rsa option

This is the “generate an rsa private-public key pair” option. Generates an RSA private-public key pair on the specified token.

#### 5.2.16 generate-dsa option

This is the “generate a dsa private-public key pair” option. Generates a DSA private-public key pair on the specified token.

#### 5.2.17 generate-ecdsa option

This is the “generate an ecdsa private-public key pair” option. Generates an ECDSA private-public key pair on the specified token.

#### 5.2.18 export-pubkey option

This is the “export the public key for a private key” option. Exports the public key for the specified private key

#### 5.2.19 mark-wrap option

This is the “marks the generated key to be a wrapping key” option.

This option has some usage constraints. It:

- can be disabled with `-no-mark-wrap`.

Marks the generated key with the CKA\_WRAP flag.

#### 5.2.20 mark-trusted option

This is the “marks the object to be written as trusted” option.

This option has some usage constraints. It:

- can be disabled with `-no-mark-trusted`.

Marks the object to be generated/copied with the CKA\_TRUST flag.

#### 5.2.21 mark-ca option

This is the “marks the object to be written as a ca” option.

This option has some usage constraints. It:

- can be disabled with `-no-mark-ca`.

Marks the object to be generated/copied with the CKA\_CERTIFICATE\_CATEGORY as CA.

### 5.2.22 mark-private option

This is the “marks the object to be written as private” option.

This option has some usage constraints. It:

- can be disabled with `-no-mark-private`.
- It is enabled by default.

Marks the object to be generated/copied with the CKA\_PRIVATE flag. The written object will require a PIN to be used.

### 5.2.23 trusted option

This is an alias for the `mark-trusted` option, see [\[p11tool mark-trusted\]](#), page 97.

### 5.2.24 ca option

This is an alias for the `mark-ca` option, see [\[p11tool mark-ca\]](#), page 97.

### 5.2.25 private option

This is an alias for the `mark-private` option, see [\[p11tool mark-private\]](#), page 97.

### 5.2.26 so-login option

This is the “force security officer login to token” option.

This option has some usage constraints. It:

- can be disabled with `-no-so-login`.

Forces login to the token as security officer (admin).

### 5.2.27 admin-login option

This is an alias for the `so-login` option, see [\[p11tool so-login\]](#), page 98.

### 5.2.28 curve option

This is the “specify the curve used for ec key generation” option. This option takes a string argument. Supported values are `secp192r1`, `secp224r1`, `secp256r1`, `secp384r1` and `secp521r1`.

### 5.2.29 sec-param option

This is the “specify the security level” option. This option takes a string argument `Security parameter`. This is alternative to the `bits` option. Available options are `[low, legacy, medium, high, ultra]`.

### 5.2.30 inder option

This is the “use der/raw format for input” option.

This option has some usage constraints. It:

- can be disabled with `-no-inder`.

Use DER/RAW format for input certificates and private keys.

### 5.2.31 inraw option

This is an alias for the `inder` option, see [\[p11tool inder\]](#), page 98.

### 5.2.32 outder option

This is the “use der format for output certificates, private keys, and dh parameters” option.

This option has some usage constraints. It:

- can be disabled with `-no-outder`.

The output will be in DER or RAW format.

### 5.2.33 outraw option

This is an alias for the `outder` option, see [\[p11tool outder\]](#), page 99.

### 5.2.34 set-pin option

This is the “specify the pin to use on token initialization” option. This option takes a string argument. Alternatively the `GNUTLS_PIN` environment variable may be used.

### 5.2.35 set-so-pin option

This is the “specify the security officer’s pin to use on token initialization” option. This option takes a string argument. Alternatively the `GNUTLS_SO_PIN` environment variable may be used.

### 5.2.36 provider option

This is the “specify the pkcs #11 provider library” option. This option takes a file argument. This will override the default options in `/etc/gnutls/pkcs11.conf`

### 5.2.37 p11tool exit status

One of the following exit values will be returned:

‘0 (EXIT\_SUCCESS)’

Successful program execution.

‘1 (EXIT\_FAILURE)’

The operation failed or the command syntax was not valid.

### 5.2.38 p11tool See Also

`certtool (1)`

### 5.2.39 p11tool Examples

To view all tokens in your system use:

```
$ p11tool --list-tokens
```

To view all objects in a token use:

```
$ p11tool --login --list-all "pkcs11:TOKEN-URL"
```

To store a private key and a certificate in a token run:

```
$ p11tool --login --write "pkcs11:URL" --load-privkey key.pem \
--label "Mykey"
$ p11tool --login --write "pkcs11:URL" --load-certificate cert.pem \
--label "Mykey"
```

Note that some tokens require the same label to be used for the certificate and its corresponding private key.

To generate an RSA private key inside the token use:

```
$ p11tool --login --generate-rsa --bits 1024 --label "MyNewKey" \
--outfile MyNewKey.pub "pkcs11:TOKEN-URL"
```

The bits parameter in the above example is explicitly set because some tokens only support limited choices in the bit length. The output file is the corresponding public key. This key can be used to generate a certificate request with certtool.

```
certtool --generate-request --load-privkey "pkcs11:KEY-URL" \
--load-pubkey MyNewKey.pub --outfile request.pem
```

## 5.3 Trusted Platform Module (TPM)

In this section we present the Trusted Platform Module (TPM) support in GnuTLS.

There was a big hype when the TPM chip was introduced into computers. Briefly it is a co-processor in your PC that allows it to perform calculations independently of the main processor. This has good and bad side-effects. In this section we focus on the good ones; these are the fact that you can use the TPM chip to perform cryptographic operations on keys stored in it, without accessing them. That is very similar to the operation of a PKCS #11 smart card. The chip allows for storage and usage of RSA keys, but has quite some operational differences from PKCS #11 module, and thus require different handling. The basic TPM operations supported and used by GnuTLS, are key generation and signing.

The next sections assume that the TPM chip in the system is already initialized and in a operational state.

In GnuTLS the TPM functionality is available in `gnutls/tpm.h`.

### 5.3.1 Keys in TPM

The RSA keys in the TPM module may either be stored in a flash memory within TPM or stored in a file in disk. In the former case the key can provide operations as with PKCS #11 and is identified by a URL. The URL is described in [TPMURI] and is of the following form.

```
tpmkey:uuid=42309df8-d101-11e1-a89a-97bb33c23ad1;storage=user
```

It consists from a unique identifier of the key as well as the part of the flash memory the key is stored at. The two options for the storage field are 'user' and 'system'. The user keys are typically only available to the generating user and the system keys to all users. The stored in TPM keys are called registered keys.

The keys that are stored in the disk are exported from the TPM but in an encrypted form. To access them two passwords are required. The first is the TPM Storage Root Key (SRK), and the other is a key-specific password. Also those keys are identified by a URL of the form:

`tpmkey:file=/path/to/file`

When objects require a PIN to be accessed the same callbacks as with PKCS #11 objects are expected (see [Section 5.2.2 \[Accessing objects that require a PIN\]](#), page 90). Note that the PIN function may be called multiple times to unlock the SRK and the specific key in use. The label in the key function will then be set to ‘SRK’ when unlocking the SRK key, or to ‘TPM’ when unlocking any other key.

### 5.3.2 Key generation

All keys used by the TPM must be generated by the TPM. This can be done using [\[gnutls-tpm-privkey-generate\]](#), page 520.

```
int gnutls_tpm_privkey_generate (gnutls_pk_algorithm_t pk,          [Function]
                                unsigned int bits, const char * srk_password, const char * key_password,
                                gnutls_tpmkey_fmt_t format, gnutls_x509_crt_fmt_t pub_format,
                                gnutls_datum_t * privkey, gnutls_datum_t * pubkey, unsigned int flags)
```

*pk*: the public key algorithm

*bits*: the security bits

*srk\_password*: a password to protect the exported key (optional)

*key\_password*: the password for the TPM (optional)

*format*: the format of the private key

*pub\_format*: the format of the public key

*privkey*: the generated key

*pubkey*: the corresponding public key (may be null)

*flags*: should be a list of GNUTLS\_TPM\_\* flags

This function will generate a private key in the TPM chip. The private key will be generated within the chip and will be exported in a wrapped with TPM’s master key form. Furthermore the wrapped key can be protected with the provided `password`.

Note that bits in TPM is quantized value. If the input value is not one of the allowed values, then it will be quantized to one of 512, 1024, 2048, 4096, 8192 and 16384.

Allowed flags are:

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.1.0

```
int [gnutls_tpm_get_registered], page 519 (gnutls_tpm_key_list_t * list)
void [gnutls_tpm_key_list_deinit], page 519 (gnutls_tpm_key_list_t list)
int [gnutls_tpm_key_list_get_url], page 519 (gnutls_tpm_key_list_t list,
unsigned int idx, char ** url, unsigned int flags)
```

```
int gnutls_tpm_privkey_delete (const char * url, const char *      [Function]
                                srk_password)
```

*url*: the URL describing the key

*srk\_password*: a password for the SRK key

This function will unregister the private key from the TPM chip.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.1.0

### 5.3.3 Using keys

#### Importing keys

The TPM keys can be used directly by the abstract key types and do not require any special structures. Moreover functions like [\[gnutls\\_certificate\\_set\\_x509\\_key\\_file2\]](#), [page 286](#) can access TPM URLs.

`int [gnutls_privkey_import_tpm_raw]`, [page 530](#) (`gnutls_privkey_t pkey`, `const gnutls_datum_t * fdata`, `gnutls_tpmkey_fmt_t format`, `const char * srk_password`, `const char * key_password`, `unsigned int flags`)

`int [gnutls_pubkey_import_tpm_raw]`, [page 541](#) (`gnutls_pubkey_t pkey`, `const gnutls_datum_t * fdata`, `gnutls_tpmkey_fmt_t format`, `const char * srk_password`, `unsigned int flags`)

`int gnutls_privkey_import_tpm_url` (`gnutls_privkey_t pkey`, `const char * url`, `const char * srk_password`, `const char * key_password`, `unsigned int flags`) [Function]

*pkey*: The private key

*url*: The URL of the TPM key to be imported

*srk\_password*: The password for the SRK key (optional)

*key\_password*: A password for the key (optional)

*flags*: One of the GNUTLS\_PRIVKEY\_\* flags

This function will import the given private key to the abstract `gnutls_privkey_t` structure.

Note that unless GNUTLS\_PRIVKEY\_DISABLE\_CALLBACKS is specified, if incorrect (or NULL) passwords are given the PKCS11 callback functions will be used to obtain the correct passwords. Otherwise if the SRK password is wrong GNUTLS\_E\_TPM\_SRK\_PASSWORD\_ERROR is returned and if the key password is wrong or not provided then GNUTLS\_E\_TPM\_KEY\_PASSWORD\_ERROR is returned.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.1.0

`int gnutls_pubkey_import_tpm_url` (`gnutls_pubkey_t pkey`, `const char * url`, `const char * srk_password`, `unsigned int flags`) [Function]

*pkey*: The public key

*url*: The URL of the TPM key to be imported

*srk\_password*: The password for the SRK key (optional)

*flags*: should be zero

This function will import the given private key to the abstract `gnutls_privkey_t` structure.

Note that unless `GNUTLS_PUBKEY_DISABLE_CALLBACKS` is specified, if incorrect (or NULL) passwords are given the PKCS11 callback functions will be used to obtain the correct passwords. Otherwise if the SRK password is wrong `GNUTLS_E_TPM_SRK_PASSWORD_ERROR` is returned.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.1.0

## Listing and deleting keys

The registered keys (that are stored in the TPM) can be listed using one of the following functions. Those keys are unfortunately only identified by their UUID and have no label or other human friendly identifier. Keys can be deleted from permanent storage using [\[gnutls\\_tpm\\_privkey\\_delete\]](#), page 519.

```
int [gnutls_tpm_get_registered], page 519 (gnutls_tpm_key_list_t * list)
void [gnutls_tpm_key_list_deinit], page 519 (gnutls_tpm_key_list_t list)
int [gnutls_tpm_key_list_get_url], page 519 (gnutls_tpm_key_list_t list,
unsigned int idx, char ** url, unsigned int flags)

int gnutls_tpm_privkey_delete (const char * url, const char *          [Function]
    srk_password)
```

*url*: the URL describing the key

*srk\_password*: a password for the SRK key

This function will unregister the private key from the TPM chip.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.1.0

### 5.3.4 Invoking tpmtool

Program that allows handling cryptographic data from the TPM chip.

This section was generated by **AutoGen**, using the `agtexi-cmd` template and the option descriptions for the `tpmtool` program. This software is released under the GNU General Public License, version 3 or later.

#### 5.3.5 tpmtool help/usage (--help)

This is the automatically generated usage text for `tpmtool`.

The text printed is the same whether selected with the `help` option (`--help`) or the `more-help` option (`--more-help`). `more-help` will print the usage text by passing it through a pager program. `more-help` is disabled on platforms without a working `fork(2)` function. The `PAGER` environment variable is used to select the program, defaulting to `more`. Both will exit with a status code of 0.

```
tpmtool is unavailable - no --help
```

#### 5.3.6 debug option (-d)

This is the “enable debugging” option. This option takes a number argument. Specifies the debug level.

### 5.3.7 generate-rsa option

This is the “generate an rsa private-public key pair” option. Generates an RSA private-public key pair in the TPM chip. The key may be stored in filesystem and protected by a PIN, or stored (registered) in the TPM chip flash.

### 5.3.8 user option

This is the “any registered key will be a user key” option.

This option has some usage constraints. It:

- must appear in combination with the following options: register.
- must not appear in combination with any of the following options: system.

The generated key will be stored in a user specific persistent storage.

### 5.3.9 system option

This is the “any registered key will be a system key” option.

This option has some usage constraints. It:

- must appear in combination with the following options: register.
- must not appear in combination with any of the following options: user.

The generated key will be stored in system persistent storage.

### 5.3.10 sec-param option

This is the “specify the security level [low, legacy, medium, high, ultra].” option. This option takes a string argument **Security parameter**. This is alternative to the bits option. Note however that the values allowed by the TPM chip are quantized and given values may be rounded up.

### 5.3.11 inder option

This is the “use the der format for keys.” option.

This option has some usage constraints. It:

- can be disabled with `-no-inder`.

The input files will be assumed to be in the portable DER format of TPM. The default format is a custom format used by various TPM tools

### 5.3.12 outder option

This is the “use der format for output keys” option.

This option has some usage constraints. It:

- can be disabled with `-no-outder`.

The output will be in the TPM portable DER format.



### 5.3.13 tpmtool exit status

One of the following exit values will be returned:

‘0 (EXIT\_SUCCESS)’

Successful program execution.

‘1 (EXIT\_FAILURE)’

The operation failed or the command syntax was not valid.

### 5.3.14 tpmtool See Also

p11tool (1), certtool (1)

### 5.3.15 tpmtool Examples

To generate a key that is to be stored in filesystem use:

```
$ tpmtool --generate-rsa --bits 2048 --outfile tpmkey.pem
```

To generate a key that is to be stored in TPM’s flash use:

```
$ tpmtool --generate-rsa --bits 2048 --register --user
```

To get the public key of a TPM key use:

```
$ tpmtool --pubkey tpmkey:uuid=58ad734b-bde6-45c7-89d8-756a55ad1891;storage=user \  
--outfile pubkey.pem
```

or if the key is stored in the filesystem:

```
$ tpmtool --pubkey tpmkey:file=tmpkey.pem --outfile pubkey.pem
```

To list all keys stored in TPM use:

```
$ tpmtool --list
```

## 6 How to use GnuTLS in applications

### 6.1 Introduction

This chapter tries to explain the basic functionality of the current GnuTLS library. Note that there may be additional functionality not discussed here but included in the library. Checking the header files in `/usr/include/gnutls/` and the manpages is recommended.

#### 6.1.1 General idea

A brief description of how GnuTLS sessions operate is shown at [Figure 6.1](#). This section will become more clear when it is completely read. As shown in the figure, there is a read-only global state that is initialized once by the global initialization function. This global structure, among others, contains the memory allocation functions used, structures needed for the ASN.1 parser and depending on the system's CPU, pointers to hardware accelerated encryption functions. This structure is never modified by any GnuTLS function, except for the deinitialization function which frees all allocated memory and must be called after the program has permanently finished using GnuTLS.

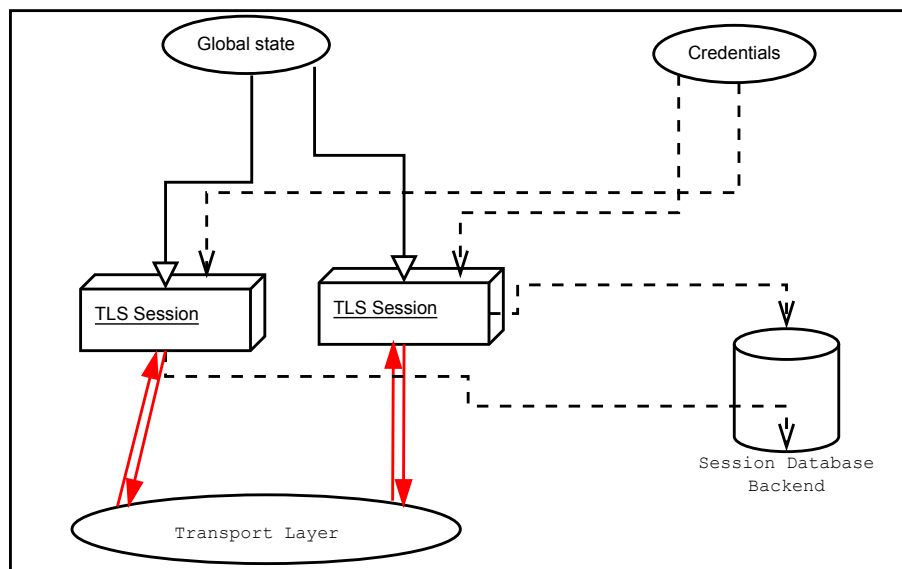


Figure 6.1: High level design of GnuTLS.

The credentials structures are used by the authentication methods, such as certificate authentication. They store certificates, private keys, and other information that is needed to prove the identity to the peer, and/or verify the identity of the peer. The information stored in the credentials structures is initialized once and then can be shared by many TLS sessions.

A GnuTLS session contains all the required state and information to handle one secure connection. The session communicates with the peers using the provided functions of the transport layer. Every session has a unique session ID shared with the peer.

Since TLS sessions can be resumed, servers need a database back-end to hold the session's parameters. Every GnuTLS session after a successful handshake calls the appropriate back-end function (see [\[resume\]](#), page 10) to store the newly negotiated session. The session database is examined by the server just after having received the client hello<sup>1</sup>, and if the session ID sent by the client, matches a stored session, the stored session will be retrieved, and the new session will be a resumed one, and will share the same session ID with the previous one.

### 6.1.2 Error handling

In GnuTLS most functions return an integer type as a result. In almost all cases a zero or a positive number means success, and a negative number indicates failure, or a situation that some action has to be taken. Thus negative error codes may be fatal or not.

Fatal errors terminate the connection immediately and further sends and receives will be disallowed. Such an example is `GNUTLS_E_DECRYPTION_FAILED`. Non-fatal errors may warn about something, i.e., a warning alert was received, or indicate the some action has to be taken. This is the case with the error code `GNUTLS_E_REHANDSHAKE` returned by [\[gnutls\\_record\\_recv\]](#), page 334. This error code indicates that the server requests a re-handshake. The client may ignore this request, or may reply with an alert. You can test if an error code is a fatal one by using the [\[gnutls\\_error\\_is\\_fatal\]](#), page 306. All errors can be converted to a descriptive string using [\[gnutls\\_strerror\]](#), page 355.

If any non fatal errors, that require an action, are to be returned by a function, these error codes will be documented in the function's reference. For example the error codes `GNUTLS_E_WARNING_ALERT_RECEIVED` and `GNUTLS_E_FATAL_ALERT_RECEIVED` that may returned when receiving data, should be handled by notifying the user of the alert (as explained in [Section 6.9 \[Handling alerts\]](#), page 131). See [Appendix C \[Error codes\]](#), page 259, for a description of the available error codes.

### 6.1.3 Common types

All strings that are to provided as input to GnuTLS functions should be in UTF-8 unless otherwise specified. Output strings are also in UTF-8 format unless otherwise specified.

When data of a fixed size are provided to GnuTLS functions then the helper structure `gnutls_datum_t` is often used. Its definition is shown below.

```
typedef struct
{
    unsigned char *data;
    unsigned int size;
} gnutls_datum_t;
```

Other functions that require data for scattered read use a structure similar to `struct iovec` typically used by `readv`. It is shown below.

```
typedef struct
{
    void *iov_base;           /* Starting address */
    size_t iov_len;           /* Number of bytes to transfer */
} giovec_t;
```

<sup>1</sup> The first message in a TLS handshake

### 6.1.4 Debugging and auditing

In many cases things may not go as expected and further information, to assist debugging, from GnuTLS is desired. Those are the cases where the `[gnutls_global_set_log_level]`, page 308 and `[gnutls_global_set_log_function]`, page 308 are to be used. Those will print verbose information on the GnuTLS functions internal flow.

```
void [gnutls_global_set_log_level], page 308 (int level)
void [gnutls_global_set_log_function], page 308 (gnutls_log_func log_func)
```

Alternatively the environment variable `GNUTLS_DEBUG_LEVEL` can be set to a logging level and GnuTLS will output debugging output to standard error. Other available environment variables are shown in Table 6.1.

Variable	Purpose
<code>GNUTLS_DEBUG_LEVEL</code>	When set to a numeric value, it sets the default debugging level for GnuTLS applications.
<code>GNUTLS_CPUID_OVERRIDE</code>	That environment variable can be used to explicitly enable/disable the use of certain CPU capabilities. Note that CPU detection cannot be overridden, i.e., VIA options cannot be enabled on an Intel CPU. The currently available options are: <ul style="list-style-type: none"> <li>• 0x1: Disable all run-time detected optimizations</li> <li>• 0x2: Enable AES-NI</li> <li>• 0x4: Enable SSSE3</li> <li>• 0x8: Enable PCLMUL</li> <li>• 0x100000: Enable VIA padlock</li> <li>• 0x200000: Enable VIA PHE</li> <li>• 0x400000: Enable VIA PHE SHA512</li> </ul>
<code>GNUTLS_FORCE_FIPS_MODE</code>	In setups where GnuTLS is compiled with support for FIPS140-2 (see <code>-enable-fips140-mode</code> in <code>configure</code> ), that option if set to one enforces the FIPS140 mode.

Table 6.1: Environment variables used by the library.

When debugging is not required, important issues, such as detected attacks on the protocol still need to be logged. This is provided by the logging function set by `[gnutls_global_set_audit_log_function]`, page 308. The provided function will receive a message and the corresponding TLS session. The session information might be used to derive IP addresses or other information about the peer involved.

```
void gnutls_global_set_audit_log_function                                     [Function]
    (gnutls_audit_log_func log_func)
    log_func: it is the audit log function
```

This is the function to set the audit logging function. This is a function to report important issues, such as possible attacks in the protocol. This is different from `gnutls_global_set_log_function()` because it will report also session-specific events. The session parameter will be null if there is no corresponding TLS session.

`gnutls_audit_log_func` is of the form, `void (*gnutls_audit_log_func)(gnutls_session_t, const char*)`;

**Since:** 3.0

### 6.1.5 Thread safety

The GnuTLS library is thread safe by design, meaning that objects of the library such as TLS sessions, can be safely divided across threads as long as a single thread accesses a single object. This is sufficient to support a server which handles several sessions per thread. If, however, an object needs to be shared across threads then access must be protected with a mutex. Read-only access to objects, for example the credentials holding structures, is also thread-safe.

A `gnutls_session_t` object can be shared by two threads, one sending, the other receiving. In that case rehandshakes, if required, must only be handled by a single thread being active. The termination of a session should be handled, either by a single thread being active, or by the sender thread using `[gnutls_bye]`, page 277 with `GNUTLS_SHUT_WR` and the receiving thread waiting for a return value of zero.

The random generator of the cryptographic back-end, utilizes mutex locks (e.g., pthreads on GNU/Linux and CriticalSection on Windows) which are setup by GnuTLS on library initialization. Prior to version 3.3.0 they were setup by calling `[gnutls_global_init]`, page 307. On special systems you could manually specify the locking system using the function `[gnutls_global_set_mutex]`, page 308 before calling any other GnuTLS function. Setting mutexes manually is not recommended. An example of non-native thread usage is shown below.

```
#include <gnutls/gnutls.h>

int main()
{
    /* When the system mutexes are not to be used
     * gnutls_global_set_mutex() must be called explicitly
     */
    gnutls_global_set_mutex (mutex_init, mutex_deinit,
                             mutex_lock, mutex_unlock);
}
```

```
void gnutls_global_set_mutex (mutex_init_func init,           [Function]
                             mutex_deinit_func deinit,
                             mutex_lock_func lock, mutex_unlock_func
                             unlock)
```

*init*: mutex initialization function

*deinit*: mutex deinitialization function

*lock*: mutex locking function

*unlock*: mutex unlocking function

With this function you are allowed to override the default mutex locks used in some parts of gnutls and dependent libraries. This function should be used if you have complete control of your program and libraries. Do not call this function from a library, or preferably from any application unless really needed to. GnuTLS will use the appropriate locks for the running system.

This function must be called prior to any other gnutls function.

**Since:** 2.12.0

### 6.1.6 Sessions and fork

A `gnutls_session_t` object can be shared by two processes after a fork, one sending, the other receiving. In that case rehandshakes, cannot and must not be performed. As with threads, the termination of a session should be handled by the sender process using `[gnutls_bye]`, page 277 with `GNUTLS_SHUT_WR` and the receiving process waiting for a return value of zero.

### 6.1.7 Callback functions

There are several cases where GnuTLS may need out of band input from your program. This is now implemented using some callback functions, which your program is expected to register.

An example of this type of functions are the push and pull callbacks which are used to specify the functions that will retrieve and send data to the transport layer.

```
void [gnutls_transport_set_push_function], page 360 (gnutls_session_t
session, gnutls_push_func push_func)
void [gnutls_transport_set_pull_function], page 359 (gnutls_session_t
session, gnutls_pull_func pull_func)
```

Other callback functions may require more complicated input and data to be allocated. Such an example is `[gnutls_srp_set_server_credentials_function]`, page 350. All callbacks should allocate and free memory using `gnutls_malloc` and `gnutls_free`.

## 6.2 Preparation

To use GnuTLS, you have to perform some changes to your sources and your build system. The necessary changes are explained in the following subsections.

### 6.2.1 Headers

All the data types and functions of the GnuTLS library are defined in the header file `gnutls/gnutls.h`. This must be included in all programs that make use of the GnuTLS library.

### 6.2.2 Initialization

The GnuTLS library is initialized on load; prior to 3.3.0 was initialized by calling `[gnutls_global_init]`, page 307<sup>2</sup>. The initialization typically enables CPU-specific

<sup>2</sup> The original behavior of requiring explicit initialization can be obtained by setting the `GNUTLS_NO_EXPLICIT_INIT` environment variable to 1, or by using the macro `GNUTLS_SKIP_GLOBAL_INIT` in a global section of your program.

acceleration, performs any required precalculations needed, opens any required system devices (e.g., `/dev/urandom` on Linux) and initializes subsystems that could be used later. The resources allocated by the initialization process will be released on library deinitialization, or explicitly by calling `[gnutls_global_deinit]`, page 307.

Note that during initialization file descriptors may be kept open by GnuTLS (e.g. `/dev/urandom`) on library load. Applications closing all unknown file descriptors must immediately call `[gnutls_global_init]`, page 307, after that, to ensure they don't disrupt GnuTLS' operation.

### 6.2.3 Version check

It is often desirable to check that the version of 'gnutls' used is indeed one which fits all requirements. Even with binary compatibility new features may have been introduced but due to problem with the dynamic linker an old version is actually used. So you may want to check that the version is okay right after program start-up. See the function `[gnutls_check_version]`, page 294.

On the other hand, it is often desirable to support more than one versions of the library. In that case you could utilize compile-time feature checks using the `GNUTLS_VERSION_NUMBER` macro. For example, to conditionally add code for GnuTLS 3.2.1 or later, you may use:

```
#if GNUTLS_VERSION_NUMBER >= 0x030201
...
#endif
```

### 6.2.4 Building the source

If you want to compile a source file including the `gnutls/gnutls.h` header file, you must make sure that the compiler can find it in the directory hierarchy. This is accomplished by adding the path to the directory in which the header file is located to the compilers include file search path (via the `-I` option).

However, the path to the include file is determined at the time the source is configured. To solve this problem, the library uses the external package `pkg-config` that knows the path to the include file and other configuration options. The options that need to be added to the compiler invocation at compile time are output by the `--cflags` option to `pkg-config gnutls`. The following example shows how it can be used at the command line:

```
gcc -c foo.c 'pkg-config gnutls --cflags'
```

Adding the output of `'pkg-config gnutls --cflags'` to the compilers command line will ensure that the compiler can find the `gnutls/gnutls.h` header file.

A similar problem occurs when linking the program with the library. Again, the compiler has to find the library files. For this to work, the path to the library files has to be added to the library search path (via the `-L` option). For this, the option `--libs` to `pkg-config gnutls` can be used. For convenience, this option also outputs all other options that are required to link the program with the library (for instance, the `'-ltaasn1'` option). The example shows how to link `foo.o` with the library to a program `foo`.

```
gcc -o foo foo.o 'pkg-config gnutls --libs'
```

Of course you can also combine both examples to a single command by specifying both options to `pkg-config`:

```
gcc -o foo foo.c `pkg-config gnutls --cflags --libs`
```

When a program uses the GNU autoconf system, then the following line or similar can be used to detect the presence of GnuTLS.

```
PKG_CHECK_MODULES([LIBGNUTLS], [gnutls >= 3.3.0])

AC_SUBST([LIBGNUTLS_CFLAGS])
AC_SUBST([LIBGNUTLS_LIBS])
```

### 6.3 Session initialization

In the previous sections we have discussed the global initialization required for GnuTLS as well as the initialization required for each authentication method's credentials (see [Section 3.5.2 \[Authentication\]](#), page 9). In this section we elaborate on the TLS or DTLS session initiation. Each session is initialized using [\[gnutls\\_init\]](#), page 315 which among others is used to specify the type of the connection (server or client), and the underlying protocol type, i.e., datagram (UDP) or reliable (TCP).

**int gnutls\_init** (*gnutls\_session\_t \* session, unsigned int flags*) [Function]

*session*: is a pointer to a `gnutls_session_t` structure.

*flags*: indicate if this session is to be used for server or client.

This function initializes the current session to null. Every session must be initialized before use, so internal structures can be allocated. This function allocates structures which can only be free'd by calling `gnutls_deinit()`. Returns `GNUTLS_E_SUCCESS` (0) on success.

*flags* can be one of `GNUTLS_CLIENT` and `GNUTLS_SERVER`. For a DTLS entity, the flags `GNUTLS_DATAGRAM` and `GNUTLS_NONBLOCK` are also available. The latter flag will enable a non-blocking operation of the DTLS timers.

The flag `GNUTLS_NO_REPLAY_PROTECTION` will disable any replay protection in DTLS mode. That must only used when replay protection is achieved using other means.

Note that since version 3.1.2 this function enables some common TLS extensions such as session tickets and OCSP certificate status request in client side by default. To prevent that use the `GNUTLS_NO_EXTENSIONS` flag.

**Returns:** `GNUTLS_E_SUCCESS` on success, or an error code.

After the session initialization details on the allowed ciphersuites and protocol versions should be set using the priority functions such as [\[gnutls\\_priority\\_set\\_direct\]](#), page 326. We elaborate on them in [Section 6.10 \[Priority Strings\]](#), page 132. The credentials used for the key exchange method, such as certificates or usernames and passwords should also be associated with the session current session using [\[gnutls\\_credentials\\_set\]](#), page 297.

**int gnutls\_credentials\_set** (*gnutls\_session\_t session,* [Function]

*gnutls\_credentials\_type\_t type, void \* cred*)

*session*: is a `gnutls_session_t` structure.

*type*: is the type of the credentials

*cred*: is a pointer to a structure.



Sets the needed credentials for the specified type. Eg username, password - or public and private keys etc. The `cred` parameter is a structure that depends on the specified type and on the current session (client or server).

In order to minimize memory usage, and share credentials between several threads gnutls keeps a pointer to cred, and not the whole cred structure. Thus you will have to keep the structure allocated until you call `gnutls_deinit()` .

For `GNUTLS_CRD_ANON` , cred should be `gnutls_anon_client_credentials_t` in case of a client. In case of a server it should be `gnutls_anon_server_credentials_t` .

For `GNUTLS_CRD_SRP` , cred should be `gnutls_srp_client_credentials_t` in case of a client, and `gnutls_srp_server_credentials_t` , in case of a server.

For `GNUTLS_CRD_CERTIFICATE` , cred should be `gnutls_certificate_credentials_t` .

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

## 6.4 Associating the credentials

Each authentication method is associated with a key exchange method, and a credentials type. The contents of the credentials is method-dependent, e.g. certificates for certificate authentication and should be initialized and associated with a session (see [\[gnutls\\_credentials\\_set\]](#), page 297). A mapping of the key exchange methods with the credential types is shown in [Table 6.2](#).

Authentication method		Key exchange	Client credentials	Server credentials
Certificate		KX_RSA, KX_DHE_RSA, KX_DHE_DSS, KX_ECDHE_RSA, KX_ECDHE_ECDSA, KX_RSA_EXPORT	CRD_CERTIFICATE	CRD_CERTIFICATE
Password certificate	and	KX_SRP_RSA, KX_SRP_DSS	CRD_SRP	CRD_CERTIFICATE, CRD_SRP
Password		KX_SRP	CRD_SRP	CRD_SRP
Anonymous		KX_ANON_DH, KX_ANON_ECDH	CRD_ANON	CRD_ANON
Pre-shared key		KX_PSK, DHE_PSK, KX_ECDHE_PSK	KX_ CRD_PSK	CRD_PSK

Table 6.2: Key exchange algorithms and the corresponding credential types.

### 6.4.1 Certificates

## Server certificate authentication

When using certificates the server is required to have at least one certificate and private key pair. Clients may not hold such a pair, but a server could require it. In this section we discuss general issues applying to both client and server certificates. The next section will elaborate on issues arising from client authentication only.

```
int [gnutls_certificate_allocate_credentials], page 278
(gnutls_certificate_credentials_t * res)
void [gnutls_certificate_free_credentials], page 279
(gnutls_certificate_credentials_t sc)
```

After the credentials structures are initialized, the certificate and key pair must be loaded. This occurs before any TLS session is initialized, and the same structures are reused for multiple sessions. Depending on the certificate type different loading functions are available, as shown below. For X.509 certificates, the functions will accept and use a certificate chain that leads to a trusted authority. The certificate chain must be ordered in such way that every certificate certifies the one before it. The trusted authority's certificate need not to be included since the peer should possess it already.

```
int [gnutls_certificate_set_x509_key_mem2], page 288
(gnutls_certificate_credentials_t res, const gnutls_datum_t * cert, const
gnutls_datum_t * key, gnutls_x509_crt_fmt_t type, const char * pass, unsigned
int flags)
int [gnutls_certificate_set_x509_key], page 285
(gnutls_certificate_credentials_t res, gnutls_x509_crt_t * cert_list, int
cert_list_size, gnutls_x509_privkey_t key)
int [gnutls_certificate_set_x509_key_file2], page 286
(gnutls_certificate_credentials_t res, const char * certfile, const char *
keyfile, gnutls_x509_crt_fmt_t type, const char * pass, unsigned int flags)
int [gnutls_certificate_set_openpgp_key_mem], page 478
(gnutls_certificate_credentials_t res, const gnutls_datum_t * cert, const
gnutls_datum_t * key, gnutls_openpgp_crt_fmt_t format)
int [gnutls_certificate_set_openpgp_key], page 477
(gnutls_certificate_credentials_t res, gnutls_openpgp_crt_t crt,
gnutls_openpgp_privkey_t pkey)
int [gnutls_certificate_set_openpgp_key_file], page 477
(gnutls_certificate_credentials_t res, const char * certfile, const char *
keyfile, gnutls_openpgp_crt_fmt_t format)
```

Note however, that since functions like `[gnutls_certificate_set_x509_key_file2]`, page 286 may accept URLs that specify objects stored in token, another important function is `[gnutls_certificate_set_pin_function]`, page 283. That allows setting a callback function to retrieve a PIN if the input keys are protected by PIN by the token.

```
void gnutls_certificate_set_pin_function [Function]
(gnutls_certificate_credentials_t cred, gnutls_pin_callback_t fn, void *
userdata)
```

`cred`: is a `gnutls_certificate_credentials_t` structure.

`fn`: A PIN callback

*userdata*: Data to be passed in the callback

This function will set a callback function to be used when required to access a protected object. This function overrides any other global PIN functions.

Note that this function must be called right after initialization to have effect.

**Since:** 3.1.0

If the imported keys and certificates need to be accessed before any TLS session is established, it is convenient to use [\[gnutls\\_certificate\\_set\\_key\]](#), page 520 in combination with [\[gnutls\\_pcert\\_import\\_x509\\_raw\]](#), page 523 and [\[gnutls\\_privkey\\_import\\_x509\\_raw\]](#), page 531.

```
int gnutls_certificate_set_key (gnutls_certificate_credentials_t [Function]
                               res, const char ** names, int names_size, gnutls_pcert_st * pcert_list, int
                               pcert_list_size, gnutls_privkey_t key)
```

*res*: is a `gnutls_certificate_credentials_t` structure.

*names*: is an array of DNS name of the certificate (NULL if none)

*names\_size*: holds the size of the names list

*pcert\_list*: contains a certificate list (path) for the specified private key

*pcert\_list\_size*: holds the size of the certificate list

*key*: is a `gnutls_privkey_t` key

This function sets a certificate/private key pair in the `gnutls_certificate_credentials_t` structure. This function may be called more than once, in case multiple keys/certificates exist for the server. For clients that wants to send more than its own end entity certificate (e.g., also an intermediate CA cert) then put the certificate chain in `pcert_list`.

Note that the `pcert_list` and `key` will become part of the credentials structure and must not be deallocated. They will be automatically deallocated when the `res` structure is deinitialized.

If that function fails to load the `res` structure is at an undefined state, it must not be reused to load other keys or certificates.

**Returns:** `GNUTLS_E_SUCCESS` (0) on success, or a negative error code.

**Since:** 3.0

If multiple certificates are used with the functions above each client's request will be served with the certificate that matches the requested name (see [Section 3.6.2 \[Server name indication\]](#), page 10).

As an alternative to loading from files or buffers, a callback may be used for the server or the client to specify the certificate and the key at the handshake time. In that case a certificate should be selected according the peer's signature algorithm preferences. To get those preferences use [\[gnutls\\_sign\\_algorithm\\_get\\_requested\]](#), page 345. Both functions are shown below.

```

void [gnutls_certificate_set_retrieve_function], page 283
(gnutls_certificate_credentials_t cred, gnutls_certificate_retrieve_function
* func)
void [gnutls_certificate_set_retrieve_function2], page 521
(gnutls_certificate_credentials_t cred,
gnutls_certificate_retrieve_function2 * func)
int [gnutls_sign_algorithm_get_requested], page 345 (gnutls_session_t
session, size_t indx, gnutls_sign_algorithm_t * algo)

```

c The functions above do not handle the requested server name automatically. A server would need to check the name requested by the client using [\[gnutls\\_server\\_name\\_get\]](#), [page 339](#), and serve the appropriate certificate. Note that some of these functions require the `gnutls_pcert_st` structure to be filled in. Helper functions to fill in the structure are listed below.

```

typedef struct gnutls_pcert_st
{
    gnutls_pubkey_t pubkey;
    gnutls_datum_t cert;
    gnutls_certificate_type_t type;
} gnutls_pcert_st;

int [gnutls_pcert_import_x509], page 522 (gnutls_pcert_st * pcert,
gnutls_x509_crt_t crt, unsigned int flags)
int [gnutls_pcert_import_openpgp], page 522 (gnutls_pcert_st * pcert,
gnutls_openpgp_crt_t crt, unsigned int flags)
int [gnutls_pcert_import_x509_raw], page 523 (gnutls_pcert_st * pcert, const
gnutls_datum_t * cert, gnutls_x509_crt_fmt_t format, unsigned int flags)
int [gnutls_pcert_import_openpgp_raw], page 522 (gnutls_pcert_st * pcert,
const gnutls_datum_t * cert, gnutls_openpgp_crt_fmt_t format,
gnutls_openpgp_keyid_t keyid, unsigned int flags)
void [gnutls_pcert_deinit], page 522 (gnutls_pcert_st * pcert)

```

In a handshake, the negotiated cipher suite depends on the certificate's parameters, so some key exchange methods might not be available with all certificates. GnuTLS will disable ciphersuites that are not compatible with the key, or the enabled authentication methods. For example keys marked as sign-only, will not be able to access the plain RSA ciphersuites, that require decryption. It is not recommended to use RSA keys for both signing and encryption. If possible use a different key for the DHE-RSA which uses signing and RSA that requires decryption. All the key exchange methods shown in [Table 4.1](#) are available in certificate authentication.

## Client certificate authentication

If a certificate is to be requested from the client during the handshake, the server will send a certificate request message. This behavior is controlled [\[gnutls\\_certificate\\_server\\_set\\_request\]](#), [page 281](#). The request contains a list of the acceptable by the server certificate signers. This list is constructed using the trusted certificate authorities of the server. In cases where the server supports a large number of certificate authorities it makes sense not to advertise all of the names to save bandwidth. That can be controlled using the function [\[gnutls\\_certificate\\_send\\_x509\\_rdn\\_sequence\]](#),

page 281. This however will have the side-effect of not restricting the client to certificates signed by server's acceptable signers.

**void gnutls\_certificate\_server\_set\_request** (*gnutls\_session\_t session*, *gnutls\_certificate\_request\_t req*) [Function]

*session*: is a *gnutls\_session\_t* structure.

*req*: is one of GNUTLS\_CERT\_REQUEST, GNUTLS\_CERT\_REQUIRE

This function specifies if we (in case of a server) are going to send a certificate request message to the client. If *req* is GNUTLS\_CERT\_REQUIRE then the server will return an error if the peer does not provide a certificate. If you do not call this function then the client will not be asked to send a certificate.

**void gnutls\_certificate\_send\_x509\_rdn\_sequence** (*gnutls\_session\_t session*, *int status*) [Function]

*session*: is a pointer to a *gnutls\_session\_t* structure.

*status*: is 0 or 1

If status is non zero, this function will order gnutls not to send the rdnSequence in the certificate request message. That is the server will not advertise its trusted CAs to the peer. If status is zero then the default behaviour will take effect, which is to advertise the server's trusted CAs.

This function has no effect in clients, and in authentication methods other than certificate with X.509 certificates.

## Client or server certificate verification

Certificate verification is possible by loading the trusted authorities into the credentials structure by using the following functions, applicable to X.509 and OpenPGP certificates.

**int [gnutls\_certificate\_set\_x509\_system\_trust]**, page 289

(*gnutls\_certificate\_credentials\_t cred*)

**int [gnutls\_certificate\_set\_x509\_trust\_file]**, page 290

(*gnutls\_certificate\_credentials\_t cred*, *const char \* cafile*, *gnutls\_x509\_crt\_fmt\_t type*)

**int [gnutls\_certificate\_set\_openpgp\_keyring\_file]**, page 479

(*gnutls\_certificate\_credentials\_t c*, *const char \* file*, *gnutls\_openpgp\_crt\_fmt\_t format*)

The peer's certificate is not automatically verified and one must call **[gnutls\_certificate\_verify\_peers3]**, page 293 after a successful handshake to verify the certificate's signature and the owner of the certificate. The verification status returned can be printed using **[gnutls\_certificate\_verification\_status\_print]**, page 292.

Alternatively the verification can occur during the handshake by using **[gnutls\_certificate\_set\_verify\_function]**, page 284.

The functions above provide a brief verification output. If a detailed output is required one should call **[gnutls\_certificate\_get\_peers]**, page 280 to obtain the raw certificate of the peer and verify it using the functions discussed in Section 4.1.1 [X.509 certificates], page 19.

```
int gnutls_certificate_verify_peers3 (gnutls_session_t session,      [Function]
                                     const char *hostname, unsigned int *status)
```

*session*: is a gnutls session

*hostname*: is the expected name of the peer; may be NULL

*status*: is the output of the verification

This function will verify the peer's certificate and store the status in the **status** variable as a bitwise or'd `gnutls_certificate_status_t` values or zero if the certificate is trusted. Note that value in **status** is set only when the return value of this function is success (i.e, failure to trust a certificate does not imply a negative return value). The default verification flags used by this function can be overridden using `gnutls_certificate_set_verify_flags()`. See the documentation of `gnutls_certificate_verify_peers2()` for details in the verification process.

If the **hostname** provided is non-NULL then this function will compare the hostname in the certificate against the given. The comparison will be accurate for ascii names; non-ascii names are compared byte-by-byte. If names do not match the `GNUTLS_CERT_UNEXPECTED_OWNER` status flag will be set.

In order to verify the purpose of the end-certificate (by checking the extended key usage), use `gnutls_certificate_verify_peers()`.

**Returns:** a negative error code on error and `GNUTLS_E_SUCCESS` (0) on success.

**Since:** 3.1.4

```
void gnutls_certificate_set_verify_function (gnutls_certificate_credentials_t cred, gnutls_certificate_verify_function *func) [Function]
```

*cred*: is a `gnutls_certificate_credentials_t` structure.

*func*: is the callback function

This function sets a callback to be called when peer's certificate has been received in order to verify it on receipt rather than doing after the handshake is completed.

The callback's function prototype is: `int (*callback)(gnutls_session_t);`

If the callback function is provided then gnutls will call it, in the handshake, just after the certificate message has been received. To verify or obtain the certificate the `gnutls_certificate_verify_peers2()`, `gnutls_certificate_type_get()`, `gnutls_certificate_get_peers()` functions can be used.

The callback function should return 0 for the handshake to continue or non-zero to terminate.

**Since:** 2.10.0

## 6.4.2 SRP

The initialization functions in SRP credentials differ between client and server. Clients supporting SRP should set the username and password prior to connection, to the credentials structure. Alternatively [\[gnutls\\_srp\\_set\\_client\\_credentials\\_function\]](#), [page 349](#) may be used instead, to specify a callback function that should return the SRP username and password. The callback is called once during the TLS handshake.

```

int [gnutls_srp_allocate_server_credentials], page 347
(gnutls_srp_server_credentials_t * sc)
int [gnutls_srp_allocate_client_credentials], page 347
(gnutls_srp_client_credentials_t * sc)
void [gnutls_srp_free_server_credentials], page 349
(gnutls_srp_server_credentials_t sc)
void [gnutls_srp_free_client_credentials], page 348
(gnutls_srp_client_credentials_t sc)
int [gnutls_srp_set_client_credentials], page 349
(gnutls_srp_client_credentials_t res, const char * username, const char *
password)

void gnutls_srp_set_client_credentials_function [Function]
(gnutls_srp_client_credentials_t cred, gnutls_srp_client_credentials_function *
func)

```

*cred*: is a `gnutls_srp_server_credentials_t` structure.

*func*: is the callback function

This function can be used to set a callback to retrieve the username and password for client SRP authentication. The callback's function form is:

```
int (*callback)(gnutls_session_t, char** username, char**password);
```

The `username` and `password` must be allocated using `gnutls_malloc()`. `username` and `password` should be ASCII strings or UTF-8 strings prepared using the "SASL-prep" profile of "stringprep".

The callback function will be called once per handshake before the initial hello message is sent.

The callback should not return a negative error code the second time called, since the handshake procedure will be aborted.

The callback function should return 0 on success. -1 indicates an error.

In server side the default behavior of GnuTLS is to read the usernames and SRP verifiers from password files. These password file format is compatible the with the *Stanford srp libraries* format. If a different password file format is to be used, then `[gnutls_srp_set_server_credentials_function]`, page 350 should be called, to set an appropriate callback.

```

int gnutls_srp_set_server_credentials_file [Function]
(gnutls_srp_server_credentials_t res, const char * password_file, const char
* password_conf_file)

```

*res*: is a `gnutls_srp_server_credentials_t` structure.

*password\_file*: is the SRP password file (tpasswd)

*password\_conf\_file*: is the SRP password conf file (tpasswd.conf)

This function sets the password files, in a `gnutls_srp_server_credentials_t` structure. Those password files hold usernames and verifiers and will be used for SRP authentication.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, or an error code.



```
void gnutls_srp_set_server_credentials_function [Function]
      (gnutls_srp_server_credentials_t cred, gnutls_srp_server_credentials_function *
       func)
```

*cred*: is a `gnutls_srp_server_credentials_t` structure.

*func*: is the callback function

This function can be used to set a callback to retrieve the user's SRP credentials. The callback's function form is:

```
int (*callback)(gnutls_session_t, const char* username, gnutls_datum_t *salt,
gnutls_datum_t *verifier, gnutls_datum_t *generator, gnutls_datum_t *prime);
```

**username** contains the actual username. The **salt**, **verifier**, **generator** and **prime** must be filled in using the `gnutls_malloc()`. For convenience **prime** and **generator** may also be one of the static parameters defined in `gnutls.h`.

Initially, the data field is NULL in every `gnutls_datum_t` structure that the callback has to fill in. When the callback is done GnuTLS deallocates all of those buffers which are non-NULL, regardless of the return value.

In order to prevent attackers from guessing valid usernames, if a user does not exist, **g** and **n** values should be filled in using a random user's parameters. In that case the callback must return the special value (1). See `gnutls_srp_set_server_fake_salt_seed` too. If this is not required for your application, return a negative number from the callback to abort the handshake.

The callback function will only be called once per handshake. The callback function should return 0 on success, while -1 indicates an error.

### 6.4.3 PSK

The initialization functions in PSK credentials differ between client and server.

```
int [gnutls_psk_allocate_server_credentials], page 328
```

```
(gnutls_psk_server_credentials_t * sc)
```

```
int [gnutls_psk_allocate_client_credentials], page 328
```

```
(gnutls_psk_client_credentials_t * sc)
```

```
void [gnutls_psk_free_server_credentials], page 329
```

```
(gnutls_psk_server_credentials_t sc)
```

```
void [gnutls_psk_free_client_credentials], page 328
```

```
(gnutls_psk_client_credentials_t sc)
```

Clients supporting PSK should supply the username and key before a TLS session is established. Alternatively `[gnutls_psk_set_client_credentials_function]`, page 329 can be used to specify a callback function. This has the advantage that the callback will be called only if PSK has been negotiated.

```
int [gnutls_psk_set_client_credentials], page 329
```

```
(gnutls_psk_client_credentials_t res, const char * username, const
```

```
gnutls_datum_t * key, gnutls_psk_key_flags flags)
```

```
void gnutls_psk_set_client_credentials_function [Function]
      (gnutls_psk_client_credentials_t cred, gnutls_psk_client_credentials_function *
       func)
```

*cred*: is a `gnutls_psk_server_credentials_t` structure.



*func*: is the callback function

This function can be used to set a callback to retrieve the username and password for client PSK authentication. The callback's function form is: `int (*callback)(gnutls_session_t, char** username, gnutls_datum_t* key);`

The **username** and **key** ->data must be allocated using `gnutls_malloc()`. **username** should be ASCII strings or UTF-8 strings prepared using the "SASLprep" profile of "stringprep".

The callback function will be called once per handshake.

The callback function should return 0 on success. -1 indicates an error.

In server side the default behavior of GnuTLS is to read the usernames and PSK keys from a password file. The password file should contain usernames and keys in hexadecimal format. The name of the password file can be stored to the credentials structure by calling `[gnutls_psk_set_server_credentials_file]`, page 330. If a different password file format is to be used, then a callback should be set instead by `[gnutls_psk_set_server_credentials_function]`, page 330.

The server can help the client chose a suitable username and password, by sending a hint. Note that there is no common profile for the PSK hint and applications are discouraged to use it. A server, may specify the hint by calling `[gnutls_psk_set_server_credentials_hint]`, page 330. The client can retrieve the hint, for example in the callback function, using `[gnutls_psk_client_get_hint]`, page 328.

`int gnutls_psk_set_server_credentials_file` [Function]

(`gnutls_psk_server_credentials_t res`, `const char * password_file`)

*res*: is a `gnutls_psk_server_credentials_t` structure.

*password\_file*: is the PSK password file (passwd.psk)

This function sets the password file, in a `gnutls_psk_server_credentials_t` structure. This password file holds usernames and keys and will be used for PSK authentication.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

`void [gnutls_psk_set_server_credentials_function]`, page 330

(`gnutls_psk_server_credentials_t cred`,  
`gnutls_psk_server_credentials_function * func`)

`int [gnutls_psk_set_server_credentials_hint]`, page 330

(`gnutls_psk_server_credentials_t res`, `const char * hint`)

`const char * [gnutls_psk_client_get_hint]`, page 328 (`gnutls_session_t session`)

#### 6.4.4 Anonymous

The key exchange methods for anonymous authentication might require Diffie-Hellman parameters to be generated by the server and associated with an anonymous credentials structure. Check [Section 6.12.3 \[Parameter generation\]](#), page 146 for more information. The initialization functions for the credentials are shown below.

```

int [gnutls_anon_allocate_server_credentials], page 275
(gnutls_anon_server_credentials_t * sc)
int [gnutls_anon_allocate_client_credentials], page 275
(gnutls_anon_client_credentials_t * sc)
void [gnutls_anon_free_server_credentials], page 275
(gnutls_anon_server_credentials_t sc)
void [gnutls_anon_free_client_credentials], page 275
(gnutls_anon_client_credentials_t sc)

```

## 6.5 Setting up the transport layer

The next step is to setup the underlying transport layer details. The Berkeley sockets are implicitly used by GnuTLS, thus a call to `[gnutls_transport_set_int]`, page 359 would be sufficient to specify the socket descriptor.

```

void [gnutls_transport_set_int], page 359 (gnutls_session_t session, int i)
void [gnutls_transport_set_int2], page 359 (gnutls_session_t session, int
recv_int, int send_int)

```

If however another transport layer than TCP is selected, then a pointer should be used instead to express the parameter to be passed to custom functions. In that case the following functions should be used instead.

```

void [gnutls_transport_set_ptr], page 359 (gnutls_session_t session,
gnutls_transport_ptr_t ptr)
void [gnutls_transport_set_ptr2], page 359 (gnutls_session_t session,
gnutls_transport_ptr_t recv_ptr, gnutls_transport_ptr_t send_ptr)

```

Moreover all of the following push and pull callbacks should be set.

```

void gnutls_transport_set_push_function (gnutls_session_t      [Function]
    session, gnutls_push_func push_func)
    session: is a gnutls_session_t structure.
    push_func: a callback function similar to write()

```

This is the function where you set a push function for gnutls to use in order to send data. If you are going to use berkeley style sockets, you do not need to use this function since the default `send(2)` will probably be ok. Otherwise you should specify this function for gnutls to be able to send data. The callback should return a positive number indicating the bytes sent, and -1 on error.

`push_func` is of the form, `ssize_t (*gnutls_push_func)(gnutls_transport_ptr_t, const void*, size_t);`

```

void gnutls_transport_set_vec_push_function (gnutls_session_t      [Function]
    session, gnutls_vec_push_func vec_func)
    session: is a gnutls_session_t structure.
    vec_func: a callback function similar to writev()

```

Using this function you can override the default `writev(2)` function for gnutls to send data. Setting this callback instead of `gnutls_transport_set_push_function()` is recommended since it introduces less overhead in the TLS handshake process.

`vec_func` is of the form, `ssize_t (*gnutls_vec_push_func) (gnutls_transport_ptr_t, const iovect_t * iov, int iovcnt);`

**Since:** 2.12.0

`void gnutls_transport_set_pull_function (gnutls_session_t session, gnutls_pull_func pull_func)` [Function]

`session`: is a `gnutls_session_t` structure.

`pull_func`: a callback function similar to `read()`

This is the function where you set a function for gnutls to receive data. Normally, if you use berkeley style sockets, do not need to use this function since the default `recv(2)` will probably be ok. The callback should return 0 on connection termination, a positive number indicating the number of bytes received, and -1 on error.

`gnutls_pull_func` is of the form, `ssize_t (*gnutls_pull_func)(gnutls_transport_ptr_t, void*, size_t);`

`void gnutls_transport_set_pull_timeout_function (gnutls_session_t session, gnutls_pull_timeout_func func)` [Function]

`session`: is a `gnutls_session_t` structure.

`func`: a callback function

This is the function where you set a function for gnutls to know whether data are ready to be received. It should wait for data a given time frame in milliseconds. The callback should return 0 on timeout, a positive number if data can be received, and -1 on error. You'll need to override this function if `select()` is not suitable for the provided transport calls.

As with `select()`, if the timeout value is zero the callback should return zero if no data are immediately available.

`gnutls_pull_timeout_func` is of the form, `int (*gnutls_pull_timeout_func)(gnutls_transport_ptr_t, unsigned int ms);`

**Since:** 3.0

The functions above accept a callback function which should return the number of bytes written, or -1 on error and should set `errno` appropriately. In some environments, setting `errno` is unreliable. For example Windows have several `errno` variables in different CRTs, or in other systems it may be a non thread-local variable. If this is a concern to you, call [\[gnutls\\_transport\\_set\\_errno\]](#), page 358 with the intended `errno` value instead of setting `errno` directly.

`void gnutls_transport_set_errno (gnutls_session_t session, int err)` [Function]

`session`: is a `gnutls_session_t` structure.

`err`: error value to store in session-specific `errno` variable.

Store `err` in the session-specific `errno` variable. Useful values for `err` are `EINTR`, `EAGAIN` and `EMSGSIZE`, other values are treated will be treated as real errors in the push/pull function.

This function is useful in replacement push and pull functions set by `gnutls_transport_set_push_function()` and `gnutls_transport_set_pull_function()`

under Windows, where the replacements may not have access to the same `errno` variable that is used by GnuTLS (e.g., the application is linked to `msvcr71.dll` and `gnutls` is linked to `msvcrt.dll`).

GnuTLS currently only interprets the `EINTR`, `EAGAIN` and `EMSGSIZE` `errno` values and returns the corresponding GnuTLS error codes:

- `GNUTLS_E_INTERRUPTED`
- `GNUTLS_E_AGAIN`
- `GNUTLS_E_LARGE_PACKET`

The `EINTR` and `EAGAIN` values are returned by interrupted system calls, or when non blocking IO is used. All GnuTLS functions can be resumed (called again), if any of the above error codes is returned. The `EMSGSIZE` value is returned when attempting to send a large datagram.

In the case of DTLS it is also desirable to override the generic transport functions with functions that emulate the operation of `recvfrom` and `sendto`. In addition DTLS requires timers during the receive of a handshake message, set using the [\[gnutls\\_transport\\_set\\_pull\\_timeout\\_function\]](#), page 360 function. To check the retransmission timers the function [\[gnutls\\_dtls\\_get\\_timeout\]](#), page 363 is provided, which returns the time remaining until the next retransmission, or better the time until [\[gnutls\\_handshake\]](#), page 309 should be called again.

`void gnutls_transport_set_pull_timeout_function` [Function]

(*gnutls\_session\_t session, gnutls\_pull\_timeout\_func func*)

*session*: is a `gnutls_session_t` structure.

*func*: a callback function

This is the function where you set a function for gnutls to know whether data are ready to be received. It should wait for data a given time frame in milliseconds. The callback should return 0 on timeout, a positive number if data can be received, and -1 on error. You'll need to override this function if `select()` is not suitable for the provided transport calls.

As with `select()`, if the timeout value is zero the callback should return zero if no data are immediately available.

`gnutls_pull_timeout_func` is of the form, `int (*gnutls_pull_timeout_func)(gnutls_transport_ptr_t, unsigned int ms);`

**Since:** 3.0

`unsigned int gnutls_dtls_get_timeout` (*gnutls\_session\_t session*) [Function]

*session*: is a `gnutls_session_t` structure.

This function will return the milliseconds remaining for a retransmission of the previously sent handshake message. This function is useful when DTLS is used in non-blocking mode, to estimate when to call `gnutls_handshake()` if no packets have been received.

**Returns:** the remaining time in milliseconds.

**Since:** 3.0

### 6.5.1 Asynchronous operation

GnuTLS can be used with asynchronous socket or event-driven programming. The approach is similar to using Berkeley sockets under such an environment. The blocking, due to network interaction, calls such as `[gnutls_handshake]`, page 309, `[gnutls_record_recv]`, page 334, can be set to non-blocking by setting the underlying sockets to non-blocking. If other push and pull functions are setup, then they should behave the same way as `recv` and `send` when used in a non-blocking way, i.e., set `errno` to `EAGAIN`. Since, during a TLS protocol session GnuTLS does not block except for network interaction, the non blocking `EAGAIN` `errno` will be propagated and GnuTLS functions will return the `GNUTLS_E_AGAIN` error code. Such calls can be resumed the same way as a system call would. The only exception is `[gnutls_record_send]`, page 335, which if interrupted subsequent calls need not to include the data to be sent (can be called with `NULL` argument).

The `select` system call can also be used in combination with the GnuTLS functions. `select` allows monitoring of sockets and notifies on them being ready for reading or writing data. Note however that this system call cannot notify on data present in GnuTLS read buffers, it is only applicable to the kernel sockets API. Thus if you are using it for reading from a GnuTLS session, make sure that any cached data are read completely. That can be achieved by checking there are no data waiting to be read (using `[gnutls_record_check_pending]`, page 332), either before the `select` system call, or after a call to `[gnutls_record_recv]`, page 334. GnuTLS does not keep a write buffer, thus when writing no additional actions are required.

Although in the TLS protocol implementation each call to receive or send function implies to restoring the same function that was interrupted, in the DTLS protocol this requirement isn't true. There are cases where a retransmission is required, which are indicated by a received message and thus `[gnutls_record_get_direction]`, page 333 must be called to decide which direction to check prior to restoring a function call.

**int gnutls\_record\_get\_direction (gnutls\_session\_t session)** [Function]  
*session*: is a `gnutls_session_t` structure.

This function provides information about the internals of the record protocol and is only useful if a prior `gnutls` function call (e.g. `gnutls_handshake()`) was interrupted for some reason, that is, if a function returned `GNUTLS_E_INTERRUPTED` or `GNUTLS_E_AGAIN`. In such a case, you might want to call `select()` or `poll()` before calling the interrupted `gnutls` function again. To tell you whether a file descriptor should be selected for either reading or writing, `gnutls_record_get_direction()` returns 0 if the interrupted function was trying to read data, and 1 if it was trying to write data.

This function's output is unreliable if you are using the `session` in different threads, for sending and receiving.

**Returns:** 0 if trying to read data, 1 if trying to write data.

Moreover, to prevent blocking from DTLS' retransmission timers to block a handshake, the `[gnutls_init]`, page 315 function should be called with the `GNUTLS_NONBLOCK` flag set (see Section 6.3 [Session initialization], page 112). In that case, in order to be able to use the DTLS handshake timers, the function `[gnutls_dtls_get_timeout]`, page 363 should be used to estimate when to call `[gnutls_handshake]`, page 309 if no packets have been received.

### 6.5.2 DTLS sessions

Because datagram TLS can operate over connections where the client cannot be reliably verified, functionality in the form of cookies, is available to prevent denial of service attacks to servers. GnuTLS requires a server to generate a secret key that is used to sign a cookie<sup>3</sup>. That cookie is sent to the client using [\[gnutls\\_dtls\\_cookie\\_send\]](#), page 362, and the client must reply using the correct cookie. The server side should verify the initial message sent by client using [\[gnutls\\_dtls\\_cookie\\_verify\]](#), page 362. If successful the session should be initialized and associated with the cookie using [\[gnutls\\_dtls\\_prestate\\_set\]](#), page 363, before proceeding to the handshake.

```
int \[gnutls\_key\_generate\], page 315 (gnutls_datum_t * key, unsigned int
key_size)
int \[gnutls\_dtls\_cookie\_send\], page 362 (gnutls_datum_t * key, void *
client_data, size_t client_data_size, gnutls_dtls_prestate_st * prestate,
gnutls_transport_ptr_t ptr, gnutls_push_func push_func)
int \[gnutls\_dtls\_cookie\_verify\], page 362 (gnutls_datum_t * key, void *
client_data, size_t client_data_size, void * _msg, size_t msg_size,
gnutls_dtls_prestate_st * prestate)
void \[gnutls\_dtls\_prestate\_set\], page 363 (gnutls_session_t session,
gnutls_dtls_prestate_st * prestate)
```

Note that the above apply to server side only and they are not mandatory to be used. Not using them, however, allows denial of service attacks. The client side cookie handling is part of [\[gnutls\\_handshake\]](#), page 309.

Datagrams are typically restricted by a maximum transfer unit (MTU). For that both client and server side should set the correct maximum transfer unit for the layer underneath GnuTLS. This will allow proper fragmentation of DTLS messages and prevent messages from being silently discarded by the transport layer. The “correct” maximum transfer unit can be obtained through a path MTU discovery mechanism [\[RFC4821\]](#).

```
void \[gnutls\_dtls\_set\_mtu\], page 364 (gnutls_session_t session, unsigned int
mtu)
unsigned int \[gnutls\_dtls\_get\_mtu\], page 363 (gnutls_session_t session)
unsigned int \[gnutls\_dtls\_get\_data\_mtu\], page 363 (gnutls_session_t session)
```

## 6.6 TLS handshake

Once a session has been initialized and a network connection has been set up, TLS and DTLS protocols perform a handshake. The handshake is the actual key exchange.

```
int gnutls_handshake (gnutls_session_t session) [Function]
    session: is a gnutls_session_t structure.
```

This function does the handshake of the TLS/SSL protocol, and initializes the TLS connection.

This function will fail if any problem is encountered, and will return a negative error code. In case of a client, if the client has asked to resume a session, but the server couldn't, then a full handshake will be performed.

<sup>3</sup> A key of 128 bits or 16 bytes should be sufficient for this purpose.

The non-fatal errors expected by this function are: `GNUTLS_E_INTERRUPTED` , `GNUTLS_E_AGAIN` , `GNUTLS_E_WARNING_ALERT_RECEIVED` , and `GNUTLS_E_GOT_APPLICATION_DATA` , the latter only in a case of rehandshake.

The former two interrupt the handshake procedure due to the lower layer being interrupted, and the latter because of an alert that may be sent by a server (it is always a good idea to check any received alerts). On these errors call this function again, until it returns 0; cf. `gnutls_record_get_direction()` and `gnutls_error_is_fatal()` . In DTLS sessions the non-fatal error `GNUTLS_E_LARGE_PACKET` is also possible, and indicates that the MTU should be adjusted.

If this function is called by a server after a rehandshake request then `GNUTLS_E_GOT_APPLICATION_DATA` or `GNUTLS_E_WARNING_ALERT_RECEIVED` may be returned. Note that these are non fatal errors, only in the specific case of a rehandshake. Their meaning is that the client rejected the rehandshake request or in the case of `GNUTLS_E_GOT_APPLICATION_DATA` it could also mean that some data were pending.

**Returns:** `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

```
void gnutls_handshake_set_timeout (gnutls_session_t session,      [Function]
                                unsigned int ms)
```

*session*: is a `gnutls_session_t` structure.

*ms*: is a timeout value in milliseconds

This function sets the timeout for the handshake process to the provided value. Use an *ms* value of zero to disable timeout, or `GNUTLS_DEFAULT_HANDSHAKE_TIMEOUT` for a reasonable default value.

**Since:** 3.1.0

The handshake process doesn't ensure the verification of the peer's identity. When certificates are in use, this can be done, either after the handshake is complete, or during the handshake if `[gnutls_certificate_set_verify_function]`, page 284 has been used. In both cases the `[gnutls_certificate_verify_peers2]`, page 293 function can be used to verify the peer's certificate (see Section 4.1 [Certificate authentication], page 18 for more information).

```
int [gnutls_certificate_verify_peers2], page 293 (gnutls_session_t session,
unsigned int * status)
```

## 6.7 Data transfer and termination

Once the handshake is complete and peer's identity has been verified data can be exchanged. The available functions resemble the POSIX `recv` and `send` functions. It is suggested to use `[gnutls_error_is_fatal]`, page 306 to check whether the error codes returned by these functions are fatal for the protocol or can be ignored.

```
ssize_t gnutls_record_send (gnutls_session_t session, const void *    [Function]
                           data, size_t data_size)
```

*session*: is a `gnutls_session_t` structure.

*data*: contains the data to send

*data\_size*: is the length of the data



This function has the similar semantics with `send()` . The only difference is that it accepts a GnuTLS session, and uses different error codes. Note that if the send buffer is full, `send()` will block this function. See the `send()` documentation for more information.

You can replace the default push function which is `send()` , by using `gnutls_transport_set_push_function()` .

If the `EINTR` is returned by the internal push function then `GNUTLS_E_INTERRUPTED` will be returned. If `GNUTLS_E_INTERRUPTED` or `GNUTLS_E_AGAIN` is returned, you must call this function again, with the exact same parameters; alternatively you could provide a `NULL` pointer for data, and 0 for size. cf. `gnutls_record_get_direction()` .

Note that in DTLS this function will return the `GNUTLS_E_LARGE_PACKET` error code if the send data exceed the data MTU value - as returned by `gnutls_dtls_get_data_mtu()` . The `errno` value `EMSGSIZE` also maps to `GNUTLS_E_LARGE_PACKET` . Note that since 3.2.13 this function can be called under cork in DTLS mode, and will refuse to send data over the MTU size by returning `GNUTLS_E_LARGE_PACKET` .

**Returns:** The number of bytes sent, or a negative error code. The number of bytes sent might be less than `data_size` . The maximum number of bytes this function can send in a single call depends on the negotiated maximum record size.

```
ssize_t gnutls_record_recv (gnutls_session_t session, void * data,      [Function]
                           size_t data_size)
```

*session*: is a `gnutls_session_t` structure.

*data*: the buffer that the data will be read into

*data\_size*: the number of requested bytes

This function has the similar semantics with `recv()` . The only difference is that it accepts a GnuTLS session, and uses different error codes. In the special case that a server requests a renegotiation, the client may receive an error code of `GNUTLS_E_REHANDSHAKE` . This message may be simply ignored, replied with an alert `GNUTLS_A_NO_RENEGOTIATION` , or replied with a new handshake, depending on the client's will. If `EINTR` is returned by the internal push function (the default is `recv()` ) then `GNUTLS_E_INTERRUPTED` will be returned. If `GNUTLS_E_INTERRUPTED` or `GNUTLS_E_AGAIN` is returned, you must call this function again to get the data. See also `gnutls_record_get_direction()` . A server may also receive `GNUTLS_E_REHANDSHAKE` when a client has initiated a handshake. In that case the server can only initiate a handshake or terminate the connection.

**Returns:** The number of bytes received and zero on EOF (for stream connections). A negative error code is returned in case of an error. The number of bytes received might be less than the requested `data_size` .

```
int gnutls_error_is_fatal (int error)                                [Function]
```

*error*: is a GnuTLS error code, a negative error code

If a GnuTLS function returns a negative error code you may feed that value to this function to see if the error condition is fatal to a TLS session (i.e., must be terminated).



Note that you may also want to check the error code manually, since some non-fatal errors to the protocol (such as a warning alert or a rehandshake request) may be fatal for your program.

This function is only useful if you are dealing with errors from functions that relate to a TLS session (e.g., record layer or handshake layer handling functions).

**Returns:** Non-zero value on fatal errors or zero on non-fatal.

Although, in the TLS protocol the receive function can be called at any time, when DTLS is used the GnuTLS receive functions must be called once a message is available for reading, even if no data are expected. This is because in DTLS various (internal) actions may be required due to retransmission timers. Moreover, an extended receive function is shown below, which allows the extraction of the message's sequence number. Due to the unreliable nature of the protocol, this field allows distinguishing out-of-order messages.

`ssize_t gnutls_record_recv_seq (gnutls_session_t session, void * data, size_t data_size, unsigned char * seq)` [Function]

*session*: is a `gnutls_session_t` structure.

*data*: the buffer that the data will be read into

*data\_size*: the number of requested bytes

*seq*: is the packet's 64-bit sequence number. Should have space for 8 bytes.

This function is the same as `gnutls_record_recv()`, except that it returns in addition to data, the sequence number of the data. This is useful in DTLS where record packets might be received out-of-order. The returned 8-byte sequence number is an integer in big-endian format and should be treated as a unique message identification.

**Returns:** The number of bytes received and zero on EOF. A negative error code is returned in case of an error. The number of bytes received might be less than `data_size`.

**Since:** 3.0

The `[gnutls_record_check_pending]`, page 332 helper function is available to allow checking whether data are available to be read in a GnuTLS session buffers. Note that this function complements but does not replace `select`, i.e., `[gnutls_record_check_pending]`, page 332 reports no data to be read, `select` should be called to check for data in the network buffers.

`size_t gnutls_record_check_pending (gnutls_session_t session)` [Function]

*session*: is a `gnutls_session_t` structure.

This function checks if there are unread data in the gnutls buffers. If the return value is non-zero the next call to `gnutls_record_recv()` is guaranteed not to block.

**Returns:** Returns the size of the data or zero.

`int [gnutls_record_get_direction]`, page 333 (`gnutls_session_t session`)

Once a TLS or DTLS session is no longer needed, it is recommended to use `[gnutls_bye]`, page 277 to terminate the session. That way the peer is notified securely about the intention of termination, which allows distinguishing it from a malicious connection termination. A session can be deinitialized with the `[gnutls_deinit]`, page 300 function.

**int gnutls\_bye** (*gnutls\_session\_t session*, *gnutls\_close\_request\_t how*) [Function]

*session*: is a `gnutls_session_t` structure.

*how*: is an integer

Terminates the current TLS/SSL connection. The connection should have been initiated using `gnutls_handshake()` . *how* should be one of `GNUTLS_SHUT_RDWR` , `GNUTLS_SHUT_WR` .

In case of `GNUTLS_SHUT_RDWR` the TLS session gets terminated and further receives and sends will be disallowed. If the return value is zero you may continue using the underlying transport layer. `GNUTLS_SHUT_RDWR` sends an alert containing a close request and waits for the peer to reply with the same message.

In case of `GNUTLS_SHUT_WR` the TLS session gets terminated and further sends will be disallowed. In order to reuse the connection you should wait for an EOF from the peer. `GNUTLS_SHUT_WR` sends an alert containing a close request.

Note that not all implementations will properly terminate a TLS connection. Some of them, usually for performance reasons, will terminate only the underlying transport layer, and thus not distinguishing between a malicious party prematurely terminating the connection and normal termination.

This function may also return `GNUTLS_E_AGAIN` or `GNUTLS_E_INTERRUPTED` ; cf. `gnutls_record_get_direction()` .

**Returns:** `GNUTLS_E_SUCCESS` on success, or an error code, see function documentation for entire semantics.

**void gnutls\_deinit** (*gnutls\_session\_t session*) [Function]

*session*: is a `gnutls_session_t` structure.

This function clears all buffers associated with the `session` . This function will also remove session data from the session database if the session was terminated abnormally.

## 6.8 Buffered data transfer

Although `[gnutls_record_send]`, page 335 is sufficient to transmit data to the peer, when many small chunks of data are to be transmitted it is inefficient and wastes bandwidth due to the TLS record overhead. In that case it is preferable to combine the small chunks before transmission. The following functions provide that functionality.

**void gnutls\_record\_cork** (*gnutls\_session\_t session*) [Function]

*session*: is a `gnutls_session_t` structure.

If called, `gnutls_record_send()` will no longer send any records. Any sent records will be cached until `gnutls_record_uncork()` is called.

This function is safe to use with DTLS after GnuTLS 3.3.0.

**Since:** 3.1.9

**int gnutls\_record\_uncork** (*gnutls\_session\_t session*, *unsigned int flags*) [Function]

*session*: is a `gnutls_session_t` structure.

*flags*: Could be zero or `GNUTLS_RECORD_WAIT`

This resets the effect of `gnutls_record_cork()` , and flushes any pending data. If the `GNUTLS_RECORD_WAIT` flag is specified then this function will block until the data is sent or a fatal error occurs (i.e., the function will retry on `GNUTLS_E_AGAIN` and `GNUTLS_E_INTERRUPTED` ).

If the flag `GNUTLS_RECORD_WAIT` is not specified and the function is interrupted then the `GNUTLS_E_AGAIN` or `GNUTLS_E_INTERRUPTED` errors will be returned. To obtain the data left in the corked buffer use `gnutls_record_check_corked()` .

**Returns:** On success the number of transmitted data is returned, or otherwise a negative error code.

**Since:** 3.1.9

## 6.9 Handling alerts

During a TLS connection alert messages may be exchanged by the two peers. Those messages may be fatal, meaning the connection must be terminated afterwards, or warning when something needs to be reported to the peer, but without interrupting the session. The error codes `GNUTLS_E_WARNING_ALERT_RECEIVED` or `GNUTLS_E_FATAL_ALERT_RECEIVED` signal those alerts when received, and may be returned by all GnuTLS functions that receive data from the peer, being [\[gnutls\\_handshake\]](#), page 309 and [\[gnutls\\_record\\_recv\]](#), page 334. If those error codes are received the alert and its level should be logged or reported to the peer using the functions below.

`gnutls_alert_description_t gnutls_alert_get (gnutls_session_t session)` [Function]

*session*: is a `gnutls_session_t` structure.

This function will return the last alert number received. This function should be called when `GNUTLS_E_WARNING_ALERT_RECEIVED` or `GNUTLS_E_FATAL_ALERT_RECEIVED` errors are returned by a gnutls function. The peer may send alerts if he encounters an error. If no alert has been received the returned value is undefined.

**Returns:** the last alert received, a `gnutls_alert_description_t` value.

`const char * gnutls_alert_get_name (gnutls_alert_description_t alert)` [Function]

*alert*: is an alert number.

This function will return a string that describes the given alert number, or `NULL` . See `gnutls_alert_get()` .

**Returns:** string corresponding to `gnutls_alert_description_t` value.

The peer may also be warned or notified of a fatal issue by using one of the functions below. All the available alerts are listed in [\[The Alert Protocol\]](#), page 8.

`int gnutls_alert_send (gnutls_session_t session, gnutls_alert_level_t level, gnutls_alert_description_t desc)` [Function]

*session*: is a `gnutls_session_t` structure.

*level*: is the level of the alert

*desc*: is the alert description

This function will send an alert to the peer in order to inform him of something important (eg. his Certificate could not be verified). If the alert level is Fatal then the peer is expected to close the connection, otherwise he may ignore the alert and continue.

The error code of the underlying record send function will be returned, so you may also receive `GNUTLS_E_INTERRUPTED` or `GNUTLS_E_AGAIN` as well.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

```
int gnutls_error_to_alert (int err, int * level) [Function]
    err: is a negative integer
```

*level*: the alert level will be stored there

Get an alert depending on the error code returned by a gnutls function. All alerts sent by this function should be considered fatal. The only exception is when `err` is `GNUTLS_E_REHANDSHAKE`, where a warning alert should be sent to the peer indicating that no renegotiation will be performed.

If there is no mapping to a valid alert the alert to indicate internal error is returned.

**Returns:** the alert code to use for a particular error code.

## 6.10 Priority strings

The GnuTLS priority strings specify the TLS session's handshake algorithms and options in a compact, easy-to-use format. That string may contain a single initial keyword such as in [Table 6.3](#) and may be followed by additional algorithm or special keywords. Note that their description is intentionally avoiding specific algorithm details, as the priority strings are not constant between gnutls versions (they are periodically updated to account for cryptographic advances while providing compatibility with old clients and servers).

```
int [gnutls_priority_set_direct], page 326 (gnutls_session_t session, const
char * priorities, const char ** err_pos)
int [gnutls_priority_set], page 326 (gnutls_session_t session,
gnutls_priority_t priority)
```

**Keyword****Description****@KEYWORD**

Means that a compile-time specified system configuration file<sup>4</sup> will be used to expand the provided keyword. That is used to impose system-specific policies. It may be followed by additional options that will be appended to the system string (e.g., "@SYSTEM:+SRP"). The system file should have the format 'KEYWORD=VALUE', e.g., 'SYSTEM=NORMAL:-ARCFOUR-128'.

**PERFORMANCE**

All the known to be secure ciphersuites are enabled, limited to 128 bit ciphers and sorted by terms of speed performance. The message authenticity security level is of 64 bits or more, and the certificate verification profile is set to GNUTLS\_PROFILE\_LOW (80-bits).

**NORMAL**

Means all the known to be secure ciphersuites. The ciphers are sorted by security margin, although the 256-bit ciphers are included as a fallback only. The message authenticity security level is of 64 bits or more, and the certificate verification profile is set to GNUTLS\_PROFILE\_LOW (80-bits).

This priority string implicitly enables ECDHE and DHE. The ECDHE ciphersuites are placed first in the priority order, but due to compatibility issues with the DHE ciphersuites they are placed last in the priority order, after the plain RSA ciphersuites.

**LEGACY**

This sets the NORMAL settings that were used for GnuTLS 3.2.x or earlier. There is no verification profile set, and the allowed DH primes are considered weak today (but are often used by misconfigured servers).

**PFS**

Means all the known to be secure ciphersuites that support perfect forward secrecy (ECDHE and DHE). The ciphers are sorted by security margin, although the 256-bit ciphers are included as a fallback only. The message authenticity security level is of 80 bits or more, and the certificate verification profile is set to GNUTLS\_PROFILE\_LOW (80-bits). This option is available since 3.2.4 or later.

**SECURE128**

Means all known to be secure ciphersuites that offer a security level 128-bit or more. The message authenticity security level is of 80 bits or more, and the certificate verification profile is set to GNUTLS\_PROFILE\_LOW (80-bits).

**SECURE192**

Means all the known to be secure ciphersuites that offer a security level 192-bit or more. The message authenticity security level is of 128 bits or more, and the certificate verification

Unless the initial keyword is "NONE" the defaults (in preference order) are for TLS protocols TLS 1.2, TLS1.1, TLS1.0, SSL3.0; for compression NULL; for certificate types X.509. In key exchange algorithms when in NORMAL or SECURE levels the perfect forward secrecy algorithms take precedence of the other protocols. In all cases all the supported key exchange algorithms are enabled.

Note that the SECURE levels distinguish between overall security level and message authenticity security level. That is because the message authenticity security level requires the adversary to break the algorithms at real-time during the protocol run, whilst the overall security level refers to off-line adversaries (e.g. adversaries breaking the ciphertext years after it was captured).

The NONE keyword, if used, must followed by keywords specifying the algorithms and protocols to be enabled. The other initial keywords do not require, but may be followed by such keywords. All level keywords can be combined, and for example a level of "SECURE256:+SECURE128" is allowed.

The order with which every algorithm or protocol is specified is significant. Algorithms specified before others will take precedence. The supported algorithms and protocols are shown in [Table 6.4](#). To avoid collisions in order to specify a compression algorithm in the priority string you have to prefix it with "COMP-", protocol versions with "VERS-", signature algorithms with "SIGN-" and certificate types with "CTYPE-". All other algorithms don't need a prefix. Each specified keyword can be prefixed with any of the following characters.

'!' or '-' appended with an algorithm will remove this algorithm.

"+" appended with an algorithm will add this algorithm.

Type	Keywords
Ciphers	AES-128-CBC, AES-256-CBC, AES-128-GCM, CAMELLIA-128-CBC, CAMELLIA-256-CBC, ARCFOUR-128, 3DES-CBC ARCFOUR-40. Catch all name is CIPHER-ALL which will add all the algorithms from NORMAL priority.
Key exchange	RSA, DHE-RSA, DHE-DSS, SRP, SRP-RSA, SRP-DSS, PSK, DHE-PSK, ECDHE-RSA, ANON-ECDH, ANON-DH. The Catch all name is KX-ALL which will add all the algorithms from NORMAL priority. Add !DHE-RSA:!DHE-DSS to the priority string to disable DHE.
MAC	MD5, SHA1, SHA256, SHA384, AEAD (used with GCM ciphers only). All algorithms from NORMAL priority can be accessed with MAC-ALL.
Compression algorithms	COMP-NULL, COMP-DEFLATE. Catch all is COMP-ALL.
TLS versions	VERS-SSL3.0, VERS-TLS1.0, VERS-TLS1.1, VERS-TLS1.2, VERS-DTLS1.2, VERS-DTLS1.0. Catch all is VERS-TLS-ALL and VERS-DTLS-ALL.
Signature algorithms	SIGN-RSA-SHA1, SIGN-RSA-SHA224, SIGN-RSA-SHA256, SIGN-RSA-SHA384, SIGN-RSA-SHA512, SIGN-DSA-SHA1, SIGN-DSA-SHA224, SIGN-DSA-SHA256, SIGN-RSA-MD5. Catch all is SIGN-ALL. This is only valid for TLS 1.2 and later.
Elliptic curves	CURVE-SECP192R1, CURVE-SECP224R1, CURVE-SECP256R1, CURVE-SECP384R1, CURVE-SECP521R1. Catch all is CURVE-ALL.

Table 6.4: The supported algorithm keywords in priority strings.

Note that the DHE key exchange methods are generally slower<sup>5</sup> than their elliptic curves counterpart (ECDHE). Moreover the plain Diffie-Hellman key exchange requires parameters to be generated and associated with a credentials structure by the server (see [Section 6.12.3 \[Parameter generation\]](#), page 146).

The available special keywords are shown in [Table 6.5](#) and [Table 6.6](#).

<sup>5</sup> It depends on the group used. Primes with lesser bits are always faster, but also easier to break. See [Section 6.11 \[Selecting cryptographic key sizes\]](#), page 138 for the acceptable security levels.

Keyword	Description
%COMPAT	will enable compatibility mode. It might mean that violations of the protocols are allowed as long as maximum compatibility with problematic clients and servers is achieved. More specifically this string would disable TLS record random padding, tolerate packets over the maximum allowed TLS record, and add a padding to TLS Client Hello packet to prevent it being in the 256-512 range which is known to be causing issues with a commonly used firewall.
%DUMBFW	will add a private extension with bogus data that make the client hello exceed 512 bytes. This avoids a black hole behavior in some firewalls. This is a non-standard TLS extension, use with care.
%NO_EXTENSIONS	will prevent the sending of any TLS extensions in client side. Note that TLS 1.2 requires extensions to be used, as well as safe renegotiation thus this option must be used with care.
%SERVER_PRECEDENCE	The ciphersuite will be selected according to server priorities and not the client's.
%SSL3_RECORD_VERSION	will use SSL3.0 record version in client hello. This is the default.
%LATEST_RECORD_VERSION	will use the latest TLS version record version in client hello.

Table 6.5: Special priority string keywords.



Keyword	Description
%STATELESS_COMPRESSION	will disable keeping state across records when compressing. This may help to mitigate attacks when compression is used but an attacker is in control of input data. This has to be used only when the data that are possibly controlled by an attacker are placed in separate records.
%DISABLE_WILDCARDS	will disable matching wildcards when comparing hostnames in certificates.
%DISABLE_SAFE_RENEGOTIATION	will completely disable safe renegotiation completely. Do not use unless you know what you are doing.
%UNSAFE_RENEGOTIATION	will allow handshakes and re-handshakes without the safe renegotiation extension. Note that for clients this mode is insecure (you may be under attack), and for servers it will allow insecure clients to connect (which could be fooled by an attacker). Do not use unless you know what you are doing and want maximum compatibility.
%PARTIAL_RENEGOTIATION	will allow initial handshakes to proceed, but not re-handshakes. This leaves the client vulnerable to attack, and servers will be compatible with non-upgraded clients for initial handshakes. This is currently the default for clients and servers, for compatibility reasons.
%SAFE_RENEGOTIATION	will enforce safe renegotiation. Clients and servers will refuse to talk to an insecure peer. Currently this causes interoperability problems, but is required for full protection.
%VERIFY_ALLOW_SIGN_RSA_MD5	will allow RSA-MD5 signatures in certificate chains.
%VERIFY_DISABLE_CRL_CHECKS	will disable CRL or OCSP checks in the verification of the certificate chain.
%VERIFY_ALLOW_X509_V1_CA_CRT	will allow V1 CAs in chains.

Finally the ciphersuites enabled by any priority string can be listed using the `gnutls-cli` application (see [Section 9.1 \[gnutls-cli Invocation\]](#), page 231), or by using the priority functions as in [Section 7.4.3 \[Listing the ciphersuites in a priority string\]](#), page 223.

Example priority strings are:

The system imposed security level:

```
"SYSTEM"
```

The default priority without the HMAC-MD5:

```
"NORMAL:-MD5"
```

Specifying RSA with AES-128-CBC:

```
"NONE:+VERS-TLS-ALL:+MAC-ALL:+RSA:+AES-128-CBC:+SIGN-ALL:+COMP-NULL"
```

Specifying the defaults except ARCFOUR-128:

```
"NORMAL:-ARCFOUR-128"
```

Enabling the 128-bit secure ciphers, while disabling SSL 3.0 and enabling compression:

```
"SECURE128:-VERS-SSL3.0:+COMP-DEFLATE"
```

Enabling the 128-bit and 192-bit secure ciphers, while disabling all TLS versions except TLS 1.2:

```
"SECURE128:+SECURE192:-VERS-TLS-ALL:+VERS-TLS1.2"
```

## 6.11 Selecting cryptographic key sizes

Because many algorithms are involved in TLS, it is not easy to set a consistent security level. For this reason in [Table 6.7](#) we present some correspondence between key sizes of symmetric algorithms and public key algorithms based on *[ECRYPT]*. Those can be used to generate certificates with appropriate key sizes as well as select parameters for Diffie-Hellman and SRP authentication.

Security bits	RSA, DH and SRP parameter size	ECC key size	Security parameter	Description
<64	<768	<128	INSECURE	Considered to be insecure
64	768	128	VERY WEAK	Short term protection against individuals
72	1008	160	WEAK	Short term protection against small organizations
80	1024	160	LOW	Very short term protection against agencies (corresponds to ENISA legacy level)
96	1776	192	LEGACY	Legacy standard level
112	2048	224	MEDIUM	Medium-term protection
128	3072	256	HIGH	Long term protection
256	15424	512	ULTRA	Foreseeable future

Table 6.7: Key sizes and security parameters.

The first column provides a security parameter in a number of bits. This gives an indication of the number of combinations to be tried by an adversary to brute force a key. For example to test all possible keys in a 112 bit security parameter  $2^{112}$  combinations have to be tried. For today's technology this is infeasible. The next two columns correlate the security parameter with actual bit sizes of parameters for DH, RSA, SRP and ECC algorithms. A mapping to `gnutls_sec_param_t` value is given for each security parameter, on the next column, and finally a brief description of the level.

Note, however, that the values suggested here are nothing more than an educated guess that is valid today. There are no guarantees that an algorithm will remain unbreakable or that these values will remain constant in time. There could be scientific breakthroughs that cannot be predicted or total failure of the current public key systems by quantum computers. On the other hand though the cryptosystems used in TLS are selected in a conservative way and such catastrophic breakthroughs or failures are believed to be unlikely. The NIST publication SP 800-57 [*NISTSP80057*] contains a similar table.

When using GnuTLS and a decision on bit sizes for a public key algorithm is required, use of the following functions is recommended:

**unsigned int gnutls\_sec\_param\_to\_pk\_bits** [Function]

(*gnutls\_pk\_algorithm\_t algo, gnutls\_sec\_param\_t param*)

*algo*: is a public key algorithm

*param*: is a security parameter

When generating private and public key pairs a difficult question is which size of "bits" the modulus will be in RSA and the group size in DSA. The easy answer is 1024, which is also wrong. This function will convert a human understandable security parameter to an appropriate size for the specific algorithm.

**Returns:** The number of bits, or (0).

**Since:** 2.12.0

**gnutls\_sec\_param\_t gnutls\_pk\_bits\_to\_sec\_param** [Function]

(*gnutls\_pk\_algorithm\_t algo, unsigned int bits*)

*algo*: is a public key algorithm

*bits*: is the number of bits

This is the inverse of `gnutls_sec_param_to_pk_bits()`. Given an algorithm and the number of bits, it will return the security parameter. This is a rough indication.

**Returns:** The security parameter.

**Since:** 2.12.0

Those functions will convert a human understandable security parameter of `gnutls_sec_param_t` type, to a number of bits suitable for a public key algorithm.

`const char * [gnutls_sec_param_get_name], page 338` (`gnutls_sec_param_t param`)

The following functions will set the minimum acceptable group size for Diffie-Hellman and SRP authentication.

`void [gnutls_dh_set_prime_bits], page 304` (`gnutls_session_t session, unsigned int bits`)

`void [gnutls_srp_set_prime_bits], page 350` (`gnutls_session_t session, unsigned int bits`)

## 6.12 Advanced topics

### 6.12.1 Session resumption

#### Client side

To reduce time and roundtrips spent in a handshake the client can request session resumption from a server that previously shared a session with the client. For that the client has to retrieve and store the session parameters. Before establishing a new session to the same server the parameters must be re-associated with the GnuTLS session using `[gnutls_session_set_data], page 343`.

```
int [gnutls_session_get_data2], page 341 (gnutls_session_t session,
gnutls_datum_t * data)
int [gnutls_session_get_id2], page 341 (gnutls_session_t session,
gnutls_datum_t * session_id)
int [gnutls_session_set_data], page 343 (gnutls_session_t session, const void
* session_data, size_t session_data_size)
```

Keep in mind that sessions will be expired after some time, depending on the server, and a server may choose not to resume a session even when requested to. The expiration is to prevent temporal session keys from becoming long-term keys. Also note that as a client you must enable, using the priority functions, at least the algorithms used in the last session.

```
int gnutls_session_is_resumed (gnutls_session_t session) [Function]
    session: is a gnutls_session_t structure.
```

Check whether session is resumed or not.

**Returns:** non zero if this session is resumed, or a zero if this is a new session.

## Server side

In order to support resumption a server can store the session security parameters in a local database or by using session tickets (see [Section 3.6.3 \[Session tickets\], page 11](#)) to delegate storage to the client. Because session tickets might not be supported by all clients, servers could combine the two methods.

A storing server needs to specify callback functions to store, retrieve and delete session data. These can be registered with the functions below. The stored sessions in the database can be checked using [\[gnutls\\_db\\_check\\_entry\], page 297](#) for expiration.

```
void [gnutls_db_set_retrieve_function], page 299 (gnutls_session_t session,
gnutls_db_retr_func retr_func)
void [gnutls_db_set_store_function], page 299 (gnutls_session_t session,
gnutls_db_store_func store_func)
void [gnutls_db_set_ptr], page 299 (gnutls_session_t session, void * ptr)
void [gnutls_db_set_remove_function], page 299 (gnutls_session_t session,
gnutls_db_remove_func rem_func)
int [gnutls_db_check_entry], page 297 (gnutls_session_t session,
gnutls_datum_t session_entry)
```

A server utilizing tickets should generate ticket encryption and authentication keys using [\[gnutls\\_session\\_ticket\\_key\\_generate\], page 344](#). Those keys should be associated with the GnuTLS session using [\[gnutls\\_session\\_ticket\\_enable\\_server\], page 344](#), and should be rotated regularly (e.g., every few hours), to prevent them from becoming long-term keys which if revealed could be used to decrypt all previous sessions.

```
int gnutls_session_ticket_enable_server (gnutls_session_t [Function]
    session, const gnutls_datum_t * key)
```

session: is a gnutls\_session\_t structure.

key: key to encrypt session parameters.

Request that the server should attempt session resumption using SessionTicket. **key** must be initialized with `gnutls_session_ticket_key_generate()`.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, or an error code.

**Since:** 2.10.0

**int gnutls\_session\_ticket\_key\_generate** (*gnutls\_datum\_t* \* **key**) [Function]

*key*: is a pointer to a *gnutls\_datum\_t* which will contain a newly created key.

Generate a random key to encrypt security parameters within SessionTicket.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, or an error code.

**Since:** 2.10.0

**int gnutls\_session\_resumption\_requested** (*gnutls\_session\_t* **session**) [Function]

*session*: is a *gnutls\_session\_t* structure.

Check whether the client has asked for session resumption. This function is valid only on server side.

**Returns:** non zero if session resumption was asked, or a zero if not.

A server enabling both session tickets and a storage for session data would use session tickets when clients support it and the storage otherwise.

## 6.12.2 Certificate verification

In this section the functionality for additional certificate verification methods is listed. These methods are intended to be used in addition to normal PKI verification, in order to reduce the risk of a compromised CA being undetected.

### 6.12.2.1 Trust on first use

The GnuTLS library includes functionality to use an SSH-like trust on first use authentication. The available functions to store and verify public keys are listed below.

**int gnutls\_verify\_stored\_pubkey** (*const char* \* **db\_name**, [Function]  
*gnutls\_tdb\_t* **tdb**, *const char* \* **host**, *const char* \* **service**,  
*gnutls\_certificate\_type\_t* **cert\_type**, *const gnutls\_datum\_t* \* **cert**, *unsigned*  
*int* **flags**)

*db\_name*: A file specifying the stored keys (use NULL for the default)

*tdb*: A storage structure or NULL to use the default

*host*: The peer's name

*service*: non-NULL if this key is specific to a service (e.g. http)

*cert\_type*: The type of the certificate

*cert*: The raw (der) data of the certificate

*flags*: should be 0.

This function will try to verify the provided (raw or DER-encoded) certificate using a list of stored public keys. The **service** field if non-NULL should be a port number.

The **retrieve** variable if non-null specifies a custom backend for the retrieval of entries. If it is NULL then the default file backend will be used. In POSIX-like systems the file backend uses the \$HOME/.gnutls/known\_hosts file.

Note that if the custom storage backend is provided the retrieval function should return `GNUTLS_E_CERTIFICATE_KEY_MISMATCH` if the host/service pair is found but key doesn't match, `GNUTLS_E_NO_CERTIFICATE_FOUND` if no such host/service with the given key is found, and 0 if it was found. The storage function should return 0 on success.

**Returns:** If no associated public key is found then `GNUTLS_E_NO_CERTIFICATE_FOUND` will be returned. If a key is found but does not match `GNUTLS_E_CERTIFICATE_KEY_MISMATCH` is returned. On success, `GNUTLS_E_SUCCESS` (0) is returned, or a negative error value on other errors.

**Since:** 3.0.13

```
int gnutls_store_pubkey (const char * db_name, gnutls_tdb_t tdb,      [Function]
                        const char * host, const char * service, gnutls_certificate_type_t cert_type,
                        const gnutls_datum_t * cert, time_t expiration, unsigned int flags)
```

*db\_name*: A file specifying the stored keys (use NULL for the default)

*tdb*: A storage structure or NULL to use the default

*host*: The peer's name

*service*: non-NULL if this key is specific to a service (e.g. http)

*cert\_type*: The type of the certificate

*cert*: The data of the certificate

*expiration*: The expiration time (use 0 to disable expiration)

*flags*: should be 0.

This function will store the provided (raw or DER-encoded) certificate to the list of stored public keys. The key will be considered valid until the provided expiration time.

The `store` variable if non-null specifies a custom backend for the storage of entries. If it is NULL then the default file backend will be used.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.0.13

In addition to the above the [\[gnutls\\_store\\_commitment\]](#), [page 354](#) can be used to implement a key-pinning architecture as in [\[KEYPIN\]](#). This provides a way for web server to commit on a public key that is not yet active.

```
int gnutls_store_commitment (const char * db_name, gnutls_tdb_t      [Function]
                             tdb, const char * host, const char * service, gnutls_digest_algorithm_t
                             hash_algo, const gnutls_datum_t * hash, time_t expiration, unsigned int
                             flags)
```

*db\_name*: A file specifying the stored keys (use NULL for the default)

*tdb*: A storage structure or NULL to use the default

*host*: The peer's name

*service*: non-NULL if this key is specific to a service (e.g. http)

*hash\_algo*: The hash algorithm type

*hash*: The raw hash

*expiration*: The expiration time (use 0 to disable expiration)

*flags*: should be 0.

This function will store the provided hash commitment to the list of stored public keys. The key with the given hash will be considered valid until the provided expiration time.

The `store` variable if non-null specifies a custom backend for the storage of entries. If it is NULL then the default file backend will be used.

Note that this function is not thread safe with the default backend.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.0

The storage and verification functions may be used with the default text file based back-end, or another back-end may be specified. That should contain storage and retrieval functions and specified as below.

```
int [gnutls_tdb_init], page 356 (gnutls_tdb_t * tdb)
void [gnutls_tdb_deinit], page 356 (gnutls_tdb_t tdb)
void [gnutls_tdb_set_verify_func], page 357 (gnutls_tdb_t tdb,
gnutls_tdb_verify_func verify)
void [gnutls_tdb_set_store_func], page 357 (gnutls_tdb_t tdb,
gnutls_tdb_store_func store)
void [gnutls_tdb_set_store_commitment_func], page 356 (gnutls_tdb_t tdb,
gnutls_tdb_store_commitment_func cstore)
```

### 6.12.2.2 DANE verification

Since the DANE library is not included in GnuTLS it requires programs to be linked against it. This can be achieved with the following commands.

```
gcc -o foo foo.c 'pkg-config gnutls-dane --cflags --libs'
```

When a program uses the GNU autoconf system, then the following line or similar can be used to detect the presence of the library.

```
PKG_CHECK_MODULES([LIBDANE], [gnutls-dane >= 3.0.0])
```

```
AC_SUBST([LIBDANE_CFLAGS])
```

```
AC_SUBST([LIBDANE_LIBS])
```

The high level functionality provided by the DANE library is shown below.

```
int dane_verify_cert (dane_state_t s, const gnutls_datum_t * chain,      [Function]
                      unsigned chain_size, gnutls_certificate_type_t chain_type, const char *
                      hostname, const char * proto, unsigned int port, unsigned int sflags,
                      unsigned int vflags, unsigned int * verify)
```

*s*: A DANE state structure (may be NULL)

*chain*: A certificate chain

*chain\_size*: The size of the chain



*chain\_type*: The type of the certificate chain

*hostname*: The hostname associated with the chain

*proto*: The protocol of the service connecting (e.g. tcp)

*port*: The port of the service connecting (e.g. 443)

*sflags*: Flags for the the initialization of *s* (if NULL)

*vflags*: Verification flags; an OR'ed list of `dane_verify_flags_t`.

*verify*: An OR'ed list of `dane_verify_status_t`.

This function will verify the given certificate chain against the CA constraints and/or the certificate available via DANE. If no information via DANE can be obtained the flag `DANE_VERIFY_NO_DANE_INFO` is set. If a DNSSEC signature is not available for the DANE record then the verify flag `DANE_VERIFY_NO_DNSSEC_DATA` is set.

Due to the many possible options of DANE, there is no single threat model countered. When notifying the user about DANE verification results it may be better to mention: DANE verification did not reject the certificate, rather than mentioning a successful DANE verification.

Note that this function is designed to be run in addition to PKIX - certificate chain - verification. To be run independently the `DANE_VFLAG_ONLY_CHECK_EE_USAGE` flag should be specified; then the function will check whether the key of the peer matches the key advertised in the DANE entry.

If the *q* parameter is provided it will be used for caching entries.

**Returns:** On success, `DANE_E_SUCCESS` (0) is returned, otherwise a negative error value.

```
int [dane_verify_session_crt], page 553 (dane_state_t s, gnutls_session_t
session, const char * hostname, const char * proto, unsigned int port, unsigned
int sflags, unsigned int vflags, unsigned int * verify)
const char * [dane_strerror], page 551 (int error)
```

Note that the `dane_state_t` structure that is accepted by both verification functions is optional. It is required when many queries are performed to facilitate caching. The following flags are returned by the verify functions to indicate the status of the verification.

`DANE_VERIFY_CA_CONSTRAINTS_VIOLATED`

The CA constraints were violated.

`DANE_VERIFY_CERT_DIFFERS`

The certificate obtained via DNS differs.

`DANE_VERIFY_UNKNOWN_DANE_INFO`

No known DANE data was found in the DNS record.

Figure 6.2: The DANE verification status flags.

In order to generate a DANE TLSA entry to use in a DNS server you may use `danetool` (see [Section 4.2.7 \[danetool Invocation\]](#), page 70).

### 6.12.3 Parameter generation

Several TLS ciphersuites require additional parameters that need to be generated or provided by the application. The Diffie-Hellman based ciphersuites (ANON-DH or DHE), require the group parameters to be provided. Those can either be generated on the fly using [\[gnutls\\_dh\\_params\\_generate2\]](#), [page 303](#) or imported from pregenerated data using [\[gnutls\\_dh\\_params\\_import\\_pkcs3\]](#), [page 303](#). The parameters can be used in a TLS session by calling [\[gnutls\\_certificate\\_set\\_dh\\_params\]](#), [page 281](#) or [\[gnutls\\_anon\\_set\\_server\\_dh\\_params\]](#), [page 276](#) for anonymous sessions.

```
int [gnutls_dh_params_generate2], page 303 (gnutls_dh_params_t dparams,
unsigned int bits)
int [gnutls_dh_params_import_pkcs3], page 303 (gnutls_dh_params_t params,
const gnutls_datum_t * pkcs3_params, gnutls_x509_crt_fmt_t format)
void [gnutls_certificate_set_dh_params], page 281
(gnutls_certificate_credentials_t res, gnutls_dh_params_t dh_params)
void [gnutls_anon_set_server_dh_params], page 276
(gnutls_anon_server_credentials_t res, gnutls_dh_params_t dh_params)
```

Due to the time-consuming calculations required for the generation of Diffie-Hellman parameters we suggest against performing generation of them within an application. The `certtool` tool can be used to generate or export known safe values that can be stored in code or in a configuration file to provide the ability to replace. We also recommend the usage of [\[gnutls\\_sec\\_param\\_to\\_pk\\_bits\]](#), [page 338](#) (see [Section 6.11 \[Selecting cryptographic key sizes\]](#), [page 138](#)) to determine the bit size of the generated parameters.

Note that the information stored in the generated PKCS #3 structure changed with GnuTLS 3.0.9. Since that version the `privateValueLength` member of the structure is set, allowing the server utilizing the parameters to use keys of the size of the security parameter. This provides better performance in key exchange.

To allow renewal of the parameters within an application without accessing the credentials, which are a shared structure, an alternative interface is available using a callback function.

```
void gnutls_certificate_set_params_function [Function]
(gnutls_certificate_credentials_t res, gnutls_params_function * func)
res: is a gnutls_certificate_credentials_t structure
func: is the function to be called
```

This function will set a callback in order for the server to get the Diffie-Hellman or RSA parameters for certificate authentication. The callback should return `GNUTLS_E_SUCCESS (0)` on success.

### 6.12.4 Deriving keys for other applications/protocols

In several cases, after a TLS connection is established, it is desirable to derive keys to be used in another application or protocol (e.g., in an other TLS session using pre-shared keys). The following describe GnuTLS' implementation of RFC5705 to extract keys based on a session's master secret.

The API to use is [\[gnutls\\_prf\]](#), [page 322](#). The function needs to be provided with a label, and additional context data to mix in the `extra` parameter. Moreover, the API allows to switch the mix of the client and server random nonces, using the `server_random_first`

parameter. In typical uses you don't need it, so a zero value should be provided in `server_random_first`.

For example, after establishing a TLS session using [\[gnutls\\_handshake\]](#), page 309, you can obtain 32-bytes to be used as key, using this call:

```
#define MYLABEL "EXPORTER-My-protocol-name"
#define MYCONTEXT "my-protocol's-1st-session"

char out[32];
rc = gnutls_prf (session, sizeof(MYLABEL)-1, MYLABEL, 0,
                sizeof(MYCONTEXT)-1, MYCONTEXT, 32, out);
```

The output key depends on TLS' master secret, and is the same on both client and server.

If you don't want to use the RFC5705 interface and not mix in the client and server random nonces, there is a low-level TLS PRF interface called [\[gnutls\\_prf\\_raw\]](#), page 322.

### 6.12.5 Channel bindings

In user authentication protocols (e.g., EAP or SASL mechanisms) it is useful to have a unique string that identifies the secure channel that is used, to bind together the user authentication with the secure channel. This can protect against man-in-the-middle attacks in some situations. That unique string is called a "channel binding". For background and discussion see [\[RFC5056\]](#).

In GnuTLS you can extract a channel binding using the [\[gnutls\\_session\\_channel\\_binding\]](#), page 339 function. Currently only the type `GNUTLS_CB_TLS_UNIQUE` is supported, which corresponds to the `tls-unique` channel binding for TLS defined in [\[RFC5929\]](#).

The following example describes how to print the channel binding data. Note that it must be run after a successful TLS handshake.

```
{
    gnutls_datum_t cb;
    int rc;

    rc = gnutls_session_channel_binding (session,
                                         GNUTLS_CB_TLS_UNIQUE,
                                         &cb);

    if (rc)
        fprintf (stderr, "Channel binding error: %s\n",
                 gnutls_strerror (rc));
    else
    {
        size_t i;
        printf ("- Channel binding 'tls-unique': ");
        for (i = 0; i < cb.size; i++)
            printf ("%02x", cb.data[i]);
        printf ("\n");
    }
}
```

### 6.12.6 Interoperability

The TLS protocols support many ciphersuites, extensions and version numbers. As a result, few implementations are not able to properly interoperate once faced with extensions or version protocols they do not support and understand. The TLS protocol allows for a graceful downgrade to the commonly supported options, but practice shows it is not always implemented correctly.

Because there is no way to achieve maximum interoperability with broken peers without sacrificing security, GnuTLS ignores such peers by default. This might not be acceptable in cases where maximum compatibility is required. Thus we allow enabling compatibility with broken peers using priority strings (see [Section 6.10 \[Priority Strings\]](#), page 132). A conservative priority string that would disable certain TLS protocol options that are known to cause compatibility problems, is shown below.

```
NORMAL:%COMPAT
```

For broken peers that do not tolerate TLS version numbers over TLS 1.0 another priority string is:

```
NORMAL:-VERS-TLS-ALL:+VERS-TLS1.0:+VERS-SSL3.0:%COMPAT
```

This priority string will in addition to above, only enable SSL 3.0 and TLS 1.0 as protocols.

### 6.12.7 Compatibility with the OpenSSL library

To ease GnuTLS' integration with existing applications, a compatibility layer with the OpenSSL library is included in the `gnutls-openssl` library. This compatibility layer is not complete and it is not intended to completely re-implement the OpenSSL API with GnuTLS. It only provides limited source-level compatibility.

The prototypes for the compatibility functions are in the `gnutls/openssl.h` header file. The limitations imposed by the compatibility layer include:

- Error handling is not thread safe.

## 7 GnuTLS application examples

In this chapter several examples of real-world use cases are listed. The examples are simplified to promote readability and contain little or no error checking.

### 7.1 Client examples

This section contains examples of TLS and SSL clients, using GnuTLS. Note that some of the examples require functions implemented by another example.

#### 7.1.1 Simple client example with X.509 certificate support

Let's assume now that we want to create a TCP client which communicates with servers that use X.509 or OpenPGP certificate authentication. The following client is a very simple TLS client, which uses the high level verification functions for certificates, but does not support session resumption.

```
/* This example code is placed in the public domain. */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <gnutls/gnutls.h>
#include <gnutls/x509.h>
#include "examples.h"

/* A very basic TLS client, with X.509 authentication and server certificate
 * verification. Note that error checking for missing files etc. is omitted
 * for simplicity.
 */

#define MAX_BUF 1024
#define CAFILE "/etc/ssl/certs/ca-certificates.crt"
#define MSG "GET / HTTP/1.0\r\n\r\n"

extern int tcp_connect(void);
extern void tcp_close(int sd);
static int _verify_certificate_callback(gnutls_session_t session);

int main(void)
{
    int ret, sd, ii;
    gnutls_session_t session;
    char buffer[MAX_BUF + 1];
    const char *err;
```

```

gnutls_certificate_credentials_t xcred;

if (gnutls_check_version("3.1.4") == NULL) {
    fprintf(stderr, "GnuTLS 3.1.4 or later is required for this example\n");
    exit(1);
}

/* for backwards compatibility with gnutls < 3.3.0 */
gnutls_global_init();

/* X509 stuff */
gnutls_certificate_allocate_credentials(&xcred);

/* sets the trusted cas file
 */
gnutls_certificate_set_x509_trust_file(xcred, CAFILE,
                                       GNUTLS_X509_FMT_PEM);
gnutls_certificate_set_verify_function(xcred,
                                       _verify_certificate_callback);

/* If client holds a certificate it can be set using the following:
 */
gnutls_certificate_set_x509_key_file (xcred,
    "cert.pem", "key.pem",
    GNUTLS_X509_FMT_PEM);
/*

/* Initialize TLS session
 */
gnutls_init(&session, GNUTLS_CLIENT);

gnutls_session_set_ptr(session, (void *) "my_host_name");

gnutls_server_name_set(session, GNUTLS_NAME_DNS, "my_host_name",
    strlen("my_host_name"));

/* use default priorities */
gnutls_set_default_priority(session);
#endif 0

/* if more fine-graned control is required */
ret = gnutls_priority_set_direct(session,
    "NORMAL", &err);
if (ret < 0) {
    if (ret == GNUTLS_E_INVALID_REQUEST) {
        fprintf(stderr, "Syntax error at: %s\n", err);
    }
    exit(1);
}

```

```

    }
#endif

    /* put the x509 credentials to the current session
    */
    gnutls_credentials_set(session, GNUTLS_CRD_CERTIFICATE, xcred);

    /* connect to the peer
    */
    sd = tcp_connect();

    gnutls_transport_set_int(session, sd);
    gnutls_handshake_set_timeout(session,
                                GNUTLS_DEFAULT_HANDSHAKE_TIMEOUT);

    /* Perform the TLS handshake
    */
    do {
        ret = gnutls_handshake(session);
    }
    while (ret < 0 && gnutls_error_is_fatal(ret) == 0);

    if (ret < 0) {
        fprintf(stderr, "*** Handshake failed\n");
        gnutls_perror(ret);
        goto end;
    } else {
        char *desc;

        desc = gnutls_session_get_desc(session);
        printf("- Session info:  %s\n", desc);
        gnutls_free(desc);
    }

    gnutls_record_send(session, MSG, strlen(MSG));

    ret = gnutls_record_recv(session, buffer, MAX_BUF);
    if (ret == 0) {
        printf("- Peer has closed the TLS connection\n");
        goto end;
    } else if (ret < 0 && gnutls_error_is_fatal(ret) == 0) {
        fprintf(stderr, "*** Warning:  %s\n", gnutls_strerror(ret));
    } else if (ret < 0) {
        fprintf(stderr, "*** Error:  %s\n", gnutls_strerror(ret));
        goto end;
    }
}

```

```

    if (ret > 0) {
        printf("- Received %d bytes: ", ret);
        for (ii = 0; ii < ret; ii++) {
            fputc(buffer[ii], stdout);
        }
        fputs("\n", stdout);
    }

    gnutls_bye(session, GNUTLS_SHUT_RDWR);

end:

    tcp_close(sd);

    gnutls_deinit(session);

    gnutls_certificate_free_credentials(xcred);

    gnutls_global_deinit();

    return 0;
}

/* This function will verify the peer's certificate, and check
 * if the hostname matches, as well as the activation, expiration dates.
 */
static int _verify_certificate_callback(gnutls_session_t session)
{
    unsigned int status;
    int ret, type;
    const char *hostname;
    gnutls_datum_t out;

    /* read hostname */
    hostname = gnutls_session_get_ptr(session);

    /* This verification function uses the trusted CAs in the credentials
     * structure. So you must have installed one or more CA certificates.
     */

    /* The following demonstrate two different verification functions,
     * the more flexible gnutls_certificate_verify_peers(), as well
     * as the old gnutls_certificate_verify_peers3(). */
#if 1
    {
        gnutls_typed_vdata_st data[2];

```



```

    memset(data, 0, sizeof(data));

    data[0].type = GNUTLS_DT_DNS_HOSTNAME;
    data[0].data = (void*)hostname;

    data[1].type = GNUTLS_DT_KEY_PURPOSE_OID;
    data[1].data = (void*)GNUTLS_KP_TLS_WWW_SERVER;

    ret = gnutls_certificate_verify_peers(session, data, 2,
                                          &status);
}
#else
ret = gnutls_certificate_verify_peers3(session, hostname,
                                       &status);
#endif
if (ret < 0) {
    printf("Error\n");
    return GNUTLS_E_CERTIFICATE_ERROR;
}

type = gnutls_certificate_type_get(session);

ret =
    gnutls_certificate_verification_status_print(status, type,
                                                &out, 0);
if (ret < 0) {
    printf("Error\n");
    return GNUTLS_E_CERTIFICATE_ERROR;
}

printf("%s", out.data);

gnutls_free(out.data);

if (status != 0) /* Certificate is not trusted */
    return GNUTLS_E_CERTIFICATE_ERROR;

/* notify gnutls to continue handshake normally */
return 0;
}

```

### 7.1.2 Simple client example with SSH-style certificate verification

This is an alternative verification function that will use the X.509 certificate authorities for verification, but also assume an trust on first use (SSH-like) authentication system. That is the user is prompted on unknown public keys and known public keys are considered trusted.

*/\* This example code is placed in the public domain. \*/*

```

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <gnutls/gnutls.h>
#include <gnutls/x509.h>
#include "examples.h"

/* This function will verify the peer's certificate, check
 * if the hostname matches. In addition it will perform an
 * SSH-style authentication, where ultimately trusted keys
 * are only the keys that have been seen before.
 */
int _ssh_verify_certificate_callback(gnutls_session_t session)
{
    unsigned int status;
    const gnutls_datum_t *cert_list;
    unsigned int cert_list_size;
    int ret, type;
    gnutls_datum_t out;
    const char *hostname;

    /* read hostname */
    hostname = gnutls_session_get_ptr(session);

    /* This verification function uses the trusted CAs in the credentials
     * structure. So you must have installed one or more CA certificates.
     */
    ret = gnutls_certificate_verify_peers3(session, hostname, &status);
    if (ret < 0) {
        printf("Error\n");
        return GNUTLS_E_CERTIFICATE_ERROR;
    }

    type = gnutls_certificate_type_get(session);

    ret =
        gnutls_certificate_verification_status_print(status, type,
                                                    &out, 0);

    if (ret < 0) {
        printf("Error\n");
        return GNUTLS_E_CERTIFICATE_ERROR;
    }
}

```

```

printf("%s", out.data);

gnutls_free(out.data);

if (status != 0)          /* Certificate is not trusted */
    return GNUTLS_E_CERTIFICATE_ERROR;

/* Do SSH verification */
cert_list = gnutls_certificate_get_peers(session, &cert_list_size);
if (cert_list == NULL) {
    printf("No certificate was found!\n");
    return GNUTLS_E_CERTIFICATE_ERROR;
}

/* service may be obtained alternatively using getservbyport() */
ret = gnutls_verify_stored_pubkey(NULL, NULL, hostname, "https",
                                  type, &cert_list[0], 0);
if (ret == GNUTLS_E_NO_CERTIFICATE_FOUND) {
    printf("Host %s is not known.", hostname);
    if (status == 0)
        printf("Its certificate is valid for %s.\n",
               hostname);

    /* the certificate must be printed and user must be asked on
       * whether it is trustworthy.  --see gnutls_x509_cert_print() */

    /* if not trusted */
    return GNUTLS_E_CERTIFICATE_ERROR;
} else if (ret == GNUTLS_E_CERTIFICATE_KEY_MISMATCH) {
    printf
        ("Warning:  host %s is known but has another key associated.",
         hostname);
    printf
        ("It might be that the server has multiple keys, or you are under attack.\n");
    if (status == 0)
        printf("Its certificate is valid for %s.\n",
               hostname);

    /* the certificate must be printed and user must be asked on
       * whether it is trustworthy.  --see gnutls_x509_cert_print() */

    /* if not trusted */
    return GNUTLS_E_CERTIFICATE_ERROR;
} else if (ret < 0) {
    printf("gnutls_verify_stored_pubkey:  %s\n",
           gnutls_strerror(ret));
}

```

```

        return ret;
    }

    /* user trusts the key -> store it */
    if (ret != 0) {
        ret = gnutls_store_pubkey(NULL, NULL, hostname, "https",
                                   type, &cert_list[0], 0, 0);

        if (ret < 0)
            printf("gnutls_store_pubkey: %s\n",
                   gnutls_strerror(ret));
    }

    /* notify gnutls to continue handshake normally */
    return 0;
}

```

### 7.1.3 Simple client example with anonymous authentication

The simplest client using TLS is the one that doesn't do any authentication. This means no external certificates or passwords are needed to set up the connection. As could be expected, the connection is vulnerable to man-in-the-middle (active or redirection) attacks. However, the data are integrity protected and encrypted from passive eavesdroppers.

Note that due to the vulnerable nature of this method very few public servers support it.

/\* This example code is placed in the public domain. \*/

```

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <gnutls/gnutls.h>

/* A very basic TLS client, with anonymous authentication.
 */

#define MAX_BUF 1024
#define MSG "GET / HTTP/1.0\r\n\r\n"

extern int tcp_connect(void);
extern void tcp_close(int sd);

```

```
int main(void)
{
    int ret, sd, ii;
    gnutls_session_t session;
    char buffer[MAX_BUF + 1];
    gnutls_anon_client_credentials_t anoncred;
    /* Need to enable anonymous KX specifically. */

    gnutls_global_init();

    gnutls_anon_allocate_client_credentials(&anoncred);

    /* Initialize TLS session
     */
    gnutls_init(&session, GNUTLS_CLIENT);

    /* Use default priorities */
    gnutls_priority_set_direct(session,
                               "PERFORMANCE:+ANON-ECDH:+ANON-DH",
                               NULL);

    /* put the anonymous credentials to the current session
     */
    gnutls_credentials_set(session, GNUTLS_CRD_ANON, anoncred);

    /* connect to the peer
     */
    sd = tcp_connect();

    gnutls_transport_set_int(session, sd);
    gnutls_handshake_set_timeout(session,
                                 GNUTLS_DEFAULT_HANDSHAKE_TIMEOUT);

    /* Perform the TLS handshake
     */
    do {
        ret = gnutls_handshake(session);
    }
    while (ret < 0 && gnutls_error_is_fatal(ret) == 0);

    if (ret < 0) {
        fprintf(stderr, "*** Handshake failed\n");
        gnutls_perror(ret);
        goto end;
    } else {
        char *desc;
```

```

        desc = gnutls_session_get_desc(session);
        printf("- Session info:  %s\n", desc);
        gnutls_free(desc);
    }

    gnutls_record_send(session, MSG, strlen(MSG));

    ret = gnutls_record_recv(session, buffer, MAX_BUF);
    if (ret == 0) {
        printf("- Peer has closed the TLS connection\n");
        goto end;
    } else if (ret < 0 && gnutls_error_is_fatal(ret) == 0) {
        fprintf(stderr, "*** Warning:  %s\n", gnutls_strerror(ret));
    } else if (ret < 0) {
        fprintf(stderr, "*** Error:  %s\n", gnutls_strerror(ret));
        goto end;
    }

    if (ret > 0) {
        printf("- Received %d bytes:  ", ret);
        for (ii = 0; ii < ret; ii++) {
            fputc(buffer[ii], stdout);
        }
        fputs("\n", stdout);
    }

    gnutls_bye(session, GNUTLS_SHUT_RDWR);

end:

    tcp_close(sd);

    gnutls_deinit(session);

    gnutls_anon_free_client_credentials(anoncred);

    gnutls_global_deinit();

    return 0;
}

```

#### 7.1.4 Simple datagram TLS client example

This is a client that uses UDP to connect to a server. This is the DTLS equivalent to the TLS example with X.509 certificates.

```
/* This example code is placed in the public domain.  */
```

[illegible]

```

gnutls_certificate_set_verify_function(xcred,
                                       verify_certificate_callback);

/* Initialize TLS session */
gnutls_init(&session, GNUTLS_CLIENT | GNUTLS_DATAGRAM);

/* Use default priorities */
ret = gnutls_priority_set_direct(session,
                                "NORMAL", &err);
if (ret < 0) {
    if (ret == GNUTLS_E_INVALID_REQUEST) {
        fprintf(stderr, "Syntax error at: %s\n", err);
    }
    exit(1);
}

/* put the x509 credentials to the current session */
gnutls_credentials_set(session, GNUTLS_CRD_CERTIFICATE, xcred);
gnutls_server_name_set(session, GNUTLS_NAME_DNS, "my_host_name",
                       strlen("my_host_name"));

/* connect to the peer */
sd = udp_connect();

gnutls_transport_set_int(session, sd);

/* set the connection MTU */
gnutls_dtls_set_mtu(session, 1000);
/* gnutls_dtls_set_timeouts(session, 1000, 60000); */

/* Perform the TLS handshake */
do {
    ret = gnutls_handshake(session);
}
while (ret == GNUTLS_E_INTERRUPTED || ret == GNUTLS_E_AGAIN);
/* Note that DTLS may also receive GNUTLS_E_LARGE_PACKET */

if (ret < 0) {
    fprintf(stderr, "*** Handshake failed\n");
    gnutls_perror(ret);
    goto end;
} else {
    char *desc;

    desc = gnutls_session_get_desc(session);
    printf("- Session info: %s\n", desc);
    gnutls_free(desc);
}

```



```

    }

    gnutls_record_send(session, MSG, strlen(MSG));

    ret = gnutls_record_recv(session, buffer, MAX_BUF);
    if (ret == 0) {
        printf("- Peer has closed the TLS connection\n");
        goto end;
    } else if (ret < 0 && gnutls_error_is_fatal(ret) == 0) {
        fprintf(stderr, "*** Warning:  %s\n", gnutls_strerror(ret));
    } else if (ret < 0) {
        fprintf(stderr, "*** Error:  %s\n", gnutls_strerror(ret));
        goto end;
    }

    if (ret > 0) {
        printf("- Received %d bytes:  ", ret);
        for (ii = 0; ii < ret; ii++) {
            fputc(buffer[ii], stdout);
        }
        fputs("\n", stdout);
    }

    /* It is suggested not to use GNUTLS_SHUT_RDWR in DTLS
     * connections because the peer's closure message might
     * be lost */
    gnutls_bye(session, GNUTLS_SHUT_WR);

end:

    udp_close(sd);

    gnutls_deinit(session);

    gnutls_certificate_free_credentials(xcred);

    gnutls_global_deinit();

    return 0;
}

```

### 7.1.5 Obtaining session information

Most of the times it is desirable to know the security properties of the current established session. This includes the underlying ciphers and the protocols involved. That is the purpose of the following function. Note that this function will print meaningful values only if called after a successful [\[gnutls\\_handshake\]](#), [page 309](#).

```

/* This example code is placed in the public domain. */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <gnutls/gnutls.h>
#include <gnutls/x509.h>

#include "examples.h"

/* This function will print some details of the
 * given session.
 */
int print_info(gnutls_session_t session)
{
    const char *tmp;
    gnutls_credentials_type_t cred;
    gnutls_kx_algorithm_t kx;
    int dhe, ecdh;

    dhe = ecdh = 0;

    /* print the key exchange's algorithm name
     */
    kx = gnutls_kx_get(session);
    tmp = gnutls_kx_get_name(kx);
    printf("- Key Exchange:  %s\n", tmp);

    /* Check the authentication type used and switch
     * to the appropriate.
     */
    cred = gnutls_auth_get_type(session);
    switch (cred) {
    case GNUTLS_CRD_IA:
        printf("- TLS/IA session\n");
        break;

#ifdef ENABLE_SRP
    case GNUTLS_CRD_SRP:
        printf("- SRP session with username %s\n",
            gnutls_srp_server_get_username(session));
        break;
#endif
    }
}
#endif

```

```

case GNUTLS_CRD_PSK:
    /* This returns NULL in server side.
    */
    if (gnutls_psk_client_get_hint(session) != NULL)
        printf("- PSK authentication. PSK hint '%s'\n",
            gnutls_psk_client_get_hint(session));
    /* This returns NULL in client side.
    */
    if (gnutls_psk_server_get_username(session) != NULL)
        printf("- PSK authentication. Connected as '%s'\n",
            gnutls_psk_server_get_username(session));

    if (kx == GNUTLS_KX_ECDHE_PSK)
        ecdh = 1;
    else if (kx == GNUTLS_KX_DHE_PSK)
        dhe = 1;
    break;

case GNUTLS_CRD_ANON: /* anonymous authentication */

    printf("- Anonymous authentication.\n");
    if (kx == GNUTLS_KX_ANON_ECDH)
        ecdh = 1;
    else if (kx == GNUTLS_KX_ANON_DH)
        dhe = 1;
    break;

case GNUTLS_CRD_CERTIFICATE: /* certificate authentication */

    /* Check if we have been using ephemeral Diffie-Hellman.
    */
    if (kx == GNUTLS_KX_DHE_RSA || kx == GNUTLS_KX_DHE_DSS)
        dhe = 1;
    else if (kx == GNUTLS_KX_ECDHE_RSA
        || kx == GNUTLS_KX_ECDHE_ECDSA)
        ecdh = 1;

    /* if the certificate list is available, then
    * print some information about it.
    */
    print_x509_certificate_info(session);

} /* switch */

if (ecdh != 0)
    printf("- Ephemeral ECDH using curve %s\n",

```

```

        gnutls_ecc_curve_get_name(gnutls_ecc_curve_get
                                (session)));
else if (dhe != 0)
    printf("- Ephemeral DH using prime of %d bits\n",
        gnutls_dh_get_prime_bits(session));

/* print the protocol's name (ie TLS 1.0)
 */
tmp =
    gnutls_protocol_get_name(gnutls_protocol_get_version(session));
printf("- Protocol:  %s\n", tmp);

/* print the certificate type of the peer.
 * ie X.509
 */
tmp =
    gnutls_certificate_type_get_name(gnutls_certificate_type_get
                                    (session));

printf("- Certificate Type:  %s\n", tmp);

/* print the compression algorithm (if any)
 */
tmp = gnutls_compression_get_name(gnutls_compression_get(session));
printf("- Compression:  %s\n", tmp);

/* print the name of the cipher used.
 * ie 3DES.
 */
tmp = gnutls_cipher_get_name(gnutls_cipher_get(session));
printf("- Cipher:  %s\n", tmp);

/* Print the MAC algorithms name.
 * ie SHA1
 */
tmp = gnutls_mac_get_name(gnutls_mac_get(session));
printf("- MAC: %s\n", tmp);

return 0;
}

```

### 7.1.6 Using a callback to select the certificate to use

There are cases where a client holds several certificate and key pairs, and may not want to load all of them in the credentials structure. The following example demonstrates the use of the certificate selection callback.

```
/* This example code is placed in the public domain. */
```

```
#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <gnutls/gnutls.h>
#include <gnutls/x509.h>
#include <gnutls/abstract.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

/* A TLS client that loads the certificate and key.
 */

#define MAX_BUF 1024
#define MSG "GET / HTTP/1.0\r\n\r\n"

#define CERT_FILE "cert.pem"
#define KEY_FILE "key.pem"
#define CAFILE "/etc/ssl/certs/ca-certificates.crt"

extern int tcp_connect(void);
extern void tcp_close(int sd);

static int
cert_callback(gnutls_session_t session,
              const gnutls_datum_t * req_ca_rdn, int nreqs,
              const gnutls_pk_algorithm_t * sign_algos,
              int sign_algos_length, gnutls_pcert_st ** pcert,
              unsigned int *pcert_length, gnutls_privkey_t * pkey);

gnutls_pcert_st pcert;
gnutls_privkey_t key;

/* Load the certificate and the private key.
 */
static void load_keys(void)
{
    int ret;
```

```

    gnutls_datum_t data;

    ret = gnutls_load_file(CERT_FILE, &data);
    if (ret < 0) {
        fprintf(stderr, "*** Error loading certificate file.\n");
        exit(1);
    }

    ret =
        gnutls_pcert_import_x509_raw(&pcrt, &data, GNUTLS_X509_FMT_PEM,
                                     0);
    if (ret < 0) {
        fprintf(stderr, "*** Error loading certificate file:  %s\n",
                gnutls_strerror(ret));
        exit(1);
    }

    gnutls_free(data.data);

    ret = gnutls_load_file(KEY_FILE, &data);
    if (ret < 0) {
        fprintf(stderr, "*** Error loading key file.\n");
        exit(1);
    }

    gnutls_privkey_init(&key);

    ret =
        gnutls_privkey_import_x509_raw(key, &data, GNUTLS_X509_FMT_PEM,
                                       NULL, 0);
    if (ret < 0) {
        fprintf(stderr, "*** Error loading key file:  %s\n",
                gnutls_strerror(ret));
        exit(1);
    }

    gnutls_free(data.data);
}

int main(void)
{
    int ret, sd, ii;
    gnutls_session_t session;
    gnutls_priority_t priorities_cache;
    char buffer[MAX_BUF + 1];
    gnutls_certificate_credentials_t xcred;

```

```
if (gnutls_check_version("3.1.4") == NULL) {
    fprintf(stderr, "GnuTLS 3.1.4 or later is required for this example\n");
    exit(1);
}

/* for backwards compatibility with gnutls < 3.3.0 */
gnutls_global_init();

load_keys();

/* X509 stuff */
gnutls_certificate_allocate_credentials(&xcred);

/* priorities */
gnutls_priority_init(&priorities_cache,
                    "NORMAL", NULL);

/* sets the trusted cas file
 */
gnutls_certificate_set_x509_trust_file(xcred, CAFILE,
                                       GNUTLS_X509_FMT_PEM);

gnutls_certificate_set_retrieve_function2(xcred, cert_callback);

/* Initialize TLS session
 */
gnutls_init(&session, GNUTLS_CLIENT);

/* Use default priorities */
gnutls_priority_set(session, priorities_cache);

/* put the x509 credentials to the current session
 */
gnutls_credentials_set(session, GNUTLS_CRD_CERTIFICATE, xcred);

/* connect to the peer
 */
sd = tcp_connect();

gnutls_transport_set_int(session, sd);

/* Perform the TLS handshake
 */
ret = gnutls_handshake(session);

if (ret < 0) {
    fprintf(stderr, "*** Handshake failed\n");
}
```

```

        gnutls_perror(ret);
        goto end;
    } else {
        char *desc;

        desc = gnutls_session_get_desc(session);
        printf("- Session info:  %s\n", desc);
        gnutls_free(desc);
    }

    gnutls_record_send(session, MSG, strlen(MSG));

    ret = gnutls_record_recv(session, buffer, MAX_BUF);
    if (ret == 0) {
        printf("- Peer has closed the TLS connection\n");
        goto end;
    } else if (ret < 0) {
        fprintf(stderr, "*** Error:  %s\n", gnutls_strerror(ret));
        goto end;
    }

    printf("- Received %d bytes:  ", ret);
    for (ii = 0; ii < ret; ii++) {
        fputc(buffer[ii], stdout);
    }
    fputs("\n", stdout);

    gnutls_bye(session, GNUTLS_SHUT_RDWR);

end:

    tcp_close(sd);

    gnutls_deinit(session);

    gnutls_certificate_free_credentials(xcred);
    gnutls_priority_deinit(priorities_cache);

    gnutls_global_deinit();

    return 0;
}

/* This callback should be associated with a session by calling
 * gnutls_certificate_client_set_retrieve_function( session, cert_callback),

```



```

    * before a handshake.
    */

static int
cert_callback(gnutls_session_t session,
              const gnutls_datum_t * req_ca_rdn, int nreqs,
              const gnutls_pk_algorithm_t * sign_algos,
              int sign_algos_length, gnutls_pcert_st ** pcert,
              unsigned int *pcert_length, gnutls_privkey_t * pkey)
{
    char issuer_dn[256];
    int i, ret;
    size_t len;
    gnutls_certificate_type_t type;

    /* Print the server's trusted CAs
     */
    if (nreqs > 0)
        printf("- Server's trusted authorities:\n");
    else
        printf
            ("- Server did not send us any trusted authorities names.\n");

    /* print the names (if any) */
    for (i = 0; i < nreqs; i++) {
        len = sizeof(issuer_dn);
        ret = gnutls_x509_rdn_get(&req_ca_rdn[i], issuer_dn, &len);
        if (ret >= 0) {
            printf("    [%d]: ", i);
            printf("%s\n", issuer_dn);
        }
    }

    /* Select a certificate and return it.
     * The certificate must be of any of the "sign algorithms"
     * supported by the server.
     */
    type = gnutls_certificate_type_get(session);
    if (type == GNUTLS_CERT_X509) {
        *pcert_length = 1;
        *pcert = &pcert;
        *pkey = key;
    } else {
        return -1;
    }

    return 0;
}

```

```
}
```

### 7.1.7 Verifying a certificate

An example is listed below which uses the high level verification functions to verify a given certificate list.

```
/* This example code is placed in the public domain. */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <gnutls/gnutls.h>
#include <gnutls/x509.h>

#include "examples.h"

/* All the available CRLs
*/
gnutls_x509_crl_t *crl_list;
int crl_list_size;

/* All the available trusted CAs
*/
gnutls_x509_cert_t *ca_list;
int ca_list_size;

static int print_details_func(gnutls_x509_cert_t cert,
                             gnutls_x509_cert_t issuer,
                             gnutls_x509_crl_t crl,
                             unsigned int verification_output);

/* This function will try to verify the peer's certificate chain, and
 * also check if the hostname matches.
 */
void
verify_certificate_chain(const char *hostname,
                        const gnutls_datum_t * cert_chain,
                        int cert_chain_length)
{
    int i;
    gnutls_x509_trust_list_t tlist;
    gnutls_x509_cert_t *cert;
```

```

unsigned int output;

/* Initialize the trusted certificate list. This should be done
 * once on initialization. gnutls_x509_cert_list_import2() and
 * gnutls_x509_crl_list_import2() can be used to load them.
 */
gnutls_x509_trust_list_init(&tlist, 0);

gnutls_x509_trust_list_add_cas(tlist, ca_list, ca_list_size, 0);
gnutls_x509_trust_list_add_crls(tlist, crl_list, crl_list_size,
                                GNUTLS_TL_VERIFY_CRL, 0);

cert = malloc(sizeof(*cert) * cert_chain_length);

/* Import all the certificates in the chain to
 * native certificate format.
 */
for (i = 0; i < cert_chain_length; i++) {
    gnutls_x509_cert_init(&cert[i]);
    gnutls_x509_cert_import(cert[i], &cert_chain[i],
                            GNUTLS_X509_FMT_DER);
}

gnutls_x509_trust_list_verify_named_cert(tlist, cert[0], hostname,
                                           strlen(hostname),
                                           GNUTLS_VERIFY_DISABLE_CRL_CHECKS,
                                           &output,
                                           print_details_func);

/* if this certificate is not explicitly trusted verify against CAs
 */
if (output != 0) {
    gnutls_x509_trust_list_verify_cert(tlist, cert,
                                       cert_chain_length, 0,
                                       &output,
                                       print_details_func);
}

if (output & GNUTLS_CERT_INVALID) {
    fprintf(stderr, "Not trusted");

    if (output & GNUTLS_CERT_SIGNER_NOT_FOUND)
        fprintf(stderr, ": no issuer was found");
    if (output & GNUTLS_CERT_SIGNER_NOT_CA)
        fprintf(stderr, ": issuer is not a CA");
    if (output & GNUTLS_CERT_NOT_ACTIVATED)

```

```

        fprintf(stderr, ": not yet activated\n");
        if (output & GNUTLS_CERT_EXPIRED)
            fprintf(stderr, ": expired\n");

        fprintf(stderr, "\n");
    } else
        fprintf(stderr, "Trusted\n");

    /* Check if the name in the first certificate matches our destination!
    */
    if (!gnutls_x509_cert_check_hostname(cert[0], hostname)) {
        printf
            ("The certificate's owner does not match hostname '%s'\n",
             hostname);
    }

    gnutls_x509_trust_list_deinit(tlist, 1);

    return;
}

static int
print_details_func(gnutls_x509_cert_t cert,
                  gnutls_x509_cert_t issuer, gnutls_x509_crl_t crl,
                  unsigned int verification_output)
{
    char name[512];
    char issuer_name[512];
    size_t name_size;
    size_t issuer_name_size;

    issuer_name_size = sizeof(issuer_name);
    gnutls_x509_cert_get_issuer_dn(cert, issuer_name,
                                    &issuer_name_size);

    name_size = sizeof(name);
    gnutls_x509_cert_get_dn(cert, name, &name_size);

    fprintf(stdout, "\tSubject:  %s\n", name);
    fprintf(stdout, "\tIssuer:   %s\n", issuer_name);

    if (issuer != NULL) {
        issuer_name_size = sizeof(issuer_name);
        gnutls_x509_cert_get_dn(issuer, issuer_name,
                                &issuer_name_size);

        fprintf(stdout, "\tVerified against:  %s\n", issuer_name);
    }
}

```

```

    }

    if (crl != NULL) {
        issuer_name_size = sizeof(issuer_name);
        gnutls_x509_crl_get_issuer_dn(crl, issuer_name,
                                      &issuer_name_size);

        fprintf(stdout, "\tVerified against CRL of:  %s\n",
                issuer_name);
    }

    fprintf(stdout, "\tVerification output:  %x\n\n",
            verification_output);

    return 0;
}

```

### 7.1.8 Using a smart card with TLS

This example will demonstrate how to load keys and certificates from a smart-card or any other PKCS #11 token, and use it in a TLS connection.

/\* This example code is placed in the public domain. \*/

```

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <gnutls/gnutls.h>
#include <gnutls/x509.h>
#include <gnutls/pkcs11.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <getpass.h>          /* for getpass() */

/* A TLS client that loads the certificate and key.
 */

#define MAX_BUF 1024
#define MSG "GET / HTTP/1.0\r\n\r\n"

```

```

#define MIN(x,y) (((x)<(y))?(x):(y))

#define CAFILE "/etc/ssl/certs/ca-certificates.crt"

/* The URLs of the objects can be obtained
 * using p11tool --list-all --login
 */
#define KEY_URL "pkcs11:manufacturer=SomeManufacturer;object=Private%20Key" \
    ";objecttype=private;id=db%5b%3e%b5%72%33"
#define CERT_URL "pkcs11:manufacturer=SomeManufacturer;object=Certificate;" \
    "objecttype=cert;id=db%5b%3e%b5%72%33"

extern int tcp_connect(void);
extern void tcp_close(int sd);

static int
pin_callback(void *user, int attempt, const char *token_url,
             const char *token_label, unsigned int flags, char *pin,
             size_t pin_max)
{
    const char *password;
    int len;

    printf("PIN required for token '%s' with URL '%s'\n", token_label,
           token_url);
    if (flags & GNUTLS_PIN_FINAL_TRY)
        printf("*** This is the final try before locking!\n");
    if (flags & GNUTLS_PIN_COUNT_LOW)
        printf("*** Only few tries left before locking!\n");
    if (flags & GNUTLS_PIN_WRONG)
        printf("*** Wrong PIN\n");

    password = getpass("Enter pin: ");
    if (password == NULL || password[0] == 0) {
        fprintf(stderr, "No password given\n");
        exit(1);
    }

    len = MIN(pin_max - 1, strlen(password));
    memcpy(pin, password, len);
    pin[len] = 0;

    return 0;
}

int main(void)
{

```

```
int ret, sd, ii;
gnutls_session_t session;
gnutls_priority_t priorities_cache;
char buffer[MAX_BUF + 1];
gnutls_certificate_credentials_t xcred;
/* Allow connections to servers that have OpenPGP keys as well.
 */

if (gnutls_check_version("3.1.4") == NULL) {
    fprintf(stderr, "GnuTLS 3.1.4 or later is required for this example\n");
    exit(1);
}

/* for backwards compatibility with gnutls < 3.3.0 */
gnutls_global_init();

/* The PKCS11 private key operations may require PIN.
 * Register a callback. */
gnutls_pkcs11_set_pin_function(pin_callback, NULL);

/* X509 stuff */
gnutls_certificate_allocate_credentials(&xcred);

/* priorities */
gnutls_priority_init(&priorities_cache,
                    "NORMAL", NULL);

/* sets the trusted cas file
 */
gnutls_certificate_set_x509_trust_file(xcred, CAFILE,
                                       GNUTLS_X509_FMT_PEM);

gnutls_certificate_set_x509_key_file(xcred, CERT_URL, KEY_URL,
                                       GNUTLS_X509_FMT_DER);

/* Initialize TLS session
 */
gnutls_init(&session, GNUTLS_CLIENT);

/* Use default priorities */
gnutls_priority_set(session, priorities_cache);

/* put the x509 credentials to the current session
 */
gnutls_credentials_set(session, GNUTLS_CRD_CERTIFICATE, xcred);

/* connect to the peer
```

```

    */
    sd = tcp_connect();

    gnutls_transport_set_int(session, sd);

    /* Perform the TLS handshake
    */
    ret = gnutls_handshake(session);

    if (ret < 0) {
        fprintf(stderr, "*** Handshake failed\n");
        gnutls_perror(ret);
        goto end;
    } else {
        char *desc;

        desc = gnutls_session_get_desc(session);
        printf("- Session info:  %s\n", desc);
        gnutls_free(desc);
    }

    gnutls_record_send(session, MSG, strlen(MSG));

    ret = gnutls_record_recv(session, buffer, MAX_BUF);
    if (ret == 0) {
        printf("- Peer has closed the TLS connection\n");
        goto end;
    } else if (ret < 0) {
        fprintf(stderr, "*** Error:  %s\n", gnutls_strerror(ret));
        goto end;
    }

    printf("- Received %d bytes:  ", ret);
    for (ii = 0; ii < ret; ii++) {
        fputc(buffer[ii], stdout);
    }
    fputs("\n", stdout);

    gnutls_bye(session, GNUTLS_SHUT_RDWR);

end:

    tcp_close(sd);

    gnutls_deinit(session);

    gnutls_certificate_free_credentials(xcred);

```



```

    gnutls_priority_deinit(priorities_cache);

    gnutls_global_deinit();

    return 0;
}

```

### 7.1.9 Client with resume capability example

This is a modification of the simple client example. Here we demonstrate the use of session resumption. The client tries to connect once using TLS, close the connection and then try to establish a new connection using the previously negotiated data.

```

/* This example code is placed in the public domain. */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <gnutls/gnutls.h>

/* Those functions are defined in other examples.
 */
extern void check_alert(gnutls_session_t session, int ret);
extern int tcp_connect(void);
extern void tcp_close(int sd);

#define MAX_BUF 1024
#define CAFILE "/etc/ssl/certs/ca-certificates.crt"
#define MSG "GET / HTTP/1.0\r\n\r\n"

int main(void)
{
    int ret;
    int sd, ii;
    gnutls_session_t session;
    char buffer[MAX_BUF + 1];
    gnutls_certificate_credentials_t xcred;

    /* variables used in session resuming
     */
    int t;
    char *session_data = NULL;
    size_t session_data_size = 0;

```

```

gnutls_global_init();

/* X509 stuff */
gnutls_certificate_allocate_credentials(&xcred);

gnutls_certificate_set_x509_trust_file(xcred, CAFILE,
                                       GNUTLS_X509_FMT_PEM);

for (t = 0; t < 2; t++) {      /* connect 2 times to the server */

    sd = tcp_connect();

    gnutls_init(&session, GNUTLS_CLIENT);

    gnutls_priority_set_direct(session,
                               "PERFORMANCE:!ARCFOUR-128",
                               NULL);

    gnutls_credentials_set(session, GNUTLS_CRD_CERTIFICATE,
                           xcred);

    if (t > 0) {
        /* if this is not the first time we connect */
        gnutls_session_set_data(session, session_data,
                                session_data_size);
        free(session_data);
    }

    gnutls_transport_set_int(session, sd);
    gnutls_handshake_set_timeout(session,
                                  GNUTLS_DEFAULT_HANDSHAKE_TIMEOUT);

    /* Perform the TLS handshake
     */
    do {
        ret = gnutls_handshake(session);
    }
    while (ret < 0 && gnutls_error_is_fatal(ret) == 0);

    if (ret < 0) {
        fprintf(stderr, "*** Handshake failed\n");
        gnutls_perror(ret);
        goto end;
    } else {
        printf("- Handshake was completed\n");
    }
}

```

```

if (t == 0) { /* the first time we connect */
    /* get the session data size */
    gnutls_session_get_data(session, NULL,
                            &session_data_size);
    session_data = malloc(session_data_size);

    /* put session data to the session variable */
    gnutls_session_get_data(session, session_data,
                            &session_data_size);

} else { /* the second time we connect */

    /* check if we actually resumed the previous session */
    if (gnutls_session_is_resumed(session) != 0) {
        printf("- Previous session was resumed\n");
    } else {
        fprintf(stderr,
                "**** Previous session was NOT resumed\n");
    }
}

/* This function was defined in a previous example
*/
/* print_info(session); */

gnutls_record_send(session, MSG, strlen(MSG));

ret = gnutls_record_recv(session, buffer, MAX_BUF);
if (ret == 0) {
    printf("- Peer has closed the TLS connection\n");
    goto end;
} else if (ret < 0 && gnutls_error_is_fatal(ret) == 0) {
    fprintf(stderr, "**** Warning: %s\n",
            gnutls_strerror(ret));
} else if (ret < 0) {
    fprintf(stderr, "**** Error: %s\n",
            gnutls_strerror(ret));
    goto end;
}

if (ret > 0) {
    printf("- Received %d bytes: ", ret);
    for (ii = 0; ii < ret; ii++) {
        fputc(buffer[ii], stdout);
    }
    fputs("\n", stdout);
}

```

```

        gnutls_bye(session, GNUTLS_SHUT_RDWR);

    end:

        tcp_close(sd);

        gnutls_deinit(session);

}                                /* for() */

    gnutls_certificate_free_credentials(xcred);

    gnutls_global_deinit();

    return 0;
}

```

### 7.1.10 Simple client example with SRP authentication

The following client is a very simple SRP TLS client which connects to a server and authenticates using a *username* and a *password*. The server may authenticate itself using a certificate, and in that case it has to be verified.

```

/* This example code is placed in the public domain. */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <gnutls/gnutls.h>

/* Those functions are defined in other examples.
 */
extern void check_alert(gnutls_session_t session, int ret);
extern int tcp_connect(void);
extern void tcp_close(int sd);

#define MAX_BUF 1024
#define USERNAME "user"
#define PASSWORD "pass"
#define CAFILE "/etc/ssl/certs/ca-certificates.crt"
#define MSG "GET / HTTP/1.0\r\n\r\n"

int main(void)

```

[illegible]

```

/* Perform the TLS handshake
 */
do {
    ret = gnutls_handshake(session);
}
while (ret < 0 && gnutls_error_is_fatal(ret) == 0);

if (ret < 0) {
    fprintf(stderr, "*** Handshake failed\n");
    gnutls_perror(ret);
    goto end;
} else {
    char *desc;

    desc = gnutls_session_get_desc(session);
    printf("- Session info:  %s\n", desc);
    gnutls_free(desc);
}

gnutls_record_send(session, MSG, strlen(MSG));

ret = gnutls_record_recv(session, buffer, MAX_BUF);
if (gnutls_error_is_fatal(ret) != 0 || ret == 0) {
    if (ret == 0) {
        printf
            ("- Peer has closed the GnuTLS connection\n");
        goto end;
    } else {
        fprintf(stderr, "*** Error:  %s\n",
            gnutls_strerror(ret));
        goto end;
    }
} else
    check_alert(session, ret);

if (ret > 0) {
    printf("- Received %d bytes:  ", ret);
    for (ii = 0; ii < ret; ii++) {
        fputc(buffer[ii], stdout);
    }
    fputs("\n", stdout);
}

gnutls_bye(session, GNUTLS_SHUT_RDWR);

end:

tcp_close(sd);

```

```

    gnutls_deinit(session);

    gnutls_srp_free_client_credentials(srp_cred);
    gnutls_certificate_free_credentials(cert_cred);

    gnutls_global_deinit();

    return 0;
}

```

### 7.1.11 Simple client example using the C++ API

The following client is a simple example of a client utilizing the GnuTLS C++ API.

```

#include <config.h>
#include <iostream>
#include <stdexcept>
#include <gnutls/gnutls.h>
#include <gnutls/gnutlsxx.h>
#include <cstring> /* for strlen */

/* A very basic TLS client, with anonymous authentication.
 * written by Eduardo Villanueva Che.
 */

#define MAX_BUF 1024
#define SA struct sockaddr

#define CAFILE "ca.pem"
#define MSG "GET / HTTP/1.0\r\n\r\n"

extern "C"
{
    int tcp_connect(void);
    void tcp_close(int sd);
}

int main(void)
{
    int sd = -1;
    gnutls_global_init();

    try
    {

        /* Allow connections to servers that have OpenPGP keys as well.

```

```

    */
    gnutls::client_session session;

    /* X509 stuff */
    gnutls::certificate_credentials credentials;

    /* sets the trusted cas file
    */
    credentials.set_x509_trust_file(CAFILE, GNUTLS_X509_FMT_PEM);
    /* put the x509 credentials to the current session
    */
    session.set_credentials(credentials);

    /* Use default priorities */
    session.set_priority ("NORMAL", NULL);

    /* connect to the peer
    */
    sd = tcp_connect();
    session.set_transport_ptr((gnutls_transport_ptr_t) (ptrdiff_t)sd);

    /* Perform the TLS handshake
    */
    int ret = session.handshake();
    if (ret < 0)
    {
        throw std::runtime_error("Handshake failed");
    }
    else
    {
        std::cout << "- Handshake was completed" << std::endl;
    }

    session.send(MSG, strlen(MSG));
    char buffer[MAX_BUF + 1];
    ret = session.recv(buffer, MAX_BUF);
    if (ret == 0)
    {
        throw std::runtime_error("Peer has closed the TLS connection");
    }
    else if (ret < 0)
    {
        throw std::runtime_error(gnutls_strerror(ret));
    }

    std::cout << "- Received " << ret << " bytes:" << std::endl;

```



```

        std::cout.write(buffer, ret);
        std::cout << std::endl;

        session.bye(GNUTLS_SHUT_RDWR);
    }
    catch (std::exception &ex)
    {
        std::cerr << "Exception caught:  " << ex.what() << std::endl;
    }

    if (sd != -1)
        tcp_close(sd);

    gnutls_global_deinit();

    return 0;
}

```

### 7.1.12 Helper functions for TCP connections

Those helper function abstract away TCP connection handling from the other examples. It is required to build some examples.

*/\* This example code is placed in the public domain. \*/*

```

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <unistd.h>

/* tcp.c */
int tcp_connect(void);
void tcp_close(int sd);

/* Connects to the peer and returns a socket
 * descriptor.
 */
extern int tcp_connect(void)
{
    const char *PORT = "5556";

```

```

    const char *SERVER = "127.0.0.1";
    int err, sd;
    struct sockaddr_in sa;

    /* connects to server
     */
    sd = socket(AF_INET, SOCK_STREAM, 0);

    memset(&sa, '\0', sizeof(sa));
    sa.sin_family = AF_INET;
    sa.sin_port = htons(atoi(PORT));
    inet_pton(AF_INET, SERVER, &sa.sin_addr);

    err = connect(sd, (struct sockaddr *) &sa, sizeof(sa));
    if (err < 0) {
        fprintf(stderr, "Connect error\n");
        exit(1);
    }

    return sd;
}

/* closes the given socket descriptor.
 */
extern void tcp_close(int sd)
{
    shutdown(sd, SHUT_RDWR);      /* no more receptions */
    close(sd);
}

```

### 7.1.13 Helper functions for UDP connections

The UDP helper functions abstract away UDP connection handling from the other examples. It is required to build the examples using UDP.

/\* This example code is placed in the public domain. \*/

```

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>

```

```

#include <unistd.h>

/* udp.c */
int udp_connect(void);
void udp_close(int sd);

/* Connects to the peer and returns a socket
 * descriptor.
 */
extern int udp_connect(void)
{
    const char *PORT = "5557";
    const char *SERVER = "127.0.0.1";
    int err, sd, optval;
    struct sockaddr_in sa;

    /* connects to server
     */
    sd = socket(AF_INET, SOCK_DGRAM, 0);

    memset(&sa, '\0', sizeof(sa));
    sa.sin_family = AF_INET;
    sa.sin_port = htons(atoi(PORT));
    inet_pton(AF_INET, SERVER, &sa.sin_addr);

    #if defined(IP_DONTFRAG)
        optval = 1;
        setsockopt(sd, IPPROTO_IP, IP_DONTFRAG,
                   (const void *) &optval, sizeof(optval));
    #elif defined(IP_MTU_DISCOVER)
        optval = IP_PMTUDISC_DO;
        setsockopt(sd, IPPROTO_IP, IP_MTU_DISCOVER,
                   (const void *) &optval, sizeof(optval));
    #endif

    err = connect(sd, (struct sockaddr *) &sa, sizeof(sa));
    if (err < 0) {
        fprintf(stderr, "Connect error\n");
        exit(1);
    }

    return sd;
}

/* closes the given socket descriptor.
 */
extern void udp_close(int sd)

```

```
{
    close(sd);
}
```

## 7.2 Server examples

This section contains examples of TLS and SSL servers, using GnuTLS.

### 7.2.1 Echo server with X.509 authentication

This example is a very simple echo server which supports X.509 authentication.

```
/* This example code is placed in the public domain. */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
#include <gnutls/gnutls.h>

#define KEYFILE "key.pem"
#define CERTFILE "cert.pem"
#define CAFILE "/etc/ssl/certs/ca-certificates.crt"
#define CRLFILE "crl.pem"

/* The OCSP status file contains up to date information about revocation
 * of the server's certificate. That can be periodically be updated
 * using:
 * $ ocsptool --ask --load-cert your_cert.pem --load-issuer your_issuer.pem
 *           --load-signer your_issuer.pem --outfile ocsf-status.der
 */
#define OCSP_STATUS_FILE "ocsp-status.der"

/* This is a sample TLS 1.0 echo server, using X.509 authentication and
 * OCSP stapling support.
 */

#define MAX_BUF 1024
#define PORT 5556 /* listen to 5556 port */
```

[illegible]

```

if (ret < 0) {
    printf("No certificate or key were found\n");
    exit(1);
}

/* loads an OCSP status request if available */
gnutls_certificate_set_ocsp_status_request_file(x509_cred,
                                                OCSP_STATUS_FILE,
                                                0);

generate_dh_params();

gnutls_priority_init(&priority_cache,
                    "PERFORMANCE:%SERVER_PRECEDENCE", NULL);

gnutls_certificate_set_dh_params(x509_cred, dh_params);

/* Socket operations
 */
listen_sd = socket(AF_INET, SOCK_STREAM, 0);

memset(&sa_serv, '\0', sizeof(sa_serv));
sa_serv.sin_family = AF_INET;
sa_serv.sin_addr.s_addr = INADDR_ANY;
sa_serv.sin_port = htons(PORT); /* Server Port number */

setsockopt(listen_sd, SOL_SOCKET, SO_REUSEADDR, (void *) &optval,
           sizeof(int));

bind(listen_sd, (struct sockaddr *) &sa_serv, sizeof(sa_serv));

listen(listen_sd, 1024);

printf("Server ready.  Listening to port '%d'.\n\n", PORT);

client_len = sizeof(sa_cli);
for (;;) {
    gnutls_init(&session, GNUTLS_SERVER);
    gnutls_priority_set(session, priority_cache);
    gnutls_credentials_set(session, GNUTLS_CRD_CERTIFICATE,
                           x509_cred);

    /* We don't request any certificate from the client.
     * If we did we would need to verify it.  One way of
     * doing that is shown in the "Verifying a certificate"
     * example.

```

```

    */
    gnutls_certificate_server_set_request(session,
                                         GNUTLS_CERT_IGNORE);

    sd = accept(listen_sd, (struct sockaddr *) &sa_cli,
                &client_len);

    printf("- connection from %s, port %d\n",
           inet_ntop(AF_INET, &sa_cli.sin_addr, topbuf,
                     sizeof(topbuf)), ntohs(sa_cli.sin_port));

    gnutls_transport_set_int(session, sd);

    do {
        ret = gnutls_handshake(session);
    }
    while (ret < 0 && gnutls_error_is_fatal(ret) == 0);

    if (ret < 0) {
        close(sd);
        gnutls_deinit(session);
        fprintf(stderr,
                "*** Handshake has failed (%s)\n\n",
                gnutls_strerror(ret));
        continue;
    }
    printf("- Handshake was completed\n");

    /* see the Getting peer's information example */
    /* print_info(session); */

    for (;;) {
        ret = gnutls_record_recv(session, buffer, MAX_BUF);

        if (ret == 0) {
            printf
                ("\n- Peer has closed the GnuTLS connection\n");
            break;
        } else if (ret < 0
                   && gnutls_error_is_fatal(ret) == 0) {
            fprintf(stderr, "*** Warning: %s\n",
                    gnutls_strerror(ret));
        } else if (ret < 0) {
            fprintf(stderr, "\n*** Received corrupted "
                    "data(%d). Closing the connection.\n\n",
                    ret);
            break;
        }
    }

```

```

        } else if (ret > 0) {
            /* echo data back to the client
             */
            gnutls_record_send(session, buffer, ret);
        }
    }
    printf("\n");
    /* do not wait for the peer to close the connection.
     */
    gnutls_bye(session, GNUTLS_SHUT_WR);

    close(sd);
    gnutls_deinit(session);

}
close(listen_sd);

gnutls_certificate_free_credentials(x509_cred);
gnutls_priority_deinit(priority_cache);

gnutls_global_deinit();

return 0;

}

```

### 7.2.2 Echo server with OpenPGP authentication

The following example is an echo server which supports OpenPGP key authentication. You can easily combine this functionality—that is have a server that supports both X.509 and OpenPGP certificates—but we separated them to keep these examples as simple as possible.

*/\* This example code is placed in the public domain. \*/*

```

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
#include <gnutls/gnutls.h>

```



```

#include <gnutls/openpgp.h>

#define KEYFILE "secret.asc"
#define CERTFILE "public.asc"
#define RINGFILE "ring.gpg"

/* This is a sample TLS 1.0-OpenPGP echo server.
   */

#define SOCKET_ERR(err,s) if(err==-1) {perror(s);return(1);}
#define MAX_BUF 1024
#define PORT 5556 /* listen to 5556 port */

/* These are global */
gnutls_dh_params_t dh_params;

static int generate_dh_params(void)
{
    unsigned int bits = gnutls_sec_param_to_pk_bits(GNUTLS_PK_DH,
                                                    GNUTLS_SEC_PARAM_LEGACY);

    /* Generate Diffie-Hellman parameters - for use with DHE
       * kx algorithms. These should be discarded and regenerated
       * once a day, once a week or once a month. Depending on the
       * security requirements.
       */
    gnutls_dh_params_init(&dh_params);
    gnutls_dh_params_generate2(dh_params, bits);

    return 0;
}

int main(void)
{
    int err, listen_sd;
    int sd, ret;
    struct sockaddr_in sa_serv;
    struct sockaddr_in sa_cli;
    socklen_t client_len;
    char topbuf[512];
    gnutls_session_t session;
    gnutls_certificate_credentials_t cred;
    char buffer[MAX_BUF + 1];
    int optval = 1;
    char name[256];

```

[illegible]

```

/* request client certificate if any.
 */
gnutls_certificate_server_set_request(session,
                                     GNUTLS_CERT_REQUEST);

sd = accept(listen_sd, (struct sockaddr *) &sa_cli,
            &client_len);

printf("- connection from %s, port %d\n",
       inet_ntop(AF_INET, &sa_cli.sin_addr, topbuf,
               sizeof(topbuf)), ntohs(sa_cli.sin_port));

gnutls_transport_set_int(session, sd);
ret = gnutls_handshake(session);
if (ret < 0) {
    close(sd);
    gnutls_deinit(session);
    fprintf(stderr,
           "*** Handshake has failed (%s)\n\n",
           gnutls_strerror(ret));
    continue;
}
printf("- Handshake was completed\n");

/* see the Getting peer's information example */
/* print_info(session); */

for (;;) {
    ret = gnutls_record_recv(session, buffer, MAX_BUF);

    if (ret == 0) {
        printf
            ("\n- Peer has closed the GnuTLS connection\n");
        break;
    } else if (ret < 0
               && gnutls_error_is_fatal(ret) == 0) {
        fprintf(stderr, "*** Warning: %s\n",
               gnutls_strerror(ret));
    } else if (ret < 0) {
        fprintf(stderr, "\n*** Received corrupted "
               "data(%d). Closing the connection.\n\n",
               ret);
        break;
    } else if (ret > 0) {
        /* echo data back to the client
        */

```

```

        gnutls_record_send(session, buffer, ret);
    }
}
printf("\n");
/* do not wait for the peer to close the connection.
   */
gnutls_bye(session, GNUTLS_SHUT_WR);

close(sd);
gnutls_deinit(session);

}
close(listen_sd);

gnutls_certificate_free_credentials(cred);

gnutls_global_deinit();

return 0;

}

```

### 7.2.3 Echo server with SRP authentication

This is a server which supports SRP authentication. It is also possible to combine this functionality with a certificate server. Here it is separate for simplicity.

*/\* This example code is placed in the public domain. \*/*

```

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
#include <gnutls/gnutls.h>

#define SRP_PASSWD "tpasswd"
#define SRP_PASSWD_CONF "tpasswd.conf"

#define KEYFILE "key.pem"

```

[illegible]

```

/* TCP socket operations
 */
listen_sd = socket(AF_INET, SOCK_STREAM, 0);
SOCKET_ERR(listen_sd, "socket");

memset(&sa_serv, '\0', sizeof(sa_serv));
sa_serv.sin_family = AF_INET;
sa_serv.sin_addr.s_addr = INADDR_ANY;
sa_serv.sin_port = htons(PORT); /* Server Port number */

setsockopt(listen_sd, SOL_SOCKET, SO_REUSEADDR, (void *) &optval,
           sizeof(int));

err =
    bind(listen_sd, (struct sockaddr *) &sa_serv, sizeof(sa_serv));
SOCKET_ERR(err, "bind");
err = listen(listen_sd, 1024);
SOCKET_ERR(err, "listen");

printf("%s ready.  Listening to port '%d'.\n\n", name, PORT);

client_len = sizeof(sa_cli);
for (;;) {
    gnutls_init(&session, GNUTLS_SERVER);
    gnutls_priority_set_direct(session,
                              "NORMAL"
                              ":-KX-ALL:+SRP:+SRP-DSS:+SRP-RSA",
                              NULL);
    gnutls_credentials_set(session, GNUTLS_CRD_SRP, srp_cred);
    /* for the certificate authenticated ciphersuites.
     */
    gnutls_credentials_set(session, GNUTLS_CRD_CERTIFICATE,
                          cert_cred);

    /* We don't request any certificate from the client.
     * If we did we would need to verify it.  One way of
     * doing that is shown in the "Verifying a certificate"
     * example.
     */
    gnutls_certificate_server_set_request(session,
                                         GNUTLS_CERT_IGNORE);

    sd = accept(listen_sd, (struct sockaddr *) &sa_cli,
                &client_len);

    printf("- connection from %s, port %d\n",
           inet_ntop(AF_INET, &sa_cli.sin_addr, topbuf,

```

```

        sizeof(topbuf)), ntohs(sa_cli.sin_port));

gnutls_transport_set_int(session, sd);

do {
    ret = gnutls_handshake(session);
}
while (ret < 0 && gnutls_error_is_fatal(ret) == 0);

if (ret < 0) {
    close(sd);
    gnutls_deinit(session);
    fprintf(stderr,
        "*** Handshake has failed (%s)\n\n",
        gnutls_strerror(ret));
    continue;
}
printf("- Handshake was completed\n");
printf("- User %s was connected\n",
    gnutls_srp_server_get_username(session));

/* print_info(session); */

for (;;) {
    ret = gnutls_record_recv(session, buffer, MAX_BUF);

    if (ret == 0) {
        printf
            ("\n- Peer has closed the GnuTLS connection\n");
        break;
    } else if (ret < 0
        && gnutls_error_is_fatal(ret) == 0) {
        fprintf(stderr, "*** Warning: %s\n",
            gnutls_strerror(ret));
    } else if (ret < 0) {
        fprintf(stderr, "\n*** Received corrupted "
            "data(%d). Closing the connection.\n\n",
            ret);
        break;
    } else if (ret > 0) {
        /* echo data back to the client
        */
        gnutls_record_send(session, buffer, ret);
    }
}

printf("\n");
/* do not wait for the peer to close the connection. */

```

```

        gnutls_bye(session, GNUTLS_SHUT_WR);

        close(sd);
        gnutls_deinit(session);

    }
    close(listen_sd);

    gnutls_srp_free_server_credentials(srp_cred);
    gnutls_certificate_free_credentials(cert_cred);

    gnutls_global_deinit();

    return 0;

}

```

#### 7.2.4 Echo server with anonymous authentication

This example server supports anonymous authentication, and could be used to serve the example client for anonymous authentication.

```

/* This example code is placed in the public domain. */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
#include <gnutls/gnutls.h>

/* This is a sample TLS 1.0 echo server, for anonymous authentication only.
   */

#define SOCKET_ERR(err,s) if(err==-1) {perror(s);return(1);}
#define MAX_BUF 1024
#define PORT 5556 /* listen to 5556 port */

/* These are global */

```



```
static gnutls_dh_params_t dh_params;

static int generate_dh_params(void)
{
    unsigned int bits = gnutls_sec_param_to_pk_bits(GNUTLS_PK_DH,
                                                    GNUTLS_SEC_PARAM_LEGACY);
    /* Generate Diffie-Hellman parameters - for use with DHE
     * kx algorithms. These should be discarded and regenerated
     * once a day, once a week or once a month. Depending on the
     * security requirements.
     */
    gnutls_dh_params_init(&dh_params);
    gnutls_dh_params_generate2(dh_params, bits);

    return 0;
}

int main(void)
{
    int err, listen_sd;
    int sd, ret;
    struct sockaddr_in sa_serv;
    struct sockaddr_in sa_cli;
    socklen_t client_len;
    char topbuf[512];
    gnutls_session_t session;
    gnutls_anon_server_credentials_t anoncred;
    char buffer[MAX_BUF + 1];
    int optval = 1;

    if (gnutls_check_version("3.1.4") == NULL) {
        fprintf(stderr, "GnuTLS 3.1.4 or later is required for this example\n");
        exit(1);
    }

    /* for backwards compatibility with gnutls < 3.3.0 */
    gnutls_global_init();

    gnutls_anon_allocate_server_credentials(&anoncred);

    generate_dh_params();

    gnutls_anon_set_server_dh_params(anoncred, dh_params);

    /* Socket operations
     */
    listen_sd = socket(AF_INET, SOCK_STREAM, 0);
```

```

SOCKET_ERR(listen_sd, "socket");

memset(&sa_serv, '\0', sizeof(sa_serv));
sa_serv.sin_family = AF_INET;
sa_serv.sin_addr.s_addr = INADDR_ANY;
sa_serv.sin_port = htons(PORT); /* Server Port number */

setsockopt(listen_sd, SOL_SOCKET, SO_REUSEADDR, (void *) &optval,
           sizeof(int));

err =
    bind(listen_sd, (struct sockaddr *) &sa_serv, sizeof(sa_serv));
SOCKET_ERR(err, "bind");
err = listen(listen_sd, 1024);
SOCKET_ERR(err, "listen");

printf("Server ready.  Listening to port '%d'.\n\n", PORT);

client_len = sizeof(sa_cli);
for (;;) {
    gnutls_init(&session, GNUTLS_SERVER);
    gnutls_priority_set_direct(session,
                              "NORMAL::+ANON-ECDH:+ANON-DH",
                              NULL);
    gnutls_credentials_set(session, GNUTLS_CRD_ANON, anoncred);

    sd = accept(listen_sd, (struct sockaddr *) &sa_cli,
                &client_len);

    printf("- connection from %s, port %d\n",
           inet_ntop(AF_INET, &sa_cli.sin_addr, topbuf,
                     sizeof(topbuf)), ntohs(sa_cli.sin_port));

    gnutls_transport_set_int(session, sd);

    do {
        ret = gnutls_handshake(session);
    }
    while (ret < 0 && gnutls_error_is_fatal(ret) == 0);

    if (ret < 0) {
        close(sd);
        gnutls_deinit(session);
        fprintf(stderr,
               "*** Handshake has failed (%s)\n\n",
               gnutls_strerror(ret));
        continue;
    }
}

```

```

    }
    printf("- Handshake was completed\n");

    /* see the Getting peer's information example */
    /* print_info(session); */

    for (;;) {
        ret = gnutls_record_recv(session, buffer, MAX_BUF);

        if (ret == 0) {
            printf
                ("\n- Peer has closed the GnuTLS connection\n");
            break;
        } else if (ret < 0)
            && gnutls_error_is_fatal(ret) == 0) {
            fprintf(stderr, "*** Warning:  %s\n",
                    gnutls_strerror(ret));
        } else if (ret < 0) {
            fprintf(stderr, "\n*** Received corrupted "
                    "data(%d).  Closing the connection.\n\n",
                    ret);
            break;
        } else if (ret > 0) {
            /* echo data back to the client
             */
            gnutls_record_send(session, buffer, ret);
        }
    }
    printf("\n");
    /* do not wait for the peer to close the connection.
     */
    gnutls_bye(session, GNUTLS_SHUT_WR);

    close(sd);
    gnutls_deinit(session);

}
close(listen_sd);

gnutls_anon_free_server_credentials(anoncred);

gnutls_global_deinit();

return 0;

}

```

### 7.2.5 DTLS echo server with X.509 authentication

This example is a very simple echo server using Datagram TLS and X.509 authentication.

```
/* This example code is placed in the public domain. */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/select.h>
#include <netdb.h>
#include <string.h>
#include <unistd.h>
#include <gnutls/gnutls.h>
#include <gnutls/dtls.h>

#define KEYFILE "key.pem"
#define CERTFILE "cert.pem"
#define CAFILE "/etc/ssl/certs/ca-certificates.crt"
#define CRLFILE "crl.pem"

/* This is a sample DTLS echo server, using X.509 authentication.
 * Note that error checking is minimal to simplify the example.
 */

#define MAX_BUFFER 1024
#define PORT 5557

typedef struct {
    gnutls_session_t session;
    int fd;
    struct sockaddr *cli_addr;
    socklen_t cli_addr_size;
} priv_data_st;

static int pull_timeout_func(gnutls_transport_ptr_t ptr, unsigned int ms);
static ssize_t push_func(gnutls_transport_ptr_t p, const void *data,
                        size_t size);
static ssize_t pull_func(gnutls_transport_ptr_t p, void *data,
                        size_t size);
```

```

static const char *human_addr(const struct sockaddr *sa, socklen_t salen,
                              char *buf, size_t buflen);
static int wait_for_connection(int fd);
static int generate_dh_params(void);

/* Use global credentials and parameters to simplify
 * the example. */
static gnutls_certificate_credentials_t x509_cred;
static gnutls_priority_t priority_cache;
static gnutls_dh_params_t dh_params;

int main(void)
{
    int listen_sd;
    int sock, ret;
    struct sockaddr_in sa_serv;
    struct sockaddr_in cli_addr;
    socklen_t cli_addr_size;
    gnutls_session_t session;
    char buffer[MAX_BUFFER];
    priv_data_st priv;
    gnutls_datum_t cookie_key;
    gnutls_dtls_prestate_st prestate;
    int mtu = 1400;
    unsigned char sequence[8];

    /* this must be called once in the program
     */
    gnutls_global_init();

    gnutls_certificate_allocate_credentials(&x509_cred);
    gnutls_certificate_set_x509_trust_file(x509_cred, CAFILE,
                                          GNUTLS_X509_FMT_PEM);

    gnutls_certificate_set_x509_crl_file(x509_cred, CRLFILE,
                                          GNUTLS_X509_FMT_PEM);

    ret =
        gnutls_certificate_set_x509_key_file(x509_cred, CERTFILE,
                                             KEYFILE,
                                             GNUTLS_X509_FMT_PEM);

    if (ret < 0) {
        printf("No certificate or key were found\n");
        exit(1);
    }

    generate_dh_params();

```

```

gnutls_certificate_set_dh_params(x509_cred, dh_params);

gnutls_priority_init(&priority_cache,
                    "PERFORMANCE:-VERS-TLS-ALL:+VERS-DTLS1.0:%SERVER_PRECEDENCE",
                    NULL);

gnutls_key_generate(&cookie_key, GNUTLS_COOKIE_KEY_SIZE);

/* Socket operations
 */
listen_sd = socket(AF_INET, SOCK_DGRAM, 0);

memset(&sa_serv, '\0', sizeof(sa_serv));
sa_serv.sin_family = AF_INET;
sa_serv.sin_addr.s_addr = INADDR_ANY;
sa_serv.sin_port = htons(PORT);

{
    /* DTLS requires the IP don't fragment (DF) bit to be set */
#ifdef IP_DONTFRAG
    int optval = 1;
    setsockopt(listen_sd, IPPROTO_IP, IP_DONTFRAG,
               (const void *) &optval, sizeof(optval));
#elif defined(IP_MTU_DISCOVER)
    int optval = IP_PMTUDISC_DO;
    setsockopt(listen_sd, IPPROTO_IP, IP_MTU_DISCOVER,
               (const void *) &optval, sizeof(optval));
#endif
}

bind(listen_sd, (struct sockaddr *) &sa_serv, sizeof(sa_serv));

printf("UDP server ready. Listening to port '%d'.\n\n", PORT);

for (;;) {
    printf("Waiting for connection...\n");
    sock = wait_for_connection(listen_sd);
    if (sock < 0)
        continue;

    cli_addr_size = sizeof(cli_addr);
    ret = recvfrom(sock, buffer, sizeof(buffer), MSG_PEEK,
                  (struct sockaddr *) &cli_addr,
                  &cli_addr_size);
    if (ret > 0) {
        memset(&prestate, 0, sizeof(prestate));
        ret =

```

```

        gnutls_dtls_cookie_verify(&cookie_key,
                                   &cli_addr,
                                   sizeof(cli_addr),
                                   buffer, ret,
                                   &prestate);
    if (ret < 0) { /* cookie not valid */
        priv_data_st s;

        memset(&s, 0, sizeof(s));
        s.fd = sock;
        s.cli_addr = (void *) &cli_addr;
        s.cli_addr_size = sizeof(cli_addr);

        printf
            ("Sending hello verify request to %s\n",
             human_addr((struct sockaddr *)
                        &cli_addr,
                        sizeof(cli_addr), buffer,
                        sizeof(buffer)));

        gnutls_dtls_cookie_send(&cookie_key,
                                 &cli_addr,
                                 sizeof(cli_addr),
                                 &prestate,
                                 (gnutls_transport_ptr_t)
                                 &s, push_func);

        /* discard peeked data */
        recvfrom(sock, buffer, sizeof(buffer), 0,
                  (struct sockaddr *) &cli_addr,
                  &cli_addr_size);
        usleep(100);
        continue;
    }
    printf("Accepted connection from %s\n",
           human_addr((struct sockaddr *)
                      &cli_addr, sizeof(cli_addr),
                      buffer, sizeof(buffer)));
} else
    continue;

gnutls_init(&session, GNUTLS_SERVER | GNUTLS_DATAGRAM);
gnutls_priority_set(session, priority_cache);
gnutls_credentials_set(session, GNUTLS_CRD_CERTIFICATE,
                       x509_cred);

gnutls_dtls_prestate_set(session, &prestate);

```

```

gnutls_dtls_set_mtu(session, mtu);

priv.session = session;
priv.fd = sock;
priv.cli_addr = (struct sockaddr *) &cli_addr;
priv.cli_addr_size = sizeof(cli_addr);

gnutls_transport_set_ptr(session, &priv);
gnutls_transport_set_push_function(session, push_func);
gnutls_transport_set_pull_function(session, pull_func);
gnutls_transport_set_pull_timeout_function(session,
                                           pull_timeout_func);

do {
    ret = gnutls_handshake(session);
}
while (ret == GNUTLS_E_INTERRUPTED
      || ret == GNUTLS_E_AGAIN);
/* Note that DTLS may also receive GNUTLS_E_LARGE_PACKET.
 * In that case the MTU should be adjusted.
 */

if (ret < 0) {
    fprintf(stderr, "Error in handshake(): %s\n",
            gnutls_strerror(ret));
    gnutls_deinit(session);
    continue;
}

printf("- Handshake was completed\n");

for (;;) {
    do {
        ret =
            gnutls_record_recv_seq(session, buffer,
                                   MAX_BUFFER,
                                   sequence);
    }
    while (ret == GNUTLS_E_AGAIN
          || ret == GNUTLS_E_INTERRUPTED);

    if (ret < 0 && gnutls_error_is_fatal(ret) == 0) {
        fprintf(stderr, "*** Warning: %s\n",
                gnutls_strerror(ret));
        continue;
    } else if (ret < 0) {
        fprintf(stderr, "Error in recv(): %s\n",

```



```

                                gnutls_strerror(ret));
                                break;
                                }

                                if (ret == 0) {
                                    printf("EOF\n\n");
                                    break;
                                }

                                buffer[ret] = 0;
                                printf
                                    ("received[%.2x%.2x%.2x%.2x%.2x%.2x%.2x]:  %s\n",
                                     sequence[0], sequence[1], sequence[2],
                                     sequence[3], sequence[4], sequence[5],
                                     sequence[6], sequence[7], buffer);

                                /* reply back */
                                ret = gnutls_record_send(session, buffer, ret);
                                if (ret < 0) {
                                    fprintf(stderr, "Error in send():  %s\n",
                                                gnutls_strerror(ret));
                                    break;
                                }
                                }

                                gnutls_bye(session, GNUTLS_SHUT_WR);
                                gnutls_deinit(session);

                                }
                                close(listen_sd);

                                gnutls_certificate_free_credentials(x509_cred);
                                gnutls_priority_deinit(priority_cache);

                                gnutls_global_deinit();

                                return 0;

                                }

static int wait_for_connection(int fd)
{
    fd_set rd, wr;
    int n;

    FD_ZERO(&rd);
    FD_ZERO(&wr);

```

```

    FD_SET(fd, &rd);

    /* waiting part */
    n = select(fd + 1, &rd, &wr, NULL, NULL);
    if (n == -1 && errno == EINTR)
        return -1;
    if (n < 0) {
        perror("select()");
        exit(1);
    }

    return fd;
}

/* Wait for data to be received within a timeout period in milliseconds
*/
static int pull_timeout_func(gnutls_transport_ptr_t ptr, unsigned int ms)
{
    fd_set rfd;
    struct timeval tv;
    priv_data_st *priv = ptr;
    struct sockaddr_in cli_addr;
    socklen_t cli_addr_size;
    int ret;
    char c;

    FD_ZERO(&rfd);
    FD_SET(priv->fd, &rfd);

    tv.tv_sec = 0;
    tv.tv_usec = ms * 1000;

    while (tv.tv_usec >= 1000000) {
        tv.tv_usec -= 1000000;
        tv.tv_sec++;
    }

    ret = select(priv->fd + 1, &rfd, NULL, NULL, &tv);

    if (ret <= 0)
        return ret;

    /* only report ok if the next message is from the peer we expect
    * from
    */
    cli_addr_size = sizeof(cli_addr);

```

```

    ret =
        recvfrom(priv->fd, &c, 1, MSG_PEEK,
                  (struct sockaddr *) &cli_addr, &cli_addr_size);
    if (ret > 0) {
        if (cli_addr_size == priv->cli_addr_size
            && memcmp(&cli_addr, priv->cli_addr,
                      sizeof(cli_addr)) == 0)
            return 1;
    }

    return 0;
}

static ssize_t
push_func(gnutls_transport_ptr_t p, const void *data, size_t size)
{
    priv_data_st *priv = p;

    return sendto(priv->fd, data, size, 0, priv->cli_addr,
                  priv->cli_addr_size);
}

static ssize_t pull_func(gnutls_transport_ptr_t p, void *data, size_t size)
{
    priv_data_st *priv = p;
    struct sockaddr_in cli_addr;
    socklen_t cli_addr_size;
    char buffer[64];
    int ret;

    cli_addr_size = sizeof(cli_addr);
    ret =
        recvfrom(priv->fd, data, size, 0,
                  (struct sockaddr *) &cli_addr, &cli_addr_size);
    if (ret == -1)
        return ret;

    if (cli_addr_size == priv->cli_addr_size
        && memcmp(&cli_addr, priv->cli_addr, sizeof(cli_addr)) == 0)
        return ret;

    printf("Denied connection from %s\n",
           human_addr((struct sockaddr *)
                      &cli_addr, sizeof(cli_addr), buffer,
                      sizeof(buffer)));

    gnutls_transport_set_errno(priv->session, EAGAIN);
}

```

```

        return -1;
    }

static const char *human_addr(const struct sockaddr *sa, socklen_t salen,
                              char *buf, size_t buflen)
{
    const char *save_buf = buf;
    size_t l;

    if (!buf || !buflen)
        return NULL;

    *buf = '\\0';

    switch (sa->sa_family) {
#ifdef HAVE_IPV6
        case AF_INET6:
            snprintf(buf, buflen, "IPv6 ");
            break;
#endif

        case AF_INET:
            snprintf(buf, buflen, "IPv4 ");
            break;
    }

    l = strlen(buf);
    buf += l;
    buflen -= l;

    if (getnameinfo(sa, salen, buf, buflen, NULL, 0, NI_NUMERICHOST) !=
        0)
        return NULL;

    l = strlen(buf);
    buf += l;
    buflen -= l;

    strncat(buf, " port ", buflen);

    l = strlen(buf);
    buf += l;
    buflen -= l;

    if (getnameinfo(sa, salen, NULL, 0, buf, buflen, NI_NUMERICSERV) !=
        0)
        return NULL;
}

```

```

        return save_buf;
    }

static int generate_dh_params(void)
{
    int bits = gnutls_sec_param_to_pk_bits(GNUTLS_PK_DH,
                                           GNUTLS_SEC_PARAM_LEGACY);

    /* Generate Diffie-Hellman parameters - for use with DHE
     * kx algorithms. When short bit length is used, it might
     * be wise to regenerate parameters often.
     */
    gnutls_dh_params_init(&dh_params);
    gnutls_dh_params_generate2(dh_params, bits);

    return 0;
}

```

## 7.3 OCSP example

### Generate OCSP request

A small tool to generate OCSP requests.

```

/* This example code is placed in the public domain. */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <gnutls/gnutls.h>
#include <gnutls/crypto.h>
#include <gnutls/ocsp.h>
#ifdef NO_LIBCURL
#include <curl/curl.h>
#endif
#include "read-file.h"

size_t get_data(void *buffer, size_t size, size_t nmemb, void *userp);
static gnutls_x509_crt_t load_cert(const char *cert_file);
static void _response_info(const gnutls_datum_t * data);
static void
_generate_request(gnutls_datum_t * rdata, gnutls_x509_crt_t cert,
                 gnutls_x509_crt_t issuer, gnutls_datum_t *nonce);

```

```

static int
_verify_response(gnutls_datum_t * data, gnutls_x509_crt_t cert,
                 gnutls_x509_crt_t signer, gnutls_datum_t *nonce);

/* This program queries an OCSP server.
   It expects three files.  argv[1] containing the certificate to
   be checked, argv[2] holding the issuer for this certificate,
   and argv[3] holding a trusted certificate to verify OCSP's response.
   argv[4] is optional and should hold the server host name.

   For simplicity the libcurl library is used.
*/

int main(int argc, char *argv[])
{
    gnutls_datum_t ud, tmp;
    int ret;
    gnutls_datum_t req;
    gnutls_x509_crt_t cert, issuer, signer;
#ifdef NO_LIBCURL
    CURL *handle;
    struct curl_slist *headers = NULL;
#endif
    int v, seq;
    const char *cert_file = argv[1];
    const char *issuer_file = argv[2];
    const char *signer_file = argv[3];
    char *hostname = NULL;
    unsigned char noncebuf[23];
    gnutls_datum_t nonce = { noncebuf, sizeof(noncebuf) };

    gnutls_global_init();

    if (argc > 4)
        hostname = argv[4];

    ret = gnutls_rnd(GNUTLS_RND_NONCE, nonce.data, nonce.size);
    if (ret < 0)
        exit(1);

    cert = load_cert(cert_file);
    issuer = load_cert(issuer_file);
    signer = load_cert(signer_file);

    if (hostname == NULL) {
        for (seq = 0;; seq++) {

```

```

        ret =
            gnutls_x509_cert_get_authority_info_access(cert,
                                                    seq,
                                                    GNUTLS_IA_OCSP_URI,
                                                    &tmp,
                                                    NULL);

        if (ret == GNUTLS_E_UNKNOWN_ALGORITHM)
            continue;
        if (ret == GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE) {
            fprintf(stderr,
                    "No URI was found in the certificate.\n");
            exit(1);
        }
        if (ret < 0) {
            fprintf(stderr, "error: %s\n",
                    gnutls_strerror(ret));
            exit(1);
        }

        printf("CA issuers URI: %.*s\n", tmp.size,
              tmp.data);

        hostname = malloc(tmp.size + 1);
        memcpy(hostname, tmp.data, tmp.size);
        hostname[tmp.size] = 0;

        gnutls_free(tmp.data);
        break;
    }

}

/* Note that the OCSP servers hostname might be available
 * using gnutls_x509_cert_get_authority_info_access() in the issuer's
 * certificate */

memset(&ud, 0, sizeof(ud));
fprintf(stderr, "Connecting to %s\n", hostname);

_generate_request(&req, cert, issuer, &nonce);

#ifdef NO_LIBCURL
curl_global_init(CURL_GLOBAL_ALL);

handle = curl_easy_init();
if (handle == NULL)
    exit(1);

```

```

    headers =
        curl_slist_append(headers,
            "Content-Type:  application/ocsp-request");

    curl_easy_setopt(handle, CURLOPT_HTTPHEADER, headers);
    curl_easy_setopt(handle, CURLOPT_POSTFIELDS, (void *) req.data);
    curl_easy_setopt(handle, CURLOPT_POSTFIELDSIZE, req.size);
    curl_easy_setopt(handle, CURLOPT_URL, hostname);
    curl_easy_setopt(handle, CURLOPT_WRITEFUNCTION, get_data);
    curl_easy_setopt(handle, CURLOPT_WRITEDATA, &ud);

    ret = curl_easy_perform(handle);
    if (ret != 0) {
        fprintf(stderr, "curl[%d] error %d\n", __LINE__, ret);
        exit(1);
    }

    curl_easy_cleanup(handle);
#endif

    _response_info(&ud);

    v = _verify_response(&ud, cert, signer, &nonce);

    gnutls_x509_cert_deinit(cert);
    gnutls_x509_cert_deinit(issuer);
    gnutls_x509_cert_deinit(signer);
    gnutls_global_deinit();

    return v;
}

static void _response_info(const gnutls_datum_t * data)
{
    gnutls_ocsp_resp_t resp;
    int ret;
    gnutls_datum buf;

    ret = gnutls_ocsp_resp_init(&resp);
    if (ret < 0)
        exit(1);

    ret = gnutls_ocsp_resp_import(resp, data);
    if (ret < 0)
        exit(1);
}

```



```

    ret = gnutls_ocsp_resp_print(resp, GNUTLS_OCSP_PRINT_FULL, &buf);
    if (ret != 0)
        exit(1);

    printf("%.s", buf.size, buf.data);
    gnutls_free(buf.data);

    gnutls_ocsp_resp_deinit(resp);
}

static gnutls_x509_crt_t load_cert(const char *cert_file)
{
    gnutls_x509_crt_t crt;
    int ret;
    gnutls_datum_t data;
    size_t size;

    ret = gnutls_x509_crt_init(&crt);
    if (ret < 0)
        exit(1);

    data.data = (void *) read_binary_file(cert_file, &size);
    data.size = size;

    if (!data.data) {
        fprintf(stderr, "Cannot open file: %s\n", cert_file);
        exit(1);
    }

    ret = gnutls_x509_crt_import(crt, &data, GNUTLS_X509_FMT_PEM);
    free(data.data);
    if (ret < 0) {
        fprintf(stderr, "Cannot import certificate in %s: %s\n",
                cert_file, gnutls_strerror(ret));
        exit(1);
    }

    return crt;
}

static void
_generate_request(gnutls_datum_t * rdata, gnutls_x509_crt_t cert,
                 gnutls_x509_crt_t issuer, gnutls_datum_t *nonce)
{
    gnutls_ocsp_req_t req;
    int ret;

```

```
    ret = gnutls_ocsp_req_init(&req);
    if (ret < 0)
        exit(1);

    ret = gnutls_ocsp_req_add_cert(req, GNUTLS_DIG_SHA1, issuer, cert);
    if (ret < 0)
        exit(1);

    ret = gnutls_ocsp_req_set_nonce(req, 0, nonce);
    if (ret < 0)
        exit(1);

    ret = gnutls_ocsp_req_export(req, rdata);
    if (ret != 0)
        exit(1);

    gnutls_ocsp_req_deinit(req);

    return;
}

static int
_verify_response(gnutls_datum_t * data, gnutls_x509_crt_t cert,
                 gnutls_x509_crt_t signer, gnutls_datum_t *nonce)
{
    gnutls_ocsp_resp_t resp;
    int ret;
    unsigned verify;
    gnutls_datum_t rnonce;

    ret = gnutls_ocsp_resp_init(&resp);
    if (ret < 0)
        exit(1);

    ret = gnutls_ocsp_resp_import(resp, data);
    if (ret < 0)
        exit(1);

    ret = gnutls_ocsp_resp_check_crt(resp, 0, cert);
    if (ret < 0)
        exit(1);

    ret = gnutls_ocsp_resp_get_nonce(resp, NULL, &rnonce);
    if (ret < 0)
        exit(1);
}
```

```

    if (rnonce.size != nonce->size || memcmp(nonce->data, rnonce.data,
        nonce->size) != 0) {
        exit(1);
    }

    ret = gnutls_ocsp_resp_verify_direct(resp, signer, &verify, 0);
    if (ret < 0)
        exit(1);

    printf("Verifying OCSP Response: ");
    if (verify == 0)
        printf("Verification success!\n");
    else
        printf("Verification error!\n");

    if (verify & GNUTLS_OCSP_VERIFY_SIGNER_NOT_FOUND)
        printf("Signer cert not found\n");

    if (verify & GNUTLS_OCSP_VERIFY_SIGNER_KEYUSAGE_ERROR)
        printf("Signer cert keyusage error\n");

    if (verify & GNUTLS_OCSP_VERIFY_UNTRUSTED_SIGNER)
        printf("Signer cert is not trusted\n");

    if (verify & GNUTLS_OCSP_VERIFY_INSECURE_ALGORITHM)
        printf("Insecure algorithm\n");

    if (verify & GNUTLS_OCSP_VERIFY_SIGNATURE_FAILURE)
        printf("Signature failure\n");

    if (verify & GNUTLS_OCSP_VERIFY_CERT_NOT_ACTIVATED)
        printf("Signer cert not yet activated\n");

    if (verify & GNUTLS_OCSP_VERIFY_CERT_EXPIRED)
        printf("Signer cert expired\n");

    gnutls_free(rnonce.data);
    gnutls_ocsp_resp_deinit(resp);

    return verify;
}

size_t get_data(void *buffer, size_t size, size_t nmemb, void *userp)
{
    gnutls_datum_t *ud = userp;

    size *= nmemb;

```

```

    ud->data = realloc(ud->data, size + ud->size);
    if (ud->data == NULL) {
        fprintf(stderr, "Not enough memory for the request\n");
        exit(1);
    }

    memcpy(&ud->data[ud->size], buffer, size);
    ud->size += size;

    return size;
}

```

## 7.4 Miscellaneous examples

### 7.4.1 Checking for an alert

This is a function that checks if an alert has been received in the current session.

/\* This example code is placed in the public domain. \*/

```

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <gnutls/gnutls.h>

#include "examples.h"

/* This function will check whether the given return code from
 * a gnutls function (recv/send), is an alert, and will print
 * that alert.
 */
void check_alert(gnutls_session_t session, int ret)
{
    int last_alert;

    if (ret == GNUTLS_E_WARNING_ALERT_RECEIVED
        || ret == GNUTLS_E_FATAL_ALERT_RECEIVED) {
        last_alert = gnutls_alert_get(session);

        /* The check for renegotiation is only useful if we are
         * a server, and we had requested a rehandshake.
         */
        if (last_alert == GNUTLS_A_NO_RENEGOTIATION &&
            ret == GNUTLS_E_WARNING_ALERT_RECEIVED)

```

```

        printf("* Received NO_RENEGOTIATION alert.  "
               "Client Does not support renegotiation.\n");
    else
        printf("* Received alert '%d':  %s.\n", last_alert,
               gnutls_alert_get_name(last_alert));
    }
}

```

### 7.4.2 X.509 certificate parsing example

To demonstrate the X.509 parsing capabilities an example program is listed below. That program reads the peer's certificate, and prints information about it.

```

/* This example code is placed in the public domain.  */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <gnutls/gnutls.h>
#include <gnutls/x509.h>

#include "examples.h"

static const char *bin2hex(const void *bin, size_t bin_size)
{
    static char printable[110];
    const unsigned char *_bin = bin;
    char *print;
    size_t i;

    if (bin_size > 50)
        bin_size = 50;

    print = printable;
    for (i = 0; i < bin_size; i++) {
        sprintf(print, "%.2x ", _bin[i]);
        print += 2;
    }

    return printable;
}

/* This function will print information about this session's peer
 * certificate.
 */

```

```

void print_x509_certificate_info(gnutls_session_t session)
{
    char serial[40];
    char dn[256];
    size_t size;
    unsigned int algo, bits;
    time_t expiration_time, activation_time;
    const gnutls_datum_t *cert_list;
    unsigned int cert_list_size = 0;
    gnutls_x509_crt_t cert;
    gnutls_datum_t cinfo;

    /* This function only works for X.509 certificates.
       */
    if (gnutls_certificate_type_get(session) != GNUTLS_CERT_X509)
        return;

    cert_list = gnutls_certificate_get_peers(session, &cert_list_size);

    printf("Peer provided %d certificates.\n", cert_list_size);

    if (cert_list_size > 0) {
        int ret;

        /* we only print information about the first certificate.
           */
        gnutls_x509_crt_init(&cert);

        gnutls_x509_crt_import(cert, &cert_list[0],
                               GNUTLS_X509_FMT_DER);

        printf("Certificate info:\n");

        /* This is the preferred way of printing short information about
           a certificate.  */

        ret =
            gnutls_x509_crt_print(cert, GNUTLS_CERT_PRINT_ONELINE,
                                  &cinfo);

        if (ret == 0) {
            printf("\t%s\n", cinfo.data);
            gnutls_free(cinfo.data);
        }

        /* If you want to extract fields manually for some other reason,
           below are popular example calls.  */
    }
}

```

```

        expiration_time =
            gnutls_x509_cert_get_expiration_time(cert);
        activation_time =
            gnutls_x509_cert_get_activation_time(cert);

        printf("\tCertificate is valid since:  %s",
            ctime(&activation_time));
        printf("\tCertificate expires:  %s",
            ctime(&expiration_time));

        /* Print the serial number of the certificate.
         */
        size = sizeof(serial);
        gnutls_x509_cert_get_serial(cert, serial, &size);

        printf("\tCertificate serial number:  %s\n",
            bin2hex(serial, size));

        /* Extract some of the public key algorithm's parameters
         */
        algo = gnutls_x509_cert_get_pk_algorithm(cert, &bits);

        printf("Certificate public key:  %s",
            gnutls_pk_algorithm_get_name(algo));

        /* Print the version of the X.509
         * certificate.
         */
        printf("\tCertificate version:  %#d\n",
            gnutls_x509_cert_get_version(cert));

        size = sizeof(dn);
        gnutls_x509_cert_get_dn(cert, dn, &size);
        printf("\tDN:  %s\n", dn);

        size = sizeof(dn);
        gnutls_x509_cert_get_issuer_dn(cert, dn, &size);
        printf("\tIssuer's DN:  %s\n", dn);

        gnutls_x509_cert_deinit(cert);

    }
}

```

### 7.4.3 Listing the ciphersuites in a priority string

This is a small program to list the enabled ciphersuites by a priority string.

```

/* This example code is placed in the public domain. */

#include <config.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <gnutls/gnutls.h>

static void print_cipher_suite_list(const char *priorities)
{
    size_t i;
    int ret;
    unsigned int idx;
    const char *name;
    const char *err;
    unsigned char id[2];
    gnutls_protocol_t version;
    gnutls_priority_t pcache;

    if (priorities != NULL) {
        printf("Cipher suites for %s\n", priorities);

        ret = gnutls_priority_init(&pcache, priorities, &err);
        if (ret < 0) {
            fprintf(stderr, "Syntax error at: %s\n", err);
            exit(1);
        }

        for (i = 0;; i++) {
            ret =
                gnutls_priority_get_cipher_suite_index(pcache,
                                                         i,
                                                         &idx);
            if (ret == GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE)
                break;
            if (ret == GNUTLS_E_UNKNOWN_CIPHER_SUITE)
                continue;

            name =
                gnutls_cipher_suite_info(idx, id, NULL, NULL,
                                         NULL, &version);

            if (name != NULL)
                printf("%-50s\t0x%02x, 0x%02x\t%s\n",
                      name, (unsigned char) id[0],
                      (unsigned char) id[1],
                      gnutls_protocol_get_name(version));
        }
    }
}

```



```

        }

        return;
    }
}

int main(int argc, char **argv)
{
    if (argc > 1)
        print_cipher_suite_list(argv[1]);
    return 0;
}

```

#### 7.4.4 PKCS #12 structure generation example

This small program demonstrates the usage of the PKCS #12 API, by generating such a structure.

```

/* This example code is placed in the public domain. */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <gnutls/gnutls.h>
#include <gnutls/pkcs12.h>

#include "examples.h"

#define OUTFILE "out.p12"

/* This function will write a pkcs12 structure into a file.
 * cert: is a DER encoded certificate
 * pkcs8_key: is a PKCS #8 encrypted key (note that this must be
 * encrypted using a PKCS #12 cipher, or some browsers will crash)
 * password: is the password used to encrypt the PKCS #12 packet.
 */
int
write_pkcs12(const gnutls_datum_t * cert,
             const gnutls_datum_t * pkcs8_key, const char *password)
{
    gnutls_pkcs12_t pkcs12;
    int ret, bag_index;
    gnutls_pkcs12_bag_t bag, key_bag;
    char pkcs12_struct[10 * 1024];
    size_t pkcs12_struct_size;

```

[illegible]

```
                                pkcs8_key);
if (ret < 0) {
    fprintf(stderr, "ret:  %s\n", gnutls_strerror(ret));
    return 1;
}

/* Note that since the PKCS #8 key is already encrypted we don't
 * bother encrypting that bag.
 */
bag_index = ret;

gnutls_pkcs12_bag_set_friendly_name(key_bag, bag_index, "My name");

gnutls_pkcs12_bag_set_key_id(key_bag, bag_index, &key_id);

/* The bags were filled.  Now create the PKCS #12 structure.
 */
gnutls_pkcs12_init(&pkcs12);

/* Insert the two bags in the PKCS #12 structure.
 */

gnutls_pkcs12_set_bag(pkcs12, bag);
gnutls_pkcs12_set_bag(pkcs12, key_bag);

/* Generate a message authentication code for the PKCS #12
 * structure.
 */
gnutls_pkcs12_generate_mac(pkcs12, password);

pkcs12_struct_size = sizeof(pkcs12_struct);
ret =
    gnutls_pkcs12_export(pkcs12, GNUTLS_X509_FMT_DER,
                        pkcs12_struct, &pkcs12_struct_size);
if (ret < 0) {
    fprintf(stderr, "ret:  %s\n", gnutls_strerror(ret));
    return 1;
}

fd = fopen(OUTFILE, "w");
if (fd == NULL) {
    fprintf(stderr, "cannot open file\n");
    return 1;
}
fwrite(pkcs12_struct, 1, pkcs12_struct_size, fd);
```

```
    fclose(fd);

    gnutls_pkcs12_bag_deinit(bag);
    gnutls_pkcs12_bag_deinit(key_bag);
    gnutls_pkcs12_deinit(pkcs12);

    return 0;
}
```

## 8 Using GnuTLS as a cryptographic library

GnuTLS is not a low-level cryptographic library, i.e., it does not provide access to basic cryptographic primitives. However it abstracts the internal cryptographic back-end (see [Section 10.5 \[Cryptographic Backend\]](#), page 250), providing symmetric crypto, hash and HMAC algorithms, as well access to the random number generation. For a low-level crypto API the usage of nettle<sup>1</sup> library is recommended.

### 8.1 Symmetric algorithms

The available functions to access symmetric crypto algorithms operations are shown below. The supported algorithms are the algorithms required by the TLS protocol. They are listed in [Table 3.1](#).

```
int [gnutls_cipher_init], page 555 (gnutls_cipher_hd_t * handle,
gnutls_cipher_algorithm_t cipher, const gnutls_datum_t * key, const
gnutls_datum_t * iv)
int [gnutls_cipher_encrypt2], page 555 (gnutls_cipher_hd_t handle, const void
* text, size_t textlen, void * ciphertext, size_t ciphertextlen)
int [gnutls_cipher_decrypt2], page 554 (gnutls_cipher_hd_t handle, const void
* ciphertext, size_t ciphertextlen, void * text, size_t textlen)
void [gnutls_cipher_set_iv], page 556 (gnutls_cipher_hd_t handle, void * iv,
size_t ivlen)
void [gnutls_cipher_deinit], page 554 (gnutls_cipher_hd_t handle)
```

In order to support authenticated encryption with associated data (AEAD) algorithms the following functions are provided to set the associated data and retrieve the authentication tag.

```
int [gnutls_cipher_add_auth], page 553 (gnutls_cipher_hd_t handle, const void
* text, size_t text_size)
int [gnutls_cipher_tag], page 556 (gnutls_cipher_hd_t handle, void * tag,
size_t tag_size)
```

### 8.2 Public key algorithms

Public key cryptography algorithms such as RSA, DSA and ECDSA, can be accessed using the abstract key API in [Section 5.1 \[Abstract key types\]](#), page 81. This is a high level API with the advantage of transparently handling keys in memory and keys present in smart cards.

### 8.3 Hash and HMAC functions

The available operations to access hash functions and hash-MAC (HMAC) algorithms are shown below. HMAC algorithms provided keyed hash functionality. They supported HMAC algorithms are listed in [Table 3.2](#).

<sup>1</sup> See <http://www.lysator.liu.se/~nisse/nettle/>.

```

int [gnutls_hmac_init], page 559 (gnutls_hmac_hd_t * dig,
gnutls_mac_algorithm_t algorithm, const void * key, size_t keylen)
int [gnutls_hmac], page 558 (gnutls_hmac_hd_t handle, const void * text,
size_t textlen)
void [gnutls_hmac_output], page 559 (gnutls_hmac_hd_t handle, void * digest)
void [gnutls_hmac_deinit], page 558 (gnutls_hmac_hd_t handle, void * digest)
int [gnutls_hmac_get_len], page 558 (gnutls_mac_algorithm_t algorithm)
int [gnutls_hmac_fast], page 558 (gnutls_mac_algorithm_t algorithm, const
void * key, size_t keylen, const void * text, size_t textlen, void * digest)

```

The available functions to access hash functions are shown below. The supported hash functions are the same as the HMAC algorithms.

```

int [gnutls_hash_init], page 557 (gnutls_hash_hd_t * dig,
gnutls_digest_algorithm_t algorithm)
int [gnutls_hash], page 556 (gnutls_hash_hd_t handle, const void * text,
size_t textlen)
void [gnutls_hash_output], page 557 (gnutls_hash_hd_t handle, void * digest)
void [gnutls_hash_deinit], page 557 (gnutls_hash_hd_t handle, void * digest)
int [gnutls_hash_get_len], page 557 (gnutls_digest_algorithm_t algorithm)
int [gnutls_hash_fast], page 557 (gnutls_digest_algorithm_t algorithm, const
void * text, size_t textlen, void * digest)
int [gnutls_fingerprint], page 307 (gnutls_digest_algorithm_t algo, const
gnutls_datum_t * data, void * result, size_t * result_size)

```

## 8.4 Random number generation

Access to the random number generator is provided using the `[gnutls_rnd]`, page 560 function. It allows obtaining random data of various levels.

### GNUTLS\_RND\_NONCE

Non-predictable random number. Fatal in parts of session if broken, i.e., vulnerable to statistical analysis.

### GNUTLS\_RND\_RANDOM

Pseudo-random cryptographic random number. Fatal in session if broken.

### GNUTLS\_RND\_KEY

Fatal in many sessions if broken.

Figure 8.1: The random number levels.

```

int gnutls_rnd (gnutls_rnd_level_t level, void * data, size_t len)      [Function]
    level: a security level

```

*data*: place to store random bytes

*len*: The requested size

This function will generate random data and store it to output buffer.

This function is thread-safe and also fork-safe.

**Returns:** Zero on success, or a negative error code on error.

**Since:** 2.12.0

## 9 Other included programs

Included with GnuTLS are also a few command line tools that let you use the library for common tasks without writing an application. The applications are discussed in this chapter.

### 9.1 Invoking gnutls-cli

Simple client program to set up a TLS connection to some other computer. It sets up a TLS connection and forwards data from the standard input to the secured socket and vice versa.

This section was generated by **AutoGen**, using the **agtexi-cmd** template and the option descriptions for the **gnutls-cli** program. This software is released under the GNU General Public License, version 3 or later.

#### **gnutls-cli help/usage (--help)**

This is the automatically generated usage text for gnutls-cli.

The text printed is the same whether selected with the **help** option (**--help**) or the **more-help** option (**--more-help**). **more-help** will print the usage text by passing it through a pager program. **more-help** is disabled on platforms without a working **fork(2)** function. The **PAGER** environment variable is used to select the program, defaulting to **more**. Both will exit with a status code of 0.

**gnutls-cli is unavailable - no --help**

#### **debug option (-d)**

This is the “enable debugging” option. This option takes a number argument. Specifies the debug level.

#### **tofu option**

This is the “enable trust on first use authentication” option.

This option has some usage constraints. It:

- can be disabled with **-no-tofu**.

This option will, in addition to certificate authentication, perform authentication based on previously seen public keys, a model similar to SSH authentication. Note that when **tofu** is specified (PKI) and DANE authentication will become advisory to assist the public key acceptance process.

#### **strict-tofu option**

This is the “fail to connect if a known certificate has changed” option.

This option has some usage constraints. It:

- can be disabled with **-no-strict-tofu**.

This option will perform authentication as with option **-tofu**; however, while **-tofu** asks whether to trust a changed public key, this option will fail in case of public key changes.

### **dane option**

This is the “enable dane certificate verification (dnssec)” option.

This option has some usage constraints. It:

- can be disabled with `-no-dane`.

This option will, in addition to certificate authentication using the trusted CAs, verify the server certificates using on the DANE information available via DNSSEC.

### **local-dns option**

This is the “use the local dns server for dnssec resolving” option.

This option has some usage constraints. It:

- can be disabled with `-no-local-dns`.

This option will use the local DNS server for DNSSEC. This is disabled by default due to many servers not allowing DNSSEC.

### **ca-verification option**

This is the “disable ca certificate verification” option.

This option has some usage constraints. It:

- can be disabled with `-no-ca-verification`.
- It is enabled by default.

This option will disable CA certificate verification. It is to be used with the `-dane` or `-tofu` options.

### **ocsp option**

This is the “enable ocsp certificate verification” option.

This option has some usage constraints. It:

- can be disabled with `-no-ocsp`.

This option will enable verification of the peer’s certificate using ocsp

### **resume option (-r)**

This is the “establish a session and resume” option. Connect, establish a session, reconnect and resume.

### **rehandshake option (-e)**

This is the “establish a session and rehandshake” option. Connect, establish a session and rehandshake immediately.

### **starttls option (-s)**

This is the “connect, establish a plain session and start tls” option. The TLS session will be initiated when EOF or a SIGALRM is received.



### **app-proto option**

This is an alias for the `starttls-proto` option, see [\[gnutls-cli starttls-proto\]](#), page 233.

### **starttls-proto option**

This is the “the application protocol to be used to obtain the server’s certificate (https, ftp, smtp, imap)” option. This option takes a string argument.

This option has some usage constraints. It:

- must not appear in combination with any of the following options: `starttls`.

Specify the application layer protocol for STARTTLS. If the protocol is supported, gnutls-cli will proceed to the TLS negotiation.

### **dh-bits option**

This is the “the minimum number of bits allowed for dh” option. This option takes a number argument. This option sets the minimum number of bits allowed for a Diffie-Hellman key exchange. You may want to lower the default value if the peer sends a weak prime and you get an connection error with unacceptable prime.

### **priority option**

This is the “priorities string” option. This option takes a string argument. TLS algorithms and protocols to enable. You can use predefined sets of ciphersuites such as PERFORMANCE, NORMAL, PFS, SECURE128, SECURE256. The default is NORMAL.

Check the GnuTLS manual on section “Priority strings” for more information on the allowed keywords

### **ranges option**

This is the “use length-hiding padding to prevent traffic analysis” option. When possible (e.g., when using CBC ciphersuites), use length-hiding padding to prevent traffic analysis.

### **list option (-l)**

This is the “print a list of the supported algorithms and modes” option.

This option has some usage constraints. It:

- must not appear in combination with any of the following options: `port`.

Print a list of the supported algorithms and modes. If a priority string is given then only the enabled ciphersuites are shown.

### **alpn option**

This is the “application layer protocol” option. This option takes a string argument.

This option has some usage constraints. It:

- may appear an unlimited number of times.

This option will set and enable the Application Layer Protocol Negotiation (ALPN) in the TLS protocol.

**disable-extensions option**

This is the “disable all the tls extensions” option. This option disables all TLS extensions. Deprecated option. Use the priority string.

**inline-commands option**

This is the “inline commands of the form `^<cmd>^`” option. Enable inline commands of the form `^<cmd>^`. The inline commands are expected to be in a line by themselves. The available commands are: resume and renegotiate.

**inline-commands-prefix option**

This is the “change the default delimiter for inline commands.” option. This option takes a string argument. Change the default delimiter (`^`) used for inline commands. The delimiter is expected to be a single US-ASCII character (octets 0 - 127). This option is only relevant if inline commands are enabled via the inline-commands option

**provider option**

This is the “specify the pkcs #11 provider library” option. This option takes a file argument. This will override the default options in `/etc/gnutls/pkcs11.conf`

**gnutls-cli exit status**

One of the following exit values will be returned:

`'0 (EXIT_SUCCESS)'`

Successful program execution.

`'1 (EXIT_FAILURE)'`

The operation failed or the command syntax was not valid.

**gnutls-cli See Also**

`gnutls-cli-debug(1)`, `gnutls-serv(1)`

**gnutls-cli Examples****Connecting using PSK authentication**

To connect to a server using PSK authentication, you need to enable the choice of PSK by using a cipher priority parameter such as in the example below.

```
$ ./gnutls-cli -p 5556 localhost --pskusername psk_identity \
    --pskey 88f3824b3e5659f52d00e959bacab954b6540344 \
    --priority NORMAL:-KX-ALL:+ECDHE-PSK:+DHE-PSK:+PSK
Resolving 'localhost'...
Connecting to '127.0.0.1:5556'...
- PSK authentication.
- Version: TLS1.1
- Key Exchange: PSK
- Cipher: AES-128-CBC
```

- MAC: SHA1
- Compression: NULL
- Handshake was completed
  
- Simple Client Mode:

By keeping the `-pskusername` parameter and removing the `-pskkey` parameter, it will query only for the password during the handshake.

## Listing ciphersuites in a priority string

To list the ciphersuites in a priority string:

```
$ ./gnutls-cli --priority SECURE192 -l
Cipher suites for SECURE192
TLS_ECDHE_ECDSA_AES_256_CBC_SHA384      0xc0, 0x24 TLS1.2
TLS_ECDHE_ECDSA_AES_256_GCM_SHA384      0xc0, 0x2e TLS1.2
TLS_ECDHE_RSA_AES_256_GCM_SHA384        0xc0, 0x30 TLS1.2
TLS_DHE_RSA_AES_256_CBC_SHA256          0x00, 0x6b TLS1.2
TLS_DHE_DSS_AES_256_CBC_SHA256          0x00, 0x6a TLS1.2
TLS_RSA_AES_256_CBC_SHA256              0x00, 0x3d TLS1.2

Certificate types: CTYPE-X.509
Protocols: VERS-TLS1.2, VERS-TLS1.1, VERS-TLS1.0, VERS-SSL3.0, VERS-DTLS1.0
Compression: COMP=NULL
Elliptic curves: CURVE-SECP384R1, CURVE-SECP521R1
PK-signatures: SIGN-RSA-SHA384, SIGN-ECDSA-SHA384, SIGN-RSA-SHA512, SIGN-ECDSA-SHA512
```

## Connecting using a PKCS #11 token

To connect to a server using a certificate and a private key present in a PKCS #11 token you need to substitute the PKCS 11 URLs in the `x509certfile` and `x509keyfile` parameters. Those can be found using `"p11tool --list-tokens"` and then listing all the objects in the needed token, and using the appropriate.

```
$ p11tool --list-tokens

Token 0:
URL: pkcs11:model=PKCS15;manufacturer=MyMan;serial=1234;token=Test
Label: Test
Manufacturer: EnterSafe
Model: PKCS15
Serial: 1234

$ p11tool --login --list-certs "pkcs11:model=PKCS15;manufacturer=MyMan;serial=1234;token=Test"

Object 0:
URL: pkcs11:model=PKCS15;manufacturer=MyMan;serial=1234;token=Test;object=client;objectid=1
Type: X.509 Certificate
Label: client
ID: 2a:97:0d:58:d1:51:3c:23:07:ae:4e:0d:72:26:03:7d:99:06:02:6a
```

```
$ export MYCERT="pkcs11:model=PKCS15;manufacturer=MyMan;serial=1234;token=Test;object=MYCERT"
$ export MYKEY="pkcs11:model=PKCS15;manufacturer=MyMan;serial=1234;token=Test;object=MYKEY"

$ gnutls-cli www.example.com --x509keyfile $MYKEY --x509certfile MYCERT
```

Notice that the private key only differs from the certificate in the object-type.

## 9.2 Invoking gnutls-serv

Server program that listens to incoming TLS connections.

This section was generated by **AutoGen**, using the **agtexi-cmd** template and the option descriptions for the **gnutls-serv** program. This software is released under the GNU General Public License, version 3 or later.

### gnutls-serv help/usage (--help)

This is the automatically generated usage text for gnutls-serv.

The text printed is the same whether selected with the **help** option (**--help**) or the **more-help** option (**--more-help**). **more-help** will print the usage text by passing it through a pager program. **more-help** is disabled on platforms without a working **fork(2)** function. The **PAGER** environment variable is used to select the program, defaulting to **more**. Both will exit with a status code of 0.

```
gnutls-serv is unavailable - no --help
```

### debug option (-d)

This is the “enable debugging” option. This option takes a number argument. Specifies the debug level.

### verify-client-cert option

This is the “if a client certificate is sent then verify it.” option. Do not require, but if a client certificate is sent then verify it and close the connection if invalid.

### heartbeat option (-b)

This is the “activate heartbeat support” option. Regularly ping client via heartbeat extension messages

### priority option

This is the “priorities string” option. This option takes a string argument. TLS algorithms and protocols to enable. You can use predefined sets of ciphersuites such as **PERFORMANCE**, **NORMAL**, **SECURE128**, **SECURE256**. The default is **NORMAL**.

Check the GnuTLS manual on section “Priority strings” for more information on allowed keywords

### ocsp-response option

This is the “the ocsp response to send to client” option. This option takes a file argument. If the client requested an OCSF response, return data from this file to the client.

**list option (-l)**

This is the “print a list of the supported algorithms and modes” option. Print a list of the supported algorithms and modes. If a priority string is given then only the enabled ciphersuites are shown.

**provider option**

This is the “specify the pkcs #11 provider library” option. This option takes a file argument. This will override the default options in `/etc/gnutls/pkcs11.conf`

**gnutls-serv exit status**

One of the following exit values will be returned:

‘0 (EXIT\_SUCCESS)’

Successful program execution.

‘1 (EXIT\_FAILURE)’

The operation failed or the command syntax was not valid.

**gnutls-serv See Also**

`gnutls-cli-debug(1)`, `gnutls-cli(1)`

**gnutls-serv Examples**

Running your own TLS server based on GnuTLS can be useful when debugging clients and/or GnuTLS itself. This section describes how to use `gnutls-serv` as a simple HTTPS server.

The most basic server can be started as:

```
gnutls-serv --http --priority "NORMAL:+ANON-ECDH:+ANON-DH"
```

It will only support anonymous ciphersuites, which many TLS clients refuse to use.

The next step is to add support for X.509. First we generate a CA:

```
$ certtool --generate-privkey > x509-ca-key.pem
$ echo 'cn = GnuTLS test CA' > ca.tmpl
$ echo 'ca' >> ca.tmpl
$ echo 'cert_signing_key' >> ca.tmpl
$ certtool --generate-self-signed --load-privkey x509-ca-key.pem \
  --template ca.tmpl --outfile x509-ca.pem
...
```

Then generate a server certificate. Remember to change the `dns_name` value to the name of your server host, or skip that command to avoid the field.

```
$ certtool --generate-privkey > x509-server-key.pem
$ echo 'organization = GnuTLS test server' > server.tmpl
$ echo 'cn = test.gnutls.org' >> server.tmpl
$ echo 'tls_www_server' >> server.tmpl
$ echo 'encryption_key' >> server.tmpl
$ echo 'signing_key' >> server.tmpl
$ echo 'dns_name = test.gnutls.org' >> server.tmpl
```

```
$ certtool --generate-certificate --load-privkey x509-server-key.pem \
--load-ca-certificate x509-ca.pem --load-ca-privkey x509-ca-key.pem \
--template server.tpl --outfile x509-server.pem
...
```

For use in the client, you may want to generate a client certificate as well.

```
$ certtool --generate-privkey > x509-client-key.pem
$ echo 'cn = GnuTLS test client' > client.tpl
$ echo 'tls_www_client' >> client.tpl
$ echo 'encryption_key' >> client.tpl
$ echo 'signing_key' >> client.tpl
$ certtool --generate-certificate --load-privkey x509-client-key.pem \
--load-ca-certificate x509-ca.pem --load-ca-privkey x509-ca-key.pem \
--template client.tpl --outfile x509-client.pem
...
```

To be able to import the client key/certificate into some applications, you will need to convert them into a PKCS#12 structure. This also encrypts the security sensitive key with a password.

```
$ certtool --to-p12 --load-ca-certificate x509-ca.pem \
--load-privkey x509-client-key.pem --load-certificate x509-client.pem \
--outder --outfile x509-client.p12
```

For icing, we'll create a proxy certificate for the client too.

```
$ certtool --generate-privkey > x509-proxy-key.pem
$ echo 'cn = GnuTLS test client proxy' > proxy.tpl
$ certtool --generate-proxy --load-privkey x509-proxy-key.pem \
--load-ca-certificate x509-client.pem --load-ca-privkey x509-client-key.pem \
--load-certificate x509-client.pem --template proxy.tpl \
--outfile x509-proxy.pem
...
```

Then start the server again:

```
$ gnutls-serv --http \
--x509cafile x509-ca.pem \
--x509keyfile x509-server-key.pem \
--x509certfile x509-server.pem
```

Try connecting to the server using your web browser. Note that the server listens to port 5556 by default.

While you are at it, to allow connections using DSA, you can also create a DSA key and certificate for the server. These credentials will be used in the final example below.

```
$ certtool --generate-privkey --dsa > x509-server-key-dsa.pem
$ certtool --generate-certificate --load-privkey x509-server-key-dsa.pem \
--load-ca-certificate x509-ca.pem --load-ca-privkey x509-ca-key.pem \
--template server.tpl --outfile x509-server-dsa.pem
...
```

The next step is to create OpenPGP credentials for the server.

```
gpg --gen-key
...enter whatever details you want, use 'test.gnutls.org' as name...
```

Make a note of the OpenPGP key identifier of the newly generated key, here it was 5D1D14D8. You will need to export the key for GnuTLS to be able to use it.

```
gpg -a --export 5D1D14D8 > openpgp-server.txt
gpg --export 5D1D14D8 > openpgp-server.bin
gpg --export-secret-keys 5D1D14D8 > openpgp-server-key.bin
gpg -a --export-secret-keys 5D1D14D8 > openpgp-server-key.txt
```

Let's start the server with support for OpenPGP credentials:

```
gnutls-serv --http --priority NORMAL:+CTYPE-OPENPGP \
--pgpkeyfile openpgp-server-key.txt \
--pgpcertfile openpgp-server.txt
```

The next step is to add support for SRP authentication. This requires an SRP password file created with `srptool`. To start the server with SRP support:

```
gnutls-serv --http --priority NORMAL:+SRP-RSA:+SRP \
--srppasswdconf srp-tpasswd.conf \
--srppasswd srp-passwd.txt
```

Let's also start a server with support for PSK. This would require a password file created with `psktool`.

```
gnutls-serv --http --priority NORMAL:+ECDHE-PSK:+PSK \
--pskpasswd psk-passwd.txt
```

Finally, we start the server with all the earlier parameters and you get this command:

```
gnutls-serv --http --priority NORMAL:+PSK:+SRP:+CTYPE-OPENPGP \
--x509cafile x509-ca.pem \
--x509keyfile x509-server-key.pem \
--x509certfile x509-server.pem \
--x509dsafile x509-server-key-dsa.pem \
--x509dsacertfile x509-server-dsa.pem \
--pgpkeyfile openpgp-server-key.txt \
--pgpcertfile openpgp-server.txt \
--srppasswdconf srp-tpasswd.conf \
--srppasswd srp-passwd.txt \
--pskpasswd psk-passwd.txt
```

### 9.3 Invoking gnutls-cli-debug

TLS debug client. It sets up multiple TLS connections to a server and queries its capabilities. It was created to assist in debugging GnuTLS, but it might be useful to extract a TLS server's capabilities. It connects to a TLS server, performs tests and print the server's capabilities. If called with the `-v` parameter more checks will be performed. Can be used to check for servers with special needs or bugs.

This section was generated by **AutoGen**, using the `agtexi-cmd` template and the option descriptions for the `gnutls-cli-debug` program. This software is released under the GNU General Public License, version 3 or later.

**gnutls-cli-debug help/usage (--help)**

This is the automatically generated usage text for gnutls-cli-debug.

The text printed is the same whether selected with the **help** option (**--help**) or the **more-help** option (**--more-help**). **more-help** will print the usage text by passing it through a pager program. **more-help** is disabled on platforms without a working **fork(2)** function. The **PAGER** environment variable is used to select the program, defaulting to **more**. Both will exit with a status code of 0.

```
gnutls-cli-debug is unavailable - no --help
```

**debug option (-d)**

This is the “enable debugging” option. This option takes a number argument. Specifies the debug level.

**gnutls-cli-debug exit status**

One of the following exit values will be returned:

```
'0 (EXIT_SUCCESS)'
```

Successful program execution.

```
'1 (EXIT_FAILURE)'
```

The operation failed or the command syntax was not valid.

**gnutls-cli-debug See Also**

gnutls-cli(1), gnutls-serv(1)

**gnutls-cli-debug Examples**

```
$ ../src/gnutls-cli-debug localhost
Resolving 'localhost'...
Connecting to '127.0.0.1:443'...
Checking for SSL 3.0 support... yes
Checking whether %COMPAT is required... no
Checking for TLS 1.0 support... yes
Checking for TLS 1.1 support... no
Checking fallback from TLS 1.1 to... TLS 1.0
Checking for TLS 1.2 support... no
Checking whether we need to disable TLS 1.0... N/A
Checking for Safe renegotiation support... yes
Checking for Safe renegotiation support (SCSV)... yes
Checking for HTTPS server name... not checked
Checking for version rollback bug in RSA PMS... no
Checking for version rollback bug in Client Hello... no
Checking whether the server ignores the RSA PMS version... no
Checking whether the server can accept Hello Extensions... yes
Checking whether the server can accept small records (512 bytes)... yes
Checking whether the server can accept cipher suites not in SSL 3.0 spec... yes
Checking whether the server can accept a bogus TLS record version in the client hell
```



```
Checking for certificate information... N/A
Checking for trusted CAs... N/A
Checking whether the server understands TLS closure alerts... partially
Checking whether the server supports session resumption... yes
Checking for export-grade ciphersuite support... no
Checking RSA-export ciphersuite info... N/A
Checking for anonymous authentication support... no
Checking anonymous Diffie-Hellman group info... N/A
Checking for ephemeral Diffie-Hellman support... no
Checking ephemeral Diffie-Hellman group info... N/A
Checking for ephemeral EC Diffie-Hellman support... yes
Checking ephemeral EC Diffie-Hellman group info...
  Curve SECP256R1
Checking for AES-GCM cipher support... no
Checking for AES-CBC cipher support... yes
Checking for CAMELLIA cipher support... no
Checking for 3DES-CBC cipher support... yes
Checking for ARCFOUR 128 cipher support... yes
Checking for ARCFOUR 40 cipher support... no
Checking for MD5 MAC support... yes
Checking for SHA1 MAC support... yes
Checking for SHA256 MAC support... no
Checking for ZLIB compression support... no
Checking for max record size... no
Checking for OpenPGP authentication support... no
```

## 10 Internal Architecture of GnuTLS

This chapter is to give a brief description of the way GnuTLS works. The focus is to give an idea to potential developers and those who want to know what happens inside the black box.

### 10.1 The TLS Protocol

The main use case for the TLS protocol is shown in [Figure 10.1](#). A user of a library implementing the protocol expects no less than this functionality, i.e., to be able to set parameters such as the accepted security level, perform a negotiation with the peer and be able to exchange data.



Figure 10.1: TLS protocol use case.

### 10.2 TLS Handshake Protocol

The GnuTLS handshake protocol is implemented as a state machine that waits for input or returns immediately when the non-blocking transport layer functions are used. The main idea is shown in [Figure 10.2](#).



Figure 10.2: GnuTLS handshake state machine.

Also the way the input is processed varies per ciphersuite. Several implementations of the internal handlers are available and `[gnutls_handshake]`, page 309 only multiplexes the input to the appropriate handler. For example a PSK ciphersuite has a different implementation of the `process_client_key_exchange` than a certificate ciphersuite. We illustrate the idea in Figure 10.3.



Figure 10.3: GnuTLS handshake process sequence.

### 10.3 TLS Authentication Methods

In GnuTLS authentication methods can be implemented quite easily. Since the required changes to add a new authentication method affect only the handshake protocol, a simple interface is used. An authentication method needs to implement the functions shown below.

```
typedef struct
{
```

```

const char *name;
int (*gnutls_generate_server_certificate) (gnutls_session_t, gnutls_buffer_st*);
int (*gnutls_generate_client_certificate) (gnutls_session_t, gnutls_buffer_st*);
int (*gnutls_generate_server_kx) (gnutls_session_t, gnutls_buffer_st*);
int (*gnutls_generate_client_kx) (gnutls_session_t, gnutls_buffer_st*);
int (*gnutls_generate_client_cert_vrfy) (gnutls_session_t, gnutls_buffer_st *);
int (*gnutls_generate_server_certificate_request) (gnutls_session_t,
                                                    gnutls_buffer_st *);

int (*gnutls_process_server_certificate) (gnutls_session_t, opaque *,
                                           size_t);
int (*gnutls_process_client_certificate) (gnutls_session_t, opaque *,
                                           size_t);
int (*gnutls_process_server_kx) (gnutls_session_t, opaque *, size_t);
int (*gnutls_process_client_kx) (gnutls_session_t, opaque *, size_t);
int (*gnutls_process_client_cert_vrfy) (gnutls_session_t, opaque *, size_t);
int (*gnutls_process_server_certificate_request) (gnutls_session_t,
                                                  opaque *, size_t);
} mod_auth_st;

```

Those functions are responsible for the interpretation of the handshake protocol messages. It is common for such functions to read data from one or more `credentials_t` structures<sup>1</sup> and write data, such as certificates, usernames etc. to `auth_info_t` structures.

Simple examples of existing authentication methods can be seen in `auth/psk.c` for PSK ciphersuites and `auth/srp.c` for SRP ciphersuites. After implementing these functions the structure holding its pointers has to be registered in `gnutls_algorithms.c` in the `_gnutls_kx_algorithms` structure.

## 10.4 TLS Extension Handling

As with authentication methods, the TLS extensions handlers can be implemented using the interface shown below.

```

typedef int (*gnutls_ext_recv_func) (gnutls_session_t session,
                                     const unsigned char *data, size_t len);
typedef int (*gnutls_ext_send_func) (gnutls_session_t session,
                                     gnutls_buffer_st *extdata);

```

Here there are two functions, one for receiving the extension data and one for sending. These functions have to check internally whether they operate in client or server side.

A simple example of an extension handler can be seen in `ext/srp.c` in GnuTLS' source code. After implementing these functions, together with the extension number they handle, they have to be registered using `_gnutls_ext_register` in `gnutls_extensions.c` typically within `_gnutls_ext_init`.

---

<sup>1</sup> such as the `gnutls_certificate_credentials_t` structures

## Adding a new TLS extension

Adding support for a new TLS extension is done from time to time, and the process to do so is not difficult. Here are the steps you need to follow if you wish to do this yourself. For sake of discussion, let's consider adding support for the hypothetical TLS extension `foobar`.

### Add configure option like `--enable-foobar` or `--disable-foobar`.

This step is useful when the extension code is large and it might be desirable to disable the extension under some circumstances. Otherwise it can be safely skipped.

Whether to chose enable or disable depends on whether you intend to make the extension be enabled by default. Look at existing checks (i.e., SRP, authz) for how to model the code. For example:

```
AC_MSG_CHECKING([whether to disable foobar support])
AC_ARG_ENABLE(foobar,
AS_HELP_STRING([--disable-foobar],
[disable foobar support]),
ac_enable_foobar=no)
if test x$ac_enable_foobar != xno; then
  AC_MSG_RESULT(no)
  AC_DEFINE(ENABLE_FOOBAR, 1, [enable foobar])
else
  ac_full=0
  AC_MSG_RESULT(yes)
fi
AM_CONDITIONAL(ENABLE_FOOBAR, test "$ac_enable_foobar" != "no")
```

These lines should go in `m4/hooks.m4`.

### Add IANA extension value to `extensions_t` in `gnutls_int.h`.

A good name for the value would be `GNUTLS_EXTENSION_FOOBAR`. Check with <http://www.iana.org/assignments/tls-extensiontype-values> for allocated values. For experiments, you could pick a number but remember that some consider it a bad idea to deploy such modified version since it will lead to interoperability problems in the future when the IANA allocates that number to someone else, or when the foobar protocol is allocated another number.

### Add an entry to `_gnutls_extensions` in `gnutls_extensions.c`.

A typical entry would be:

```
int ret;

#ifdef ENABLE_FOOBAR
  ret = _gnutls_ext_register (&foobar_ext);
  if (ret != GNUTLS_E_SUCCESS)
    return ret;
#endif
```

Most likely you'll need to add an `#include "ext/foobar.h"`, that will contain something like like:

```

extension_entry_st foobar_ext = {
    .name = "FOOBAR",
    .type = GNUTLS_EXTENSION_FOOBAR,
    .parse_type = GNUTLS_EXT_TLS,
    .recv_func = _foobar_recv_params,
    .send_func = _foobar_send_params,
    .pack_func = _foobar_pack,
    .unpack_func = _foobar_unpack,
    .deinit_func = NULL
}

```

The `GNUTLS_EXTENSION_FOOBAR` is the integer value you added to `gnutls_int.h` earlier. In this structure you specify the functions to read the extension from the hello message, the function to send the reply to, and two more functions to pack and unpack from stored session data (e.g. when resumming a session). The `deinit` function will be called to deinitialize the extension's private parameters, if any.

Note that the conditional `ENABLE_FOOBAR` definition should only be used if step 1 with the `configure` options has taken place.

### Add new files that implement the extension.

The functions you are responsible to add are those mentioned in the previous step. They should be added in a file such as `ext/foobar.c` and headers should be placed in `ext/foobar.h`. As a starter, you could add this:

```

int
_foobar_recv_params (gnutls_session_t session, const opaque * data,
                    size_t data_size)
{
    return 0;
}

int
_foobar_send_params (gnutls_session_t session, gnutls_buffer_st* data)
{
    return 0;
}

int
_foobar_pack (extension_priv_data_t epriv, gnutls_buffer_st * ps)
{
    /* Append the extension's internal state to buffer */
    return 0;
}

int
_foobar_unpack (gnutls_buffer_st * ps, extension_priv_data_t * epriv)
{
    /* Read the internal state from buffer */

```

```
    return 0;
}
```

The `_foobar_recv_params` function is responsible for parsing incoming extension data (both in the client and server).

The `_foobar_send_params` function is responsible for sending extension data (both in the client and server).

If you receive length fields that don't match, return `GNUTLS_E_UNEXPECTED_PACKET_LENGTH`. If you receive invalid data, return `GNUTLS_E_RECEIVED_ILLEGAL_PARAMETER`. You can use other error codes from the list in [Appendix C \[Error codes\]](#), page 259. Return 0 on success.

An extension typically stores private information in the `session` data for later usage. That can be done using the functions `_gnutls_ext_set_session_data` and `_gnutls_ext_get_session_data`. You can check simple examples at `ext/max_record.c` and `ext/server_name.c` extensions. That private information can be saved and restored across session resumption if the following functions are set:

The `_foobar_pack` function is responsible for packing internal extension data to save them in the session resumption storage.

The `_foobar_unpack` function is responsible for restoring session data from the session resumption storage.

Recall that both the client and server, send and receive parameters, and your code most likely will need to do different things depending on which mode it is in. It may be useful to make this distinction explicit in the code. Thus, for example, a better template than above would be:

```
int
_gnutls_foobar_recv_params (gnutls_session_t session,
                           const opaque * data,
                           size_t data_size)
{
    if (session->security_parameters.entity == GNUTLS_CLIENT)
        return foobar_recv_client (session, data, data_size);
    else
        return foobar_recv_server (session, data, data_size);
}

int
_gnutls_foobar_send_params (gnutls_session_t session,
                           gnutls_buffer_st * data)
{
    if (session->security_parameters.entity == GNUTLS_CLIENT)
        return foobar_send_client (session, data);
    else
        return foobar_send_server (session, data);
}
```

The functions used would be declared as `static` functions, of the appropriate prototype, in the same file. When adding the files, you'll need to add them to `ext/Makefile.am` as well, for example:

```

if ENABLE_FOOBAR
libgnutls_ext_la_SOURCES += ext/foobar.c ext/foobar.h
endif

```

### Add API functions to enable/disable the extension.

It might be desirable to allow users of the extension to request use of the extension, or set extension specific data. This can be implemented by adding extension specific function calls that can be added to `includes/gnutls/gnutls.h`, as long as the LGPLv2.1+ applies. The implementation of the function should lie in the `ext/foobar.c` file.

To make the API available in the shared library you need to add the symbol in `lib/libgnutls.map`, so that the symbol is exported properly.

When writing GTK-DOC style documentation for your new APIs, don't forget to add `Since:` tags to indicate the GnuTLS version the API was introduced in.

### Heartbeat extension.

One such extension is HeartBeat protocol (RFC6520: <https://tools.ietf.org/html/rfc6520>) implementation. To enable it use option `-heartbeat` with example client and server supplied with gnutls:

```

./doc/credentials/gnutls-http-serv --priority "NORMAL:-CIPHER-ALL:+NULL" -d 100 \
--heartbeat --echo
./src/gnutls-cli --priority "NORMAL:-CIPHER-ALL:+NULL" -d 100 localhost -p 5556 \
--insecure --heartbeat

```

After that pasting

```
**HEARTBEAT**
```

command into `gnutls-cli` will trigger corresponding command on the server and it will send HeartBeat Request with random length to client.

Another way is to run capabilities check with:

```

./doc/credentials/gnutls-http-serv -d 100 --heartbeat
./src/gnutls-cli-debug localhost -p 5556

```

### Adding a new Supplemental Data Handshake Message

TLS handshake extensions allow to send so called supplemental data handshake messages [RFC4680]. This short section explains how to implement a supplemental data handshake message for a given TLS extension.

First of all, modify your extension `foobar` in the way, the that flags `session->security_parameters.do_send_supplemental` and `session->security_parameters.do_recv_supplemental` are set:

```

int
_gnutls_foobar_rcv_params (gnutls_session_t session, const opaque * data,
                           size_t _data_size)
{
    ...
    session->security_parameters.do_recv_supplemental=1;
    ...
}

```



```

    }

    int
    _gnutls_foobar_send_params (gnutls_session_t session, gnutls_buffer_st *extdata)
    {
        ...
        session->security_parameters.do_send_supplemental=1;
        ...
    }

```

Furthermore add the functions `_foobar_supp_rcv_params` and `_foobar_supp_send_params` to `_foobar.h` and `_foobar.c`. The following example code shows how to send a “Hello World” string in the supplemental data handshake message:

```

    int
    _foobar_supp_rcv_params(gnutls_session_t session, const opaque *data, size_t _data_size)
    {
        uint8_t len = _data_size;
        unsigned char *msg;

        msg = gnutls_malloc(len);
        if (msg == NULL) return GNUTLS_E_MEMORY_ERROR;

        memcpy(msg, data, len);
        msg[len]='\0';

        /* do something with msg */
        gnutls_free(msg);

        return len;
    }

    int
    _foobar_supp_send_params(gnutls_session_t session, gnutls_buffer_st *buf)
    {
        unsigned char *msg = "hello world";
        int len = strlen(msg);

        _gnutls_buffer_append_data_prefix(buf, 8, msg, len);

        return len;
    }

```

Afterwards, add the new supplemental data handshake message to `lib/gnutls_supplemental.c` by adding a new entry to the `_gnutls_supplemental[]` structure:

```

gnutls_supplemental_entry _gnutls_supplemental[] =
{
    {"foobar",
     GNUTLS_SUPPLEMENTAL_FOOBAR_DATA,

```

```

        _foobar_supp_recv_params,
        _foobar_supp_send_params},
    {0, 0, 0, 0}
};

```

You have to include your `foobar.h` header file as well:

```
#include "foobar.h"
```

Lastly, add the new supplemental data type to `lib/includes/gnutls/gnutls.h`:

```

typedef enum
{
    GNUTLS_SUPPLEMENTAL_USER_MAPPING_DATA = 0,
    GNUTLS_SUPPLEMENTAL_FOOBAR_DATA = 1
} gnutls_supplemental_data_format_type_t;

```

## 10.5 Cryptographic Backend

Today most new processors, either for embedded or desktop systems include either instructions intended to speed up cryptographic operations, or a co-processor with cryptographic capabilities. Taking advantage of those is a challenging task for every cryptographic application or library. Unfortunately the cryptographic library that GnuTLS is based on takes no advantage of these capabilities. For this reason GnuTLS handles this internally by following a layered approach to accessing cryptographic operations as in [Figure 10.4](#).

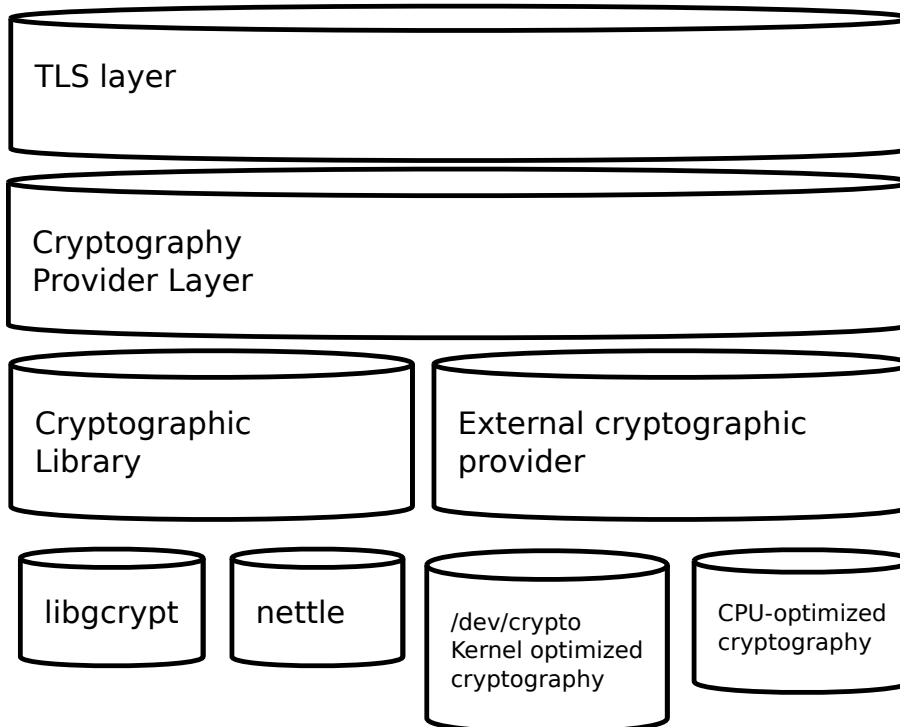


Figure 10.4: GnuTLS cryptographic back-end design.

The TLS layer uses a cryptographic provider layer, that will in turn either use the default crypto provider – a software crypto library, or use an external crypto provider, if available in the local system. The reason of handling the external cryptographic provider in GnuTLS and not delegating it to the cryptographic libraries, is that none of the supported cryptographic libraries support `/dev/crypto` or CPU-optimized cryptography in an efficient way.

## Cryptographic library layer

The Cryptographic library layer, currently supports only libnettle. Older versions of GnuTLS used to support libgcrypt, but it was switched with nettle mainly for performance reasons<sup>2</sup> and secondary because it is a simpler library to use. In the future other cryptographic libraries might be supported as well.

## External cryptography provider

Systems that include a cryptographic co-processor, typically come with kernel drivers to utilize the operations from software. For this reason GnuTLS provides a layer where each individual algorithm used can be replaced by another implementation, i.e., the one provided by the driver. The FreeBSD, OpenBSD and Linux kernels<sup>3</sup> include already a number of hardware assisted implementations, and also provide an interface to access them, called `/dev/crypto`. GnuTLS will take advantage of this interface if compiled with special options. That is because in most systems where hardware-assisted cryptographic operations are not available, using this interface might actually harm performance.

In systems that include cryptographic instructions with the CPU's instructions set, using the kernel interface will introduce an unneeded layer. For this reason GnuTLS includes such optimizations found in popular processors such as the AES-NI or VIA PADLOCK instruction sets. This is achieved using a mechanism that detects CPU capabilities and overrides parts of crypto back-end at runtime. The next section discusses the registration of a detected algorithm optimization. For more information please consult the GnuTLS source code in `lib/accelerated/`.

## Overriding specific algorithms

When an optimized implementation of a single algorithm is available, say a hardware assisted version of AES-CBC then the following (internal) functions, from `crypto-backend.h`, can be used to register those algorithms.

- `gnutls_crypto_single_cipher_register`: To register a cipher algorithm.
- `gnutls_crypto_single_digest_register`: To register a hash (digest) or MAC algorithm.

Those registration functions will only replace the specified algorithm and leave the rest of subsystem intact.

---

<sup>2</sup> See <http://lists.gnu.org/archive/html/gnutls-devel/2011-02/msg00079.html>.

<sup>3</sup> Check <http://home.gna.org/cryptodev-linux/> for the Linux kernel implementation of `/dev/crypto`.

## Overriding the cryptographic library

In some systems, that might contain a broad acceleration engine, it might be desirable to override big parts of the cryptographic back-end, or even all of them. The following functions are provided for this reason.

- `gnutls_crypto_cipher_register`: To override the cryptographic algorithms back-end.
- `gnutls_crypto_digest_register`: To override the digest algorithms back-end.
- `gnutls_crypto_rnd_register`: To override the random number generator back-end.
- `gnutls_crypto_bigint_register`: To override the big number number operations back-end.
- `gnutls_crypto_pk_register`: To override the public key encryption back-end. This is tied to the big number operations so either none or both of them should be overridden.

## Appendix A Upgrading from previous versions

The GnuTLS library typically maintains binary and source code compatibility across versions. The releases that have the major version increased break binary compatibility but source compatibility is provided. This section lists exceptional cases where changes to existing code are required due to library changes.

### Upgrading to 2.12.x from previous versions

GnuTLS 2.12.x is binary compatible with previous versions but changes the semantics of `gnutls_transport_set_lowat`, which might cause breakage in applications that relied on its default value be 1. Two fixes are proposed:

- Quick fix. Explicitly call `gnutls_transport_set_lowat(session, 1)`; after [\[gnutls\\_init\]](#), page 315.
- Long term fix. Because later versions of gnutls abolish the functionality of using the system call `select` to check for gnutls pending data, the function [\[gnutls\\_record\\_check\\_pending\]](#), page 332 has to be used to achieve the same functionality as described in [Section 6.5.1 \[Asynchronous operation\]](#), page 125.

### Upgrading to 3.0.x from 2.12.x

GnuTLS 3.0.x is source compatible with previous versions except for the functions listed below.

Old function	Replacement
<code>gnutls_transport_set_lowat</code>	To replace its functionality the function <a href="#">[gnutls_record_check_pending]</a> , page 332 has to be used, as described in <a href="#">Section 6.5.1 [Asynchronous operation]</a> , page 125
<code>gnutls_session_get_server_random</code> , <code>gnutls_session_get_client_random</code>	They are replaced by the safer function <a href="#">[gnutls_session_get_random]</a> , page 342
<code>gnutls_session_get_master_secret</code>	Replaced by the keying material exporters discussed in <a href="#">Section 6.12.4 [Deriving keys for other applications/protocols]</a> , page 146
<code>gnutls_transport_set_global_errno</code>	Replaced by using the system's <code>errno</code> facility or <a href="#">[gnutls_transport_set_errno]</a> , page 358.
<code>gnutls_x509_privkey_verify_data</code>	Replaced by <a href="#">[gnutls_pubkey_verify_data]</a> , page 544.
<code>gnutls_certificate_verify_peers</code>	Replaced by <a href="#">[gnutls_certificate_verify_peers2]</a> , page 293.

<code>gnutls_psk_netconf_derive_key</code>	Removed. The key derivation function was never standardized.
<code>gnutls_session_set_finished_function</code>	Removed.
<code>gnutls_ext_register</code>	Removed. Extension registration API is now internal to allow easier changes in the API.
<code>gnutls_certificate_get_x509_crls</code> , <code>gnutls_certificate_get_x509_cas</code>	Removed to allow updating the internal structures. Replaced by <a href="#">[gnutls_certificate_get_issuer]</a> , page 280.
<code>gnutls_certificate_get_openpgp_keyring</code>	Removed.
<code>gnutls_ia_</code>	Removed. The inner application extensions were completely removed (they failed to be standardized).

## Upgrading to 3.1.x from 3.0.x

GnuTLS 3.1.x is source and binary compatible with GnuTLS 3.0.x releases. Few functions have been deprecated and are listed below.

Old function	Replacement
<code>gnutls_pubkey_verify_hash</code>	The function <a href="#">[gnutls_pubkey_verify_hash2]</a> , page 545 is provided and is functionally equivalent and safer to use.
<code>gnutls_pubkey_verify_data</code>	The function <a href="#">[gnutls_pubkey_verify_data2]</a> , page 545 is provided and is functionally equivalent and safer to use.

## Upgrading to 3.2.x from 3.1.x

GnuTLS 3.2.x is source and binary compatible with GnuTLS 3.1.x releases. Few functions have been deprecated and are listed below.

Old function	Replacement
<code>gnutls_privkey_sign_raw_data</code>	The function <a href="#">[gnutls_privkey_sign_hash]</a> , page 533 is equivalent when the flag <code>GNUTLS_PRIVKEY_SIGN_FLAG_TLS1_RSA</code> is specified.

## Upgrading to 3.3.x from 3.2.x

GnuTLS 3.3.x is source and binary compatible with GnuTLS 3.2.x releases; however there are few changes in semantics which are listed below.

Old function	Replacement
<code>gnutls_global_init</code>	No longer required. The library is initialized using a constructor.
<code>gnutls_global_deinit</code>	No longer required. The library is deinitialized using a destructor.

## Appendix B Support

### B.1 Getting Help

A mailing list where users may help each other exists, and you can reach it by sending e-mail to [gnutls-help@gnutls.org](mailto:gnutls-help@gnutls.org). Archives of the mailing list discussions, and an interface to manage subscriptions, is available through the World Wide Web at <http://lists.gnutls.org/pipermail/gnutls-help/>.

A mailing list for developers are also available, see <http://www.gnutls.org/lists.html>. Bug reports should be sent to [bugs@gnutls.org](mailto:bugs@gnutls.org), see Section B.3 [Bug Reports], page 256.

### B.2 Commercial Support

Commercial support is available for users of GnuTLS. The kind of support that can be purchased may include:

- Implement new features. Such as a new TLS extension.
- Port GnuTLS to new platforms. This could include porting to an embedded platforms that may need memory or size optimization.
- Integrating TLS as a security environment in your existing project.
- System design of components related to TLS.

If you are interested, please write to:

Simon Josefsson Datakonsult  
Hagagatan 24  
113 47 Stockholm  
Sweden

E-mail: [simon@josefsson.org](mailto:simon@josefsson.org)

If your company provides support related to GnuTLS and would like to be mentioned here, contact the authors.

### B.3 Bug Reports

If you think you have found a bug in GnuTLS, please investigate it and report it.

- Please make sure that the bug is really in GnuTLS, and preferably also check that it hasn't already been fixed in the latest version.
- You have to send us a test case that makes it possible for us to reproduce the bug.
- You also have to explain what is wrong; if you get a crash, or if the results printed are not good and in that case, in what way. Make sure that the bug report includes all information you would need to fix this kind of bug for someone else.

Please make an effort to produce a self-contained report, with something definite that can be tested or debugged. Vague queries or piecemeal messages are difficult to act on and don't help the development effort.

If your bug report is good, we will do our best to help you to get a corrected version of the software; if the bug report is poor, we won't do anything about it (apart from asking you to send better bug reports).



If you think something in this manual is unclear, or downright incorrect, or if the language needs to be improved, please also send a note.

Send your bug report to:

`'bugs@gnutls.org'`

## B.4 Contributing

If you want to submit a patch for inclusion – from solving a typo you discovered, up to adding support for a new feature – you should submit it as a bug report, using the process in [Section B.3 \[Bug Reports\]](#), [page 256](#). There are some things that you can do to increase the chances for it to be included in the official package.

Unless your patch is very small (say, under 10 lines) we require that you assign the copyright of your work to the Free Software Foundation. This is to protect the freedom of the project. If you have not already signed papers, we will send you the necessary information when you submit your contribution.

For contributions that doesn't consist of actual programming code, the only guidelines are common sense. For code contributions, a number of style guides will help you:

- Coding Style. Follow the GNU Standards document.  
If you normally code using another coding standard, there is no problem, but you should use `'indent'` to reformat the code before submitting your work.
- Use the unified diff format `'diff -u'`.
- Return errors. No reason whatsoever should abort the execution of the library. Even memory allocation errors, e.g. when malloc return NULL, should work although result in an error code.
- Design with thread safety in mind. Don't use global variables. Don't even write to per-handle global variables unless the documented behaviour of the function you write is to write to the per-handle global variable.
- Avoid using the C math library. It causes problems for embedded implementations, and in most situations it is very easy to avoid using it.
- Document your functions. Use comments before each function headers, that, if properly formatted, are extracted into Texinfo manuals and GTK-DOC web pages.
- Supply a ChangeLog and NEWS entries, where appropriate.

## B.5 Certification

Many cryptographic libraries claim certifications from national or international bodies. These certifications are tied on a specific (and often restricted) version of the library or a specific product using the library, and typically in the case of software they assure that the algorithms implemented are correct. The major certifications known are:

- USA's FIPS 140-2 at Level 1 which certifies that approved algorithms are used (see [http://en.wikipedia.org/wiki/FIPS\\_140-2](http://en.wikipedia.org/wiki/FIPS_140-2));
- Common Criteria for Information Technology Security Evaluation (CC), an international standard for verification of elaborate security claims (see [http://en.wikipedia.org/wiki/Common\\_Criteria](http://en.wikipedia.org/wiki/Common_Criteria)).

Obtaining such a certification is an expensive and elaborate job that has no immediate value for a continuously developed free software library (as the certification is tied to the particular version tested). While, as a free software project, we are not actively pursuing this kind of certification, GnuTLS has been FIPS-140-2 certified in several systems by third parties. If you are, interested, see [Section B.2 \[Commercial Support\]](#), page 256.

## Appendix C Error Codes and Descriptions

The error codes used throughout the library are described below. The return code `GNUTLS_E_SUCCESS` indicates a successful operation, and is guaranteed to have the value 0, so you can use it in logical expressions.

0	<code>GNUTLS_E_SUCCESS</code>	Success.
-3	<code>GNUTLS_E_UNKNOWN_- COMPRESSION_ALGORITHM</code>	Could not negotiate a supported compression method.
-6	<code>GNUTLS_E_UNKNOWN_- CIPHER_TYPE</code>	The cipher type is unsupported.
-7	<code>GNUTLS_E_LARGE_PACKET</code>	The transmitted packet is too large (EMSGSIZE).
-8	<code>GNUTLS_E_UNSUPPORTED_- VERSION_PACKET</code>	A packet with illegal or unsupported version was received.
-9	<code>GNUTLS_E_UNEXPECTED_- PACKET_LENGTH</code>	A TLS packet with unexpected length was received.
-10	<code>GNUTLS_E_INVALID_SESSION</code>	The specified session has been invalidated for some reason.
-12	<code>GNUTLS_E_FATAL_ALERT_- RECEIVED</code>	A TLS fatal alert has been received.
-15	<code>GNUTLS_E_UNEXPECTED_- PACKET</code>	An unexpected TLS packet was received.
-16	<code>GNUTLS_E_WARNING_- ALERT_RECEIVED</code>	A TLS warning alert has been received.
-18	<code>GNUTLS_E_ERROR_IN_- FINISHED_PACKET</code>	An error was encountered at the TLS Finished packet calculation.
-19	<code>GNUTLS_E_UNEXPECTED_- HANDSHAKE_PACKET</code>	An unexpected TLS handshake packet was received.
-21	<code>GNUTLS_E_UNKNOWN_- CIPHER_SUITE</code>	Could not negotiate a supported cipher suite.
-22	<code>GNUTLS_E_UNWANTED_- ALGORITHM</code>	An algorithm that is not enabled was negotiated.
-23	<code>GNUTLS_E_MPL_SCAN_- FAILED</code>	The scanning of a large integer has failed.
-24	<code>GNUTLS_E_DECRYPTION_- FAILED</code>	Decryption has failed.
-25	<code>GNUTLS_E_MEMORY_ERROR</code>	Internal error in memory allocation.
-26	<code>GNUTLS_E_- DECOMPRESSION_FAILED</code>	Decompression of the TLS record packet has failed.
-27	<code>GNUTLS_E_COMPRESSION_- FAILED</code>	Compression of the TLS record packet has failed.
-28	<code>GNUTLS_E_AGAIN</code>	Resource temporarily unavailable, try again.

-29	GNUTLS_E_EXPIRED	The requested session has expired.
-30	GNUTLS_E_DB_ERROR	Error in Database backend.
-31	GNUTLS_E_SRP_PWD_ERROR	Error in password file.
-32	GNUTLS_E_INSUFFICIENT_CREDENTIALS	Insufficient credentials for that request.
-33	GNUTLS_E_HASH_FAILED	Hashing has failed.
-34	GNUTLS_E_BASE64_DECODING_ERROR	Base64 decoding error.
-35	GNUTLS_E_MPI_PRINT_FAILED	Could not export a large integer.
-37	GNUTLS_E_REHANDSHAKE	Rehandshake was requested by the peer.
-38	GNUTLS_E_GOT_APPLICATION_DATA	TLS Application data were received, while expecting handshake data.
-39	GNUTLS_E_RECORD_LIMIT_REACHED	The upper limit of record packet sequence numbers has been reached. Wow!
-40	GNUTLS_E_ENCRYPTION_FAILED	Encryption has failed.
-43	GNUTLS_E_CERTIFICATE_ERROR	Error in the certificate.
-44	GNUTLS_E_PK_ENCRYPTION_FAILED	Public key encryption has failed.
-45	GNUTLS_E_PK_DECRYPTION_FAILED	Public key decryption has failed.
-46	GNUTLS_E_PK_SIGN_FAILED	Public key signing has failed.
-47	GNUTLS_E_X509_UNSUPPORTED_CRITICAL_EXTENSION	Unsupported critical extension in X.509 certificate.
-48	GNUTLS_E_KEY_USAGE_VIOLATION	Key usage violation in certificate has been detected.
-49	GNUTLS_E_NO_CERTIFICATE_FOUND	No certificate was found.
-50	GNUTLS_E_INVALID_REQUEST	The request is invalid.
-51	GNUTLS_E_SHORT_MEMORY_BUFFER	The given memory buffer is too short to hold parameters.
-52	GNUTLS_E_INTERRUPTED	Function was interrupted.
-53	GNUTLS_E_PUSH_ERROR	Error in the push function.
-54	GNUTLS_E_PULL_ERROR	Error in the pull function.
-55	GNUTLS_E_RECEIVED_ILLEGAL_PARAMETER	An illegal parameter has been received.

-56	GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE	The requested data were not available.
-57	GNUTLS_E_PKCS1_WRONG_PAD	Wrong padding in PKCS1 packet.
-58	GNUTLS_E_RECEIVED_ILLEGAL_EXTENSION	An illegal TLS extension was received.
-59	GNUTLS_E_INTERNAL_ERROR	GnuTLS internal error.
-60	GNUTLS_E_CERTIFICATE_KEY_MISMATCH	The certificate and the given key do not match.
-61	GNUTLS_E_UNSUPPORTED_CERTIFICATE_TYPE	The certificate type is not supported.
-62	GNUTLS_E_X509_UNKNOWN_SAN	Unknown Subject Alternative name in X.509 certificate.
-63	GNUTLS_E_DH_PRIME_UNACCEPTABLE	The Diffie-Hellman prime sent by the server is not acceptable (not long enough).
-64	GNUTLS_E_FILE_ERROR	Error while reading file.
-67	GNUTLS_E_ASN1_ELEMENT_NOT_FOUND	ASN1 parser: Element was not found.
-68	GNUTLS_E_ASN1_IDENTIFIER_NOT_FOUND	ASN1 parser: Identifier was not found
-69	GNUTLS_E_ASN1_DER_ERROR	ASN1 parser: Error in DER parsing.
-70	GNUTLS_E_ASN1_VALUE_NOT_FOUND	ASN1 parser: Value was not found.
-71	GNUTLS_E_ASN1_GENERIC_ERROR	ASN1 parser: Generic parsing error.
-72	GNUTLS_E_ASN1_VALUE_NOT_VALID	ASN1 parser: Value is not valid.
-73	GNUTLS_E_ASN1_TAG_ERROR	ASN1 parser: Error in TAG.
-74	GNUTLS_E_ASN1_TAG_IMPLICIT	ASN1 parser: error in implicit tag
-75	GNUTLS_E_ASN1_TYPE_ANY_ERROR	ASN1 parser: Error in type 'ANY'.
-76	GNUTLS_E_ASN1_SYNTAX_ERROR	ASN1 parser: Syntax error.
-77	GNUTLS_E_ASN1_DER_OVERFLOW	ASN1 parser: Overflow in DER parsing.
-78	GNUTLS_E_TOO_MANY_EMPTY_PACKETS	Too many empty record packets have been received.
-79	GNUTLS_E_OPENPGP_UID_REVOKED	The OpenPGP User ID is revoked.

-80	GNUTLS_E_UNKNOWN_PK_- ALGORITHM	An unknown public key algo- rithm was encountered.
-81	GNUTLS_E_TOO_MANY_- HANDSHAKE_PACKETS	Too many handshake packets have been received.
-84	GNUTLS_E_NO_- TEMPORARY_RSA_PARAMS	No temporary RSA parameters were found.
-86	GNUTLS_E_NO_- COMPRESSION_- ALGORITHMS	No supported compression al- gorithms have been found.
-87	GNUTLS_E_NO_CIPHER_- SUITES	No supported cipher suites have been found.
-88	GNUTLS_E_OPENPGP_- GETKEY_FAILED	Could not get OpenPGP key.
-89	GNUTLS_E_PK_SIG_VERIFY_- FAILED	Public key signature verifica- tion has failed.
-90	GNUTLS_E_ILLEGAL_SRP_- USERNAME	The SRP username supplied is illegal.
-91	GNUTLS_E_SRP_PWD_- PARSING_ERROR	Parsing error in password file.
-93	GNUTLS_E_NO_- TEMPORARY_DH_PARAMS	No temporary DH parameters were found.
-94	GNUTLS_E_OPENPGP_- FINGERPRINT_- UNSUPPORTED	The OpenPGP fingerprint is not supported.
-95	GNUTLS_E_X509_- UNSUPPORTED_ATTRIBUTE	The certificate has unsup- ported attributes.
-96	GNUTLS_E_UNKNOWN_- HASH_ALGORITHM	The hash algorithm is unknown.
-97	GNUTLS_E_UNKNOWN_- PKCS_CONTENT_TYPE	The PKCS structure's content type is unknown.
-98	GNUTLS_E_UNKNOWN_- PKCS_BAG_TYPE	The PKCS structure's bag type is unknown.
-99	GNUTLS_E_INVALID_- PASSWORD	The given password contains invalid characters.
-100	GNUTLS_E_MAC_VERIFY_- FAILED	The Message Authentication Code verification failed.
-101	GNUTLS_E_CONSTRAINT_- ERROR	Some constraint limits were reached.
-104	GNUTLS_E_IA_VERIFY_- FAILED	Verifying TLS/IA phase check- sum failed
-105	GNUTLS_E_UNKNOWN_- ALGORITHM	The specified algorithm or pro- tocol is unknown.
-106	GNUTLS_E_UNSUPPORTED_- SIGNATURE_ALGORITHM	The signature algorithm is not supported.

-107	GNUTLS_E_SAFE_- RENEGOTIATION_FAILED	Safe renegotiation failed.
-108	GNUTLS_E_UNSAFE_- RENEGOTIATION_DENIED	Unsafe renegotiation denied.
-109	GNUTLS_E_UNKNOWN_SRP_- USERNAME	The SRP username supplied is unknown.
-110	GNUTLS_E_PREMATURE_- TERMINATION	The TLS connection was non-properly terminated.
-201	GNUTLS_E_BASE64_- ENCODING_ERROR	Base64 encoding error.
-202	GNUTLS_E_INCOMPATIBLE_- GCRYPT_LIBRARY	The crypto library version is too old.
-203	GNUTLS_E_INCOMPATIBLE_- LIBTASN1_LIBRARY	The tasn1 library version is too old.
-204	GNUTLS_E_OPENPGP_- KEYRING_ERROR	Error loading the keyring.
-205	GNUTLS_E_X509_- UNSUPPORTED_OID	The OID is not supported.
-206	GNUTLS_E_RANDOM_FAILED	Failed to acquire random data.
-207	GNUTLS_E_BASE64_- UNEXPECTED_HEADER_- ERROR	Base64 unexpected header error.
-208	GNUTLS_E_OPENPGP_- SUBKEY_ERROR	Could not find OpenPGP subkey.
-209	GNUTLS_E_CRYPTO_- ALREADY_REGISTERED	There is already a crypto algorithm with lower priority.
-210	GNUTLS_E_HANDSHAKE_- TOO_LARGE	The handshake data size is too large.
-211	GNUTLS_E_CRYPTODEV_- IOCTL_ERROR	Error interfacing with /dev/crypto
-212	GNUTLS_E_CRYPTODEV_- DEVICE_ERROR	Error opening /dev/crypto
-213	GNUTLS_E_CHANNEL_- BINDING_NOT_AVAILABLE	Channel binding data not available
-214	GNUTLS_E_BAD_COOKIE	The cookie was bad.
-215	GNUTLS_E_OPENPGP_- PREFERRED_KEY_ERROR	The OpenPGP key has not a preferred key set.
-216	GNUTLS_E_INCOMPAT_DSA_- KEY_WITH_TLS_PROTOCOL	The given DSA key is incompatible with the selected TLS protocol.
-292	GNUTLS_E_HEARTBEAT_- PONG_RECEIVED	A heartbeat pong message was received.
-293	GNUTLS_E_HEARTBEAT_- PING_RECEIVED	A heartbeat ping message was received.
-300	GNUTLS_E_PKCS11_ERROR	PKCS #11 error.

-301	GNUTLS_E_PKCS11_LOAD_ERROR	PKCS #11 initialization error.
-302	GNUTLS_E_PARSING_ERROR	Error in parsing.
-303	GNUTLS_E_PKCS11_PIN_ERROR	Error in provided PIN.
-305	GNUTLS_E_PKCS11_SLOT_ERROR	PKCS #11 error in slot
-306	GNUTLS_E_LOCKING_ERROR	Thread locking error
-307	GNUTLS_E_PKCS11_ATTRIBUTE_ERROR	PKCS #11 error in attribute
-308	GNUTLS_E_PKCS11_DEVICE_ERROR	PKCS #11 error in device
-309	GNUTLS_E_PKCS11_DATA_ERROR	PKCS #11 error in data
-310	GNUTLS_E_PKCS11_UNSUPPORTED_FEATURE_ERROR	PKCS #11 unsupported feature
-311	GNUTLS_E_PKCS11_KEY_ERROR	PKCS #11 error in key
-312	GNUTLS_E_PKCS11_PIN_EXPIRED	PKCS #11 PIN expired
-313	GNUTLS_E_PKCS11_PIN_LOCKED	PKCS #11 PIN locked
-314	GNUTLS_E_PKCS11_SESSION_ERROR	PKCS #11 error in session
-315	GNUTLS_E_PKCS11_SIGNATURE_ERROR	PKCS #11 error in signature
-316	GNUTLS_E_PKCS11_TOKEN_ERROR	PKCS #11 error in token
-317	GNUTLS_E_PKCS11_USER_ERROR	PKCS #11 user error
-318	GNUTLS_E_CRYPTTO_INIT_FAILED	The initialization of crypto backend has failed.
-319	GNUTLS_E_TIMEDOUT	The operation timed out
-320	GNUTLS_E_USER_ERROR	The operation was cancelled due to user error
-321	GNUTLS_E_ECC_NO_SUPPORTED_CURVES	No supported ECC curves were found
-322	GNUTLS_E_ECC_UNSUPPORTED_CURVE	The curve is unsupported
-323	GNUTLS_E_PKCS11_REQUESTED_OBJECT_NOT_AVAILABLE	The requested PKCS #11 object is not available



-324	GNUTLS_E_CERTIFICATE_- LIST_UNSORTED	The provided X.509 certificate list is not sorted (in subject to issuer order)
-325	GNUTLS_E_ILLEGAL_- PARAMETER	An illegal parameter was found.
-326	GNUTLS_E_NO_PRIORITIES_- WERE_SET	No or insufficient priorities were set.
-327	GNUTLS_E_X509_- UNSUPPORTED_EXTENSION	Unsupported extension in X.509 certificate.
-328	GNUTLS_E_SESSION_EOF	Peer has terminated the connection
-329	GNUTLS_E_TPM_ERROR	TPM error.
-330	GNUTLS_E_TPM_KEY_- PASSWORD_ERROR	Error in provided password for key to be loaded in TPM.
-331	GNUTLS_E_TPM_SRK_- PASSWORD_ERROR	Error in provided SRK password for TPM.
-332	GNUTLS_E_TPM_SESSION_- ERROR	Cannot initialize a session with the TPM.
-333	GNUTLS_E_TPM_KEY_NOT_- FOUND	TPM key was not found in persistent storage.
-334	GNUTLS_E_TPM_- UNINITIALIZED	TPM is not initialized.
-340	GNUTLS_E_NO_- CERTIFICATE_STATUS	There is no certificate status (OCSP).
-341	GNUTLS_E_OCSP_- RESPONSE_ERROR	The OCSP response is invalid
-342	GNUTLS_E_RANDOM_- DEVICE_ERROR	Error in the system's randomness device.
-343	GNUTLS_E_AUTH_ERROR	Could not authenticate peer.
-344	GNUTLS_E_NO_- APPLICATION_PROTOCOL	No common application protocol could be negotiated.
-345	GNUTLS_E_SOCKETS_INIT_- ERROR	Error in sockets initialization.
-400	GNUTLS_E_SELF_TEST_- ERROR	Error while performing self checks.
-401	GNUTLS_E_NO_SELF_TEST	There is no self test for this algorithm.
-402	GNUTLS_E_LIB_IN_ERROR_- STATE	An error has been detected in the library and cannot continue operations.
-403	GNUTLS_E_PK_- GENERATION_ERROR	Error in public key generation.

## Appendix D Supported Ciphersuites

### Ciphersuites

Ciphersuite name	TLS ID	Since
TLS_RSA_NULL_MD5	0x00 0x01	SSL3.0
TLS_RSA_NULL_SHA1	0x00 0x02	SSL3.0
TLS_RSA_NULL_SHA256	0x00 0x3B	TLS1.2
TLS_RSA_ARCFOUR_128_SHA1	0x00 0x05	SSL3.0
TLS_RSA_ARCFOUR_128_MD5	0x00 0x04	SSL3.0
TLS_RSA_3DES_EDE_CBC_SHA1	0x00 0x0A	SSL3.0
TLS_RSA_AES_128_CBC_SHA1	0x00 0x2F	SSL3.0
TLS_RSA_AES_256_CBC_SHA1	0x00 0x35	SSL3.0
TLS_RSA_CAMELLIA_128_CBC_SHA256	0x00 0xBA	TLS1.2
TLS_RSA_CAMELLIA_256_CBC_SHA256	0x00 0xC0	TLS1.2
TLS_RSA_CAMELLIA_128_CBC_SHA1	0x00 0x41	SSL3.0
TLS_RSA_CAMELLIA_256_CBC_SHA1	0x00 0x84	SSL3.0
TLS_RSA_AES_128_CBC_SHA256	0x00 0x3C	TLS1.2
TLS_RSA_AES_256_CBC_SHA256	0x00 0x3D	TLS1.2
TLS_RSA_AES_128_GCM_SHA256	0x00 0x9C	TLS1.2
TLS_RSA_AES_256_GCM_SHA384	0x00 0x9D	TLS1.2
TLS_RSA_CAMELLIA_128_GCM_SHA256	0xC0 0x7A	TLS1.2
TLS_RSA_CAMELLIA_256_GCM_SHA384	0xC0 0x7B	TLS1.2
TLS_RSA_SALSA20_256_SHA1	0xE4 0x11	SSL3.0
TLS_RSA_ESTREAM_SALSA20_256_SHA1	0xE4 0x10	SSL3.0
TLS_DHE_DSS_ARCFOUR_128_SHA1	0x00 0x66	SSL3.0
TLS_DHE_DSS_3DES_EDE_CBC_SHA1	0x00 0x13	SSL3.0
TLS_DHE_DSS_AES_128_CBC_SHA1	0x00 0x32	SSL3.0
TLS_DHE_DSS_AES_256_CBC_SHA1	0x00 0x38	SSL3.0
TLS_DHE_DSS_CAMELLIA_128_CBC_SHA256	0x00 0xBD	TLS1.2
TLS_DHE_DSS_CAMELLIA_256_CBC_SHA256	0x00 0xC3	TLS1.2
TLS_DHE_DSS_CAMELLIA_128_CBC_SHA1	0x00 0x44	SSL3.0
TLS_DHE_DSS_CAMELLIA_256_CBC_SHA1	0x00 0x87	SSL3.0
TLS_DHE_DSS_AES_128_CBC_SHA256	0x00 0x40	TLS1.2
TLS_DHE_DSS_AES_256_CBC_SHA256	0x00 0x6A	TLS1.2
TLS_DHE_DSS_AES_128_GCM_SHA256	0x00 0xA2	TLS1.2
TLS_DHE_DSS_AES_256_GCM_SHA384	0x00 0xA3	TLS1.2
TLS_DHE_DSS_CAMELLIA_128_GCM_SHA256	0xC0 0x80	TLS1.2
TLS_DHE_DSS_CAMELLIA_256_GCM_SHA384	0xC0 0x81	TLS1.2
TLS_DHE_RSA_3DES_EDE_CBC_SHA1	0x00 0x16	SSL3.0
TLS_DHE_RSA_AES_128_CBC_SHA1	0x00 0x33	SSL3.0
TLS_DHE_RSA_AES_256_CBC_SHA1	0x00 0x39	SSL3.0
TLS_DHE_RSA_CAMELLIA_128_CBC_SHA256	0x00 0xBE	TLS1.2
TLS_DHE_RSA_CAMELLIA_256_CBC_SHA256	0x00 0xC4	TLS1.2
TLS_DHE_RSA_CAMELLIA_128_CBC_SHA1	0x00 0x45	SSL3.0

TLS_DHE_RSA_CAMELLIA_256_CBC_SHA1	0x00 0x88	SSL3.0
TLS_DHE_RSA_AES_128_CBC_SHA256	0x00 0x67	TLS1.2
TLS_DHE_RSA_AES_256_CBC_SHA256	0x00 0x6B	TLS1.2
TLS_DHE_RSA_AES_128_GCM_SHA256	0x00 0x9E	TLS1.2
TLS_DHE_RSA_AES_256_GCM_SHA384	0x00 0x9F	TLS1.2
TLS_DHE_RSA_CAMELLIA_128_GCM_SHA256	0xC0 0x7C	TLS1.2
TLS_DHE_RSA_CAMELLIA_256_GCM_SHA384	0xC0 0x7D	TLS1.2
TLS_ECDHE_RSA_NULL_SHA1	0xC0 0x10	SSL3.0
TLS_ECDHE_RSA_3DES_EDE_CBC_SHA1	0xC0 0x12	SSL3.0
TLS_ECDHE_RSA_AES_128_CBC_SHA1	0xC0 0x13	SSL3.0
TLS_ECDHE_RSA_AES_256_CBC_SHA1	0xC0 0x14	SSL3.0
TLS_ECDHE_RSA_AES_256_CBC_SHA384	0xC0 0x28	TLS1.2
TLS_ECDHE_RSA_ARCFOUR_128_SHA1	0xC0 0x11	SSL3.0
TLS_ECDHE_RSA_CAMELLIA_128_CBC_SHA256	0xC0 0x76	TLS1.2
TLS_ECDHE_RSA_CAMELLIA_256_CBC_SHA384	0xC0 0x77	TLS1.2
TLS_ECDHE_ECDSA_NULL_SHA1	0xC0 0x06	SSL3.0
TLS_ECDHE_ECDSA_3DES_EDE_CBC_SHA1	0xC0 0x08	SSL3.0
TLS_ECDHE_ECDSA_AES_128_CBC_SHA1	0xC0 0x09	SSL3.0
TLS_ECDHE_ECDSA_AES_256_CBC_SHA1	0xC0 0x0A	SSL3.0
TLS_ECDHE_ECDSA_ARCFOUR_128_SHA1	0xC0 0x07	SSL3.0
TLS_ECDHE_ECDSA_CAMELLIA_128_CBC_- SHA256	0xC0 0x72	TLS1.2
TLS_ECDHE_ECDSA_CAMELLIA_256_CBC_- SHA384	0xC0 0x73	TLS1.2
TLS_ECDHE_ECDSA_AES_128_CBC_SHA256	0xC0 0x23	TLS1.2
TLS_ECDHE_RSA_AES_128_CBC_SHA256	0xC0 0x27	TLS1.2
TLS_ECDHE_ECDSA_CAMELLIA_128_GCM_- SHA256	0xC0 0x86	TLS1.2
TLS_ECDHE_ECDSA_CAMELLIA_256_GCM_- SHA384	0xC0 0x87	TLS1.2
TLS_ECDHE_ECDSA_AES_128_GCM_SHA256	0xC0 0x2B	TLS1.2
TLS_ECDHE_ECDSA_AES_256_GCM_SHA384	0xC0 0x2C	TLS1.2
TLS_ECDHE_RSA_AES_128_GCM_SHA256	0xC0 0x2F	TLS1.2
TLS_ECDHE_RSA_AES_256_GCM_SHA384	0xC0 0x30	TLS1.2
TLS_ECDHE_ECDSA_AES_256_CBC_SHA384	0xC0 0x24	TLS1.2
TLS_ECDHE_RSA_CAMELLIA_128_GCM_SHA256	0xC0 0x8A	TLS1.2
TLS_ECDHE_RSA_CAMELLIA_256_GCM_SHA384	0xC0 0x8B	TLS1.2
TLS_ECDHE_RSA_SALSA20_256_SHA1	0xE4 0x13	SSL3.0
TLS_ECDHE_ECDSA_SALSA20_256_SHA1	0xE4 0x15	SSL3.0
TLS_ECDHE_RSA_ESTREAM_SALSA20_256_SHA1	0xE4 0x12	SSL3.0
TLS_ECDHE_ECDSA_ESTREAM_SALSA20_256_- SHA1	0xE4 0x14	SSL3.0
TLS_ECDHE_PSK_3DES_EDE_CBC_SHA1	0xC0 0x34	SSL3.0
TLS_ECDHE_PSK_AES_128_CBC_SHA1	0xC0 0x35	SSL3.0
TLS_ECDHE_PSK_AES_256_CBC_SHA1	0xC0 0x36	SSL3.0
TLS_ECDHE_PSK_AES_128_CBC_SHA256	0xC0 0x37	TLS1.2

TLS_ECDHE_PSK_AES_256_CBC_SHA384	0xC0 0x38	TLS1.2
TLS_ECDHE_PSK_ARCFOUR_128_SHA1	0xC0 0x33	SSL3.0
TLS_ECDHE_PSK_NULL_SHA1	0xC0 0x39	SSL3.0
TLS_ECDHE_PSK_NULL_SHA256	0xC0 0x3A	TLS1.2
TLS_ECDHE_PSK_NULL_SHA384	0xC0 0x3B	TLS1.0
TLS_ECDHE_PSK_CAMELLIA_128_CBC_SHA256	0xC0 0x9A	TLS1.2
TLS_ECDHE_PSK_CAMELLIA_256_CBC_SHA384	0xC0 0x9B	TLS1.2
TLS_ECDHE_PSK_SALSA20_256_SHA1	0xE4 0x19	SSL3.0
TLS_ECDHE_PSK_ESTREAM_SALSA20_256_SHA1	0xE4 0x18	SSL3.0
TLS_PSK_ARCFOUR_128_SHA1	0x00 0x8A	SSL3.0
TLS_PSK_3DES_EDE_CBC_SHA1	0x00 0x8B	SSL3.0
TLS_PSK_AES_128_CBC_SHA1	0x00 0x8C	SSL3.0
TLS_PSK_AES_256_CBC_SHA1	0x00 0x8D	SSL3.0
TLS_PSK_AES_128_CBC_SHA256	0x00 0xAE	TLS1.2
TLS_PSK_AES_256_GCM_SHA384	0x00 0xA9	TLS1.2
TLS_PSK_CAMELLIA_128_GCM_SHA256	0xC0 0x8E	TLS1.2
TLS_PSK_CAMELLIA_256_GCM_SHA384	0xC0 0x8F	TLS1.2
TLS_PSK_AES_128_GCM_SHA256	0x00 0xA8	TLS1.2
TLS_PSK_NULL_SHA1	0x00 0x2C	SSL3.0
TLS_PSK_NULL_SHA256	0x00 0xB0	TLS1.2
TLS_PSK_CAMELLIA_128_CBC_SHA256	0xC0 0x94	TLS1.2
TLS_PSK_CAMELLIA_256_CBC_SHA384	0xC0 0x95	TLS1.2
TLS_PSK_SALSA20_256_SHA1	0xE4 0x17	SSL3.0
TLS_PSK_ESTREAM_SALSA20_256_SHA1	0xE4 0x16	SSL3.0
TLS_PSK_AES_256_CBC_SHA384	0x00 0xAF	TLS1.2
TLS_PSK_NULL_SHA384	0x00 0xB1	TLS1.2
TLS_RSA_PSK_ARCFOUR_128_SHA1	0x00 0x92	TLS1.0
TLS_RSA_PSK_3DES_EDE_CBC_SHA1	0x00 0x93	TLS1.0
TLS_RSA_PSK_AES_128_CBC_SHA1	0x00 0x94	TLS1.0
TLS_RSA_PSK_AES_256_CBC_SHA1	0x00 0x95	TLS1.0
TLS_RSA_PSK_CAMELLIA_128_GCM_SHA256	0xC0 0x92	TLS1.2
TLS_RSA_PSK_CAMELLIA_256_GCM_SHA384	0xC0 0x93	TLS1.2
TLS_RSA_PSK_AES_128_GCM_SHA256	0x00 0xAC	TLS1.2
TLS_RSA_PSK_AES_128_CBC_SHA256	0x00 0xB6	TLS1.2
TLS_RSA_PSK_NULL_SHA1	0x00 0x2E	TLS1.0
TLS_RSA_PSK_NULL_SHA256	0x00 0xB8	TLS1.2
TLS_RSA_PSK_AES_256_GCM_SHA384	0x00 0xAD	TLS1.2
TLS_RSA_PSK_AES_256_CBC_SHA384	0x00 0xB7	TLS1.2
TLS_RSA_PSK_NULL_SHA384	0x00 0xB9	TLS1.2
TLS_RSA_PSK_CAMELLIA_128_CBC_SHA256	0xC0 0x98	TLS1.2
TLS_RSA_PSK_CAMELLIA_256_CBC_SHA384	0xC0 0x99	TLS1.2
TLS_DHE_PSK_ARCFOUR_128_SHA1	0x00 0x8E	SSL3.0
TLS_DHE_PSK_3DES_EDE_CBC_SHA1	0x00 0x8F	SSL3.0
TLS_DHE_PSK_AES_128_CBC_SHA1	0x00 0x90	SSL3.0
TLS_DHE_PSK_AES_256_CBC_SHA1	0x00 0x91	SSL3.0
TLS_DHE_PSK_AES_128_CBC_SHA256	0x00 0xB2	TLS1.2
TLS_DHE_PSK_AES_128_GCM_SHA256	0x00 0xAA	TLS1.2

TLS_DHE_PSK_NULL_SHA1	0x00 0x2D	SSL3.0
TLS_DHE_PSK_NULL_SHA256	0x00 0xB4	TLS1.2
TLS_DHE_PSK_NULL_SHA384	0x00 0xB5	TLS1.2
TLS_DHE_PSK_AES_256_CBC_SHA384	0x00 0xB3	TLS1.2
TLS_DHE_PSK_AES_256_GCM_SHA384	0x00 0xAB	TLS1.2
TLS_DHE_PSK_CAMELLIA_128_CBC_SHA256	0xC0 0x96	TLS1.2
TLS_DHE_PSK_CAMELLIA_256_CBC_SHA384	0xC0 0x97	TLS1.2
TLS_DHE_PSK_CAMELLIA_128_GCM_SHA256	0xC0 0x90	TLS1.2
TLS_DHE_PSK_CAMELLIA_256_GCM_SHA384	0xC0 0x91	TLS1.2
TLS_DH_ANON_ARCFOUR_128_MD5	0x00 0x18	SSL3.0
TLS_DH_ANON_3DES_EDE_CBC_SHA1	0x00 0x1B	SSL3.0
TLS_DH_ANON_AES_128_CBC_SHA1	0x00 0x34	SSL3.0
TLS_DH_ANON_AES_256_CBC_SHA1	0x00 0x3A	SSL3.0
TLS_DH_ANON_CAMELLIA_128_CBC_SHA256	0x00 0xBF	TLS1.2
TLS_DH_ANON_CAMELLIA_256_CBC_SHA256	0x00 0xC5	TLS1.2
TLS_DH_ANON_CAMELLIA_128_CBC_SHA1	0x00 0x46	SSL3.0
TLS_DH_ANON_CAMELLIA_256_CBC_SHA1	0x00 0x89	SSL3.0
TLS_DH_ANON_AES_128_CBC_SHA256	0x00 0x6C	TLS1.2
TLS_DH_ANON_AES_256_CBC_SHA256	0x00 0x6D	TLS1.2
TLS_DH_ANON_AES_128_GCM_SHA256	0x00 0xA6	TLS1.2
TLS_DH_ANON_AES_256_GCM_SHA384	0x00 0xA7	TLS1.2
TLS_DH_ANON_CAMELLIA_128_GCM_SHA256	0xC0 0x84	TLS1.2
TLS_DH_ANON_CAMELLIA_256_GCM_SHA384	0xC0 0x85	TLS1.2
TLS_ECDH_ANON_NULL_SHA1	0xC0 0x15	SSL3.0
TLS_ECDH_ANON_3DES_EDE_CBC_SHA1	0xC0 0x17	SSL3.0
TLS_ECDH_ANON_AES_128_CBC_SHA1	0xC0 0x18	SSL3.0
TLS_ECDH_ANON_AES_256_CBC_SHA1	0xC0 0x19	SSL3.0
TLS_ECDH_ANON_ARCFOUR_128_SHA1	0xC0 0x16	SSL3.0
TLS_SRP_SHA_3DES_EDE_CBC_SHA1	0xC0 0x1A	SSL3.0
TLS_SRP_SHA_AES_128_CBC_SHA1	0xC0 0x1D	SSL3.0
TLS_SRP_SHA_AES_256_CBC_SHA1	0xC0 0x20	SSL3.0
TLS_SRP_SHA_DSS_3DES_EDE_CBC_SHA1	0xC0 0x1C	SSL3.0
TLS_SRP_SHA_RSA_3DES_EDE_CBC_SHA1	0xC0 0x1B	SSL3.0
TLS_SRP_SHA_DSS_AES_128_CBC_SHA1	0xC0 0x1F	SSL3.0
TLS_SRP_SHA_RSA_AES_128_CBC_SHA1	0xC0 0x1E	SSL3.0
TLS_SRP_SHA_DSS_AES_256_CBC_SHA1	0xC0 0x22	SSL3.0
TLS_SRP_SHA_RSA_AES_256_CBC_SHA1	0xC0 0x21	SSL3.0

## Certificate types

X.509

OPENPGP

## Protocols

SSL3.0

TLS1.0

TLS1.1

TLS1.2

DTLS0.9

DTLS1.0

DTLS1.2

## Ciphers

AES-256-CBC

AES-192-CBC

AES-128-CBC

AES-128-GCM

AES-256-GCM

ARCFOUR-128

ESTREAM-SALSA20-256

SALSA20-256

CAMELLIA-256-CBC

CAMELLIA-192-CBC

CAMELLIA-128-CBC

CAMELLIA-128-GCM

CAMELLIA-256-GCM

3DES-CBC

DES-CBC

ARCFOUR-40

RC2-40

NULL

## MAC algorithms

SHA1

MD5

SHA256

SHA384

SHA512

SHA224

UMAC-96

UMAC-128

AEAD

## Key exchange methods

ANON-DH

ANON-ECDH

RSA

DHE-RSA

DHE-DSS

ECDHE-RSA

ECDHE-ECDSA

SRP-DSS

SRP-RSA

SRP

PSK

RSA-PSK

DHE-PSK

ECDHE-PSK

RSA-EXPORT

## Public key algorithms

RSA

DSA

EC

## Public key signature algorithms

RSA-SHA1

RSA-SHA1

RSA-SHA224

RSA-SHA256

RSA-SHA384

RSA-SHA512

RSA-RMD160

DSA-SHA1

DSA-SHA1

DSA-SHA224

DSA-SHA256

RSA-MD5

RSA-MD5

RSA-MD2

ECDSA-SHA1  
ECDSA-SHA224  
ECDSA-SHA256  
ECDSA-SHA384  
ECDSA-SHA512

## **Elliptic curves**

SECP192R1  
SECP224R1  
SECP256R1  
SECP384R1  
SECP521R1

## **Compression methods**

DEFLATE  
NULL



## Appendix E API reference

### E.1 Core TLS API

The prototypes for the following functions lie in `gnutls/gnutls.h`.

#### `gnutls_alert_get`

`gnutls_alert_description_t gnutls_alert_get (gnutls_session_t session)` [Function]

*session*: is a `gnutls_session_t` structure.

This function will return the last alert number received. This function should be called when `GNUTLS_E_WARNING_ALERT_RECEIVED` or `GNUTLS_E_FATAL_ALERT_RECEIVED` errors are returned by a gnutls function. The peer may send alerts if he encounters an error. If no alert has been received the returned value is undefined.

**Returns:** the last alert received, a `gnutls_alert_description_t` value.

#### `gnutls_alert_get_name`

`const char * gnutls_alert_get_name (gnutls_alert_description_t alert)` [Function]

*alert*: is an alert number.

This function will return a string that describes the given alert number, or `NULL`. See `gnutls_alert_get()`.

**Returns:** string corresponding to `gnutls_alert_description_t` value.

#### `gnutls_alert_get_strname`

`const char * gnutls_alert_get_strname (gnutls_alert_description_t alert)` [Function]

*alert*: is an alert number.

This function will return a string of the name of the alert.

**Returns:** string corresponding to `gnutls_alert_description_t` value.

**Since:** 3.0

#### `gnutls_alert_send`

`int gnutls_alert_send (gnutls_session_t session, gnutls_alert_level_t level, gnutls_alert_description_t desc)` [Function]

*session*: is a `gnutls_session_t` structure.

*level*: is the level of the alert

*desc*: is the alert description

This function will send an alert to the peer in order to inform him of something important (eg. his Certificate could not be verified). If the alert level is Fatal then the peer is expected to close the connection, otherwise he may ignore the alert and continue.

The error code of the underlying record send function will be returned, so you may also receive `GNUTLS_E_INTERRUPTED` or `GNUTLS_E_AGAIN` as well.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

## **gnutls\_alert\_send\_appropriate**

**int gnutls\_alert\_send\_appropriate** (*gnutls\_session\_t session*, *int err*) [Function]

*session*: is a `gnutls_session_t` structure.

*err*: is an integer

Sends an alert to the peer depending on the error code returned by a gnutls function. This function will call `gnutls_error_to_alert()` to determine the appropriate alert to send.

This function may also return `GNUTLS_E_AGAIN`, or `GNUTLS_E_INTERRUPTED`.

If the return value is `GNUTLS_E_INVALID_REQUEST`, then no alert has been sent to the peer.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

## **gnutls\_alpn\_get\_selected\_protocol**

**int gnutls\_alpn\_get\_selected\_protocol** (*gnutls\_session\_t session*, *gnutls\_datum\_t \*protocol*) [Function]

*session*: is a `gnutls_session_t` structure.

*protocol*: will hold the protocol name

This function allows you to get the negotiated protocol name. The returned protocol should be treated as opaque, constant value and only valid during the session life.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

Since 3.2.0

## **gnutls\_alpn\_set\_protocols**

**int gnutls\_alpn\_set\_protocols** (*gnutls\_session\_t session*, *const gnutls\_datum\_t \*protocols*, *unsigned protocols\_size*, *unsigned int flags*) [Function]

*session*: is a `gnutls_session_t` structure.

*protocols*: is the protocol names to add.

*protocols\_size*: the number of protocols to add.

*flags*: zero or `GNUTLS_ALPN_*`

This function is to be used by both clients and servers, to declare the supported ALPN protocols, which are used during negotiation with peer.

If `GNUTLS_ALPN_MAND` is specified the connection will be aborted if no matching ALPN protocol is found.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error code is returned.

Since 3.2.0

### **gnutls\_anon\_allocate\_client\_credentials**

**int gnutls\_anon\_allocate\_client\_credentials** [Function]  
     (*gnutls\_anon\_client\_credentials\_t \* sc*)

*sc*: is a pointer to a **gnutls\_anon\_client\_credentials\_t** structure.

This structure is complex enough to manipulate directly thus this helper function is provided in order to allocate it.

**Returns:** GNUTLS\_E\_SUCCESS on success, or an error code.

### **gnutls\_anon\_allocate\_server\_credentials**

**int gnutls\_anon\_allocate\_server\_credentials** [Function]  
     (*gnutls\_anon\_server\_credentials\_t \* sc*)

*sc*: is a pointer to a **gnutls\_anon\_server\_credentials\_t** structure.

This structure is complex enough to manipulate directly thus this helper function is provided in order to allocate it.

**Returns:** GNUTLS\_E\_SUCCESS on success, or an error code.

### **gnutls\_anon\_free\_client\_credentials**

**void gnutls\_anon\_free\_client\_credentials** [Function]  
     (*gnutls\_anon\_client\_credentials\_t sc*)

*sc*: is a **gnutls\_anon\_client\_credentials\_t** structure.

This structure is complex enough to manipulate directly thus this helper function is provided in order to free (deallocate) it.

### **gnutls\_anon\_free\_server\_credentials**

**void gnutls\_anon\_free\_server\_credentials** [Function]  
     (*gnutls\_anon\_server\_credentials\_t sc*)

*sc*: is a **gnutls\_anon\_server\_credentials\_t** structure.

This structure is complex enough to manipulate directly thus this helper function is provided in order to free (deallocate) it.

### **gnutls\_anon\_set\_params\_function**

**void gnutls\_anon\_set\_params\_function** [Function]  
     (*gnutls\_anon\_server\_credentials\_t res, gnutls\_params\_function \* func*)

*res*: is a **gnutls\_anon\_server\_credentials\_t** structure

*func*: is the function to be called

This function will set a callback in order for the server to get the Diffie-Hellman or RSA parameters for anonymous authentication. The callback should return GNUTLS\_E\_SUCCESS (0) on success.

## gnutls\_anon\_set\_server\_dh\_params

`void gnutls_anon_set_server_dh_params` [Function]

(*gnutls\_anon\_server\_credentials\_t* **res**, *gnutls\_dh\_params\_t* **dh\_params**)

*res*: is a *gnutls\_anon\_server\_credentials\_t* structure

*dh\_params*: is a structure that holds Diffie-Hellman parameters.

This function will set the Diffie-Hellman parameters for an anonymous server to use.

These parameters will be used in Anonymous Diffie-Hellman cipher suites.

## gnutls\_anon\_set\_server\_params\_function

`void gnutls_anon_set_server_params_function` [Function]

(*gnutls\_anon\_server\_credentials\_t* **res**, *gnutls\_params\_function* \* **func**)

*res*: is a *gnutls\_certificate\_credentials\_t* structure

*func*: is the function to be called

This function will set a callback in order for the server to get the Diffie-Hellman parameters for anonymous authentication. The callback should return `GNUTLS_E_SUCCESS` (0) on success.

## gnutls\_auth\_client\_get\_type

`gnutls_credentials_type_t gnutls_auth_client_get_type` [Function]

(*gnutls\_session\_t* **session**)

*session*: is a *gnutls\_session\_t* structure.

Returns the type of credentials that were used for client authentication. The returned information is to be used to distinguish the function used to access authentication data.

**Returns:** The type of credentials for the client authentication schema, a *gnutls\_credentials\_type\_t* type.

## gnutls\_auth\_get\_type

`gnutls_credentials_type_t gnutls_auth_get_type` [Function]

(*gnutls\_session\_t* **session**)

*session*: is a *gnutls\_session\_t* structure.

Returns type of credentials for the current authentication schema. The returned information is to be used to distinguish the function used to access authentication data.

Eg. for CERTIFICATE ciphersuites (key exchange algorithms: `GNUTLS_KX_RSA` , `GNUTLS_KX_DHE_RSA` ), the same function are to be used to access the authentication data.

**Returns:** The type of credentials for the current authentication schema, a *gnutls\_credentials\_type\_t* type.

## gnutls\_auth\_server\_get\_type

`gnutls_credentials_type_t gnutls_auth_server_get_type` [Function]  
 (`gnutls_session_t session`)

*session*: is a `gnutls_session_t` structure.

Returns the type of credentials that were used for server authentication. The returned information is to be used to distinguish the function used to access authentication data.

**Returns:** The type of credentials for the server authentication schema, a `gnutls_credentials_type_t` type.

## gnutls\_bye

`int gnutls_bye` (`gnutls_session_t session`, `gnutls_close_request_t how`) [Function]  
*session*: is a `gnutls_session_t` structure.

*how*: is an integer

Terminates the current TLS/SSL connection. The connection should have been initiated using `gnutls_handshake()`. *how* should be one of `GNUTLS_SHUT_RDWR`, `GNUTLS_SHUT_WR`.

In case of `GNUTLS_SHUT_RDWR` the TLS session gets terminated and further receives and sends will be disallowed. If the return value is zero you may continue using the underlying transport layer. `GNUTLS_SHUT_RDWR` sends an alert containing a close request and waits for the peer to reply with the same message.

In case of `GNUTLS_SHUT_WR` the TLS session gets terminated and further sends will be disallowed. In order to reuse the connection you should wait for an EOF from the peer. `GNUTLS_SHUT_WR` sends an alert containing a close request.

Note that not all implementations will properly terminate a TLS connection. Some of them, usually for performance reasons, will terminate only the underlying transport layer, and thus not distinguishing between a malicious party prematurely terminating the connection and normal termination.

This function may also return `GNUTLS_E_AGAIN` or `GNUTLS_E_INTERRUPTED`; cf. `gnutls_record_get_direction()`.

**Returns:** `GNUTLS_E_SUCCESS` on success, or an error code, see function documentation for entire semantics.

## gnutls\_certificate\_activation\_time\_peers

`time_t gnutls_certificate_activation_time_peers` [Function]  
 (`gnutls_session_t session`)

*session*: is a gnutls session

This function will return the peer's certificate activation time. This is the creation time for openpgp keys.

**Returns:** (time\_t)-1 on error.

**Deprecated:** `gnutls_certificate_verify_peers2()` now verifies activation times.

**gnutls\_certificate\_allocate\_credentials**

**int gnutls\_certificate\_allocate\_credentials** [Function]  
     (*gnutls\_certificate\_credentials\_t* \* *res*)

*res*: is a pointer to a *gnutls\_certificate\_credentials\_t* structure.

This structure is complex enough to manipulate directly thus this helper function is provided in order to allocate it.

**Returns:** GNUTLS\_E\_SUCCESS on success, or an error code.

**gnutls\_certificate\_client\_get\_request\_status**

**int gnutls\_certificate\_client\_get\_request\_status** [Function]  
     (*gnutls\_session\_t* *session*)

*session*: is a gnutls session

Get whether client certificate is requested or not.

**Returns:** 0 if the peer (server) did not request client authentication or 1 otherwise.

**gnutls\_certificate\_expiration\_time\_peers**

**time\_t gnutls\_certificate\_expiration\_time\_peers** [Function]  
     (*gnutls\_session\_t* *session*)

*session*: is a gnutls session

This function will return the peer's certificate expiration time.

**Returns:** (time\_t)-1 on error.

**Deprecated:** *gnutls\_certificate\_verify\_peers2()* now verifies expiration times.

**gnutls\_certificate\_free\_ca\_names**

**void gnutls\_certificate\_free\_ca\_names** [Function]  
     (*gnutls\_certificate\_credentials\_t* *sc*)

*sc*: is a *gnutls\_certificate\_credentials\_t* structure.

This function will delete all the CA name in the given credentials. Clients may call this to save some memory since in client side the CA names are not used. Servers might want to use this function if a large list of trusted CAs is present and sending the names of it would just consume bandwidth without providing information to client.

CA names are used by servers to advertise the CAs they support to clients.

**gnutls\_certificate\_free\_cas**

**void gnutls\_certificate\_free\_cas** (*gnutls\_certificate\_credentials\_t* [Function]  
     *sc*)

*sc*: is a *gnutls\_certificate\_credentials\_t* structure.

This function will delete all the CAs associated with the given credentials. Servers that do not use *gnutls\_certificate\_verify\_peers2()* may call this to save some memory.

**gnutls\_certificate\_free\_credentials**

**void gnutls\_certificate\_free\_credentials** [Function]  
     (*gnutls\_certificate\_credentials\_t sc*)

*sc*: is a *gnutls\_certificate\_credentials\_t* structure.

This structure is complex enough to manipulate directly thus this helper function is provided in order to free (deallocate) it.

This function does not free any temporary parameters associated with this structure (ie RSA and DH parameters are not freed by this function).

**gnutls\_certificate\_free\_crls**

**void gnutls\_certificate\_free\_crls** (*gnutls\_certificate\_credentials\_t* [Function]  
     *sc*)

*sc*: is a *gnutls\_certificate\_credentials\_t* structure.

This function will delete all the CRLs associated with the given credentials.

**gnutls\_certificate\_free\_keys**

**void gnutls\_certificate\_free\_keys** (*gnutls\_certificate\_credentials\_t* [Function]  
     *sc*)

*sc*: is a *gnutls\_certificate\_credentials\_t* structure.

This function will delete all the keys and the certificates associated with the given credentials. This function must not be called when a TLS negotiation that uses the credentials is in progress.

**gnutls\_certificate\_get\_crt\_raw**

**int gnutls\_certificate\_get\_crt\_raw** [Function]  
     (*gnutls\_certificate\_credentials\_t sc*, unsigned *idx1*, unsigned *idx2*,  
     *gnutls\_datum\_t* \* *cert*)

*sc*: is a *gnutls\_certificate\_credentials\_t* structure.

*idx1*: the index of the certificate if multiple are present

*idx2*: the index in the certificate list. Zero gives the server's certificate.

*cert*: Will hold the DER encoded certificate.

This function will return the DER encoded certificate of the server or any other certificate on its certificate chain (based on *idx2* ). The returned data should be treated as constant and only accessible during the lifetime of *sc* .

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value. In case the indexes are out of bounds GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE is returned.

**Since:** 3.2.5

## gnutls\_certificate\_get\_issuer

`int gnutls_certificate_get_issuer (gnutls_certificate_credentials_t [Function]  
                                   sc, gnutls_x509_crt_t cert, gnutls_x509_crt_t * issuer, unsigned int flags)`

*sc*: is a `gnutls_certificate_credentials_t` structure.

*cert*: is the certificate to find issuer for

*issuer*: Will hold the issuer if any. Should be treated as constant.

*flags*: Use zero or `GNUTLS_TL_GET_COPY`

This function will return the issuer of a given certificate. As with `gnutls_x509_trust_list_get_issuer()` this function requires the `GNUTLS_TL_GET_COPY` flag in order to operate with PKCS 11 trust lists. In that case the issuer must be freed using `gnutls_x509_crt_deinit()` .

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.0

## gnutls\_certificate\_get\_ours

`const gnutls_datum_t * gnutls_certificate_get_ours [Function]  
                                   (gnutls_session_t session)`

*session*: is a gnutls session

Gets the certificate as sent to the peer in the last handshake. The certificate is in raw (DER) format. No certificate list is being returned. Only the first certificate.

**Returns:** a pointer to a `gnutls_datum_t` containing our certificate, or `NULL` in case of an error or if no certificate was used.

## gnutls\_certificate\_get\_peers

`const gnutls_datum_t * gnutls_certificate_get_peers [Function]  
                                   (gnutls_session_t session, unsigned int * list_size)`

*session*: is a gnutls session

*list\_size*: is the length of the certificate list (may be `NULL` )

Get the peer's raw certificate (chain) as sent by the peer. These certificates are in raw format (DER encoded for X.509). In case of a X.509 then a certificate list may be present. The first certificate in the list is the peer's certificate, following the issuer's certificate, then the issuer's issuer etc.

In case of OpenPGP keys a single key will be returned in raw format.

**Returns:** a pointer to a `gnutls_datum_t` containing the peer's certificates, or `NULL` in case of an error or if no certificate was used.

## gnutls\_certificate\_get\_peers\_subkey\_id

`int gnutls_certificate_get_peers_subkey_id (gnutls_session_t [Function]  
                                   session, gnutls_datum_t * id)`

*session*: is a gnutls session



*id*: will contain the ID

Get the peer's subkey ID when OpenPGP certificates are used. The returned *id* should be treated as constant.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise an error code is returned.

**Since:** 3.1.3

## gnutls\_certificate\_send\_x509\_rdn\_sequence

void gnutls\_certificate\_send\_x509\_rdn\_sequence (gnutls\_session\_t *session*, int *status*) [Function]

*session*: is a pointer to a gnutls\_session\_t structure.

*status*: is 0 or 1

If *status* is non zero, this function will order gnutls not to send the rdnSequence in the certificate request message. That is the server will not advertise its trusted CAs to the peer. If *status* is zero then the default behaviour will take effect, which is to advertise the server's trusted CAs.

This function has no effect in clients, and in authentication methods other than certificate with X.509 certificates.

## gnutls\_certificate\_server\_set\_request

void gnutls\_certificate\_server\_set\_request (gnutls\_session\_t *session*, gnutls\_certificate\_request\_t *req*) [Function]

*session*: is a gnutls\_session\_t structure.

*req*: is one of GNUTLS\_CERT\_REQUEST, GNUTLS\_CERT\_REQUIRE

This function specifies if we (in case of a server) are going to send a certificate request message to the client. If *req* is GNUTLS\_CERT\_REQUIRE then the server will return an error if the peer does not provide a certificate. If you do not call this function then the client will not be asked to send a certificate.

## gnutls\_certificate\_set\_dh\_params

void gnutls\_certificate\_set\_dh\_params (gnutls\_certificate\_credentials\_t *res*, gnutls\_dh\_params\_t *dh\_params*) [Function]

*res*: is a gnutls\_certificate\_credentials\_t structure

*dh\_params*: is a structure that holds Diffie-Hellman parameters.

This function will set the Diffie-Hellman parameters for a certificate server to use. These parameters will be used in Ephemeral Diffie-Hellman cipher suites. Note that only a pointer to the parameters are stored in the certificate handle, so you must not deallocate the parameters before the certificate is deallocated.

## gnutls\_certificate\_set\_ocsp\_status\_request\_file

int gnutls\_certificate\_set\_ocsp\_status\_request\_file (gnutls\_certificate\_credentials\_t *sc*, const char \* *response\_file*, unsigned int *flags*) [Function]

*sc*: is a credentials structure.

*response\_file*: a filename of the OCSF response

*flags*: should be zero

This function sets the filename of an OCSF response, that will be sent to the client if requests an OCSF certificate status. This is a convenience function which is inefficient on busy servers since the file is opened on every access. Use `gnutls_certificate_set_ocsp_status_request_function()` to fine-tune file accesses.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

**Since:** 3.1.3

## `gnutls_certificate_set_ocsp_status_request_function`

```
void gnutls_certificate_set_ocsp_status_request_function      [Function]
    (gnutls_certificate_credentials_t sc, gnutls_status_request_ocsp_func
    ocsp_func, void *ptr)
```

*sc*: is a `gnutls_certificate_credentials_t` structure.

*ocsp\_func*: function pointer to OCSF status request callback.

*ptr*: opaque pointer passed to callback function

This function is to be used by server to register a callback to handle OCSF status requests from the client. The callback will be invoked if the client supplied a status-request OCSF extension. The callback function prototype is:

```
typedef int (*gnutls_status_request_ocsp_func) (gnutls_session_t session, void *ptr,
gnutls_datum_t *ocsp_response);
```

The callback will be invoked if the client requests an OCSF certificate status. The callback may return `GNUTLS_E_NO_CERTIFICATE_STATUS` , if there is no recent OCSF response. If the callback returns `GNUTLS_E_SUCCESS` , the server will provide the client with the `ocsp_response`.

The response must be a value allocated using `gnutls_malloc()` , and will be deinitialized when needed.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

**Since:** 3.1.3

## `gnutls_certificate_set_params_function`

```
void gnutls_certificate_set_params_function                  [Function]
    (gnutls_certificate_credentials_t res, gnutls_params_function *func)
```

*res*: is a `gnutls_certificate_credentials_t` structure

*func*: is the function to be called

This function will set a callback in order for the server to get the Diffie-Hellman or RSA parameters for certificate authentication. The callback should return `GNUTLS_E_SUCCESS` (0) on success.

## gnutls\_certificate\_set\_pin\_function

```
void gnutls_certificate_set_pin_function [Function]
    (gnutls_certificate_credentials_t cred, gnutls_pin_callback_t fn, void *
    userdata)
```

*cred*: is a `gnutls_certificate_credentials_t` structure.

*fn*: A PIN callback

*userdata*: Data to be passed in the callback

This function will set a callback function to be used when required to access a protected object. This function overrides any other global PIN functions.

Note that this function must be called right after initialization to have effect.

**Since:** 3.1.0

## gnutls\_certificate\_set\_retrieve\_function

```
void gnutls_certificate_set_retrieve_function [Function]
    (gnutls_certificate_credentials_t cred, gnutls_certificate_retrieve_function *
    func)
```

*cred*: is a `gnutls_certificate_credentials_t` structure.

*func*: is the callback function

This function sets a callback to be called in order to retrieve the certificate to be used in the handshake. You are advised to use `gnutls_certificate_set_retrieve_function2()` because it is much more efficient in the processing it requires from gnutls.

The callback's function prototype is: `int (*callback)(gnutls_session_t, const gnutls_datum_t* req_ca_dn, int nreqs, const gnutls_pk_algorithm_t* pk_algos, int pk_algos_length, gnutls_retr2_st* st);`

`req_ca_dn` is only used in X.509 certificates. Contains a list with the CA names that the server considers trusted. Normally we should send a certificate that is signed by one of these CAs. These names are DER encoded. To get a more meaningful value use the function `gnutls_x509_rdn_get()`.

`pk_algos` contains a list with server's acceptable signature algorithms. The certificate returned should support the server's given algorithms.

`st` should contain the certificates and private keys.

If the callback function is provided then gnutls will call it, in the handshake, after the certificate request message has been received.

In server side `pk_algos` and `req_ca_dn` are NULL.

The callback function should set the certificate list to be sent, and return 0 on success. If no certificate was selected then the number of certificates should be set to zero. The value (-1) indicates error and the handshake will be terminated.

**Since:** 3.0

## gnutls\_certificate\_set\_verify\_flags

**void gnutls\_certificate\_set\_verify\_flags** [Function]

(*gnutls\_certificate\_credentials\_t res, unsigned int flags*)

*res*: is a `gnutls_certificate_credentials_t` structure

*flags*: are the flags

This function will set the flags to be used for verification of certificates and override any defaults. The provided flags must be an OR of the `gnutls_certificate_verify_flags` enumerations.

## gnutls\_certificate\_set\_verify\_function

**void gnutls\_certificate\_set\_verify\_function** [Function]

(*gnutls\_certificate\_credentials\_t cred, gnutls\_certificate\_verify\_function \* func*)

*cred*: is a `gnutls_certificate_credentials_t` structure.

*func*: is the callback function

This function sets a callback to be called when peer's certificate has been received in order to verify it on receipt rather than doing after the handshake is completed.

The callback's function prototype is: `int (*callback)(gnutls_session_t);`

If the callback function is provided then gnutls will call it, in the handshake, just after the certificate message has been received. To verify or obtain the certificate the `gnutls_certificate_verify_peers2()` , `gnutls_certificate_type_get()` , `gnutls_certificate_get_peers()` functions can be used.

The callback function should return 0 for the handshake to continue or non-zero to terminate.

**Since:** 2.10.0

## gnutls\_certificate\_set\_verify\_limits

**void gnutls\_certificate\_set\_verify\_limits** [Function]

(*gnutls\_certificate\_credentials\_t res, unsigned int max\_bits, unsigned int max\_depth*)

*res*: is a `gnutls_certificate_credentials` structure

*max\_bits*: is the number of bits of an acceptable certificate (default 8200)

*max\_depth*: is maximum depth of the verification of a certificate chain (default 5)

This function will set some upper limits for the default verification function, `gnutls_certificate_verify_peers2()` , to avoid denial of service attacks. You can set them to zero to disable limits.

## gnutls\_certificate\_set\_x509\_crl

**int gnutls\_certificate\_set\_x509\_crl** [Function]

(*gnutls\_certificate\_credentials\_t res, gnutls\_x509\_crl\_t \* crl\_list, int crl\_list\_size*)

*res*: is a `gnutls_certificate_credentials_t` structure.

*crl\_list*: is a list of trusted CRLs. They should have been verified before.

*crl\_list\_size*: holds the size of the *crl\_list*

This function adds the trusted CRLs in order to verify client or server certificates. In case of a client this is not required to be called if the certificates are not verified using `gnutls_certificate_verify_peers2()`. This function may be called multiple times.

**Returns:** number of CRLs processed, or a negative error code on error.

**Since:** 2.4.0

## **gnutls\_certificate\_set\_x509\_crl\_file**

```
int gnutls_certificate_set_x509_crl_file           [Function]
    (gnutls_certificate_credentials_t res, const char * crlfile,
     gnutls_x509_crt_fmt_t type)
```

*res*: is a `gnutls_certificate_credentials_t` structure.

*crlfile*: is a file containing the list of verified CRLs (DER or PEM list)

*type*: is PEM or DER

This function adds the trusted CRLs in order to verify client or server certificates. In case of a client this is not required to be called if the certificates are not verified using `gnutls_certificate_verify_peers2()`. This function may be called multiple times.

**Returns:** number of CRLs processed or a negative error code on error.

## **gnutls\_certificate\_set\_x509\_crl\_mem**

```
int gnutls_certificate_set_x509_crl_mem           [Function]
    (gnutls_certificate_credentials_t res, const gnutls_datum_t * CRL,
     gnutls_x509_crt_fmt_t type)
```

*res*: is a `gnutls_certificate_credentials_t` structure.

*CRL*: is a list of trusted CRLs. They should have been verified before.

*type*: is DER or PEM

This function adds the trusted CRLs in order to verify client or server certificates. In case of a client this is not required to be called if the certificates are not verified using `gnutls_certificate_verify_peers2()`. This function may be called multiple times.

**Returns:** number of CRLs processed, or a negative error code on error.

## **gnutls\_certificate\_set\_x509\_key**

```
int gnutls_certificate_set_x509_key               [Function]
    (gnutls_certificate_credentials_t res, gnutls_x509_crt_t * cert_list, int
     cert_list_size, gnutls_x509_privkey_t key)
```

*res*: is a `gnutls_certificate_credentials_t` structure.

*cert\_list*: contains a certificate list (path) for the specified private key

*cert\_list\_size*: holds the size of the certificate list

*key*: is a `gnutls_x509_privkey_t` key

This function sets a certificate/private key pair in the `gnutls_certificate_credentials_t` structure. This function may be called more than once, in case multiple keys/certificates exist for the server. For clients that wants to send more than their own end entity certificate (e.g., also an intermediate CA cert) then put the certificate chain in `cert_list`.

Note that the certificates and keys provided, can be safely deinitialized after this function is called.

If that function fails to load the `res` structure is at an undefined state, it must not be reused to load other keys or certificates.

**Returns:** GNUTLS\_E\_SUCCESS (0) on success, or a negative error code.

**Since:** 2.4.0

### **gnutls\_certificate\_set\_x509\_key\_file**

```
int gnutls_certificate_set_x509_key_file [Function]
    (gnutls_certificate_credentials_t res, const char * certfile, const char *
    keyfile, gnutls_x509_crt_fmt_t type)
```

*res*: is a `gnutls_certificate_credentials_t` structure.

*certfile*: is a file that containing the certificate list (path) for the specified private key, in PKCS7 format, or a list of certificates

*keyfile*: is a file that contains the private key

*type*: is PEM or DER

This function sets a certificate/private key pair in the `gnutls_certificate_credentials_t` structure. This function may be called more than once, in case multiple keys/certificates exist for the server. For clients that need to send more than its own end entity certificate, e.g., also an intermediate CA cert, then the `certfile` must contain the ordered certificate chain.

Note that the names in the certificate provided will be considered when selecting the appropriate certificate to use (in case of multiple certificate/key pairs).

This function can also accept URLs at `keyfile` and `certfile`. In that case it will import the private key and certificate indicated by the URLs. Note that the supported URLs are the ones indicated by `gnutls_url_is_supported()`.

In case the `certfile` is provided as a PKCS 11 URL, then the certificate, and its present issuers in the token are imported (i.e., the required trust chain).

If that function fails to load the `res` structure is at an undefined state, it must not be reused to load other keys or certificates.

**Returns:** GNUTLS\_E\_SUCCESS (0) on success, or a negative error code.

### **gnutls\_certificate\_set\_x509\_key\_file2**

```
int gnutls_certificate_set_x509_key_file2 [Function]
    (gnutls_certificate_credentials_t res, const char * certfile, const char *
    keyfile, gnutls_x509_crt_fmt_t type, const char * pass, unsigned int flags)
```

*res*: is a `gnutls_certificate_credentials_t` structure.

*certfile*: is a file that containing the certificate list (path) for the specified private key, in PKCS7 format, or a list of certificates

*keyfile*: is a file that contains the private key

*type*: is PEM or DER

*pass*: is the password of the key

*flags*: an ORed sequence of `gnutls_pkcs_encrypt_flags_t`

This function sets a certificate/private key pair in the `gnutls_certificate_credentials_t` structure. This function may be called more than once, in case multiple keys/certificates exist for the server. For clients that need to send more than its own end entity certificate, e.g., also an intermediate CA cert, then the `certfile` must contain the ordered certificate chain.

Note that the names in the certificate provided will be considered when selecting the appropriate certificate to use (in case of multiple certificate/key pairs).

This function can also accept URLs at `keyfile` and `certfile`. In that case it will import the private key and certificate indicated by the URLs. Note that the supported URLs are the ones indicated by `gnutls_url_is_supported()`.

In case the `certfile` is provided as a PKCS 11 URL, then the certificate, and its present issuers in the token are imported (i.e., the required trust chain).

If that function fails to load the `res` structure is at an undefined state, it must not be reused to load other keys or certificates.

**Returns:** `GNUTLS_E_SUCCESS` (0) on success, or a negative error code.

## `gnutls_certificate_set_x509_key_mem`

`int gnutls_certificate_set_x509_key_mem` [Function]  
     (`gnutls_certificate_credentials_t res`, `const gnutls_datum_t * cert`, `const gnutls_datum_t * key`, `gnutls_x509_crt_fmt_t type`)

*res*: is a `gnutls_certificate_credentials_t` structure.

*cert*: contains a certificate list (path) for the specified private key

*key*: is the private key, or NULL

*type*: is PEM or DER

This function sets a certificate/private key pair in the `gnutls_certificate_credentials_t` structure. This function may be called more than once, in case multiple keys/certificates exist for the server.

Note that the keyUsage (2.5.29.15) PKIX extension in X.509 certificates is supported. This means that certificates intended for signing cannot be used for ciphersuites that require encryption.

If the certificate and the private key are given in PEM encoding then the strings that hold their values must be null terminated.

The `key` may be NULL if you are using a sign callback, see `gnutls_sign_callback_set()`.

**Returns:** `GNUTLS_E_SUCCESS` (0) on success, or a negative error code.

**gnutls\_certificate\_set\_x509\_key\_mem2**

**int gnutls\_certificate\_set\_x509\_key\_mem2** [Function]  
 (gnutls\_certificate\_credentials\_t *res*, const gnutls\_datum\_t \* *cert*, const  
 gnutls\_datum\_t \* *key*, gnutls\_x509\_crt\_fmt\_t *type*, const char \* *pass*,  
 unsigned int *flags*)

*res*: is a gnutls\_certificate\_credentials\_t structure.

*cert*: contains a certificate list (path) for the specified private key

*key*: is the private key, or NULL

*type*: is PEM or DER

*pass*: is the key's password

*flags*: an ORed sequence of gnutls\_pkcs\_encrypt\_flags\_t

This function sets a certificate/private key pair in the gnutls\_certificate\_credentials\_t structure. This function may be called more than once, in case multiple keys/certificates exist for the server.

Note that the keyUsage (2.5.29.15) PKIX extension in X.509 certificates is supported. This means that certificates intended for signing cannot be used for ciphersuites that require encryption.

If the certificate and the private key are given in PEM encoding then the strings that hold their values must be null terminated.

The *key* may be NULL if you are using a sign callback, see gnutls\_sign\_callback\_set() .

**Returns:** GNUTLS\_E\_SUCCESS (0) on success, or a negative error code.

**gnutls\_certificate\_set\_x509\_simple\_pkcs12\_file**

**int gnutls\_certificate\_set\_x509\_simple\_pkcs12\_file** [Function]  
 (gnutls\_certificate\_credentials\_t *res*, const char \* *pkcs12file*,  
 gnutls\_x509\_crt\_fmt\_t *type*, const char \* *password*)

*res*: is a gnutls\_certificate\_credentials\_t structure.

*pkcs12file*: filename of file containing PKCS12 blob.

*type*: is PEM or DER of the *pkcs12file* .

*password*: optional password used to decrypt PKCS12 file, bags and keys.

This function sets a certificate/private key pair and/or a CRL in the gnutls\_certificate\_credentials\_t structure. This function may be called more than once (in case multiple keys/certificates exist for the server).

PKCS12 files with a MAC, encrypted bags and PKCS 8 private keys are supported. However, only password based security, and the same password for all operations, are supported.

PKCS12 file may contain many keys and/or certificates, and this function will try to auto-detect based on the key ID the certificate and key pair to use. If the PKCS12 file contain the issuer of the selected certificate, it will be appended to the certificate to form a chain.



If more than one private keys are stored in the PKCS12 file, then only one key will be read (and it is undefined which one).

It is believed that the limitations of this function is acceptable for most usage, and that any more flexibility would introduce complexity that would make it harder to use this functionality at all.

**Returns:** GNUTLS\_E\_SUCCESS (0) on success, or a negative error code.

### **gnutls\_certificate\_set\_x509\_simple\_pkcs12\_mem**

**int gnutls\_certificate\_set\_x509\_simple\_pkcs12\_mem** [Function]  
     (*gnutls\_certificate\_credentials\_t* **res**, *const gnutls\_datum\_t* \* **p12blob**,  
     *gnutls\_x509\_crt\_fmt\_t* **type**, *const char* \* **password**)  
**res:** is a *gnutls\_certificate\_credentials\_t* structure.  
**p12blob:** the PKCS12 blob.  
**type:** is PEM or DER of the *pkcs12file* .

**password:** optional password used to decrypt PKCS12 file, bags and keys.

This function sets a certificate/private key pair and/or a CRL in the *gnutls\_certificate\_credentials\_t* structure. This function may be called more than once (in case multiple keys/certificates exist for the server).

Encrypted PKCS12 bags and PKCS8 private keys are supported. However, only password based security, and the same password for all operations, are supported.

PKCS12 file may contain many keys and/or certificates, and this function will try to auto-detect based on the key ID the certificate and key pair to use. If the PKCS12 file contain the issuer of the selected certificate, it will be appended to the certificate to form a chain.

If more than one private keys are stored in the PKCS12 file, then only one key will be read (and it is undefined which one).

It is believed that the limitations of this function is acceptable for most usage, and that any more flexibility would introduce complexity that would make it harder to use this functionality at all.

**Returns:** GNUTLS\_E\_SUCCESS (0) on success, or a negative error code.

**Since:** 2.8.0

### **gnutls\_certificate\_set\_x509\_system\_trust**

**int gnutls\_certificate\_set\_x509\_system\_trust** [Function]  
     (*gnutls\_certificate\_credentials\_t* **cred**)  
**cred:** is a *gnutls\_certificate\_credentials\_t* structure.

This function adds the system's default trusted CAs in order to verify client or server certificates.

In the case the system is currently unsupported GNUTLS\_E\_UNIMPLEMENTED\_FEATURE is returned.

**Returns:** the number of certificates processed or a negative error code on error.

**Since:** 3.0.20

**gnutls\_certificate\_set\_x509\_trust**

**int gnutls\_certificate\_set\_x509\_trust** [Function]  
 (*gnutls\_certificate\_credentials\_t* *res*, *gnutls\_x509\_crt\_t* \* *ca\_list*, *int*  
*ca\_list\_size*)

*res*: is a *gnutls\_certificate\_credentials\_t* structure.

*ca\_list*: is a list of trusted CAs

*ca\_list\_size*: holds the size of the CA list

This function adds the trusted CAs in order to verify client or server certificates. In case of a client this is not required to be called if the certificates are not verified using *gnutls\_certificate\_verify\_peers2()* . This function may be called multiple times.

In case of a server the CAs set here will be sent to the client if a certificate request is sent. This can be disabled using *gnutls\_certificate\_send\_x509\_rdn\_sequence()* .

**Returns:** the number of certificates processed or a negative error code on error.

**Since:** 2.4.0

**gnutls\_certificate\_set\_x509\_trust\_dir**

**int gnutls\_certificate\_set\_x509\_trust\_dir** [Function]  
 (*gnutls\_certificate\_credentials\_t* *cred*, *const char* \* *ca\_dir*,  
*gnutls\_x509\_crt\_fmt\_t* *type*)

*cred*: is a *gnutls\_certificate\_credentials\_t* structure.

*ca\_dir*: is a directory containing the list of trusted CAs (DER or PEM list)

*type*: is PEM or DER

This function adds the trusted CAs present in the directory in order to verify client or server certificates. This function is identical to *gnutls\_certificate\_set\_x509\_trust\_file()* but loads all certificates in a directory.

**Returns:** the number of certificates processed

**Since:** 3.3.6

**gnutls\_certificate\_set\_x509\_trust\_file**

**int gnutls\_certificate\_set\_x509\_trust\_file** [Function]  
 (*gnutls\_certificate\_credentials\_t* *cred*, *const char* \* *cafile*,  
*gnutls\_x509\_crt\_fmt\_t* *type*)

*cred*: is a *gnutls\_certificate\_credentials\_t* structure.

*cafile*: is a file containing the list of trusted CAs (DER or PEM list)

*type*: is PEM or DER

This function adds the trusted CAs in order to verify client or server certificates. In case of a client this is not required to be called if the certificates are not verified using *gnutls\_certificate\_verify\_peers2()* . This function may be called multiple times.

In case of a server the names of the CAs set here will be sent to the client if a certificate request is sent. This can be disabled using `gnutls_certificate_send_x509_rdn_sequence()` .

This function can also accept URLs. In that case it will import all certificates that are marked as trusted. Note that the supported URLs are the ones indicated by `gnutls_url_is_supported()` .

**Returns:** the number of certificates processed

### **gnutls\_certificate\_set\_x509\_trust\_mem**

`int gnutls_certificate_set_x509_trust_mem` [Function]  
     (*gnutls\_certificate\_credentials\_t* **res**, *const gnutls\_datum\_t* \* **ca**,  
     *gnutls\_x509\_crt\_fmt\_t* **type**)

*res*: is a `gnutls_certificate_credentials_t` structure.

*ca*: is a list of trusted CAs or a DER certificate

*type*: is DER or PEM

This function adds the trusted CAs in order to verify client or server certificates. In case of a client this is not required to be called if the certificates are not verified using `gnutls_certificate_verify_peers2()` . This function may be called multiple times.

In case of a server the CAs set here will be sent to the client if a certificate request is sent. This can be disabled using `gnutls_certificate_send_x509_rdn_sequence()` .

**Returns:** the number of certificates processed or a negative error code on error.

### **gnutls\_certificate\_type\_get**

`gnutls_certificate_type_t gnutls_certificate_type_get` [Function]  
     (*gnutls\_session\_t* **session**)

*session*: is a `gnutls_session_t` structure.

The certificate type is by default X.509, unless it is negotiated as a TLS extension.

**Returns:** the currently used `gnutls_certificate_type_t` certificate type.

### **gnutls\_certificate\_type\_get\_id**

`gnutls_certificate_type_t gnutls_certificate_type_get_id` [Function]  
     (*const char* \* **name**)

*name*: is a certificate type name

The names are compared in a case insensitive way.

**Returns:** a `gnutls_certificate_type_t` for the specified in a string certificate type, or `GNUTLS_CERT_UNKNOWN` on error.

### **gnutls\_certificate\_type\_get\_name**

`const char * gnutls_certificate_type_get_name` [Function]  
     (*gnutls\_certificate\_type\_t* **type**)

*type*: is a certificate type

Convert a `gnutls_certificate_type_t` type to a string.

**Returns:** a string that contains the name of the specified certificate type, or `NULL` in case of unknown types.

## `gnutls_certificate_type_list`

```
const gnutls_certificate_type_t *          [Function]
      gnutls_certificate_type_list ( void)
```

Get a list of certificate types.

**Returns:** a (0)-terminated list of `gnutls_certificate_type_t` integers indicating the available certificate types.

## `gnutls_certificate_verification_status_print`

```
int gnutls_certificate_verification_status_print (unsigned      [Function]
      int status, gnutls_certificate_type_t type, gnutls_datum_t * out, unsigned int
      flags)
```

*status*: The status flags to be printed

*type*: The certificate type

*out*: Newly allocated datum with (0) terminated string.

*flags*: should be zero

This function will pretty print the status of a verification process – eg. the one obtained by `gnutls_certificate_verify_peers3()` .

The output *out* needs to be deallocated using `gnutls_free()` .

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.1.4

## `gnutls_certificate_verify_peers`

```
int gnutls_certificate_verify_peers (gnutls_session_t session,    [Function]
      gnutls_typed_vdata_st * data, unsigned int elements, unsigned int * status)
```

*session*: is a gnutls session

*data*: an array of typed data

*elements*: the number of data elements

*status*: is the output of the verification

This function will verify the peer's certificate and store the status in the `status` variable as a bitwise or'd `gnutls_certificate_status_t` values or zero if the certificate is trusted. Note that value in `status` is set only when the return value of this function is success (i.e, failure to trust a certificate does not imply a negative return value). The default verification flags used by this function can be overridden using `gnutls_certificate_set_verify_flags()` . See the documentation of `gnutls_certificate_verify_peers2()` for details in the verification process.

The acceptable *data* types are `GNUTLS_DT_DNS_HOSTNAME` and `GNUTLS_DT_KEY_PURPOSE_OID` . The former accepts as data a null-terminated hostname, and the latter

a null-terminated object identifier (e.g., `GNUTLS_KP_TLS_WWW_SERVER` ). If a DNS hostname is provided then this function will compare the hostname in the certificate against the given. If names do not match the `GNUTLS_CERT_UNEXPECTED_OWNER` status flag will be set. If a key purpose OID is provided and the end-certificate contains the extended key usage PKIX extension, it will be required to have the provided key purpose or be marked for any purpose, otherwise verification will fail with `GNUTLS_CERT_SIGNER_CONSTRAINTS_FAILURE` status.

**Returns:** a negative error code on error and `GNUTLS_E_SUCCESS` (0) on success.

**Since:** 3.3.0

## `gnutls_certificate_verify_peers2`

```
int gnutls_certificate_verify_peers2 (gnutls_session_t session,    [Function]
                                     unsigned int * status)
```

*session*: is a gnutls session

*status*: is the output of the verification

This function will verify the peer's certificate and store the status in the `status` variable as a bitwise or'd `gnutls_certificate_status_t` values or zero if the certificate is trusted. Note that value in `status` is set only when the return value of this function is success (i.e, failure to trust a certificate does not imply a negative return value). The default verification flags used by this function can be overridden using `gnutls_certificate_set_verify_flags()` .

This function will take into account the OCSP Certificate Status TLS extension, as well as the following X.509 certificate extensions: Name Constraints, Key Usage, and Basic Constraints (pathlen).

To avoid denial of service attacks some default upper limits regarding the certificate key size and chain size are set. To override them use `gnutls_certificate_set_verify_limits()` .

Note that you must also check the peer's name in order to check if the verified certificate belongs to the actual peer, see `gnutls_x509_crt_check_hostname()` , or use `gnutls_certificate_verify_peers3()` .

**Returns:** a negative error code on error and `GNUTLS_E_SUCCESS` (0) on success.

## `gnutls_certificate_verify_peers3`

```
int gnutls_certificate_verify_peers3 (gnutls_session_t session,    [Function]
                                     const char * hostname, unsigned int * status)
```

*session*: is a gnutls session

*hostname*: is the expected name of the peer; may be NULL

*status*: is the output of the verification

This function will verify the peer's certificate and store the status in the `status` variable as a bitwise or'd `gnutls_certificate_status_t` values or zero if the certificate is trusted. Note that value in `status` is set only when the return value of this function is success (i.e, failure to trust a certificate does not imply a negative return value). The default verification flags used by this function can be overridden using

`gnutls_certificate_set_verify_flags()` . See the documentation of `gnutls_certificate_verify_peers2()` for details in the verification process.

If the `hostname` provided is non-NULL then this function will compare the hostname in the certificate against the given. The comparison will be accurate for ascii names; non-ascii names are compared byte-by-byte. If names do not match the `GNUTLS_CERT_UNEXPECTED_OWNER` status flag will be set.

In order to verify the purpose of the end-certificate (by checking the extended key usage), use `gnutls_certificate_verify_peers()` .

**Returns:** a negative error code on error and `GNUTLS_E_SUCCESS (0)` on success.

**Since:** 3.1.4

## gnutls\_check\_version

`const char * gnutls_check_version (const char * req_version)` [Function]  
*req\_version*: version string to compare with, or NULL .

Check GnuTLS Library version.

See `GNUTLS_VERSION` for a suitable `req_version` string.

**Returns:** Check that the version of the library is at minimum the one given as a string in `req_version` and return the actual version string of the library; return NULL if the condition is not met. If NULL is passed to this function no check is done and only the version string is returned.

## gnutls\_cipher\_get

`gnutls_cipher_algorithm_t gnutls_cipher_get (gnutls_session_t session)` [Function]

*session*: is a `gnutls_session_t` structure.

Get currently used cipher.

**Returns:** the currently used cipher, a `gnutls_cipher_algorithm_t` type.

## gnutls\_cipher\_get\_id

`gnutls_cipher_algorithm_t gnutls_cipher_get_id (const char * name)` [Function]

*name*: is a cipher algorithm name

The names are compared in a case insensitive way.

**Returns:** return a `gnutls_cipher_algorithm_t` value corresponding to the specified cipher, or `GNUTLS_CIPHER_UNKNOWN` on error.

## gnutls\_cipher\_get\_key\_size

`size_t gnutls_cipher_get_key_size (gnutls_cipher_algorithm_t algorithm)` [Function]

*algorithm*: is an encryption algorithm

Get key size for cipher.

**Returns:** length (in bytes) of the given cipher's key size, or 0 if the given cipher is invalid.

## gnutls\_cipher\_get\_name

```
const char * gnutls_cipher_get_name (gnutls_cipher_algorithm_t      [Function]
                                     algorithm)
```

*algorithm*: is an encryption algorithm

Convert a `gnutls_cipher_algorithm_t` type to a string.

**Returns:** a pointer to a string that contains the name of the specified cipher, or NULL .

## gnutls\_cipher\_list

```
const gnutls_cipher_algorithm_t * gnutls_cipher_list (              [Function]
                                                        void)
```

Get a list of supported cipher algorithms. Note that not necessarily all ciphers are supported as TLS cipher suites. For example, DES is not supported as a cipher suite, but is supported for other purposes (e.g., PKCS8 or similar).

This function is not thread safe.

**Returns:** a (0)-terminated list of `gnutls_cipher_algorithm_t` integers indicating the available ciphers.

## gnutls\_cipher\_suite\_get\_name

```
const char * gnutls_cipher_suite_get_name                          [Function]
(gnutls_kx_algorithm_t kx_algorithm, gnutls_cipher_algorithm_t
 cipher_algorithm, gnutls_mac_algorithm_t mac_algorithm)
```

*kx\_algorithm*: is a Key exchange algorithm

*cipher\_algorithm*: is a cipher algorithm

*mac\_algorithm*: is a MAC algorithm

Note that the full cipher suite name must be prepended by TLS or SSL depending of the protocol in use.

**Returns:** a string that contains the name of a TLS cipher suite, specified by the given algorithms, or NULL .

## gnutls\_cipher\_suite\_info

```
const char * gnutls_cipher_suite_info (size_t idx, unsigned char    [Function]
                                     * cs_id, gnutls_kx_algorithm_t * kx, gnutls_cipher_algorithm_t * cipher,
                                     gnutls_mac_algorithm_t * mac, gnutls_protocol_t * min_version)
```

*idx*: index of cipher suite to get information about, starts on 0.

*cs\_id*: output buffer with room for 2 bytes, indicating cipher suite value

*kx*: output variable indicating key exchange algorithm, or NULL .

*cipher*: output variable indicating cipher, or NULL .

*mac*: output variable indicating MAC algorithm, or NULL .

*min\_version*: output variable indicating TLS protocol version, or NULL .

Get information about supported cipher suites. Use the function iteratively to get information about all supported cipher suites. Call with `idx=0` to get information about first cipher suite, then `idx=1` and so on until the function returns `NULL`.

**Returns:** the name of `idx` cipher suite, and set the information about the cipher suite in the output variables. If `idx` is out of bounds, `NULL` is returned.

## gnutls\_compression\_get

`gnutls_compression_method_t gnutls_compression_get` [Function]  
(*gnutls\_session\_t session*)

*session*: is a `gnutls_session_t` structure.

Get currently used compression algorithm.

**Returns:** the currently used compression method, a `gnutls_compression_method_t` value.

## gnutls\_compression\_get\_id

`gnutls_compression_method_t gnutls_compression_get_id` [Function]  
(*const char \* name*)

*name*: is a compression method name

The names are compared in a case insensitive way.

**Returns:** an id of the specified in a string compression method, or `GNUTLS_COMP_UNKNOWN` on error.

## gnutls\_compression\_get\_name

`const char * gnutls_compression_get_name` [Function]  
(*gnutls\_compression\_method\_t algorithm*)

*algorithm*: is a Compression algorithm

Convert a `gnutls_compression_method_t` value to a string.

**Returns:** a pointer to a string that contains the name of the specified compression algorithm, or `NULL`.

## gnutls\_compression\_list

`const gnutls_compression_method_t *` [Function]  
`gnutls_compression_list ( void)`

Get a list of compression methods.

**Returns:** a zero-terminated list of `gnutls_compression_method_t` integers indicating the available compression methods.

## gnutls\_credentials\_clear

`void gnutls_credentials_clear (gnutls_session_t session)` [Function]  
*session*: is a `gnutls_session_t` structure.

Clears all the credentials previously set in this session.



## gnutls\_credentials\_get

**int gnutls\_credentials\_get** (*gnutls\_session\_t session*, [Function]  
*gnutls\_credentials\_type\_t type*, void \*\* *cred*)

*session*: is a `gnutls_session_t` structure.

*type*: is the type of the credentials to return

*cred*: will contain the pointer to the credentials structure.

Returns the previously provided credentials structures.

For `GNUTLS_CRD_ANON` , *cred* will be `gnutls_anon_client_credentials_t` in case of a client. In case of a server it should be `gnutls_anon_server_credentials_t` .

For `GNUTLS_CRD_SRP` , *cred* will be `gnutls_srp_client_credentials_t` in case of a client, and `gnutls_srp_server_credentials_t` , in case of a server.

For `GNUTLS_CRD_CERTIFICATE` , *cred* will be `gnutls_certificate_credentials_t` .

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

## gnutls\_credentials\_set

**int gnutls\_credentials\_set** (*gnutls\_session\_t session*, [Function]  
*gnutls\_credentials\_type\_t type*, void \* *cred*)

*session*: is a `gnutls_session_t` structure.

*type*: is the type of the credentials

*cred*: is a pointer to a structure.

Sets the needed credentials for the specified type. Eg username, password - or public and private keys etc. The *cred* parameter is a structure that depends on the specified type and on the current session (client or server).

In order to minimize memory usage, and share credentials between several threads gnutls keeps a pointer to *cred*, and not the whole cred structure. Thus you will have to keep the structure allocated until you call `gnutls_deinit()` .

For `GNUTLS_CRD_ANON` , *cred* should be `gnutls_anon_client_credentials_t` in case of a client. In case of a server it should be `gnutls_anon_server_credentials_t` .

For `GNUTLS_CRD_SRP` , *cred* should be `gnutls_srp_client_credentials_t` in case of a client, and `gnutls_srp_server_credentials_t` , in case of a server.

For `GNUTLS_CRD_CERTIFICATE` , *cred* should be `gnutls_certificate_credentials_t` .

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

## gnutls\_db\_check\_entry

**int gnutls\_db\_check\_entry** (*gnutls\_session\_t session*, [Function]  
*gnutls\_datum\_t session\_entry*)

*session*: is a `gnutls_session_t` structure.

*session\_entry*: is the session data (not key)

This function has no effect.

**Returns:** Returns GNUTLS\_E\_EXPIRED , if the database entry has expired or 0 otherwise.

## gnutls\_db\_check\_entry\_time

`time_t gnutls_db_check_entry_time (gnutls_datum_t * entry)` [Function]  
*entry*: is a pointer to a `gnutls_datum_t` structure.

This function returns the time that this entry was active. It can be used for database entry expiration.

**Returns:** The time this entry was created, or zero on error.

## gnutls\_db\_get\_default\_cache\_expiration

`unsigned gnutls_db_get_default_cache_expiration ( void)` [Function]  
 Returns the expiration time (in seconds) of stored sessions for resumption.

## gnutls\_db\_get\_ptr

`void * gnutls_db_get_ptr (gnutls_session_t session)` [Function]  
*session*: is a `gnutls_session_t` structure.

Get db function pointer.

**Returns:** the pointer that will be sent to db store, retrieve and delete functions, as the first argument.

## gnutls\_db\_remove\_session

`void gnutls_db_remove_session (gnutls_session_t session)` [Function]  
*session*: is a `gnutls_session_t` structure.

This function will remove the current session data from the session database. This will prevent future handshakes reusing these session data. This function should be called if a session was terminated abnormally, and before `gnutls_deinit()` is called.

Normally `gnutls_deinit()` will remove abnormally terminated sessions.

## gnutls\_db\_set\_cache\_expiration

`void gnutls_db_set_cache_expiration (gnutls_session_t session,  
   int seconds)` [Function]

*session*: is a `gnutls_session_t` structure.

*seconds*: is the number of seconds.

Set the expiration time for resumed sessions. The default is 3600 (one hour) at the time of this writing.

## **gnutls\_db\_set\_ptr**

**void gnutls\_db\_set\_ptr** (*gnutls\_session\_t session*, *void \* ptr*) [Function]

*session*: is a **gnutls\_session\_t** structure.

*ptr*: is the pointer

Sets the pointer that will be provided to db store, retrieve and delete functions, as the first argument.

## **gnutls\_db\_set\_remove\_function**

**void gnutls\_db\_set\_remove\_function** (*gnutls\_session\_t session*,  
*gnutls\_db\_remove\_func rem\_func*) [Function]

*session*: is a **gnutls\_session\_t** structure.

*rem\_func*: is the function.

Sets the function that will be used to remove data from the resumed sessions database. This function must return 0 on success.

The first argument to **rem\_func** will be null unless **gnutls\_db\_set\_ptr()** has been called.

## **gnutls\_db\_set\_retrieve\_function**

**void gnutls\_db\_set\_retrieve\_function** (*gnutls\_session\_t session*,  
*gnutls\_db\_retr\_func retr\_func*) [Function]

*session*: is a **gnutls\_session\_t** structure.

*retr\_func*: is the function.

Sets the function that will be used to retrieve data from the resumed sessions database. This function must return a **gnutls\_datum\_t** containing the data on success, or a **gnutls\_datum\_t** containing null and 0 on failure.

The datum's data must be allocated using the function **gnutls\_malloc()** .

The first argument to **retr\_func** will be null unless **gnutls\_db\_set\_ptr()** has been called.

## **gnutls\_db\_set\_store\_function**

**void gnutls\_db\_set\_store\_function** (*gnutls\_session\_t session*,  
*gnutls\_db\_store\_func store\_func*) [Function]

*session*: is a **gnutls\_session\_t** structure.

*store\_func*: is the function

Sets the function that will be used to store data in the resumed sessions database. This function must return 0 on success.

The first argument to **store\_func** will be null unless **gnutls\_db\_set\_ptr()** has been called.

## gnutls\_deinit

**void gnutls\_deinit** (*gnutls\_session\_t session*) [Function]

*session*: is a **gnutls\_session\_t** structure.

This function clears all buffers associated with the **session** . This function will also remove session data from the session database if the session was terminated abnormally.

## gnutls\_dh\_get\_group

**int gnutls\_dh\_get\_group** (*gnutls\_session\_t session*, *gnutls\_datum\_t \* raw\_gen*, *gnutls\_datum\_t \* raw\_prime*) [Function]

*session*: is a gnutls session

*raw\_gen*: will hold the generator.

*raw\_prime*: will hold the prime.

This function will return the group parameters used in the last Diffie-Hellman key exchange with the peer. These are the prime and the generator used. This function should be used for both anonymous and ephemeral Diffie-Hellman. The output parameters must be freed with **gnutls\_free()** .

Note, that the prime and generator are exported as non-negative integers and may include a leading zero byte.

**Returns:** On success, **GNUTLS\_E\_SUCCESS** (0) is returned, otherwise an error code is returned.

## gnutls\_dh\_get\_peers\_public\_bits

**int gnutls\_dh\_get\_peers\_public\_bits** (*gnutls\_session\_t session*) [Function]

*session*: is a gnutls session

Get the Diffie-Hellman public key bit size. Can be used for both anonymous and ephemeral Diffie-Hellman.

**Returns:** The public key bit size used in the last Diffie-Hellman key exchange with the peer, or a negative error code in case of error.

## gnutls\_dh\_get\_prime\_bits

**int gnutls\_dh\_get\_prime\_bits** (*gnutls\_session\_t session*) [Function]

*session*: is a gnutls session

This function will return the bits of the prime used in the last Diffie-Hellman key exchange with the peer. Should be used for both anonymous and ephemeral Diffie-Hellman. Note that some ciphers, like RSA and DSA without DHE, do not use a Diffie-Hellman key exchange, and then this function will return 0.

**Returns:** The Diffie-Hellman bit strength is returned, or 0 if no Diffie-Hellman key exchange was done, or a negative error code on failure.

**gnutls\_dh\_get\_pubkey**

**int gnutls\_dh\_get\_pubkey** (*gnutls\_session\_t session*, *gnutls\_datum\_t \*raw\_key*) [Function]

*session*: is a gnutls session

*raw\_key*: will hold the public key.

This function will return the peer's public key used in the last Diffie-Hellman key exchange. This function should be used for both anonymous and ephemeral Diffie-Hellman. The output parameters must be freed with **gnutls\_free()** .

Note, that public key is exported as non-negative integer and may include a leading zero byte.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise an error code is returned.

**gnutls\_dh\_get\_secret\_bits**

**int gnutls\_dh\_get\_secret\_bits** (*gnutls\_session\_t session*) [Function]

*session*: is a gnutls session

This function will return the bits used in the last Diffie-Hellman key exchange with the peer. Should be used for both anonymous and ephemeral Diffie-Hellman.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise an error code is returned.

**gnutls\_dh\_params\_cpy**

**int gnutls\_dh\_params\_cpy** (*gnutls\_dh\_params\_t dst*, *gnutls\_dh\_params\_t src*) [Function]

*dst*: Is the destination structure, which should be initialized.

*src*: Is the source structure

This function will copy the DH parameters structure from source to destination.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error code is returned.

**gnutls\_dh\_params\_deinit**

**void gnutls\_dh\_params\_deinit** (*gnutls\_dh\_params\_t dh\_params*) [Function]

*dh\_params*: Is a structure that holds the prime numbers

This function will deinitialize the DH parameters structure.

**gnutls\_dh\_params\_export2\_pkcs3**

**int gnutls\_dh\_params\_export2\_pkcs3** (*gnutls\_dh\_params\_t params*, *gnutls\_x509\_crt\_fmt\_t format*, *gnutls\_datum\_t \*out*) [Function]

*params*: Holds the DH parameters

*format*: the format of output params. One of PEM or DER.

*out*: will contain a PKCS3 DHParams structure PEM or DER encoded

This function will export the given dh parameters to a PKCS3 DHParams structure. This is the format generated by "openssl dhparam" tool. The data in `out` will be allocated using `gnutls_malloc()`.

If the structure is PEM encoded, it will have a header of "BEGIN DH PARAMETERS".

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

**Since:** 3.1.3

### `gnutls_dh_params_export_pkcs3`

```
int gnutls_dh_params_export_pkcs3 (gnutls_dh_params_t params,      [Function]
                                   gnutls_x509_crt_fmt_t format, unsigned char * params_data, size_t *
                                   params_data_size)
```

*params*: Holds the DH parameters

*format*: the format of output params. One of PEM or DER.

*params\_data*: will contain a PKCS3 DHParams structure PEM or DER encoded

*params\_data\_size*: holds the size of *params\_data* (and will be replaced by the actual size of parameters)

This function will export the given dh parameters to a PKCS3 DHParams structure. This is the format generated by "openssl dhparam" tool. If the buffer provided is not long enough to hold the output, then `GNUTLS_E_SHORT_MEMORY_BUFFER` will be returned.

If the structure is PEM encoded, it will have a header of "BEGIN DH PARAMETERS".

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

### `gnutls_dh_params_export_raw`

```
int gnutls_dh_params_export_raw (gnutls_dh_params_t params,      [Function]
                                 gnutls_datum_t * prime, gnutls_datum_t * generator, unsigned int * bits)
```

*params*: Holds the DH parameters

*prime*: will hold the new prime

*generator*: will hold the new generator

*bits*: if non null will hold the secret key's number of bits

This function will export the pair of prime and generator for use in the Diffie-Hellman key exchange. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

## gnutls\_dh\_params\_generate2

**int gnutls\_dh\_params\_generate2** (*gnutls\_dh\_params\_t* *dparams*, [Function]  
*unsigned int bits*)

*dparams*: Is the structure that the DH parameters will be stored

*bits*: is the prime's number of bits

This function will generate a new pair of prime and generator for use in the Diffie-Hellman key exchange. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum. This function is normally slow.

Do not set the number of bits directly, use `gnutls_sec_param_to_pk_bits()` to get bits for `GNUTLS_PK_DSA`. Also note that the DH parameters are only useful to servers. Since clients use the parameters sent by the server, it's of no use to call this in client side.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

## gnutls\_dh\_params\_import\_pkcs3

**int gnutls\_dh\_params\_import\_pkcs3** (*gnutls\_dh\_params\_t* *params*, [Function]  
*const gnutls\_datum\_t \*pkcs3\_params*, *gnutls\_x509\_crt\_fmt\_t format*)

*params*: A structure where the parameters will be copied to

*pkcs3\_params*: should contain a PKCS3 DHParams structure PEM or DER encoded

*format*: the format of params. PEM or DER.

This function will extract the DHParams found in a PKCS3 formatted structure. This is the format generated by "openssl dhparam" tool.

If the structure is PEM encoded, it should have a header of "BEGIN DH PARAMETERS".

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

## gnutls\_dh\_params\_import\_raw

**int gnutls\_dh\_params\_import\_raw** (*gnutls\_dh\_params\_t* *dh\_params*, [Function]  
*const gnutls\_datum\_t \*prime*, *const gnutls\_datum\_t \*generator*)

*dh\_params*: Is a structure that will hold the prime numbers

*prime*: holds the new prime

*generator*: holds the new generator

This function will replace the pair of prime and generator for use in the Diffie-Hellman key exchange. The new parameters should be stored in the appropriate `gnutls_datum`.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

## gnutls\_dh\_params\_init

`int gnutls_dh_params_init (gnutls_dh_params_t * dh_params)` [Function]

*dh\_params*: Is a structure that will hold the prime numbers

This function will initialize the DH parameters structure.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error code is returned.

## gnutls\_dh\_set\_prime\_bits

`void gnutls_dh_set_prime_bits (gnutls_session_t session, unsigned int bits)` [Function]

*session*: is a gnutls\_session\_t structure.

*bits*: is the number of bits

This function sets the number of bits, for use in a Diffie-Hellman key exchange. This is used both in DH ephemeral and DH anonymous cipher suites. This will set the minimum size of the prime that will be used for the handshake.

In the client side it sets the minimum accepted number of bits. If a server sends a prime with less bits than that GNUTLS\_E\_DH\_PRIME\_UNACCEPTABLE will be returned by the handshake.

Note that this function will warn via the audit log for value that are believed to be weak.

The function has no effect in server side.

Note that since 3.1.7 this function is deprecated. The minimum number of bits is set by the priority string level. Also this function must be called after gnutls\_priority\_set\_direct() or the set value may be overridden by the selected priority options.

## gnutls\_digest\_get\_id

`gnutls_digest_algorithm_t gnutls_digest_get_id (const char * name)` [Function]

*name*: is a digest algorithm name

Convert a string to a gnutls\_digest\_algorithm\_t value. The names are compared in a case insensitive way.

**Returns:** a gnutls\_digest\_algorithm\_t id of the specified MAC algorithm string, or GNUTLS\_DIG\_UNKNOWN on failures.

## gnutls\_digest\_get\_name

`const char * gnutls_digest_get_name (gnutls_digest_algorithm_t algorithm)` [Function]

*algorithm*: is a digest algorithm

Convert a gnutls\_digest\_algorithm\_t value to a string.

**Returns:** a string that contains the name of the specified digest algorithm, or NULL .



**gnutls\_digest\_list**

`const gnutls_digest_algorithm_t * gnutls_digest_list ( [Function]  
void)`

Get a list of hash (digest) algorithms supported by GnuTLS.

This function is not thread safe.

**Returns:** Return a (0)-terminated list of `gnutls_digest_algorithm_t` integers indicating the available digests.

**gnutls\_ecc\_curve\_get**

`gnutls_ecc_curve_t gnutls_ecc_curve_get (gnutls_session_t [Function]  
session)`

*session*: is a `gnutls_session_t` structure.

Returns the currently used elliptic curve. Only valid when using an elliptic curve ciphersuite.

**Returns:** the currently used curve, a `gnutls_ecc_curve_t` type.

**Since:** 3.0

**gnutls\_ecc\_curve\_get\_name**

`const char * gnutls_ecc_curve_get_name (gnutls_ecc_curve_t [Function]  
curve)`

*curve*: is an ECC curve

Convert a `gnutls_ecc_curve_t` value to a string.

**Returns:** a string that contains the name of the specified curve or NULL .

**Since:** 3.0

**gnutls\_ecc\_curve\_get\_size**

`int gnutls_ecc_curve_get_size (gnutls_ecc_curve_t curve) [Function]`  
*curve*: is an ECC curve

Returns the size in bytes of the curve.

**Returns:** a the size or (0).

**Since:** 3.0

**gnutls\_ecc\_curve\_list**

`const gnutls_ecc_curve_t * gnutls_ecc_curve_list ( void) [Function]`

Get the list of supported elliptic curves.

This function is not thread safe.

**Returns:** Return a (0)-terminated list of `gnutls_ecc_curve_t` integers indicating the available curves.

## gnutls\_error\_is\_fatal

`int gnutls_error_is_fatal (int error)` [Function]

*error*: is a GnuTLS error code, a negative error code

If a GnuTLS function returns a negative error code you may feed that value to this function to see if the error condition is fatal to a TLS session (i.e., must be terminated).

Note that you may also want to check the error code manually, since some non-fatal errors to the protocol (such as a warning alert or a rehandshake request) may be fatal for your program.

This function is only useful if you are dealing with errors from functions that relate to a TLS session (e.g., record layer or handshake layer handling functions).

**Returns:** Non-zero value on fatal errors or zero on non-fatal.

## gnutls\_error\_to\_alert

`int gnutls_error_to_alert (int err, int * level)` [Function]

*err*: is a negative integer

*level*: the alert level will be stored there

Get an alert depending on the error code returned by a gnutls function. All alerts sent by this function should be considered fatal. The only exception is when **err** is **GNUTLS\_E\_REHANDSHAKE**, where a warning alert should be sent to the peer indicating that no renegotiation will be performed.

If there is no mapping to a valid alert the alert to indicate internal error is returned.

**Returns:** the alert code to use for a particular error code.

## gnutls\_est\_record\_overhead\_size

`size_t gnutls_est_record_overhead_size (gnutls_protocol_t version, gnutls_cipher_algorithm_t cipher, gnutls_mac_algorithm_t mac, gnutls_compression_method_t comp, unsigned int flags)` [Function]

*version*: is a gnutls\_protocol\_t value

*cipher*: is a gnutls\_cipher\_algorithm\_t value

*mac*: is a gnutls\_mac\_algorithm\_t value

*comp*: is a gnutls\_compression\_method\_t value

*flags*: must be zero

This function will return the set size in bytes of the overhead due to TLS (or DTLS) per record.

Note that this function may provide inaccurate values when TLS extensions that modify the record format are negotiated. In these cases a more accurate value can be obtained using `gnutls_record_overhead_size()` after a completed handshake.

**Since:** 3.2.2

## gnutls\_fingerprint

**int gnutls\_fingerprint** (*gnutls\_digest\_algorithm\_t algo*, *const gnutls\_datum\_t \*data*, *void \*result*, *size\_t \*result\_size*) [Function]

*algo*: is a digest algorithm

*data*: is the data

*result*: is the place where the result will be copied (may be null).

*result\_size*: should hold the size of the result. The actual size of the returned result will also be copied there.

This function will calculate a fingerprint (actually a hash), of the given data. The result is not printable data. You should convert it to hex, or to something else printable.

This is the usual way to calculate a fingerprint of an X.509 DER encoded certificate. Note however that the fingerprint of an OpenPGP certificate is not just a hash and cannot be calculated with this function.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise an error code is returned.

## gnutls\_fips140\_mode\_enabled

**int gnutls\_fips140\_mode\_enabled** (*void*) [Function]

Checks whether this library is in FIPS140 mode.

**Returns:** return non-zero if true or zero if false.

**Since:** 3.3.0

## gnutls\_global\_deinit

**void gnutls\_global\_deinit** (*void*) [Function]

This function deinitializes the global data, that were initialized using `gnutls_global_init()`.

## gnutls\_global\_init

**int gnutls\_global\_init** (*void*) [Function]

This function performs any required precalculations, detects the supported CPU capabilities and initializes the underlying cryptographic backend. In order to free any resources taken by this call you should `gnutls_global_deinit()` when gnutls usage is no longer needed.

This function increments a global counter, so that `gnutls_global_deinit()` only releases resources when it has been called as many times as `gnutls_global_init()`. This is useful when GnuTLS is used by more than one library in an application. This function can be called many times, but will only do something the first time.

Since GnuTLS 3.3.0 this function is only required in systems that do not support library constructors and static linking. This function also became thread safe.

A subsequent call of this function if the initial has failed will return the same error code.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error code is returned.

## gnutls\_global\_set\_audit\_log\_function

void gnutls\_global\_set\_audit\_log\_function (gnutls\_audit\_log\_func log\_func) [Function]

*log\_func*: it is the audit log function

This is the function to set the audit logging function. This is a function to report important issues, such as possible attacks in the protocol. This is different from `gnutls_global_set_log_function()` because it will report also session-specific events. The session parameter will be null if there is no corresponding TLS session.

`gnutls_audit_log_func` is of the form, void (\*gnutls\_audit\_log\_func)(gnutls\_session\_t, const char\*);

**Since:** 3.0

## gnutls\_global\_set\_log\_function

void gnutls\_global\_set\_log\_function (gnutls\_log\_func log\_func) [Function]

*log\_func*: it's a log function

This is the function where you set the logging function gnutls is going to use. This function only accepts a character array. Normally you may not use this function since it is only used for debugging purposes.

`gnutls_log_func` is of the form, void (\*gnutls\_log\_func)( int level, const char\*);

## gnutls\_global\_set\_log\_level

void gnutls\_global\_set\_log\_level (int level) [Function]

*level*: it's an integer from 0 to 99.

This is the function that allows you to set the log level. The level is an integer between 0 and 9. Higher values mean more verbosity. The default value is 0. Larger values should only be used with care, since they may reveal sensitive information.

Use a log level over 10 to enable all debugging options.

## gnutls\_global\_set\_mutex

void gnutls\_global\_set\_mutex (mutex\_init\_func init, mutex\_deinit\_func deinit, mutex\_lock\_func lock, mutex\_unlock\_func unlock) [Function]

*init*: mutex initialization function

*deinit*: mutex deinitialization function

*lock*: mutex locking function

*unlock*: mutex unlocking function

With this function you are allowed to override the default mutex locks used in some parts of gnutls and dependent libraries. This function should be used if you have complete control of your program and libraries. Do not call this function from a

library, or preferably from any application unless really needed to. GnuTLS will use the appropriate locks for the running system.

This function must be called prior to any other gnutls function.

**Since:** 2.12.0

## gnutls\_global\_set\_time\_function

**void gnutls\_global\_set\_time\_function** (*gnutls\_time\_func* *time\_func*) [Function]

*time\_func*: it's the system time function, a **gnutls\_time\_func()** callback.

This is the function where you can override the default system time function. The application provided function should behave the same as the standard function.

**Since:** 2.12.0

## gnutls\_handshake

**int gnutls\_handshake** (*gnutls\_session\_t session*) [Function]

*session*: is a **gnutls\_session\_t** structure.

This function does the handshake of the TLS/SSL protocol, and initializes the TLS connection.

This function will fail if any problem is encountered, and will return a negative error code. In case of a client, if the client has asked to resume a session, but the server couldn't, then a full handshake will be performed.

The non-fatal errors expected by this function are: **GNUTLS\_E\_INTERRUPTED** , **GNUTLS\_E\_AGAIN** , **GNUTLS\_E\_WARNING\_ALERT\_RECEIVED** , and **GNUTLS\_E\_GOT\_APPLICATION\_DATA** , the latter only in a case of rehandshake.

The former two interrupt the handshake procedure due to the lower layer being interrupted, and the latter because of an alert that may be sent by a server (it is always a good idea to check any received alerts). On these errors call this function again, until it returns 0; cf. **gnutls\_record\_get\_direction()** and **gnutls\_error\_is\_fatal()** . In DTLS sessions the non-fatal error **GNUTLS\_E\_LARGE\_PACKET** is also possible, and indicates that the MTU should be adjusted.

If this function is called by a server after a rehandshake request then **GNUTLS\_E\_GOT\_APPLICATION\_DATA** or **GNUTLS\_E\_WARNING\_ALERT\_RECEIVED** may be returned. Note that these are non fatal errors, only in the specific case of a rehandshake. Their meaning is that the client rejected the rehandshake request or in the case of **GNUTLS\_E\_GOT\_APPLICATION\_DATA** it could also mean that some data were pending.

**Returns:** **GNUTLS\_E\_SUCCESS** on success, otherwise a negative error code.

## gnutls\_handshake\_description\_get\_name

**const char \*** **gnutls\_handshake\_description\_get\_name** (*gnutls\_handshake\_description\_t type*) [Function]

*type*: is a handshake message description

Convert a **gnutls\_handshake\_description\_t** value to a string.

**Returns:** a string that contains the name of the specified handshake message or NULL

## gnutls\_handshake\_get\_last\_in

`gnutls_handshake_description_t` [Function]

`gnutls_handshake_get_last_in (gnutls_session_t session)`

*session*: is a `gnutls_session_t` structure.

This function is only useful to check where the last performed handshake failed. If the previous handshake succeed or was not performed at all then no meaningful value will be returned.

Check `gnutls_handshake_description_t` in `gnutls.h` for the available handshake descriptions.

**Returns:** the last handshake message type received, a `gnutls_handshake_description_t`.

## gnutls\_handshake\_get\_last\_out

`gnutls_handshake_description_t` [Function]

`gnutls_handshake_get_last_out (gnutls_session_t session)`

*session*: is a `gnutls_session_t` structure.

This function is only useful to check where the last performed handshake failed. If the previous handshake succeed or was not performed at all then no meaningful value will be returned.

Check `gnutls_handshake_description_t` in `gnutls.h` for the available handshake descriptions.

**Returns:** the last handshake message type sent, a `gnutls_handshake_description_t`.

## gnutls\_handshake\_set\_hook\_function

`void gnutls_handshake_set_hook_function (gnutls_session_t session, unsigned int htype, int post, gnutls_handshake_hook_func func)` [Function]

*session*: is a `gnutls_session_t` structure

*htype*: the `gnutls_handshake_description_t` of the message to hook at

*post*: `GNUTLS_HOOK_*` depending on when the hook function should be called

*func*: is the function to be called

This function will set a callback to be called after or before the specified handshake message has been received or generated. This is a generalization of `gnutls_handshake_set_post_client_hello_function()`.

To call the hook function prior to the message being sent/generated use `GNUTLS_HOOK_PRE` as *post* parameter, `GNUTLS_HOOK_POST` to call after, and `GNUTLS_HOOK_BOTH` for both cases.

This callback must return 0 on success or a gnutls error code to terminate the handshake.

Note to hook at all handshake messages use an *htype* of `GNUTLS_HANDSHAKE_ANY`.

**Warning:** You should not use this function to terminate the handshake based on client input unless you know what you are doing. Before the handshake is finished there is no way to know if there is a man-in-the-middle attack being performed.

## gnutls\_handshake\_set\_max\_packet\_length

**void gnutls\_handshake\_set\_max\_packet\_length** (*gnutls\_session\_t session, size\_t max*) [Function]

*session*: is a `gnutls_session_t` structure.

*max*: is the maximum number.

This function will set the maximum size of all handshake messages. Handshakes over this size are rejected with `GNUTLS_E_HANDSHAKE_TOO_LARGE` error code. The default value is 48kb which is typically large enough. Set this to 0 if you do not want to set an upper limit.

The reason for restricting the handshake message sizes are to limit Denial of Service attacks.

## gnutls\_handshake\_set\_post\_client\_hello\_function

**void gnutls\_handshake\_set\_post\_client\_hello\_function** (*gnutls\_session\_t session, gnutls\_handshake\_post\_client\_hello\_func func*) [Function]

*session*: is a `gnutls_session_t` structure.

*func*: is the function to be called

This function will set a callback to be called after the client hello has been received (callback valid in server side only). This allows the server to adjust settings based on received extensions.

Those settings could be ciphersuites, requesting certificate, or anything else except for version negotiation (this is done before the hello message is parsed).

This callback must return 0 on success or a gnutls error code to terminate the handshake.

Since GnuTLS 3.3.5 the callback is allowed to return `GNUTLS_E_AGAIN` or `GNUTLS_E_INTERRUPTED` to put the handshake on hold. In that case `gnutls_handshake()` will return `GNUTLS_E_INTERRUPTED` and can be resumed when needed.

**Warning:** You should not use this function to terminate the handshake based on client input unless you know what you are doing. Before the handshake is finished there is no way to know if there is a man-in-the-middle attack being performed.

## gnutls\_handshake\_set\_private\_extensions

**void gnutls\_handshake\_set\_private\_extensions** (*gnutls\_session\_t session, int allow*) [Function]

*session*: is a `gnutls_session_t` structure.

*allow*: is an integer (0 or 1)

This function will enable or disable the use of private cipher suites (the ones that start with 0xFF). By default or if `allow` is 0 then these cipher suites will not be advertised nor used.

Currently GnuTLS does not include such cipher-suites or compression algorithms.

Enabling the private ciphersuites when talking to other than gnutls servers and clients may cause interoperability problems.

## gnutls\_handshake\_set\_random

**int gnutls\_handshake\_set\_random** (*gnutls\_session\_t session*, *const gnutls\_datum\_t \* random*) [Function]

*session*: is a **gnutls\_session\_t** structure.

*random*: a random value of 32-bytes

This function will explicitly set the server or client hello random value in the subsequent TLS handshake. The random value should be a 32-byte value.

Note that this function should not normally be used as gnutls will select automatically a random value for the handshake.

This function should not be used when resuming a session.

**Returns:** GNUTLS\_E\_SUCCESS on success, or an error code.

Since 3.1.9

## gnutls\_handshake\_set\_timeout

**void gnutls\_handshake\_set\_timeout** (*gnutls\_session\_t session*, *unsigned int ms*) [Function]

*session*: is a **gnutls\_session\_t** structure.

*ms*: is a timeout value in milliseconds

This function sets the timeout for the handshake process to the provided value. Use an *ms* value of zero to disable timeout, or GNUTLS\_DEFAULT\_HANDSHAKE\_TIMEOUT for a reasonable default value.

**Since:** 3.1.0

## gnutls\_heartbeat\_allowed

**int gnutls\_heartbeat\_allowed** (*gnutls\_session\_t session*, *unsigned int type*) [Function]

*session*: is a **gnutls\_session\_t** structure.

*type*: one of GNUTLS\_HB\_LOCAL\_ALLOWED\_TO\_SEND and GNUTLS\_HB\_PEER\_ALLOWED\_TO\_SEND

This function will check whether heartbeats are allowed to be sent or received in this session.

**Returns:** Non zero if heartbeats are allowed.

**Since:** 3.1.2

## gnutls\_heartbeat\_enable

**void gnutls\_heartbeat\_enable** (*gnutls\_session\_t session*, *unsigned int type*) [Function]

*session*: is a **gnutls\_session\_t** structure.

*type*: one of the GNUTLS\_HB\_\* flags

If this function is called with the GNUTLS\_HB\_PEER\_ALLOWED\_TO\_SEND type, GnuTLS will allow heartbeat messages to be received. Moreover it also request the peer to accept heartbeat messages.



If the `type` used is `GNUTLS_HB_LOCAL_ALLOWED_TO_SEND`, then the peer will be asked to accept heartbeat messages but not send ones.

The function `gnutls_heartbeat_allowed()` can be used to test Whether locally generated heartbeat messages can be accepted by the peer.

**Since:** 3.1.2

## `gnutls_heartbeat_get_timeout`

`unsigned int gnutls_heartbeat_get_timeout (gnutls_session_t session)` [Function]

*session*: is a `gnutls_session_t` structure.

This function will return the milliseconds remaining for a retransmission of the previously sent ping message. This function is useful when ping is used in non-blocking mode, to estimate when to call `gnutls_heartbeat_ping()` if no packets have been received.

**Returns:** the remaining time in milliseconds.

**Since:** 3.1.2

## `gnutls_heartbeat_ping`

`int gnutls_heartbeat_ping (gnutls_session_t session, size_t data_size, unsigned int max_tries, unsigned int flags)` [Function]

*session*: is a `gnutls_session_t` structure.

*data\_size*: is the length of the ping payload.

*max\_tries*: if `flags` is `GNUTLS_HEARTBEAT_WAIT` then this sets the number of retransmissions. Use zero for indefinite (until timeout).

*flags*: if `GNUTLS_HEARTBEAT_WAIT` then wait for pong or timeout instead of returning immediately.

This function sends a ping to the peer. If the `flags` is set to `GNUTLS_HEARTBEAT_WAIT` then it waits for a reply from the peer.

Note that it is highly recommended to use this function with the flag `GNUTLS_HEARTBEAT_WAIT`, or you need to handle retransmissions and timeouts manually.

**Returns:** `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

**Since:** 3.1.2

## `gnutls_heartbeat_pong`

`int gnutls_heartbeat_pong (gnutls_session_t session, unsigned int flags)` [Function]

*session*: is a `gnutls_session_t` structure.

*flags*: should be zero

This function replies to a ping by sending a pong to the peer.

**Returns:** `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

**Since:** 3.1.2

## gnutls\_heartbeat\_set\_timeouts

**void gnutls\_heartbeat\_set\_timeouts** (*gnutls\_session\_t session*, [Function]  
*unsigned int retrans\_timeout, unsigned int total\_timeout*)

*session*: is a `gnutls_session_t` structure.

*retrans\_timeout*: The time at which a retransmission will occur in milliseconds

*total\_timeout*: The time at which the connection will be aborted, in milliseconds.

This function will override the timeouts for the DTLS heartbeat protocol. The retransmission timeout is the time after which a message from the peer is not received, the previous request will be retransmitted. The total timeout is the time after which the handshake will be aborted with `GNUTLS_E_TIMEDOUT`.

If the retransmission timeout is zero then the handshake will operate in a non-blocking way, i.e., return `GNUTLS_E_AGAIN`.

**Since:** 3.1.2

## gnutls\_hex2bin

**int gnutls\_hex2bin** (*const char \* hex\_data, size\_t hex\_size, void \* bin\_data, size\_t \* bin\_size*) [Function]

*hex\_data*: string with data in hex format

*hex\_size*: size of hex data

*bin\_data*: output array with binary data

*bin\_size*: when calling should hold maximum size of `bin_data`, on return will hold actual length of `bin_data`.

Convert a buffer with hex data to binary data.

**Returns:** `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

**Since:** 2.4.0

## gnutls\_hex\_decode

**int gnutls\_hex\_decode** (*const gnutls\_datum\_t \* hex\_data, void \* result, size\_t \* result\_size*) [Function]

*hex\_data*: contain the encoded data

*result*: the place where decoded data will be copied

*result\_size*: holds the size of the result

This function will decode the given encoded data, using the hex encoding used by PSK password files.

Note that `hex_data` should be null terminated.

**Returns:** `GNUTLS_E_SHORT_MEMORY_BUFFER` if the buffer given is not long enough, or 0 on success.

## gnutls\_hex\_encode

**int gnutls\_hex\_encode** (*const gnutls\_datum\_t \* data*, *char \* result*, [Function]  
*size\_t \* result\_size*)

*data*: contain the raw data

*result*: the place where hex data will be copied

*result\_size*: holds the size of the result

This function will convert the given data to printable data, using the hex encoding, as used in the PSK password files.

Note that the size of the result includes the null terminator.

**Returns:** GNUTLS\_E\_SHORT\_MEMORY\_BUFFER if the buffer given is not long enough, or 0 on success.

## gnutls\_init

**int gnutls\_init** (*gnutls\_session\_t \* session*, *unsigned int flags*) [Function]  
*session*: is a pointer to a **gnutls\_session\_t** structure.

*flags*: indicate if this session is to be used for server or client.

This function initializes the current session to null. Every session must be initialized before use, so internal structures can be allocated. This function allocates structures which can only be free'd by calling **gnutls\_deinit()** . Returns **GNUTLS\_E\_SUCCESS** (0) on success.

**flags** can be one of **GNUTLS\_CLIENT** and **GNUTLS\_SERVER** . For a DTLS entity, the flags **GNUTLS\_DATAGRAM** and **GNUTLS\_NONBLOCK** are also available. The latter flag will enable a non-blocking operation of the DTLS timers.

The flag **GNUTLS\_NO\_REPLAY\_PROTECTION** will disable any replay protection in DTLS mode. That must only used when replay protection is achieved using other means.

Note that since version 3.1.2 this function enables some common TLS extensions such as session tickets and OCSP certificate status request in client side by default. To prevent that use the **GNUTLS\_NO\_EXTENSIONS** flag.

**Returns:** **GNUTLS\_E\_SUCCESS** on success, or an error code.

## gnutls\_key\_generate

**int gnutls\_key\_generate** (*gnutls\_datum\_t \* key*, *unsigned int* [Function]  
*key\_size*)

*key*: is a pointer to a **gnutls\_datum\_t** which will contain a newly created key.

*key\_size*: The number of bytes of the key.

Generates a random key of **key\_size** bytes.

**Returns:** On success, **GNUTLS\_E\_SUCCESS** (0) is returned, or an error code.

**Since:** 3.0

## gnutls\_kx\_get

`gnutls_kx_algorithm_t gnutls_kx_get (gnutls_session_t session)` [Function]

*session*: is a `gnutls_session_t` structure.

Get currently used key exchange algorithm.

**Returns:** the key exchange algorithm used in the last handshake, a `gnutls_kx_algorithm_t` value.

## gnutls\_kx\_get\_id

`gnutls_kx_algorithm_t gnutls_kx_get_id (const char * name)` [Function]

*name*: is a KX name

Convert a string to a `gnutls_kx_algorithm_t` value. The names are compared in a case insensitive way.

**Returns:** an id of the specified KX algorithm, or `GNUTLS_KX_UNKNOWN` on error.

## gnutls\_kx\_get\_name

`const char * gnutls_kx_get_name (gnutls_kx_algorithm_t algorithm)` [Function]

*algorithm*: is a key exchange algorithm

Convert a `gnutls_kx_algorithm_t` value to a string.

**Returns:** a pointer to a string that contains the name of the specified key exchange algorithm, or `NULL` .

## gnutls\_kx\_list

`const gnutls_kx_algorithm_t * gnutls_kx_list ( void)` [Function]

Get a list of supported key exchange algorithms.

This function is not thread safe.

**Returns:** a (0)-terminated list of `gnutls_kx_algorithm_t` integers indicating the available key exchange algorithms.

## gnutls\_load\_file

`int gnutls_load_file (const char * filename, gnutls_datum_t * data)` [Function]

*filename*: the name of the file to load

*data*: Where the file will be stored

This function will load a file into a datum. The data are zero terminated but the terminating null is not included in length. The returned data are allocated using `gnutls_malloc()` .

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

Since 3.1.0

## gnutls\_mac\_get

`gnutls_mac_algorithm_t gnutls_mac_get (gnutls_session_t session)` [Function]

*session*: is a `gnutls_session_t` structure.

Get currently used MAC algorithm.

**Returns:** the currently used mac algorithm, a `gnutls_mac_algorithm_t` value.

## gnutls\_mac\_get\_id

`gnutls_mac_algorithm_t gnutls_mac_get_id (const char * name)` [Function]

*name*: is a MAC algorithm name

Convert a string to a `gnutls_mac_algorithm_t` value. The names are compared in a case insensitive way.

**Returns:** a `gnutls_mac_algorithm_t` id of the specified MAC algorithm string, or `GNUTLS_MAC_UNKNOWN` on failures.

## gnutls\_mac\_get\_key\_size

`size_t gnutls_mac_get_key_size (gnutls_mac_algorithm_t algorithm)` [Function]

*algorithm*: is an encryption algorithm

Returns the size of the MAC key used in TLS.

**Returns:** length (in bytes) of the given MAC key size, or 0 if the given MAC algorithm is invalid.

## gnutls\_mac\_get\_name

`const char * gnutls_mac_get_name (gnutls_mac_algorithm_t algorithm)` [Function]

*algorithm*: is a MAC algorithm

Convert a `gnutls_mac_algorithm_t` value to a string.

**Returns:** a string that contains the name of the specified MAC algorithm, or `NULL`.

## gnutls\_mac\_list

`const gnutls_mac_algorithm_t * gnutls_mac_list ( void)` [Function]

Get a list of hash algorithms for use as MACs. Note that not necessarily all MACs are supported in TLS cipher suites. This function is not thread safe.

**Returns:** Return a (0)-terminated list of `gnutls_mac_algorithm_t` integers indicating the available MACs.

## gnutls\_ocsp\_status\_request\_enable\_client

`int gnutls_ocsp_status_request_enable_client (gnutls_session_t session, gnutls_datum_t * responder_id, size_t responder_id_size, gnutls_datum_t * extensions)` [Function]

*session*: is a `gnutls_session_t` structure.

*responder\_id*: array with `gnutls_datum_t` with DER data of responder id

*responder\_id\_size*: number of members in `responder_id` array

*extensions*: a `gnutls_datum_t` with DER encoded OCSP extensions

This function is to be used by clients to request OCSP response from the server, using the "status\_request" TLS extension. Only OCSP status type is supported. A typical server has a single OCSP response cached, so `responder_id` and `extensions` should be null.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

**Since:** 3.1.3

### `gnutls_ocsp_status_request_get`

`int gnutls_ocsp_status_request_get (gnutls_session_t session, [Function]  
gnutls_datum_t * response)`

*session*: is a `gnutls_session_t` structure.

*response*: a `gnutls_datum_t` with DER encoded OCSP response

This function returns the OCSP status response received from the TLS server. The `response` should be treated as constant. If no OCSP response is available then `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

**Since:** 3.1.3

### `gnutls_ocsp_status_request_is_checked`

`int gnutls_ocsp_status_request_is_checked (gnutls_session_t [Function]  
session, unsigned int flags)`

*session*: is a gnutls session

*flags*: should be zero

Check whether an OCSP status response was included in the handshake and whether it was checked and valid (not too old or superseded). This is a helper function when needing to decide whether to perform an OCSP validity check on the peer's certificate. Must be called after `gnutls_certificate_verify_peers3()` is called.

**Returns:** non zero it was valid, or a zero if it wasn't sent, or sent and was invalid.

### `gnutls_openpgp_send_cert`

`void gnutls_openpgp_send_cert (gnutls_session_t session, [Function]  
gnutls_openpgp_cert_status_t status)`

*session*: is a pointer to a `gnutls_session_t` structure.

*status*: is one of `GNUTLS_OPENPGP_CERT`, or `GNUTLS_OPENPGP_CERT_FINGERPRINT`

This function will order gnutls to send the key fingerprint instead of the key in the initial handshake procedure. This should be used with care and only when there is indication or knowledge that the server can obtain the client's key.

**gnutls\_packet\_deinit**

**void gnutls\_packet\_deinit** (*gnutls\_packet\_t packet*) [Function]

*packet*: is a pointer to a **gnutls\_packet\_st** structure.

This function will deinitialize all data associated with the received packet.

**Since:** 3.3.5

**gnutls\_packet\_get**

**void gnutls\_packet\_get** (*gnutls\_packet\_t packet*, *gnutls\_datum\_t \* data*, *unsigned char \* sequence*) [Function]

*packet*: is a **gnutls\_packet\_t** structure.

*data*: will contain the data present in the **packet** structure (may be NULL )

*sequence*: the 8-bytes of the packet sequence number (may be NULL )

This function returns the data and sequence number associated with the received packet.

**Since:** 3.3.5

**gnutls\_pem\_base64\_decode**

**int gnutls\_pem\_base64\_decode** (*const char \* header*, *const gnutls\_datum\_t \* b64\_data*, *unsigned char \* result*, *size\_t \* result\_size*) [Function]

*header*: A null terminated string with the PEM header (eg. CERTIFICATE)

*b64\_data*: contain the encoded data

*result*: the place where decoded data will be copied

*result\_size*: holds the size of the result

This function will decode the given encoded data. If the header given is non null this function will search for "—BEGIN header" and decode only this part. Otherwise it will decode the first PEM packet found.

**Returns:** On success **GNUTLS\_E\_SUCCESS** (0) is returned, **GNUTLS\_E\_SHORT\_MEMORY\_BUFFER** is returned if the buffer given is not long enough, or 0 on success.

**gnutls\_pem\_base64\_decode\_alloc**

**int gnutls\_pem\_base64\_decode\_alloc** (*const char \* header*, *const gnutls\_datum\_t \* b64\_data*, *gnutls\_datum\_t \* result*) [Function]

*header*: The PEM header (eg. CERTIFICATE)

*b64\_data*: contains the encoded data

*result*: the place where decoded data lie

This function will decode the given encoded data. The decoded data will be allocated, and stored into *result*. If the header given is non null this function will search for "—BEGIN header" and decode only this part. Otherwise it will decode the first PEM packet found.

You should use **gnutls\_free()** to free the returned data.

**Returns:** On success, **GNUTLS\_E\_SUCCESS** (0) is returned, otherwise an error code is returned.

## gnutls\_pem\_base64\_encode

`int gnutls_pem_base64_encode (const char *msg, const gnutls_datum_t *data, char *result, size_t *result_size)` [Function]

*msg*: is a message to be put in the header

*data*: contain the raw data

*result*: the place where base64 data will be copied

*result\_size*: holds the size of the result

This function will convert the given data to printable data, using the base64 encoding. This is the encoding used in PEM messages.

The output string will be null terminated, although the size will not include the terminating null.

**Returns:** On success GNUTLS\_E\_SUCCESS (0) is returned, GNUTLS\_E\_SHORT\_MEMORY\_BUFFER is returned if the buffer given is not long enough, or 0 on success.

## gnutls\_pem\_base64\_encode\_alloc

`int gnutls_pem_base64_encode_alloc (const char *msg, const gnutls_datum_t *data, gnutls_datum_t *result)` [Function]

*msg*: is a message to be put in the encoded header

*data*: contains the raw data

*result*: will hold the newly allocated encoded data

This function will convert the given data to printable data, using the base64 encoding. This is the encoding used in PEM messages. This function will allocate the required memory to hold the encoded data.

You should use `gnutls_free()` to free the returned data.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise an error code is returned.

## gnutls\_perror

`void gnutls_perror (int error)` [Function]

*error*: is a GnuTLS error code, a negative error code

This function is like `perror()` . The only difference is that it accepts an error number returned by a gnutls function.

## gnutls\_pk\_algorithm\_get\_name

`const char * gnutls_pk_algorithm_get_name (gnutls_pk_algorithm_t algorithm)` [Function]

*algorithm*: is a pk algorithm

Convert a `gnutls_pk_algorithm_t` value to a string.

**Returns:** a string that contains the name of the specified public key algorithm, or NULL .



## gnutls\_pk\_bits\_to\_sec\_param

`gnutls_sec_param_t gnutls_pk_bits_to_sec_param` [Function]

(*gnutls\_pk\_algorithm\_t algo, unsigned int bits*)

*algo*: is a public key algorithm

*bits*: is the number of bits

This is the inverse of `gnutls_sec_param_to_pk_bits()` . Given an algorithm and the number of bits, it will return the security parameter. This is a rough indication.

**Returns:** The security parameter.

**Since:** 2.12.0

## gnutls\_pk\_get\_id

`gnutls_pk_algorithm_t gnutls_pk_get_id (const char * name)` [Function]

*name*: is a string containing a public key algorithm name.

Convert a string to a `gnutls_pk_algorithm_t` value. The names are compared in a case insensitive way. For example, `gnutls_pk_get_id("RSA")` will return `GNUTLS_PK_RSA` .

**Returns:** a `gnutls_pk_algorithm_t` id of the specified public key algorithm string, or `GNUTLS_PK_UNKNOWN` on failures.

**Since:** 2.6.0

## gnutls\_pk\_get\_name

`const char * gnutls_pk_get_name (gnutls_pk_algorithm_t algorithm)` [Function]

*algorithm*: is a public key algorithm

Convert a `gnutls_pk_algorithm_t` value to a string.

**Returns:** a pointer to a string that contains the name of the specified public key algorithm, or `NULL` .

**Since:** 2.6.0

## gnutls\_pk\_list

`const gnutls_pk_algorithm_t * gnutls_pk_list ( void)` [Function]

Get a list of supported public key algorithms.

This function is not thread safe.

**Returns:** a (0)-terminated list of `gnutls_pk_algorithm_t` integers indicating the available ciphers.

**Since:** 2.6.0

## gnutls\_pk\_to\_sign

`gnutls_sign_algorithm_t gnutls_pk_to_sign` [Function]

(*gnutls\_pk\_algorithm\_t pk, gnutls\_digest\_algorithm\_t hash*)

*pk*: is a public key algorithm

*hash*: a hash algorithm

This function maps public key and hash algorithms combinations to signature algorithms.

**Returns:** return a `gnutls_sign_algorithm_t` value, or `GNUTLS_SIGN_UNKNOWN` on error.

## gnutls\_prf

```
int gnutls_prf (gnutls_session_t session, size_t label_size, const [Function]
                char * label, int server_random_first, size_t extra_size, const char *
                extra, size_t outsize, char * out)
```

*session*: is a `gnutls_session_t` structure.

*label\_size*: length of the `label` variable.

*label*: label used in PRF computation, typically a short string.

*server\_random\_first*: non-zero if server random field should be first in seed

*extra\_size*: length of the `extra` variable.

*extra*: optional extra data to seed the PRF with.

*outsize*: size of pre-allocated output buffer to hold the output.

*out*: pre-allocated buffer to hold the generated data.

Applies the TLS Pseudo-Random-Function (PRF) on the master secret and the provided data, seeded with the client and server random fields.

The output of this function is identical to RFC5705 extractor if `extra` and `extra_size` are set to zero. Otherwise, `extra` should contain the context value prefixed by a two-byte length.

The `label` variable usually contains a string denoting the purpose for the generated data. The `server_random_first` indicates whether the client random field or the server random field should be first in the seed. Non-zero indicates that the server random field is first, 0 that the client random field is first.

The `extra` variable can be used to add more data to the seed, after the random variables. It can be used to make sure the generated output is strongly connected to some additional data (e.g., a string used in user authentication).

The output is placed in `out`, which must be pre-allocated.

**Returns:** `GNUTLS_E_SUCCESS` on success, or an error code.

## gnutls\_prf\_raw

```
int gnutls_prf_raw (gnutls_session_t session, size_t label_size, [Function]
                    const char * label, size_t seed_size, const char * seed, size_t outsize,
                    char * out)
```

*session*: is a `gnutls_session_t` structure.

*label\_size*: length of the `label` variable.

*label*: label used in PRF computation, typically a short string.

*seed\_size*: length of the `seed` variable.

*seed*: optional extra data to seed the PRF with.

*outsize*: size of pre-allocated output buffer to hold the output.

*out*: pre-allocated buffer to hold the generated data.

Apply the TLS Pseudo-Random-Function (PRF) on the master secret and the provided data.

The `label` variable usually contains a string denoting the purpose for the generated data. The `seed` usually contains data such as the client and server random, perhaps together with some additional data that is added to guarantee uniqueness of the output for a particular purpose.

Because the output is not guaranteed to be unique for a particular session unless `seed` includes the client random and server random fields (the PRF would output the same data on another connection resumed from the first one), it is not recommended to use this function directly. The `gnutls_prf()` function seeds the PRF with the client and server random fields directly, and is recommended if you want to generate pseudo random data unique for each session.

**Returns:** GNUTLS\_E\_SUCCESS on success, or an error code.

### gnutls\_priority\_certificate\_type\_list

`int gnutls_priority_certificate_type_list (gnutls_priority_t pcache, const unsigned int **list)` [Function]

*pcache*: is a `gnutls_priority_t` structure.

*list*: will point to an integer list

Get a list of available certificate types in the priority structure.

**Returns:** the number of certificate types, or an error code.

**Since:** 3.0

### gnutls\_priority\_cipher\_list

`int gnutls_priority_cipher_list (gnutls_priority_t pcache, const unsigned int **list)` [Function]

*pcache*: is a `gnutls_priority_t` structure.

*list*: will point to an integer list

Get a list of available ciphers in the priority structure.

**Returns:** the number of curves, or an error code.

**Since:** 3.2.3

### gnutls\_priority\_compression\_list

`int gnutls_priority_compression_list (gnutls_priority_t pcache, const unsigned int **list)` [Function]

*pcache*: is a `gnutls_priority_t` structure.

*list*: will point to an integer list

Get a list of available compression method in the priority structure.

**Returns:** the number of methods, or an error code.

**Since:** 3.0

## gnutls\_priority\_deinit

**void gnutls\_priority\_deinit** (*gnutls\_priority\_t* *priority\_cache*) [Function]  
*priority\_cache*: is a *gnutls\_priority\_t* structure.  
 Deinitializes the priority cache.

## gnutls\_priority\_ecc\_curve\_list

**int gnutls\_priority\_ecc\_curve\_list** (*gnutls\_priority\_t* *pcache*, [Function]  
     *const unsigned int \*\*list*)  
*pcache*: is a *gnutls\_priority\_t* structure.  
*list*: will point to an integer list  
 Get a list of available elliptic curves in the priority structure.  
**Returns:** the number of curves, or an error code.  
**Since:** 3.0

## gnutls\_priority\_get\_cipher\_suite\_index

**int gnutls\_priority\_get\_cipher\_suite\_index** (*gnutls\_priority\_t* [Function]  
     *pcache*, *unsigned int idx*, *unsigned int \*sidx*)  
*pcache*: is a *gnutls\_priority\_t* structure.  
*idx*: is an index number.  
*sidx*: internal index of cipher suite to get information about.  
 Provides the internal ciphersuite index to be used with *gnutls\_cipher\_suite\_info()* . The index *idx* provided is an index kept at the priorities structure. It might be that a valid priorities index does not correspond to a ciphersuite and in that case *GNUTLS\_E\_UNKNOWN\_CIPHER\_SUITE* will be returned. Once the last available index is crossed then *GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE* will be returned.  
**Returns:** On success it returns *GNUTLS\_E\_SUCCESS* (0), or a negative error value otherwise.

## gnutls\_priority\_init

**int gnutls\_priority\_init** (*gnutls\_priority\_t \*priority\_cache*, [Function]  
     *const char \*priorities*, *const char \*\*err\_pos*)  
*priority\_cache*: is a *gnutls\_priority\_t* structure.  
*priorities*: is a string describing priorities (may be NULL )  
*err\_pos*: In case of an error this will have the position in the string the error occurred  
 Sets priorities for the ciphers, key exchange methods, macs and compression methods.  
 The *priorities* option allows you to specify a colon separated list of the cipher priorities to enable. Some keywords are defined to provide quick access to common preferences.  
 Unless there is a special need, use the "NORMAL" keyword to apply a reasonable security level, or "NORMAL:COMPAT " for compatibility.

"PERFORMANCE" means all the "secure" ciphersuites are enabled, limited to 128 bit ciphers and sorted by terms of speed performance.

"LEGACY" the NORMAL settings for GnuTLS 3.2.x or earlier. There is no verification profile set, and the allowed DH primes are considered weak today.

"NORMAL" means all "secure" ciphersuites. The 256-bit ciphers are included as a fallback only. The ciphers are sorted by security margin.

"PFS" means all "secure" ciphersuites that support perfect forward secrecy. The 256-bit ciphers are included as a fallback only. The ciphers are sorted by security margin.

"SECURE128" means all "secure" ciphersuites of security level 128-bit or more.

"SECURE192" means all "secure" ciphersuites of security level 192-bit or more.

"SUITEB128" means all the NSA SuiteB ciphersuites with security level of 128.

"SUITEB192" means all the NSA SuiteB ciphersuites with security level of 192.

"EXPORT" means all ciphersuites are enabled, including the low-security 40 bit ciphers.

"NONE" means nothing is enabled. This disables even protocols and compression methods.

" **KEYWORD** " The system administrator imposed settings. The provided keywords will be expanded from a configuration-time provided file - default is: /etc/gnutls/default-priorities. Any keywords that follow it, will be appended to the expanded string. If there is no system string, then the function will fail. The system file should be formatted as "KEYWORD=VALUE", e.g., "SYSTEM=NORMAL:-ARCFOUR-128".

Special keywords are "!", "-", and "+". "!" or "-" appended with an algorithm will remove this algorithm. "+" appended with an algorithm will add this algorithm.

Check the GnuTLS manual section "Priority strings" for detailed information.

**Examples:** "NONE:+VERS-TLS-ALL:+MAC-ALL:+RSA:+AES-128-CBC:+SIGN-ALL:+COMP-NULL"

"NORMAL:-ARCFOUR-128" means normal ciphers except for ARCFOUR-128.

"SECURE128:-VERS-SSL3.0:+COMP-DEFLATE" means that only secure ciphers are enabled, SSL3.0 is disabled, and libz compression enabled.

"NONE:+VERS-TLS-ALL:+AES-128-CBC:+RSA:+SHA1:+COMP-NULL:+SIGN-RSA-SHA1",

"NONE:+VERS-TLS-ALL:+AES-128-CBC:+ECDHE-RSA:+SHA1:+COMP-NULL:+SIGN-RSA-SHA1:+CURVE-SECP256R1",

"SECURE256:+SECURE128",

Note that "NORMAL:COMPAT " is the most compatible mode.

A **NULL priorities** string indicates the default priorities to be used (this is available since GnuTLS 3.3.0).

**Returns:** On syntax error **GNUTLS\_E\_INVALID\_REQUEST** is returned, **GNUTLS\_E\_SUCCESS** on success, or an error code.

**gnutls\_priority\_kx\_list**

**int gnutls\_priority\_kx\_list** (*gnutls\_priority\_t* *pcache*, *const unsigned int \*\* list*) [Function]

*pcache*: is a *gnutls\_priority\_t* structure.  
*list*: will point to an integer list

Get a list of available key exchange methods in the priority structure.

**Returns:** the number of curves, or an error code.

**Since:** 3.2.3

**gnutls\_priority\_mac\_list**

**int gnutls\_priority\_mac\_list** (*gnutls\_priority\_t* *pcache*, *const unsigned int \*\* list*) [Function]

*pcache*: is a *gnutls\_priority\_t* structure.  
*list*: will point to an integer list

Get a list of available MAC algorithms in the priority structure.

**Returns:** the number of curves, or an error code.

**Since:** 3.2.3

**gnutls\_priority\_protocol\_list**

**int gnutls\_priority\_protocol\_list** (*gnutls\_priority\_t* *pcache*, *const unsigned int \*\* list*) [Function]

*pcache*: is a *gnutls\_priority\_t* structure.  
*list*: will point to an integer list

Get a list of available TLS version numbers in the priority structure.

**Returns:** the number of protocols, or an error code.

**Since:** 3.0

**gnutls\_priority\_set**

**int gnutls\_priority\_set** (*gnutls\_session\_t* *session*, *gnutls\_priority\_t* *priority*) [Function]

*session*: is a *gnutls\_session\_t* structure.  
*priority*: is a *gnutls\_priority\_t* structure.

Sets the priorities to use on the ciphers, key exchange methods, macs and compression methods.

**Returns:** GNUTLS\_E\_SUCCESS on success, or an error code.

**gnutls\_priority\_set\_direct**

**int gnutls\_priority\_set\_direct** (*gnutls\_session\_t* *session*, *const char \* priorities*, *const char \*\* err\_pos*) [Function]

*session*: is a *gnutls\_session\_t* structure.

*priorities*: is a string describing priorities

*err\_pos*: In case of an error this will have the position in the string the error occurred

Sets the priorities to use on the ciphers, key exchange methods, macs and compression methods. This function avoids keeping a priority cache and is used to directly set string priorities to a TLS session. For documentation check the `gnutls_priority_init()` .

To simply use a reasonable default, consider using `gnutls_set_default_priority()` .

**Returns:** On syntax error `GNUTLS_E_INVALID_REQUEST` is returned, `GNUTLS_E_SUCCESS` on success, or an error code.

### **gnutls\_priority\_sign\_list**

`int gnutls_priority_sign_list (gnutls_priority_t pcache, const [Function]  
                                  unsigned int ** list)`

*pcache*: is a `gnutls_priority_t` structure.

*list*: will point to an integer list

Get a list of available signature algorithms in the priority structure.

**Returns:** the number of algorithms, or an error code.

**Since:** 3.0

### **gnutls\_protocol\_get\_id**

`gnutls_protocol_t gnutls_protocol_get_id (const char * name) [Function]  
                                  name: is a protocol name`

The names are compared in a case insensitive way.

**Returns:** an id of the specified protocol, or `GNUTLS_VERSION_UNKNOWN` on error.

### **gnutls\_protocol\_get\_name**

`const char * gnutls_protocol_get_name (gnutls_protocol_t [Function]  
                                  version)`

*version*: is a (`gnutls`) version number

Convert a `gnutls_protocol_t` value to a string.

**Returns:** a string that contains the name of the specified TLS version (e.g., "TLS1.0"), or `NULL` .

### **gnutls\_protocol\_get\_version**

`gnutls_protocol_t gnutls_protocol_get_version [Function]  
                                  (gnutls_session_t session)`

*session*: is a `gnutls_session_t` structure.

Get TLS version, a `gnutls_protocol_t` value.

**Returns:** The version of the currently used protocol.

## gnutls\_protocol\_list

`const gnutls_protocol_t * gnutls_protocol_list ( void)` [Function]

Get a list of supported protocols, e.g. SSL 3.0, TLS 1.0 etc.

This function is not thread safe.

**Returns:** a (0)-terminated list of `gnutls_protocol_t` integers indicating the available protocols.

## gnutls\_psk\_allocate\_client\_credentials

`int gnutls_psk_allocate_client_credentials` [Function]  
(*gnutls\_psk\_client\_credentials\_t \* sc*)

*sc*: is a pointer to a `gnutls_psk_server_credentials_t` structure.

This structure is complex enough to manipulate directly thus this helper function is provided in order to allocate it.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

## gnutls\_psk\_allocate\_server\_credentials

`int gnutls_psk_allocate_server_credentials` [Function]  
(*gnutls\_psk\_server\_credentials\_t \* sc*)

*sc*: is a pointer to a `gnutls_psk_server_credentials_t` structure.

This structure is complex enough to manipulate directly thus this helper function is provided in order to allocate it.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

## gnutls\_psk\_client\_get\_hint

`const char * gnutls_psk_client_get_hint (gnutls_session_t` [Function]  
*session*)

*session*: is a gnutls session

The PSK identity hint may give the client help in deciding which username to use. This should only be called in case of PSK authentication and in case of a client.

**Returns:** the identity hint of the peer, or NULL in case of an error.

**Since:** 2.4.0

## gnutls\_psk\_free\_client\_credentials

`void gnutls_psk_free_client_credentials` [Function]  
(*gnutls\_psk\_client\_credentials\_t sc*)

*sc*: is a `gnutls_psk_client_credentials_t` structure.

This structure is complex enough to manipulate directly thus this helper function is provided in order to free (deallocate) it.



## gnutls\_psk\_free\_server\_credentials

`void gnutls_psk_free_server_credentials` [Function]  
     (*gnutls\_psk\_server\_credentials\_t* *sc*)

*sc*: is a `gnutls_psk_server_credentials_t` structure.

This structure is complex enough to manipulate directly thus this helper function is provided in order to free (deallocate) it.

## gnutls\_psk\_server\_get\_username

`const char * gnutls_psk_server_get_username` (*gnutls\_session\_t* *session*) [Function]

*session*: is a gnutls session

This should only be called in case of PSK authentication and in case of a server.

**Returns:** the username of the peer, or NULL in case of an error.

## gnutls\_psk\_set\_client\_credentials

`int gnutls_psk_set_client_credentials` [Function]  
     (*gnutls\_psk\_client\_credentials\_t* *res*, *const char \* username*, *const*  
     *gnutls\_datum\_t \* key*, *gnutls\_psk\_key\_flags flags*)

*res*: is a `gnutls_psk_client_credentials_t` structure.

*username*: is the user's zero-terminated userid

*key*: is the user's key

*flags*: indicate the format of the key, either `GNUTLS_PSK_KEY_RAW` or `GNUTLS_PSK_KEY_HEX`.

This function sets the username and password, in a `gnutls_psk_client_credentials_t` structure. Those will be used in PSK authentication. *username* should be an ASCII string or UTF-8 strings prepared using the "SASLprep" profile of "stringprep". The key can be either in raw byte format or in Hex format (without the 0x prefix).

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

## gnutls\_psk\_set\_client\_credentials\_function

`void gnutls_psk_set_client_credentials_function` [Function]  
     (*gnutls\_psk\_client\_credentials\_t* *cred*, *gnutls\_psk\_client\_credentials\_function \* func*)

*cred*: is a `gnutls_psk_server_credentials_t` structure.

*func*: is the callback function

This function can be used to set a callback to retrieve the username and password for client PSK authentication. The callback's function form is: `int (*callback)(gnutls_session_t, char** username, gnutls_datum_t* key);`

The *username* and *key* ->data must be allocated using `gnutls_malloc()`. *username* should be ASCII strings or UTF-8 strings prepared using the "SASLprep" profile of "stringprep".

The callback function will be called once per handshake.

The callback function should return 0 on success. -1 indicates an error.

### **gnutls\_psk\_set\_params\_function**

**void gnutls\_psk\_set\_params\_function** [Function]  
     (*gnutls\_psk\_server\_credentials\_t* **res**, *gnutls\_params\_function* \* **func**)

**res**: is a *gnutls\_psk\_server\_credentials\_t* structure

**func**: is the function to be called

This function will set a callback in order for the server to get the Diffie-Hellman or RSA parameters for PSK authentication. The callback should return **GNUTLS\_E\_SUCCESS** (0) on success.

### **gnutls\_psk\_set\_server\_credentials\_file**

**int gnutls\_psk\_set\_server\_credentials\_file** [Function]  
     (*gnutls\_psk\_server\_credentials\_t* **res**, *const char \****password\_file**)

**res**: is a *gnutls\_psk\_server\_credentials\_t* structure.

**password\_file**: is the PSK password file (passwd.psk)

This function sets the password file, in a *gnutls\_psk\_server\_credentials\_t* structure. This password file holds usernames and keys and will be used for PSK authentication.

**Returns:** On success, **GNUTLS\_E\_SUCCESS** (0) is returned, otherwise an error code is returned.

### **gnutls\_psk\_set\_server\_credentials\_function**

**void gnutls\_psk\_set\_server\_credentials\_function** [Function]  
     (*gnutls\_psk\_server\_credentials\_t* **cred**, *gnutls\_psk\_server\_credentials\_function* \* **func**)

**cred**: is a *gnutls\_psk\_server\_credentials\_t* structure.

**func**: is the callback function

This function can be used to set a callback to retrieve the user's PSK credentials. The callback's function form is: `int (*callback)(gnutls_session_t, const char* username, gnutls_datum_t* key);`

**username** contains the actual username. The **key** must be filled in using the **gnutls\_malloc()** .

In case the callback returned a negative number then gnutls will assume that the username does not exist.

The callback function will only be called once per handshake. The callback function should return 0 on success, while -1 indicates an error.

### **gnutls\_psk\_set\_server\_credentials\_hint**

**int gnutls\_psk\_set\_server\_credentials\_hint** [Function]  
     (*gnutls\_psk\_server\_credentials\_t* **res**, *const char \****hint**)

**res**: is a *gnutls\_psk\_server\_credentials\_t* structure.

*hint*: is the PSK identity hint string

This function sets the identity hint, in a `gnutls_psk_server_credentials_t` structure. This hint is sent to the client to help it chose a good PSK credential (i.e., username and password).

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

**Since:** 2.4.0

## `gnutls_psk_set_server_dh_params`

`void gnutls_psk_set_server_dh_params` [Function]  
     (`gnutls_psk_server_credentials_t res`, `gnutls_dh_params_t dh_params`)

*res*: is a `gnutls_psk_server_credentials_t` structure

*dh\_params*: is a structure that holds Diffie-Hellman parameters.

This function will set the Diffie-Hellman parameters for an anonymous server to use. These parameters will be used in Diffie-Hellman exchange with PSK cipher suites.

## `gnutls_psk_set_server_params_function`

`void gnutls_psk_set_server_params_function` [Function]  
     (`gnutls_psk_server_credentials_t res`, `gnutls_params_function * func`)

*res*: is a `gnutls_certificate_credentials_t` structure

*func*: is the function to be called

This function will set a callback in order for the server to get the Diffie-Hellman parameters for PSK authentication. The callback should return `GNUTLS_E_SUCCESS` (0) on success.

## `gnutls_random_art`

`int gnutls_random_art` (`gnutls_random_art_t type`, `const char *` [Function]  
     `key_type`, `unsigned int key_size`, `void * fpr`, `size_t fpr_size`,  
     `gnutls_datum_t * art`)

*type*: The type of the random art (for now only `GNUTLS_RANDOM_ART_OPENSSH` is supported)

*key\_type*: The type of the key (RSA, DSA etc.)

*key\_size*: The size of the key in bits

*fpr*: The fingerprint of the key

*fpr\_size*: The size of the fingerprint

*art*: The returned random art

This function will convert a given fingerprint to an "artistic" image. The returned image is allocated using `gnutls_malloc()`.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

## gnutls\_range\_split

**int gnutls\_range\_split** (*gnutls\_session\_t session*, *const* [Function]  
*gnutls\_range\_st \* orig*, *gnutls\_range\_st \* next*, *gnutls\_range\_st \* remainder*)

*session*: is a **gnutls\_session\_t** structure

*orig*: is the original range provided by the user

*next*: is the returned range that can be conveyed in a TLS record

*remainder*: is the returned remaining range

This function should be used when it is required to hide the length of very long data that cannot be directly provided to **gnutls\_record\_send\_range()**. In that case this function should be called with the desired length hiding range in **orig**. The returned **next** value should then be used in the next call to **gnutls\_record\_send\_range()** with the partial data. That process should be repeated until **remainder** is (0,0).

**Returns:** 0 in case splitting succeeds, non zero in case of error. Note that **orig** is not changed, while the values of **next** and **remainder** are modified to store the resulting values.

## gnutls\_record\_can\_use\_length\_hiding

**int gnutls\_record\_can\_use\_length\_hiding** (*gnutls\_session\_t* [Function]  
*session*)

*session*: is a **gnutls\_session\_t** structure.

If the session supports length-hiding padding, you can invoke **gnutls\_range\_send\_message()** to send a message whose length is hidden in the given range. If the session does not support length hiding padding, you can use the standard **gnutls\_record\_send()** function, or **gnutls\_range\_send\_message()** making sure that the range is the same as the length of the message you are trying to send.

**Returns:** true (1) if the current session supports length-hiding padding, false (0) if the current session does not.

## gnutls\_record\_check\_corked

**size\_t gnutls\_record\_check\_corked** (*gnutls\_session\_t session*) [Function]  
*session*: is a **gnutls\_session\_t** structure.

This function checks if there pending corked data in the gnutls buffers –see **gnutls\_record\_cork()**.

**Returns:** Returns the size of the corked data or zero.

**Since:** 3.2.8

## gnutls\_record\_check\_pending

**size\_t gnutls\_record\_check\_pending** (*gnutls\_session\_t session*) [Function]  
*session*: is a **gnutls\_session\_t** structure.

This function checks if there are unread data in the gnutls buffers. If the return value is non-zero the next call to **gnutls\_record\_recv()** is guaranteed not to block.

**Returns:** Returns the size of the data or zero.

## gnutls\_record\_cork

**void gnutls\_record\_cork** (*gnutls\_session\_t session*) [Function]

*session*: is a `gnutls_session_t` structure.

If called, `gnutls_record_send()` will no longer send any records. Any sent records will be cached until `gnutls_record_uncork()` is called.

This function is safe to use with DTLS after GnuTLS 3.3.0.

**Since:** 3.1.9

## gnutls\_record\_disable\_padding

**void gnutls\_record\_disable\_padding** (*gnutls\_session\_t session*) [Function]

*session*: is a `gnutls_session_t` structure.

Used to disabled padding in TLS 1.0 and above. Normally you do not need to use this function, but there are buggy clients that complain if a server pads the encrypted data. This of course will disable protection against statistical attacks on the data.

This functions is defunt since 3.1.7. Random padding is disabled by default unless requested using `gnutls_range_send_message()` .

## gnutls\_record\_get\_direction

**int gnutls\_record\_get\_direction** (*gnutls\_session\_t session*) [Function]

*session*: is a `gnutls_session_t` structure.

This function provides information about the internals of the record protocol and is only useful if a prior `gnutls` function call (e.g. `gnutls_handshake()` ) was interrupted for some reason, that is, if a function returned `GNUTLS_E_INTERRUPTED` or `GNUTLS_E_AGAIN` . In such a case, you might want to call `select()` or `poll()` before calling the interrupted `gnutls` function again. To tell you whether a file descriptor should be selected for either reading or writing, `gnutls_record_get_direction()` returns 0 if the interrupted function was trying to read data, and 1 if it was trying to write data.

This function's output is unreliable if you are using the `session` in different threads, for sending and receiving.

**Returns:** 0 if trying to read data, 1 if trying to write data.

## gnutls\_record\_get\_max\_size

**size\_t gnutls\_record\_get\_max\_size** (*gnutls\_session\_t session*) [Function]

*session*: is a `gnutls_session_t` structure.

Get the record size. The maximum record size is negotiated by the client after the first handshake message.

**Returns:** The maximum record packet size in this connection.

## gnutls\_record\_overhead\_size

**size\_t gnutls\_record\_overhead\_size** (*gnutls\_session\_t session*) [Function]

*session*: is `gnutls_session_t`

This function will return the set size in bytes of the overhead due to TLS (or DTLS) per record.

**Since:** 3.2.2

## gnutls\_record\_recv

`ssize_t gnutls_record_recv (gnutls_session_t session, void * data, [Function]  
size_t data_size)`

*session*: is a `gnutls_session_t` structure.

*data*: the buffer that the data will be read into

*data\_size*: the number of requested bytes

This function has the similar semantics with `recv()` . The only difference is that it accepts a GnuTLS session, and uses different error codes. In the special case that a server requests a renegotiation, the client may receive an error code of `GNUTLS_E_REHANDSHAKE` . This message may be simply ignored, replied with an alert `GNUTLS_A_NO_RENEGOTIATION` , or replied with a new handshake, depending on the client's will. If `EINTR` is returned by the internal push function (the default is `recv()` ) then `GNUTLS_E_INTERRUPTED` will be returned. If `GNUTLS_E_INTERRUPTED` or `GNUTLS_E_AGAIN` is returned, you must call this function again to get the data. See also `gnutls_record_get_direction()` . A server may also receive `GNUTLS_E_REHANDSHAKE` when a client has initiated a handshake. In that case the server can only initiate a handshake or terminate the connection.

**Returns:** The number of bytes received and zero on EOF (for stream connections). A negative error code is returned in case of an error. The number of bytes received might be less than the requested `data_size` .

## gnutls\_record\_recv\_packet

`ssize_t gnutls_record_recv_packet (gnutls_session_t session, [Function]  
gnutls_packet_t * packet)`

*session*: is a `gnutls_session_t` structure.

*packet*: the structure that will hold the packet data

This is a lower-level function than `gnutls_record_recv()` and allows to directly receive the whole decrypted packet. That avoids a memory copy, and is mostly applicable to applications seeking high performance.

The received packet is accessed using `gnutls_packet_get()` and must be deinitialized using `gnutls_packet_deinit()` . The returned packet will be `NULL` if the return value is zero (EOF).

**Returns:** The number of bytes received and zero on EOF (for stream connections). A negative error code is returned in case of an error.

**Since:** 3.3.5

## gnutls\_record\_recv\_seq

`ssize_t gnutls_record_recv_seq (gnutls_session_t session, void * [Function]  
data, size_t data_size, unsigned char * seq)`

*session*: is a `gnutls_session_t` structure.

*data*: the buffer that the data will be read into

*data\_size*: the number of requested bytes

*seq*: is the packet's 64-bit sequence number. Should have space for 8 bytes.

This function is the same as `gnutls_record_recv()` , except that it returns in addition to data, the sequence number of the data. This is useful in DTLS where record packets might be received out-of-order. The returned 8-byte sequence number is an integer in big-endian format and should be treated as a unique message identification.

**Returns:** The number of bytes received and zero on EOF. A negative error code is returned in case of an error. The number of bytes received might be less than *data\_size* .

**Since:** 3.0

## gnutls\_record\_send

`ssize_t gnutls_record_send (gnutls_session_t session, const void * data, size_t data_size)` [Function]

*session*: is a `gnutls_session_t` structure.

*data*: contains the data to send

*data\_size*: is the length of the data

This function has the similar semantics with `send()` . The only difference is that it accepts a GnuTLS session, and uses different error codes. Note that if the send buffer is full, `send()` will block this function. See the `send()` documentation for more information.

You can replace the default push function which is `send()` , by using `gnutls_transport_set_push_function()` .

If the `EINTR` is returned by the internal push function then `GNUTLS_E_INTERRUPTED` will be returned. If `GNUTLS_E_INTERRUPTED` or `GNUTLS_E_AGAIN` is returned, you must call this function again, with the exact same parameters; alternatively you could provide a `NULL` pointer for data, and 0 for size. cf. `gnutls_record_get_direction()` .

Note that in DTLS this function will return the `GNUTLS_E_LARGE_PACKET` error code if the send data exceed the data MTU value - as returned by `gnutls_dtls_get_data_mtu()` . The `errno` value `EMSGSIZE` also maps to `GNUTLS_E_LARGE_PACKET` . Note that since 3.2.13 this function can be called under cork in DTLS mode, and will refuse to send data over the MTU size by returning `GNUTLS_E_LARGE_PACKET` .

**Returns:** The number of bytes sent, or a negative error code. The number of bytes sent might be less than *data\_size* . The maximum number of bytes this function can send in a single call depends on the negotiated maximum record size.

## gnutls\_record\_send\_range

`ssize_t gnutls_record_send_range (gnutls_session_t session, const void * data, size_t data_size, const gnutls_range_st * range)` [Function]

*session*: is a `gnutls_session_t` structure.

*data*: contains the data to send.

*data\_size*: is the length of the data.

*range*: is the range of lengths in which the real data length must be hidden.

This function operates like `gnutls_record_send()` but, while `gnutls_record_send()` adds minimal padding to each TLS record, this function uses the TLS extra-padding feature to conceal the real data size within the range of lengths provided. Some TLS sessions do not support extra padding (e.g. stream ciphers in standard TLS or SSL3 sessions). To know whether the current session supports extra padding, and hence length hiding, use the `gnutls_record_can_use_length_hiding()` function.

**Note:** This function currently is only limited to blocking sockets.

**Returns:** The number of bytes sent (that is *data\_size* in a successful invocation), or a negative error code.

## `gnutls_record_set_max_empty_records`

`void gnutls_record_set_max_empty_records (gnutls_session_t session, const unsigned int i)` [Function]

*session*: is a `gnutls_session_t` structure.

*i*: is the desired value of maximum empty records that can be accepted in a row.

Used to set the maximum number of empty fragments that can be accepted in a row. Accepting many empty fragments is useful for receiving length-hidden content, where empty fragments filled with pad are sent to hide the real length of a message. However, a malicious peer could send empty fragments to mount a DoS attack, so as a safety measure, a maximum number of empty fragments is accepted by default. If you know your application must accept a given number of empty fragments in a row, you can use this function to set the desired value.

## `gnutls_record_set_max_size`

`ssize_t gnutls_record_set_max_size (gnutls_session_t session, size_t size)` [Function]

*session*: is a `gnutls_session_t` structure.

*size*: is the new size

This function sets the maximum record packet size in this connection. This property can only be set to clients. The server may choose not to accept the requested size.

Acceptable values are 512(=2<sup>9</sup>), 1024(=2<sup>10</sup>), 2048(=2<sup>11</sup>) and 4096(=2<sup>12</sup>). The requested record size does get in effect immediately only while sending data. The receive part will take effect after a successful handshake.

This function uses a TLS extension called 'max record size'. Not all TLS implementations use or even understand this extension.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.



## gnutls\_record\_set\_timeout

**void gnutls\_record\_set\_timeout** (*gnutls\_session\_t session*, [Function]  
*unsigned int ms*)

*session*: is a **gnutls\_session\_t** structure.

*ms*: is a timeout value in milliseconds

This function sets the receive timeout for the record layer to the provided value. Use an *ms* value of zero to disable timeout (the default).

**Since:** 3.1.7

## gnutls\_record\_uncork

**int gnutls\_record\_uncork** (*gnutls\_session\_t session*, *unsigned int flags*) [Function]

*session*: is a **gnutls\_session\_t** structure.

*flags*: Could be zero or **GNUTLS\_RECORD\_WAIT**

This resets the effect of **gnutls\_record\_cork()** , and flushes any pending data. If the **GNUTLS\_RECORD\_WAIT** flag is specified then this function will block until the data is sent or a fatal error occurs (i.e., the function will retry on **GNUTLS\_E\_AGAIN** and **GNUTLS\_E\_INTERRUPTED** ).

If the flag **GNUTLS\_RECORD\_WAIT** is not specified and the function is interrupted then the **GNUTLS\_E\_AGAIN** or **GNUTLS\_E\_INTERRUPTED** errors will be returned. To obtain the data left in the corked buffer use **gnutls\_record\_check\_corked()** .

**Returns:** On success the number of transmitted data is returned, or otherwise a negative error code.

**Since:** 3.1.9

## gnutls\_rehandshake

**int gnutls\_rehandshake** (*gnutls\_session\_t session*) [Function]

*session*: is a **gnutls\_session\_t** structure.

This function will renegotiate security parameters with the client. This should only be called in case of a server.

This message informs the peer that we want to renegotiate parameters (perform a handshake).

If this function succeeds (returns 0), you must call the **gnutls\_handshake()** function in order to negotiate the new parameters.

Since TLS is full duplex some application data might have been sent during peer's processing of this message. In that case one should call **gnutls\_record\_recv()** until **GNUTLS\_E\_REHANDSHAKE** is returned to clear any pending data. Care must be taken if rehandshake is mandatory to terminate if it does not start after some threshold.

If the client does not wish to renegotiate parameters he should reply with an alert message, thus the return code will be **GNUTLS\_E\_WARNING\_ALERT\_RECEIVED** and the

alert will be `GNUTLS_A_NO_RENEGOTIATION` . A client may also choose to ignore this message.

**Returns:** `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

### **gnutls\_safe\_renegotiation\_status**

`int gnutls_safe_renegotiation_status (gnutls_session_t session)` [Function]  
*session*: is a `gnutls_session_t` structure.

Can be used to check whether safe renegotiation is being used in the current session.

**Returns:** 0 when safe renegotiation is not used and non (0) when safe renegotiation is used.

**Since:** 2.10.0

### **gnutls\_sec\_param\_get\_name**

`const char * gnutls_sec_param_get_name (gnutls_sec_param_t param)` [Function]

*param*: is a security parameter

Convert a `gnutls_sec_param_t` value to a string.

**Returns:** a pointer to a string that contains the name of the specified security level, or `NULL` .

**Since:** 2.12.0

### **gnutls\_sec\_param\_to\_pk\_bits**

`unsigned int gnutls_sec_param_to_pk_bits (gnutls_pk_algorithm_t algo, gnutls_sec_param_t param)` [Function]

*algo*: is a public key algorithm

*param*: is a security parameter

When generating private and public key pairs a difficult question is which size of "bits" the modulus will be in RSA and the group size in DSA. The easy answer is 1024, which is also wrong. This function will convert a human understandable security parameter to an appropriate size for the specific algorithm.

**Returns:** The number of bits, or (0).

**Since:** 2.12.0

### **gnutls\_sec\_param\_to\_symmetric\_bits**

`unsigned int gnutls_sec_param_to_symmetric_bits (gnutls_sec_param_t param)` [Function]

*param*: is a security parameter

This function will return the number of bits that correspond to symmetric cipher strength for the given security parameter.

**Returns:** The number of bits, or (0).

**Since:** 3.3.0

## gnutls\_server\_name\_get

**int gnutls\_server\_name\_get** (*gnutls\_session\_t session*, *void \* data*, [Function]  
*size\_t \* data\_length*, *unsigned int \* type*, *unsigned int indx*)

*session*: is a `gnutls_session_t` structure.

*data*: will hold the data

*data\_length*: will hold the data length. Must hold the maximum size of data.

*type*: will hold the server name indicator type

*indx*: is the index of the server name

This function will allow you to get the name indication (if any), a client has sent. The name indication may be any of the enumeration `gnutls_server_name_type_t`.

If *type* is `GNUTLS_NAME_DNS`, then this function is to be used by servers that support virtual hosting, and the data will be a null terminated UTF-8 string.

If *data* has not enough size to hold the server name `GNUTLS_E_SHORT_MEMORY_BUFFER` is returned, and *data\_length* will hold the required size.

*index* is used to retrieve more than one server names (if sent by the client). The first server name has an index of 0, the second 1 and so on. If no name with the given index exists `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

## gnutls\_server\_name\_set

**int gnutls\_server\_name\_set** (*gnutls\_session\_t session*, [Function]  
*gnutls\_server\_name\_type\_t type*, *const void \* name*, *size\_t name\_length*)

*session*: is a `gnutls_session_t` structure.

*type*: specifies the indicator type

*name*: is a string that contains the server name.

*name\_length*: holds the length of name

This function is to be used by clients that want to inform (via a TLS extension mechanism) the server of the name they connected to. This should be used by clients that connect to servers that do virtual hosting.

The value of *name* depends on the *type* type. In case of `GNUTLS_NAME_DNS`, an ASCII (0)-terminated domain name string, without the trailing dot, is expected. IPv4 or IPv6 addresses are not permitted.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

## gnutls\_session\_channel\_binding

**int gnutls\_session\_channel\_binding** (*gnutls\_session\_t session*, [Function]  
*gnutls\_channel\_binding\_t cbtype*, *gnutls\_datum\_t \* cb*)

*session*: is a `gnutls_session_t` structure.

*cbtype*: an `gnutls_channel_binding_t` enumeration type

*cb*: output buffer array with data

Extract given channel binding data of the *cbtype* (e.g., `GNUTLS_CB_TLS_UNIQUE`) type.

**Returns:** `GNUTLS_E_SUCCESS` on success, `GNUTLS_E_UNIMPLEMENTED_FEATURE` if the *cbtype* is unsupported, `GNUTLS_E_CHANNEL_BINDING_NOT_AVAILABLE` if the data is not currently available, or an error code.

**Since:** 2.12.0

## `gnutls_session_enable_compatibility_mode`

`void gnutls_session_enable_compatibility_mode` [Function]  
     (*gnutls\_session\_t session*)

*session*: is a `gnutls_session_t` structure.

This function can be used to disable certain (security) features in TLS in order to maintain maximum compatibility with buggy clients. Because several trade-offs with security are enabled, if required they will be reported through the audit subsystem.

Normally only servers that require maximum compatibility with everything out there, need to call this function.

Note that this function must be called after any call to `gnutls_priority` functions.

## `gnutls_session_force_valid`

`void gnutls_session_force_valid` (*gnutls\_session\_t session*) [Function]  
     *session*: is a `gnutls_session_t` structure.

Clears the invalid flag in a session. That means that sessions were corrupt or invalid data were received can be re-used. Use only when debugging or experimenting with the TLS protocol. Should not be used in typical applications.

## `gnutls_session_get_data`

`int gnutls_session_get_data` (*gnutls\_session\_t session*, *void \* session\_data*, *size\_t \* session\_data\_size*) [Function]

*session*: is a `gnutls_session_t` structure.

*session\_data*: is a pointer to space to hold the session.

*session\_data\_size*: is the *session\_data*'s size, or it will be set by the function.

Returns all session parameters needed to be stored to support resumption. The client should call this, and store the returned session data. A session may be resumed later by calling `gnutls_session_set_data()`. This function must be called after a successful (full) handshake. It should not be used in resumed sessions –see `gnutls_session_is_resumed()`.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

## gnutls\_session\_get\_data2

`int gnutls_session_get_data2 (gnutls_session_t session, gnutls_datum_t * data)` [Function]

*session*: is a `gnutls_session_t` structure.

*data*: is a pointer to a datum that will hold the session.

Returns all session parameters needed to be stored to support resumption. The client should call this, and store the returned session data. A session may be resumed later by calling `gnutls_session_set_data()`. This function must be called after a successful (full) handshake. It should not be used in resumed sessions –see `gnutls_session_is_resumed()`.

The returned `data` are allocated and must be released using `gnutls_free()`.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

## gnutls\_session\_get\_desc

`char * gnutls_session_get_desc (gnutls_session_t session)` [Function]

*session*: is a gnutls session

This function returns a string describing the current session. The string is null terminated and allocated using `gnutls_malloc()`.

**Returns:** a description of the protocols and algorithms in the current session.

**Since:** 3.1.10

## gnutls\_session\_get\_id

`int gnutls_session_get_id (gnutls_session_t session, void * session_id, size_t * session_id_size)` [Function]

*session*: is a `gnutls_session_t` structure.

*session\_id*: is a pointer to space to hold the session id.

*session\_id\_size*: initially should contain the maximum `session_id` size and will be updated.

Returns the current session ID. This can be used if you want to check if the next session you tried to resume was actually resumed. That is because resumed sessions share the same session ID with the original session.

The session ID is selected by the server, that identify the current session. In TLS 1.0 and SSL 3.0 session id is always less than 32 bytes.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

## gnutls\_session\_get\_id2

`int gnutls_session_get_id2 (gnutls_session_t session, gnutls_datum_t * session_id)` [Function]

*session*: is a `gnutls_session_t` structure.

*session\_id*: will point to the session ID.

Returns the current session ID. The returned data should be treated as constant.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise an error code is returned.

**Since:** 3.1.4

## gnutls\_session\_get\_ptr

`void * gnutls_session_get_ptr (gnutls_session_t session)` [Function]

*session*: is a `gnutls_session_t` structure.

Get user pointer for session. Useful in callbacks. This is the pointer set with `gnutls_session_set_ptr()`.

**Returns:** the user given pointer from the session structure, or NULL if it was never set.

## gnutls\_session\_get\_random

`void gnutls_session_get_random (gnutls_session_t session, gnutls_datum_t * client, gnutls_datum_t * server)` [Function]

*session*: is a `gnutls_session_t` structure.

*client*: the client part of the random

*server*: the server part of the random

This function returns pointers to the client and server random fields used in the TLS handshake. The pointers are not to be modified or deallocated.

If a client random value has not yet been established, the output will be garbage.

**Since:** 3.0

## gnutls\_session\_is\_resumed

`int gnutls_session_is_resumed (gnutls_session_t session)` [Function]

*session*: is a `gnutls_session_t` structure.

Check whether session is resumed or not.

**Returns:** non zero if this session is resumed, or a zero if this is a new session.

## gnutls\_session\_resumption\_requested

`int gnutls_session_resumption_requested (gnutls_session_t session)` [Function]

*session*: is a `gnutls_session_t` structure.

Check whether the client has asked for session resumption. This function is valid only on server side.

**Returns:** non zero if session resumption was asked, or a zero if not.

## gnutls\_session\_set\_data

```
int gnutls_session_set_data (gnutls_session_t session, const void * [Function]
                             session_data, size_t session_data_size)
```

*session*: is a `gnutls_session_t` structure.

*session\_data*: is a pointer to space to hold the session.

*session\_data\_size*: is the session's size

Sets all session parameters, in order to resume a previously established session. The session data given must be the one returned by `gnutls_session_get_data()` . This function should be called before `gnutls_handshake()` .

Keep in mind that session resuming is advisory. The server may choose not to resume the session, thus a full handshake will be performed.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

## gnutls\_session\_set\_id

```
int gnutls_session_set_id (gnutls_session_t session, const [Function]
                           gnutls_datum_t * sid)
```

*session*: is a `gnutls_session_t` structure.

*sid*: the session identifier

This function sets the session ID to be used in a client hello. This is a function intended for exceptional uses. Do not use this function unless you are implementing a custom protocol.

To set session resumption parameters use `gnutls_session_set_data()` instead.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

## gnutls\_session\_set\_premaster

```
int gnutls_session_set_premaster (gnutls_session_t session, [Function]
                                   unsigned int entity, gnutls_protocol_t version, gnutls_kx_algorithm_t kx,
                                   gnutls_cipher_algorithm_t cipher, gnutls_mac_algorithm_t mac,
                                   gnutls_compression_method_t comp, const gnutls_datum_t * master, const
                                   gnutls_datum_t * session_id)
```

*session*: is a `gnutls_session_t` structure.

*entity*: `GNUTLS_SERVER` or `GNUTLS_CLIENT`

*version*: the TLS protocol version

*kx*: the key exchange method

*cipher*: the cipher

*mac*: the MAC algorithm

*comp*: the compression method

*master*: the master key to use

*session\_id*: the session identifier

This function sets the premaster secret in a session. This is a function intended for exceptional uses. Do not use this function unless you are implementing a legacy protocol. Use `gnutls_session_set_data()` instead.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise an error code is returned.

### **gnutls\_session\_set\_ptr**

**void gnutls\_session\_set\_ptr** (*gnutls\_session\_t session*, *void \*ptr*) [Function]  
*session*: is a `gnutls_session_t` structure.

*ptr*: is the user pointer

This function will set (associate) the user given pointer `ptr` to the session structure. This pointer can be accessed with `gnutls_session_get_ptr()` .

### **gnutls\_session\_ticket\_enable\_client**

**int gnutls\_session\_ticket\_enable\_client** (*gnutls\_session\_t session*) [Function]  
*session*: is a `gnutls_session_t` structure.

Request that the client should attempt session resumption using SessionTicket.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, or an error code.

**Since:** 2.10.0

### **gnutls\_session\_ticket\_enable\_server**

**int gnutls\_session\_ticket\_enable\_server** (*gnutls\_session\_t session*, *const gnutls\_datum\_t \*key*) [Function]  
*session*: is a `gnutls_session_t` structure.

*key*: key to encrypt session parameters.

Request that the server should attempt session resumption using SessionTicket. `key` must be initialized with `gnutls_session_ticket_key_generate()` .

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, or an error code.

**Since:** 2.10.0

### **gnutls\_session\_ticket\_key\_generate**

**int gnutls\_session\_ticket\_key\_generate** (*gnutls\_datum\_t \*key*) [Function]  
*key*: is a pointer to a `gnutls_datum_t` which will contain a newly created key.

Generate a random key to encrypt security parameters within SessionTicket.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, or an error code.

**Since:** 2.10.0



## gnutls\_set\_default\_priority

**int gnutls\_set\_default\_priority** (*gnutls\_session\_t session*) [Function]

*session*: is a `gnutls_session_t` structure.

Sets the default priority on the ciphers, key exchange methods, macs and compression methods. For more fine-tuning you could use `gnutls_priority_set_direct()` or `gnutls_priority_set()` instead.

**Returns:** GNUTLS\_E\_SUCCESS on success, or an error code.

## gnutls\_sign\_algorithm\_get

**int gnutls\_sign\_algorithm\_get** (*gnutls\_session\_t session*) [Function]

*session*: is a `gnutls_session_t` structure.

Returns the signature algorithm that is (or will be) used in this session by the server to sign data.

**Returns:** The sign algorithm or GNUTLS\_SIGN\_UNKNOWN .

**Since:** 3.1.1

## gnutls\_sign\_algorithm\_get\_client

**int gnutls\_sign\_algorithm\_get\_client** (*gnutls\_session\_t session*) [Function]

*session*: is a `gnutls_session_t` structure.

Returns the signature algorithm that is (or will be) used in this session by the client to sign data.

**Returns:** The sign algorithm or GNUTLS\_SIGN\_UNKNOWN .

**Since:** 3.1.11

## gnutls\_sign\_algorithm\_get\_requested

**int gnutls\_sign\_algorithm\_get\_requested** (*gnutls\_session\_t session, size\_t indx, gnutls\_sign\_algorithm\_t \* algo*) [Function]

*session*: is a `gnutls_session_t` structure.

*indx*: is an index of the signature algorithm to return

*algo*: the returned certificate type will be stored there

Returns the signature algorithm specified by index that was requested by the peer. If the specified index has no data available this function returns GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE . If the negotiated TLS version does not support signature algorithms then GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE will be returned even for the first index. The first index is 0.

This function is useful in the certificate callback functions to assist in selecting the correct certificate.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise an error code is returned.

**Since:** 2.10.0

## gnutls\_sign\_get\_hash\_algorithm

`gnutls_digest_algorithm_t gnutls_sign_get_hash_algorithm` [Function]  
 (`gnutls_sign_algorithm_t sign`)

*sign*: is a signature algorithm

This function returns the digest algorithm corresponding to the given signature algorithms.

**Since:** 3.1.1

**Returns:** return a `gnutls_digest_algorithm_t` value, or `GNUTLS_DIG_UNKNOWN` on error.

## gnutls\_sign\_get\_id

`gnutls_sign_algorithm_t gnutls_sign_get_id` (`const char * name`) [Function]  
*name*: is a sign algorithm name

The names are compared in a case insensitive way.

**Returns:** return a `gnutls_sign_algorithm_t` value corresponding to the specified algorithm, or `GNUTLS_SIGN_UNKNOWN` on error.

## gnutls\_sign\_get\_name

`const char * gnutls_sign_get_name` (`gnutls_sign_algorithm_t algorithm`) [Function]

*algorithm*: is a sign algorithm

Convert a `gnutls_sign_algorithm_t` value to a string.

**Returns:** a string that contains the name of the specified sign algorithm, or `NULL`.

## gnutls\_sign\_get\_pk\_algorithm

`gnutls_pk_algorithm_t gnutls_sign_get_pk_algorithm` [Function]  
 (`gnutls_sign_algorithm_t sign`)

*sign*: is a signature algorithm

This function returns the public key algorithm corresponding to the given signature algorithms.

**Since:** 3.1.1

**Returns:** return a `gnutls_pk_algorithm_t` value, or `GNUTLS_PK_UNKNOWN` on error.

## gnutls\_sign\_is\_secure

`int gnutls_sign_is_secure` (`gnutls_sign_algorithm_t algorithm`) [Function]  
*algorithm*: is a sign algorithm

**Returns:** Non-zero if the provided signature algorithm is considered to be secure.

## gnutls\_sign\_list

`const gnutls_sign_algorithm_t * gnutls_sign_list ( void)` [Function]

Get a list of supported public key signature algorithms.

**Returns:** a (0)-terminated list of `gnutls_sign_algorithm_t` integers indicating the available ciphers.

## gnutls\_srp\_allocate\_client\_credentials

`int gnutls_srp_allocate_client_credentials` [Function]

(`gnutls_srp_client_credentials_t * sc`)

`sc`: is a pointer to a `gnutls_srp_server_credentials_t` structure.

This structure is complex enough to manipulate directly thus this helper function is provided in order to allocate it.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, or an error code.

## gnutls\_srp\_allocate\_server\_credentials

`int gnutls_srp_allocate_server_credentials` [Function]

(`gnutls_srp_server_credentials_t * sc`)

`sc`: is a pointer to a `gnutls_srp_server_credentials_t` structure.

This structure is complex enough to manipulate directly thus this helper function is provided in order to allocate it.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, or an error code.

## gnutls\_srp\_base64\_decode

`int gnutls_srp_base64_decode (const gnutls_datum_t * b64_data,` [Function]

`char * result, size_t * result_size)`

`b64_data`: contain the encoded data

`result`: the place where decoded data will be copied

`result_size`: holds the size of the result

This function will decode the given encoded data, using the base64 encoding found in libsrp.

Note that `b64_data` should be null terminated.

Warning! This base64 encoding is not the "standard" encoding, so do not use it for non-SRP purposes.

**Returns:** `GNUTLS_E_SHORT_MEMORY_BUFFER` if the buffer given is not long enough, or 0 on success.

## gnutls\_srp\_base64\_decode\_alloc

`int gnutls_srp_base64_decode_alloc (const gnutls_datum_t *` [Function]

`b64_data, gnutls_datum_t * result)`

`b64_data`: contains the encoded data

`result`: the place where decoded data lie

This function will decode the given encoded data. The decoded data will be allocated, and stored into result. It will decode using the base64 algorithm as used in libsrp.

You should use `gnutls_free()` to free the returned data.

Warning! This base64 encoding is not the "standard" encoding, so do not use it for non-SRP purposes.

**Returns:** 0 on success, or an error code.

## gnutls\_srp\_base64\_encode

```
int gnutls_srp_base64_encode (const gnutls_datum_t * data, char * result, size_t * result_size) [Function]
```

*data*: contain the raw data

*result*: the place where base64 data will be copied

*result\_size*: holds the size of the result

This function will convert the given data to printable data, using the base64 encoding, as used in the libsrp. This is the encoding used in SRP password files. If the provided buffer is not long enough GNUTLS\_E\_SHORT\_MEMORY\_BUFFER is returned.

Warning! This base64 encoding is not the "standard" encoding, so do not use it for non-SRP purposes.

**Returns:** GNUTLS\_E\_SHORT\_MEMORY\_BUFFER if the buffer given is not long enough, or 0 on success.

## gnutls\_srp\_base64\_encode\_alloc

```
int gnutls_srp_base64_encode_alloc (const gnutls_datum_t * data, gnutls_datum_t * result) [Function]
```

*data*: contains the raw data

*result*: will hold the newly allocated encoded data

This function will convert the given data to printable data, using the base64 encoding. This is the encoding used in SRP password files. This function will allocate the required memory to hold the encoded data.

You should use `gnutls_free()` to free the returned data.

Warning! This base64 encoding is not the "standard" encoding, so do not use it for non-SRP purposes.

**Returns:** 0 on success, or an error code.

## gnutls\_srp\_free\_client\_credentials

```
void gnutls_srp_free_client_credentials (gnutls_srp_client_credentials_t sc) [Function]
```

*sc*: is a `gnutls_srp_client_credentials_t` structure.

This structure is complex enough to manipulate directly thus this helper function is provided in order to free (deallocate) it.

## gnutls\_srp\_free\_server\_credentials

`void gnutls_srp_free_server_credentials` [Function]  
     (*gnutls\_srp\_server\_credentials\_t* *sc*)

*sc*: is a `gnutls_srp_server_credentials_t` structure.

This structure is complex enough to manipulate directly thus this helper function is provided in order to free (deallocate) it.

## gnutls\_srp\_server\_get\_username

`const char * gnutls_srp_server_get_username` (*gnutls\_session\_t* *session*) [Function]

*session*: is a gnutls session

This function will return the username of the peer. This should only be called in case of SRP authentication and in case of a server. Returns NULL in case of an error.

**Returns:** SRP username of the peer, or NULL in case of error.

## gnutls\_srp\_set\_client\_credentials

`int gnutls_srp_set_client_credentials` [Function]  
     (*gnutls\_srp\_client\_credentials\_t* *res*, *const char \* username*, *const char \* password*)

*res*: is a `gnutls_srp_client_credentials_t` structure.

*username*: is the user's userid

*password*: is the user's password

This function sets the username and password, in a `gnutls_srp_client_credentials_t` structure. Those will be used in SRP authentication. `username` and `password` should be ASCII strings or UTF-8 strings prepared using the "SASLprep" profile of "stringprep".

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, or an error code.

## gnutls\_srp\_set\_client\_credentials\_function

`void gnutls_srp_set_client_credentials_function` [Function]  
     (*gnutls\_srp\_client\_credentials\_t* *cred*, *gnutls\_srp\_client\_credentials\_function \* func*)

*cred*: is a `gnutls_srp_server_credentials_t` structure.

*func*: is the callback function

This function can be used to set a callback to retrieve the username and password for client SRP authentication. The callback's function form is:

```
int (*callback)(gnutls_session_t, char** username, char**password);
```

The `username` and `password` must be allocated using `gnutls_malloc()`. `username` and `password` should be ASCII strings or UTF-8 strings prepared using the "SASLprep" profile of "stringprep".

The callback function will be called once per handshake before the initial hello message is sent.

The callback should not return a negative error code the second time called, since the handshake procedure will be aborted.

The callback function should return 0 on success. -1 indicates an error.

## gnutls\_srp\_set\_prime\_bits

**void gnutls\_srp\_set\_prime\_bits** (*gnutls\_session\_t session*, [Function]  
*unsigned int bits*)

*session*: is a **gnutls\_session\_t** structure.

*bits*: is the number of bits

This function sets the minimum accepted number of bits, for use in an SRP key exchange. If zero, the default 2048 bits will be used.

In the client side it sets the minimum accepted number of bits. If a server sends a prime with less bits than that **GNUTLS\_E\_RECEIVED\_ILLEGAL\_PARAMETER** will be returned by the handshake.

This function has no effect in server side.

**Since:** 2.6.0

## gnutls\_srp\_set\_server\_credentials\_file

**int gnutls\_srp\_set\_server\_credentials\_file** [Function]  
(*gnutls\_srp\_server\_credentials\_t res*, *const char \*password\_file*, *const char \*password\_conf\_file*)

*res*: is a **gnutls\_srp\_server\_credentials\_t** structure.

*password\_file*: is the SRP password file (tpasswd)

*password\_conf\_file*: is the SRP password conf file (tpasswd.conf)

This function sets the password files, in a **gnutls\_srp\_server\_credentials\_t** structure. Those password files hold usernames and verifiers and will be used for SRP authentication.

**Returns:** On success, **GNUTLS\_E\_SUCCESS** (0) is returned, or an error code.

## gnutls\_srp\_set\_server\_credentials\_function

**void gnutls\_srp\_set\_server\_credentials\_function** [Function]  
(*gnutls\_srp\_server\_credentials\_t cred*, *gnutls\_srp\_server\_credentials\_function \*func*)

*cred*: is a **gnutls\_srp\_server\_credentials\_t** structure.

*func*: is the callback function

This function can be used to set a callback to retrieve the user's SRP credentials. The callback's function form is:

```
int (*callback)(gnutls_session_t, const char* username, gnutls_datum_t *salt,
gnutls_datum_t *verifier, gnutls_datum_t *generator, gnutls_datum_t *prime);
```

**username** contains the actual username. The **salt**, **verifier**, **generator** and **prime** must be filled in using the **gnutls\_malloc()**. For convenience **prime** and **generator** may also be one of the static parameters defined in **gnutls.h**.

Initially, the data field is NULL in every `gnutls_datum_t` structure that the callback has to fill in. When the callback is done GnuTLS deallocates all of those buffers which are non-NULL, regardless of the return value.

In order to prevent attackers from guessing valid usernames, if a user does not exist, `g` and `n` values should be filled in using a random user's parameters. In that case the callback must return the special value (1). See `gnutls_srp_set_server_fake_salt_seed` too. If this is not required for your application, return a negative number from the callback to abort the handshake.

The callback function will only be called once per handshake. The callback function should return 0 on success, while -1 indicates an error.

## `gnutls_srp_set_server_fake_salt_seed`

`void gnutls_srp_set_server_fake_salt_seed` [Function]  
 (`gnutls_srp_server_credentials_t cred`, `const gnutls_datum_t * seed`, `unsigned int salt_length`)

`cred`: is a `gnutls_srp_server_credentials_t` structure

`seed`: is the seed data, only needs to be valid until the function returns; size of the seed must be greater than zero

`salt_length`: is the length of the generated fake salts

This function sets the seed that is used to generate salts for invalid (non-existent) usernames.

In order to prevent attackers from guessing valid usernames, when a user does not exist gnutls generates a salt and a verifier and proceeds with the protocol as usual. The authentication will ultimately fail, but the client cannot tell whether the username is valid (exists) or invalid.

If an attacker learns the seed, given a salt (which is part of the handshake) which was generated when the seed was in use, it can tell whether or not the authentication failed because of an unknown username. This seed cannot be used to reveal application data or passwords.

`salt_length` should represent the salt length your application uses. Generating fake salts longer than 20 bytes is not supported.

By default the seed is a random value, different each time a `gnutls_srp_server_credentials_t` is allocated and fake salts are 16 bytes long.

**Since:** 3.3.0

## `gnutls_srp_verifier`

`int gnutls_srp_verifier` (`const char * username`, `const char * password`, `const gnutls_datum_t * salt`, `const gnutls_datum_t * generator`, `const gnutls_datum_t * prime`, `gnutls_datum_t * res`) [Function]

`username`: is the user's name

`password`: is the user's password

`salt`: should be some randomly generated bytes

`generator`: is the generator of the group

*prime*: is the group's prime

*res*: where the verifier will be stored.

This function will create an SRP verifier, as specified in RFC2945. The **prime** and **generator** should be one of the static parameters defined in `gnutls/gnutls.h` or may be generated.

The verifier will be allocated with `gnutls_malloc ()` and will be stored in **res** using binary format.

**Returns:** On success, `GNUTLS_E_SUCCESS (0)` is returned, or an error code.

## **gnutls\_srtplib\_get\_keys**

```
int gnutls_srtplib_get_keys (gnutls_session_t session, void * [Function]
    key_material, unsigned int key_material_size, gnutls_datum_t *
    client_key, gnutls_datum_t * client_salt, gnutls_datum_t *
    server_key, gnutls_datum_t * server_salt)
```

*session*: is a `gnutls_session_t` structure.

*key\_material*: Space to hold the generated key material

*key\_material\_size*: The maximum size of the key material

*client\_key*: The master client write key, pointing inside the key material

*client\_salt*: The master client write salt, pointing inside the key material

*server\_key*: The master server write key, pointing inside the key material

*server\_salt*: The master server write salt, pointing inside the key material

This is a helper function to generate the keying material for SRTP. It requires the space of the key material to be pre-allocated (should be at least 2x the maximum key size and salt size). The `client_key`, `client_salt`, `server_key` and `server_salt` are convenience datums that point inside the key material. They may be `NULL`.

**Returns:** On success the size of the key material is returned, otherwise, `GNUTLS_E_SHORT_MEMORY_BUFFER` if the buffer given is not sufficient, or a negative error code.

Since 3.1.4

## **gnutls\_srtplib\_get\_mki**

```
int gnutls_srtplib_get_mki (gnutls_session_t session, gnutls_datum_t * [Function]
    mki)
```

*session*: is a `gnutls_session_t` structure.

*mki*: will hold the MKI

This function exports the negotiated Master Key Identifier, received by the peer if any. The returned value in `mki` should be treated as constant and valid only during the session's lifetime.

**Returns:** On success, `GNUTLS_E_SUCCESS (0)` is returned, otherwise a negative error code is returned.

Since 3.1.4



## gnutls\_srtp\_get\_profile\_id

`int gnutls_srtp_get_profile_id (const char * name, [Function]  
                                gnutls_srtp_profile_t * profile)`

*name*: The name of the profile to look up

*profile*: Will hold the profile id

This function allows you to look up a profile based on a string.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error code is returned.

Since 3.1.4

## gnutls\_srtp\_get\_profile\_name

`const char * gnutls_srtp_get_profile_name (gnutls_srtp_profile_t [Function]  
  profile)`

*profile*: The profile to look up a string for

This function allows you to get the corresponding name for a SRTP protection profile.

**Returns:** On success, the name of a SRTP profile as a string, otherwise NULL.

Since 3.1.4

## gnutls\_srtp\_get\_selected\_profile

`int gnutls_srtp_get_selected_profile (gnutls_session_t session, [Function]  
                                      gnutls_srtp_profile_t * profile)`

*session*: is a `gnutls_session_t` structure.

*profile*: will hold the profile

This function allows you to get the negotiated SRTP profile.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error code is returned.

Since 3.1.4

## gnutls\_srtp\_set\_mki

`int gnutls_srtp_set_mki (gnutls_session_t session, const [Function]  
                          gnutls_datum_t * mki)`

*session*: is a `gnutls_session_t` structure.

*mki*: holds the MKI

This function sets the Master Key Identifier, to be used by this session (if any).

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error code is returned.

Since 3.1.4

## gnutls\_srtplib\_set\_profile

**int gnutls\_srtplib\_set\_profile** (*gnutls\_session\_t session*, [Function]  
*gnutls\_srtplib\_profile\_t profile*)

*session*: is a **gnutls\_session\_t** structure.

*profile*: is the profile id to add.

This function is to be used by both clients and servers, to declare what SRTP profiles they support, to negotiate with the peer.

**Returns:** On success, **GNUTLS\_E\_SUCCESS** (0) is returned, otherwise a negative error code is returned.

Since 3.1.4

## gnutls\_srtplib\_set\_profile\_direct

**int gnutls\_srtplib\_set\_profile\_direct** (*gnutls\_session\_t session*, [Function]  
*const char \* profiles*, *const char \*\* err\_pos*)

*session*: is a **gnutls\_session\_t** structure.

*profiles*: is a string that contains the supported SRTP profiles, separated by colons.

*err\_pos*: In case of an error this will have the position in the string the error occurred, may be NULL.

This function is to be used by both clients and servers, to declare what SRTP profiles they support, to negotiate with the peer.

**Returns:** On syntax error **GNUTLS\_E\_INVALID\_REQUEST** is returned, **GNUTLS\_E\_SUCCESS** on success, or an error code.

Since 3.1.4

## gnutls\_store\_commitment

**int gnutls\_store\_commitment** (*const char \* db\_name*, *gnutls\_tdb\_t* [Function]  
*tdb*, *const char \* host*, *const char \* service*, *gnutls\_digest\_algorithm\_t*  
*hash\_algo*, *const gnutls\_datum\_t \* hash*, *time\_t expiration*, *unsigned int*  
*flags*)

*db\_name*: A file specifying the stored keys (use NULL for the default)

*tdb*: A storage structure or NULL to use the default

*host*: The peer's name

*service*: non-NULL if this key is specific to a service (e.g. http)

*hash\_algo*: The hash algorithm type

*hash*: The raw hash

*expiration*: The expiration time (use 0 to disable expiration)

*flags*: should be 0.

This function will store the provided hash commitment to the list of stored public keys. The key with the given hash will be considered valid until the provided expiration time.

The `store` variable if non-null specifies a custom backend for the storage of entries. If it is NULL then the default file backend will be used.

Note that this function is not thread safe with the default backend.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.0

## gnutls\_store\_pubkey

```
int gnutls_store_pubkey (const char * db_name, gnutls_tdb_t tdb,          [Function]
                        const char * host, const char * service, gnutls_certificate_type_t cert_type,
                        const gnutls_datum_t * cert, time_t expiration, unsigned int flags)
```

*db\_name*: A file specifying the stored keys (use NULL for the default)

*tdb*: A storage structure or NULL to use the default

*host*: The peer's name

*service*: non-NULL if this key is specific to a service (e.g. http)

*cert\_type*: The type of the certificate

*cert*: The data of the certificate

*expiration*: The expiration time (use 0 to disable expiration)

*flags*: should be 0.

This function will store the provided (raw or DER-encoded) certificate to the list of stored public keys. The key will be considered valid until the provided expiration time.

The `store` variable if non-null specifies a custom backend for the storage of entries. If it is NULL then the default file backend will be used.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.0.13

## gnutls\_strerror

```
const char * gnutls_strerror (int error)                                [Function]
error: is a GnuTLS error code, a negative error code
```

This function is similar to `strerror`. The difference is that it accepts an error number returned by a gnutls function; In case of an unknown error a descriptive string is sent instead of NULL .

Error codes are always a negative error code.

**Returns:** A string explaining the GnuTLS error message.

## gnutls\_strerror\_name

```
const char * gnutls_strerror_name (int error)                          [Function]
error: is an error returned by a gnutls function.
```

Return the GnuTLS error code define as a string. For example, `gnutls_strerror_name(GNUTLS_E_DH_PRIME_UNACCEPTABLE)` will return the string "GNUTLS\_E\_DH\_PRIME\_UNACCEPTABLE".

**Returns:** A string corresponding to the symbol name of the error code.

**Since:** 2.6.0

## **gnutls\_supplemental\_get\_name**

`const char * gnutls_supplemental_get_name` [Function]  
     (*gnutls\_supplemental\_data\_format\_type\_t type*)  
*type*: is a supplemental data format type

Convert a `gnutls_supplemental_data_format_type_t` value to a string.

**Returns:** a string that contains the name of the specified supplemental data format type, or NULL for unknown types.

## **gnutls\_tdb\_deinit**

`void gnutls_tdb_deinit` (*gnutls\_tdb\_t tdb*) [Function]  
*tdb*: The structure to be deinitialized

This function will deinitialize a public key trust storage structure.

## **gnutls\_tdb\_init**

`int gnutls_tdb_init` (*gnutls\_tdb\_t \* tdb*) [Function]  
*tdb*: The structure to be initialized

This function will initialize a public key trust storage structure.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

## **gnutls\_tdb\_set\_store\_commitment\_func**

`void gnutls_tdb_set_store_commitment_func` (*gnutls\_tdb\_t tdb*, [Function]  
     *gnutls\_tdb\_store\_commitment\_func cstore*)  
*tdb*: The trust storage

*cstore*: The commitment storage function

This function will associate a commitment (hash) storage function with the trust storage structure. The function is of the following form.

```
int gnutls_tdb_store_commitment_func(const char* db_name, const char* host, const
char* service, time_t expiration, gnutls_digest_algorithm_t, const gnutls_datum_t*
hash);
```

The `db_name` should be used to pass any private data to this function.

**gnutls\_tdb\_set\_store\_func**

**void gnutls\_tdb\_set\_store\_func** (*gnutls\_tdb\_t tdb*, [Function]  
*gnutls\_tdb\_store\_func store*)

*tdb*: The trust storage

*store*: The storage function

This function will associate a storage function with the trust storage structure. The function is of the following form.

```
int gnutls_tdb_store_func(const char* db_name, const char* host, const char* service,
time_t expiration, const gnutls_datum_t* pubkey);
```

The *db\_name* should be used to pass any private data to this function.

**gnutls\_tdb\_set\_verify\_func**

**void gnutls\_tdb\_set\_verify\_func** (*gnutls\_tdb\_t tdb*, [Function]  
*gnutls\_tdb\_verify\_func verify*)

*tdb*: The trust storage

*verify*: The verification function

This function will associate a retrieval function with the trust storage structure. The function is of the following form.

```
int gnutls_tdb_verify_func(const char* db_name, const char* host, const char* service,
const gnutls_datum_t* pubkey);
```

The verify function should return zero on a match, GNUTLS\_E\_CERTIFICATE\_KEY\_MISMATCH if there is a mismatch and any other negative error code otherwise.

The *db\_name* should be used to pass any private data to this function.

**gnutls\_transport\_get\_int**

**int gnutls\_transport\_get\_int** (*gnutls\_session\_t session*) [Function]  
*session*: is a *gnutls\_session\_t* structure.

Used to get the first argument of the transport function (like PUSH and PULL). This must have been set using *gnutls\_transport\_set\_int()* .

**Returns:** The first argument of the transport function.

**Since:** 3.1.9

**gnutls\_transport\_get\_int2**

**void gnutls\_transport\_get\_int2** (*gnutls\_session\_t session*, *int \*recv\_int*, *int \*send\_int*) [Function]

*session*: is a *gnutls\_session\_t* structure.

*recv\_int*: will hold the value for the pull function

*send\_int*: will hold the value for the push function

Used to get the arguments of the transport functions (like PUSH and PULL). These should have been set using *gnutls\_transport\_set\_int2()* .

**Since:** 3.1.9

## gnutls\_transport\_get\_ptr

`gnutls_transport_ptr_t gnutls_transport_get_ptr` [Function]  
     (*gnutls\_session\_t session*)

*session*: is a `gnutls_session_t` structure.

Used to get the first argument of the transport function (like PUSH and PULL). This must have been set using `gnutls_transport_set_ptr()` .

**Returns:** The first argument of the transport function.

## gnutls\_transport\_get\_ptr2

`void gnutls_transport_get_ptr2` (*gnutls\_session\_t session*, [Function]  
     *gnutls\_transport\_ptr\_t \*recv\_ptr*, *gnutls\_transport\_ptr\_t \*send\_ptr*)

*session*: is a `gnutls_session_t` structure.

*recv\_ptr*: will hold the value for the pull function

*send\_ptr*: will hold the value for the push function

Used to get the arguments of the transport functions (like PUSH and PULL). These should have been set using `gnutls_transport_set_ptr2()` .

## gnutls\_transport\_set\_errno

`void gnutls_transport_set_errno` (*gnutls\_session\_t session*, *int* [Function]  
     *err*)

*session*: is a `gnutls_session_t` structure.

*err*: error value to store in session-specific errno variable.

Store **err** in the session-specific errno variable. Useful values for **err** are EINTR, EAGAIN and EMSGSIZE, other values will be treated as real errors in the push/pull function.

This function is useful in replacement push and pull functions set by `gnutls_transport_set_push_function()` and `gnutls_transport_set_pull_function()` under Windows, where the replacements may not have access to the same **errno** variable that is used by GnuTLS (e.g., the application is linked to `msvcr71.dll` and `gnutls` is linked to `msvcr71.dll`).

## gnutls\_transport\_set\_errno\_function

`void gnutls_transport_set_errno_function` (*gnutls\_session\_t* [Function]  
     *session*, *gnutls\_errno\_func* *errno\_func*)

*session*: is a `gnutls_session_t` structure.

*errno\_func*: a callback function similar to `write()`

This is the function where you set a function to retrieve errno after a failed push or pull operation.

**errno\_func** is of the form, `int (*gnutls_errno_func)(gnutls_transport_ptr_t)`; and should return the errno.

**Since:** 2.12.0

**gnutls\_transport\_set\_int**

**void gnutls\_transport\_set\_int** (*gnutls\_session\_t session*, *int i*) [Function]

*session*: is a **gnutls\_session\_t** structure.

*i*: is the value.

Used to set the first argument of the transport function (for push and pull callbacks) for berkeley style sockets.

**Since:** 3.1.9

**gnutls\_transport\_set\_int2**

**void gnutls\_transport\_set\_int2** (*gnutls\_session\_t session*, *int recv\_int*, *int send\_int*) [Function]

*session*: is a **gnutls\_session\_t** structure.

*recv\_int*: is the value for the pull function

*send\_int*: is the value for the push function

Used to set the first argument of the transport function (for push and pull callbacks), when using the berkeley style sockets. With this function you can set two different pointers for receiving and sending.

**Since:** 3.1.9

**gnutls\_transport\_set\_ptr**

**void gnutls\_transport\_set\_ptr** (*gnutls\_session\_t session*, *gnutls\_transport\_ptr\_t ptr*) [Function]

*session*: is a **gnutls\_session\_t** structure.

*ptr*: is the value.

Used to set the first argument of the transport function (for push and pull callbacks). In berkeley style sockets this function will set the connection descriptor.

**gnutls\_transport\_set\_ptr2**

**void gnutls\_transport\_set\_ptr2** (*gnutls\_session\_t session*, *gnutls\_transport\_ptr\_t recv\_ptr*, *gnutls\_transport\_ptr\_t send\_ptr*) [Function]

*session*: is a **gnutls\_session\_t** structure.

*recv\_ptr*: is the value for the pull function

*send\_ptr*: is the value for the push function

Used to set the first argument of the transport function (for push and pull callbacks). In berkeley style sockets this function will set the connection descriptor. With this function you can use two different pointers for receiving and sending.

**gnutls\_transport\_set\_pull\_function**

**void gnutls\_transport\_set\_pull\_function** (*gnutls\_session\_t session*, *gnutls\_pull\_func pull\_func*) [Function]

*session*: is a **gnutls\_session\_t** structure.

*pull\_func*: a callback function similar to `read()`

This is the function where you set a function for gnutls to receive data. Normally, if you use berkeley style sockets, do not need to use this function since the default `recv(2)` will probably be ok. The callback should return 0 on connection termination, a positive number indicating the number of bytes received, and -1 on error.

`gnutls_pull_func` is of the form, `ssize_t (*gnutls_pull_func)(gnutls_transport_ptr_t, void*, size_t)`;

## **gnutls\_transport\_set\_pull\_timeout\_function**

`void gnutls_transport_set_pull_timeout_function` [Function]  
     (*gnutls\_session\_t session, gnutls\_pull\_timeout\_func func*)

*session*: is a `gnutls_session_t` structure.

*func*: a callback function

This is the function where you set a function for gnutls to know whether data are ready to be received. It should wait for data a given time frame in milliseconds. The callback should return 0 on timeout, a positive number if data can be received, and -1 on error. You'll need to override this function if `select()` is not suitable for the provided transport calls.

As with `select()`, if the timeout value is zero the callback should return zero if no data are immediately available.

`gnutls_pull_timeout_func` is of the form, `int (*gnutls_pull_timeout_func)(gnutls_transport_ptr_t, unsigned int ms)`;

**Since:** 3.0

## **gnutls\_transport\_set\_push\_function**

`void gnutls_transport_set_push_function` (*gnutls\_session\_t* [Function]  
     *session, gnutls\_push\_func push\_func*)

*session*: is a `gnutls_session_t` structure.

*push\_func*: a callback function similar to `write()`

This is the function where you set a push function for gnutls to use in order to send data. If you are going to use berkeley style sockets, you do not need to use this function since the default `send(2)` will probably be ok. Otherwise you should specify this function for gnutls to be able to send data. The callback should return a positive number indicating the bytes sent, and -1 on error.

`push_func` is of the form, `ssize_t (*gnutls_push_func)(gnutls_transport_ptr_t, const void*, size_t)`;

## **gnutls\_transport\_set\_vec\_push\_function**

`void gnutls_transport_set_vec_push_function` (*gnutls\_session\_t* [Function]  
     *session, gnutls\_vec\_push\_func vec\_func*)

*session*: is a `gnutls_session_t` structure.

*vec\_func*: a callback function similar to `writenv()`



Using this function you can override the default `writv(2)` function for gnutls to send data. Setting this callback instead of `gnutls_transport_set_push_function()` is recommended since it introduces less overhead in the TLS handshake process.

`vec_func` is of the form, `ssize_t (*gnutls_vec_push_func) (gnutls_transport_ptr_t, const giovec_t * iov, int iovcnt);`

**Since:** 2.12.0

## gnutls\_url\_is\_supported

`int gnutls_url_is_supported (const char *url)` [Function]

*url*: A PKCS 11 url

Check whether url is supported. Depending on the system libraries GnuTLS may support pkcs11 or tpmkey URLs.

**Returns:** return non-zero if the given URL is supported, and zero if it is not known.

**Since:** 3.1.0

## gnutls\_verify\_stored\_pubkey

`int gnutls_verify_stored_pubkey (const char *db_name, [Function]  
gnutls_tdb_t tdb, const char *host, const char *service,  
gnutls_certificate_type_t cert_type, const gnutls_datum_t *cert, unsigned  
int flags)`

*db\_name*: A file specifying the stored keys (use NULL for the default)

*tdb*: A storage structure or NULL to use the default

*host*: The peer's name

*service*: non-NULL if this key is specific to a service (e.g. http)

*cert\_type*: The type of the certificate

*cert*: The raw (der) data of the certificate

*flags*: should be 0.

This function will try to verify the provided (raw or DER-encoded) certificate using a list of stored public keys. The `service` field if non-NULL should be a port number.

The `retrieve` variable if non-null specifies a custom backend for the retrieval of entries. If it is NULL then the default file backend will be used. In POSIX-like systems the file backend uses the `$HOME/.gnutls/known_hosts` file.

Note that if the custom storage backend is provided the retrieval function should return `GNUTLS_E_CERTIFICATE_KEY_MISMATCH` if the host/service pair is found but key doesn't match, `GNUTLS_E_NO_CERTIFICATE_FOUND` if no such host/service with the given key is found, and 0 if it was found. The storage function should return 0 on success.

**Returns:** If no associated public key is found then `GNUTLS_E_NO_CERTIFICATE_FOUND` will be returned. If a key is found but does not match `GNUTLS_E_CERTIFICATE_KEY_MISMATCH` is returned. On success, `GNUTLS_E_SUCCESS` (0) is returned, or a negative error value on other errors.

**Since:** 3.0.13

## E.2 Datagram TLS API

The prototypes for the following functions lie in `gnutls/dtls.h`.

### `gnutls_dtls_cookie_send`

```
int gnutls_dtls_cookie_send (gnutls_datum_t *key, void * [Function]
                             client_data, size_t client_data_size, gnutls_dtls_prestate_st *
                             prestate, gnutls_transport_ptr_t ptr, gnutls_push_func push_func)
```

*key*: is a random key to be used at cookie generation

*client\_data*: contains data identifying the client (i.e. address)

*client\_data\_size*: The size of client's data

*prestate*: The previous cookie returned by `gnutls_dtls_cookie_verify()`

*ptr*: A transport pointer to be used by *push\_func*

*push\_func*: A function that will be used to reply

This function can be used to prevent denial of service attacks to a DTLS server by requiring the client to reply using a cookie sent by this function. That way it can be ensured that a client we allocated resources for (i.e. `gnutls_session_t`) is the one that the original incoming packet was originated from.

This function must be called at the first incoming packet, prior to allocating any resources and must be succeeded by `gnutls_dtls_cookie_verify()`.

**Returns:** the number of bytes sent, or a negative error code.

**Since:** 3.0

### `gnutls_dtls_cookie_verify`

```
int gnutls_dtls_cookie_verify (gnutls_datum_t *key, void * [Function]
                               client_data, size_t client_data_size, void * _msg, size_t msg_size,
                               gnutls_dtls_prestate_st *prestate)
```

*key*: is a random key to be used at cookie generation

*client\_data*: contains data identifying the client (i.e. address)

*client\_data\_size*: The size of client's data

*\_msg*: An incoming message that initiates a connection.

*msg\_size*: The size of the message.

*prestate*: The cookie of this client.

This function will verify the received message for a valid cookie. If a valid cookie is returned then it should be associated with the session using `gnutls_dtls_prestate_set()` ;

This function must be called after `gnutls_dtls_cookie_send()`.

**Returns:** GNUTLS\_E\_SUCCESS (0) on success, or a negative error code.

**Since:** 3.0

## gnutls\_dtls\_get\_data\_mtu

`unsigned int gnutls_dtls_get_data_mtu (gnutls_session_t session)` [Function]

*session*: is a `gnutls_session_t` structure.

This function will return the actual maximum transfer unit for application data. I.e. DTLS headers are subtracted from the actual MTU which is set using `gnutls_dtls_set_mtu()`.

**Returns:** the maximum allowed transfer unit.

**Since:** 3.0

## gnutls\_dtls\_get\_mtu

`unsigned int gnutls_dtls_get_mtu (gnutls_session_t session)` [Function]

*session*: is a `gnutls_session_t` structure.

This function will return the MTU size as set with `gnutls_dtls_set_mtu()`. This is not the actual MTU of data you can transmit. Use `gnutls_dtls_get_data_mtu()` for that reason.

**Returns:** the set maximum transfer unit.

**Since:** 3.0

## gnutls\_dtls\_get\_timeout

`unsigned int gnutls_dtls_get_timeout (gnutls_session_t session)` [Function]

*session*: is a `gnutls_session_t` structure.

This function will return the milliseconds remaining for a retransmission of the previously sent handshake message. This function is useful when DTLS is used in non-blocking mode, to estimate when to call `gnutls_handshake()` if no packets have been received.

**Returns:** the remaining time in milliseconds.

**Since:** 3.0

## gnutls\_dtls\_prestate\_set

`void gnutls_dtls_prestate_set (gnutls_session_t session, gnutls_dtls_prestate_st *prestate)` [Function]

*session*: a new session

*prestate*: contains the client's prestate

This function will associate the prestate acquired by the cookie authentication with the client, with the newly established session.

This functions must be called after a successful `gnutls_dtls_cookie_verify()` and should be succeeded by the actual DTLS handshake using `gnutls_handshake()`.

**Since:** 3.0

**gnutls\_dtls\_set\_data\_mtu**

**int gnutls\_dtls\_set\_data\_mtu** (*gnutls\_session\_t session*, *unsigned int mtu*) [Function]

*session*: is a **gnutls\_session\_t** structure.

*mtu*: The maximum unencrypted transfer unit of the session

This function will set the maximum size of the \*unencrypted\* records which will be sent over a DTLS session. It is equivalent to calculating the DTLS packet overhead with the current encryption parameters, and calling **gnutls\_dtls\_set\_mtu()** with that value. In particular, this means that you may need to call this function again after any negotiation or renegotiation, in order to ensure that the MTU is still sufficient to account for the new protocol overhead.

In most cases you only need to call **gnutls\_dtls\_set\_mtu()** with the maximum MTU of your transport layer.

**Returns:** GNUTLS\_E\_SUCCESS (0) on success, or a negative error code.

**Since:** 3.1

**gnutls\_dtls\_set\_mtu**

**void gnutls\_dtls\_set\_mtu** (*gnutls\_session\_t session*, *unsigned int mtu*) [Function]

*session*: is a **gnutls\_session\_t** structure.

*mtu*: The maximum transfer unit of the transport

This function will set the maximum transfer unit of the transport that DTLS packets are sent over. Note that this should exclude the IP (or IPv6) and UDP headers. So for DTLS over IPv6 on an Ethernet device with MTU 1500, the DTLS MTU set with this function would be 1500 - 40 (IPV6 header) - 8 (UDP header) = 1452.

**Since:** 3.0

**gnutls\_dtls\_set\_timeouts**

**void gnutls\_dtls\_set\_timeouts** (*gnutls\_session\_t session*, *unsigned int retrans\_timeout*, *unsigned int total\_timeout*) [Function]

*session*: is a **gnutls\_session\_t** structure.

*retrans\_timeout*: The time at which a retransmission will occur in milliseconds

*total\_timeout*: The time at which the connection will be aborted, in milliseconds.

This function will set the timeouts required for the DTLS handshake protocol. The retransmission timeout is the time after which a message from the peer is not received, the previous messages will be retransmitted. The total timeout is the time after which the handshake will be aborted with GNUTLS\_E\_TIMEDOUT .

The DTLS protocol recommends the values of 1 sec and 60 seconds respectively.

To disable retransmissions set a **retrans\_timeout** larger than the **total\_timeout** .

**Since:** 3.0

## gnutls\_record\_get\_discarded

`unsigned int gnutls_record_get_discarded (gnutls_session_t session)` [Function]

*session*: is a `gnutls_session_t` structure.

Returns the number of discarded packets in a DTLS connection.

**Returns:** The number of discarded packets.

**Since:** 3.0

## E.3 X.509 certificate API

The following functions are to be used for X.509 certificate handling. Their prototypes lie in `gnutls/x509.h`.

### gnutls\_certificate\_set\_trust\_list

`void gnutls_certificate_set_trust_list (gnutls_certificate_credentials_t res, gnutls_x509_trust_list_t tlist, unsigned flags)` [Function]

*res*: is a `gnutls_certificate_credentials_t` structure.

*tlist*: is a `gnutls_x509_trust_list_t` structure

*flags*: must be zero

This function sets a trust list in the `gnutls_certificate_credentials_t` structure.

Note that the `tlist` will become part of the credentials structure and must not be deallocated. It will be automatically deallocated when the `res` structure is deinitialized.

**Returns:** `GNUTLS_E_SUCCESS` (0) on success, or a negative error code.

**Since:** 3.2.2

### gnutls\_pkcs7\_deinit

`void gnutls_pkcs7_deinit (gnutls_pkcs7_t pkcs7)` [Function]

*pkcs7*: The structure to be initialized

This function will deinitialize a PKCS7 structure.

### gnutls\_pkcs7\_delete\_crl

`int gnutls_pkcs7_delete_crl (gnutls_pkcs7_t pkcs7, int indx)` [Function]

*pkcs7*: should contain a `gnutls_pkcs7_t` structure

*indx*: the index of the crl to delete

This function will delete a crl from a PKCS7 or RFC2630 crl set. Index starts from 0. Returns 0 on success.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

## gnutls\_pkcs7\_delete\_cert

**int gnutls\_pkcs7\_delete\_cert** (*gnutls\_pkcs7\_t pkcs7*, *int indx*) [Function]

*pkcs7*: should contain a gnutls\_pkcs7\_t structure

*indx*: the index of the certificate to delete

This function will delete a certificate from a PKCS7 or RFC2630 certificate set. Index starts from 0. Returns 0 on success.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

## gnutls\_pkcs7\_export

**int gnutls\_pkcs7\_export** (*gnutls\_pkcs7\_t pkcs7*, [Function]

*gnutls\_x509\_cert\_fmt\_t format*, *void \* output\_data*, *size\_t \* output\_data\_size*)

*pkcs7*: Holds the pkcs7 structure

*format*: the format of output params. One of PEM or DER.

*output\_data*: will contain a structure PEM or DER encoded

*output\_data\_size*: holds the size of output\_data (and will be replaced by the actual size of parameters)

This function will export the pkcs7 structure to DER or PEM format.

If the buffer provided is not long enough to hold the output, then \* *output\_data\_size* is updated and GNUTLS\_E\_SHORT\_MEMORY\_BUFFER will be returned.

If the structure is PEM encoded, it will have a header of "BEGIN PKCS7".

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

## gnutls\_pkcs7\_export2

**int gnutls\_pkcs7\_export2** (*gnutls\_pkcs7\_t pkcs7*, [Function]

*gnutls\_x509\_cert\_fmt\_t format*, *gnutls\_datum\_t \* out*)

*pkcs7*: Holds the pkcs7 structure

*format*: the format of output params. One of PEM or DER.

*out*: will contain a structure PEM or DER encoded

This function will export the pkcs7 structure to DER or PEM format.

The output buffer is allocated using *gnutls\_malloc()* .

If the structure is PEM encoded, it will have a header of "BEGIN PKCS7".

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.1.3

## gnutls\_pkcs7\_get\_crl\_count

int gnutls\_pkcs7\_get\_crl\_count (gnutls\_pkcs7\_t pkcs7) [Function]

*pkcs7*: should contain a gnutls\_pkcs7\_t structure

This function will return the number of certificates in the PKCS7 or RFC2630 crl set.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

## gnutls\_pkcs7\_get\_crl\_raw

int gnutls\_pkcs7\_get\_crl\_raw (gnutls\_pkcs7\_t pkcs7, int indx, void \*crl, size\_t \*crl\_size) [Function]

*pkcs7*: should contain a gnutls\_pkcs7\_t structure

*indx*: contains the index of the crl to extract

*crl*: the contents of the crl will be copied there (may be null)

*crl\_size*: should hold the size of the crl

This function will return a crl of the PKCS7 or RFC2630 crl set.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value. If the provided buffer is not long enough, then *crl\_size* is updated and GNUTLS\_E\_SHORT\_MEMORY\_BUFFER is returned. After the last crl has been read GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE will be returned.

## gnutls\_pkcs7\_get\_cert\_count

int gnutls\_pkcs7\_get\_cert\_count (gnutls\_pkcs7\_t pkcs7) [Function]

*pkcs7*: should contain a gnutls\_pkcs7\_t structure

This function will return the number of certificates in the PKCS7 or RFC2630 certificate set.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

## gnutls\_pkcs7\_get\_cert\_raw

int gnutls\_pkcs7\_get\_cert\_raw (gnutls\_pkcs7\_t pkcs7, int indx, void \*certificate, size\_t \*certificate\_size) [Function]

*pkcs7*: should contain a gnutls\_pkcs7\_t structure

*indx*: contains the index of the certificate to extract

*certificate*: the contents of the certificate will be copied there (may be null)

*certificate\_size*: should hold the size of the certificate

This function will return a certificate of the PKCS7 or RFC2630 certificate set.

After the last certificate has been read GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE will be returned.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value. If the provided buffer is not long enough, then *certificate\_size* is updated and GNUTLS\_E\_SHORT\_MEMORY\_BUFFER is returned.

## gnutls\_pkcs7\_import

**int gnutls\_pkcs7\_import** (*gnutls\_pkcs7\_t pkcs7, const* [Function]  
*gnutls\_datum\_t \* data, gnutls\_x509\_crt\_fmt\_t format)*

*pkcs7*: The structure to store the parsed PKCS7.

*data*: The DER or PEM encoded PKCS7.

*format*: One of DER or PEM

This function will convert the given DER or PEM encoded PKCS7 to the native `gnutls_pkcs7_t` format. The output will be stored in `pkcs7`.

If the PKCS7 is PEM encoded it should have a header of "PKCS7".

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

## gnutls\_pkcs7\_init

**int gnutls\_pkcs7\_init** (*gnutls\_pkcs7\_t \* pkcs7*) [Function]

*pkcs7*: The structure to be initialized

This function will initialize a PKCS7 structure. PKCS7 structures usually contain lists of X.509 Certificates and X.509 Certificate revocation lists.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

## gnutls\_pkcs7\_set\_crl

**int gnutls\_pkcs7\_set\_crl** (*gnutls\_pkcs7\_t pkcs7, gnutls\_x509\_crl\_t* [Function]  
*crl*)

*pkcs7*: should contain a `gnutls_pkcs7_t` structure

*crl*: the DER encoded crl to be added

This function will add a parsed CRL to the PKCS7 or RFC2630 crl set.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

## gnutls\_pkcs7\_set\_crl\_raw

**int gnutls\_pkcs7\_set\_crl\_raw** (*gnutls\_pkcs7\_t pkcs7, const* [Function]  
*gnutls\_datum\_t \* crl*)

*pkcs7*: should contain a `gnutls_pkcs7_t` structure

*crl*: the DER encoded crl to be added

This function will add a crl to the PKCS7 or RFC2630 crl set.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.



**gnutls\_pkcs7\_set\_cert**

**int gnutls\_pkcs7\_set\_cert** (*gnutls\_pkcs7\_t pkcs7, gnutls\_x509\_cert\_t crt*) [Function]

*pkcs7*: should contain a **gnutls\_pkcs7\_t** structure

*crt*: the certificate to be copied.

This function will add a parsed certificate to the PKCS7 or RFC2630 certificate set. This is a wrapper function over **gnutls\_pkcs7\_set\_cert\_raw()** .

**Returns:** On success, **GNUTLS\_E\_SUCCESS** (0) is returned, otherwise a negative error value.

**gnutls\_pkcs7\_set\_cert\_raw**

**int gnutls\_pkcs7\_set\_cert\_raw** (*gnutls\_pkcs7\_t pkcs7, const gnutls\_datum\_t \* crt*) [Function]

*pkcs7*: should contain a **gnutls\_pkcs7\_t** structure

*crt*: the DER encoded certificate to be added

This function will add a certificate to the PKCS7 or RFC2630 certificate set.

**Returns:** On success, **GNUTLS\_E\_SUCCESS** (0) is returned, otherwise a negative error value.

**gnutls\_subject\_alt\_names\_deinit**

**void gnutls\_subject\_alt\_names\_deinit** (*gnutls\_subject\_alt\_names\_t sans*) [Function]

*sans*: The alternative names structure

This function will deinitialize an alternative names structure.

**Since:** 3.3.0

**gnutls\_subject\_alt\_names\_get**

**int gnutls\_subject\_alt\_names\_get** (*gnutls\_subject\_alt\_names\_t sans, unsigned int seq, unsigned int \* san\_type, gnutls\_datum\_t \* san, gnutls\_datum\_t \* othername\_oid*) [Function]

*sans*: The alternative names structure

*seq*: The index of the name to get

*san\_type*: Will hold the type of the name (of **gnutls\_subject\_alt\_names\_t** )

*san*: The alternative name data (should be treated as constant)

*othername\_oid*: The object identifier if *san\_type* is **GNUTLS\_SAN\_OTHERNAME** (should be treated as constant)

This function will return a specific alternative name as stored in the **sans** structure. The returned values should be treated as constant and valid for the lifetime of **sans** .

**Returns:** On success, **GNUTLS\_E\_SUCCESS** (0) is returned, **GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE** if the index is out of bounds, otherwise a negative error value.

**Since:** 3.3.0

**gnutls\_subject\_alt\_names\_init**

**int gnutls\_subject\_alt\_names\_init** (*gnutls\_subject\_alt\_names\_t* \* *sans*) [Function]

*sans*: The alternative names structure

This function will initialize an alternative names structure.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

**gnutls\_subject\_alt\_names\_set**

**int gnutls\_subject\_alt\_names\_set** (*gnutls\_subject\_alt\_names\_t* \* *sans*, unsigned int *san\_type*, const *gnutls\_datum\_t* \* *san*, const char \* *othername\_oid*) [Function]

*sans*: The alternative names structure

*san\_type*: The type of the name (of *gnutls\_subject\_alt\_names\_t* )

*san*: The alternative name data

*othername\_oid*: The object identifier if *san\_type* is GNUTLS\_SAN\_OTHERNAME

This function will store the specified alternative name in the *sans* structure.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0), otherwise a negative error value.

**Since:** 3.3.0

**gnutls\_x509\_aia\_deinit**

**void gnutls\_x509\_aia\_deinit** (*gnutls\_x509\_aia\_t* *aia*) [Function]

*aia*: The authority info access structure

This function will deinitialize a CRL distribution points structure.

**Since:** 3.3.0

**gnutls\_x509\_aia\_get**

**int gnutls\_x509\_aia\_get** (*gnutls\_x509\_aia\_t* *aia*, unsigned int *seq*, *gnutls\_datum\_t* \* *oid*, unsigned \* *san\_type*, *gnutls\_datum\_t* \* *san*) [Function]

*aia*: The authority info access structure

*seq*: specifies the sequence number of the access descriptor (0 for the first one, 1 for the second etc.)

*oid*: the type of available data; to be treated as constant.

*san\_type*: Will hold the type of the name of *gnutls\_subject\_alt\_names\_t* (may be null).

*san*: the access location name; to be treated as constant (may be null).

This function reads from the Authority Information Access structure.

The *seq* input parameter is used to indicate which member of the sequence the caller is interested in. The first member is 0, the second member 1 and so on. When the *seq* value is out of bounds, GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE is returned.

Typically `oid` is `GNUTLS_OID_AD_CAISSEURS` or `GNUTLS_OID_AD_OCSP` .

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

## **gnutls\_x509\_aia\_init**

**int gnutls\_x509\_aia\_init** (*gnutls\_x509\_aia\_t* \* *aia*) [Function]

*aia*: The authority info access structure

This function will initialize a CRL distribution points structure.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

## **gnutls\_x509\_aia\_set**

**int gnutls\_x509\_aia\_set** (*gnutls\_x509\_aia\_t* *aia*, *const char* \* *oid*, [Function]  
*unsigned san\_type*, *const gnutls\_datum\_t* \* *san*)

*aia*: The authority info access structure

*oid*: the type of data.

*san\_type*: The type of the name (of `gnutls_subject_alt_names_t` )

*san*: The alternative name data

This function will store the specified alternative name in the *aia* structure.

Typically the value for *oid* should be `GNUTLS_OID_AD_OCSP` , or `GNUTLS_OID_AD_CAISSEURS` .

**Returns:** On success, `GNUTLS_E_SUCCESS` (0), otherwise a negative error value.

**Since:** 3.3.0

## **gnutls\_x509\_aki\_deinit**

**void gnutls\_x509\_aki\_deinit** (*gnutls\_x509\_aki\_t* *aki*) [Function]

*aki*: The authority key identifier structure

This function will deinitialize an authority key identifier structure.

**Since:** 3.3.0

## **gnutls\_x509\_aki\_get\_cert\_issuer**

**int gnutls\_x509\_aki\_get\_cert\_issuer** (*gnutls\_x509\_aki\_t* *aki*, [Function]  
*unsigned int seq*, *unsigned int* \* *san\_type*, *gnutls\_datum\_t* \* *san*,  
*gnutls\_datum\_t* \* *othername\_oid*, *gnutls\_datum\_t* \* *serial*)

*aki*: The authority key ID structure

*seq*: The index of the name to get

*san\_type*: Will hold the type of the name (of `gnutls_subject_alt_names_t` )

*san*: The alternative name data

*othername\_oid*: The object identifier if *san\_type* is `GNUTLS_SAN_OTHERNAME`

*serial*: The authorityCertSerialNumber number

This function will return a specific authorityCertIssuer name as stored in the `aki` structure, as well as the authorityCertSerialNumber. All the returned values should be treated as constant, and may be set to NULL when are not required.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE if the index is out of bounds, otherwise a negative error value.

**Since:** 3.3.0

## gnutls\_x509\_aki\_get\_id

```
int gnutls_x509_aki_get_id (gnutls_x509_aki_t aki, gnutls_datum_t id) [Function]
```

*aki*: The authority key ID structure

*id*: Will hold the identifier

This function will return the key identifier as stored in the `aki` structure. The identifier should be treated as constant.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE if the index is out of bounds, otherwise a negative error value.

**Since:** 3.3.0

## gnutls\_x509\_aki\_init

```
int gnutls_x509_aki_init (gnutls_x509_aki_t * aki) [Function]
```

*aki*: The authority key ID structure

This function will initialize an authority key ID structure.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

## gnutls\_x509\_aki\_set\_cert\_issuer

```
int gnutls_x509_aki_set_cert_issuer (gnutls_x509_aki_t aki, [Function]
                                     unsigned int san_type, const gnutls_datum_t * san, const char *
                                     othername_oid, const gnutls_datum_t * serial)
```

*aki*: The authority key ID structure

*san\_type*: the type of the name (of `gnutls_subject_alt_names_t`), may be null

*san*: The alternative name data

*othername\_oid*: The object identifier if *san\_type* is GNUTLS\_SAN\_OTHERNAME

*serial*: The authorityCertSerialNumber number (may be null)

This function will set the authorityCertIssuer name and the authorityCertSerialNumber to be stored in the `aki` structure. When storing multiple names, the serial should be set on the first call, and subsequent calls should use a NULL serial.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

**gnutls\_x509\_aki\_set\_id**

**int gnutls\_x509\_aki\_set\_id** (*gnutls\_x509\_aki\_t aki, const gnutls\_datum\_t \* id*) [Function]

*aki*: The authority key ID structure

*id*: the key identifier

This function will set the keyIdentifier to be stored in the **aki** structure.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

**gnutls\_x509\_crl\_check\_issuer**

**int gnutls\_x509\_crl\_check\_issuer** (*gnutls\_x509\_crl\_t crl, gnutls\_x509\_cert\_t issuer*) [Function]

*crl*: is the CRL to be checked

*issuer*: is the certificate of a possible issuer

This function will check if the given CRL was issued by the given issuer certificate.

**Returns:** true (1) if the given CRL was issued by the given issuer, and false (0) if not.

**gnutls\_x509\_crl\_deinit**

**void gnutls\_x509\_crl\_deinit** (*gnutls\_x509\_crl\_t crl*) [Function]

*crl*: The structure to be deinitialized

This function will deinitialize a CRL structure.

**gnutls\_x509\_crl\_dist\_points\_deinit**

**void gnutls\_x509\_crl\_dist\_points\_deinit** (*gnutls\_x509\_crl\_dist\_points\_t cdp*) [Function]

*cdp*: The CRL distribution points structure

This function will deinitialize a CRL distribution points structure.

**Since:** 3.3.0

**gnutls\_x509\_crl\_dist\_points\_get**

**int gnutls\_x509\_crl\_dist\_points\_get** (*gnutls\_x509\_crl\_dist\_points\_t cdp, unsigned int seq, unsigned int \* type, gnutls\_datum\_t \* san, unsigned int \* reasons*) [Function]

*cdp*: The CRL distribution points structure

*seq*: specifies the sequence number of the distribution point (0 for the first one, 1 for the second etc.)

*type*: The name type of the corresponding name (*gnutls\_x509\_subject\_alt\_name\_t*)

*san*: The distribution point names (to be treated as constant)

*reasons*: Revocation reasons. An ORed sequence of flags from *gnutls\_x509\_crl\_reason\_flags\_t*.

This function retrieves the individual CRL distribution points (2.5.29.31), contained in provided structure.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if the index is out of bounds, otherwise a negative error value.

### **gnutls\_x509\_crl\_dist\_points\_init**

**int gnutls\_x509\_crl\_dist\_points\_init** [Function]  
     (*gnutls\_x509\_crl\_dist\_points\_t* \* *cdp*)

*cdp*: The CRL distribution points structure

This function will initialize a CRL distribution points structure.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

### **gnutls\_x509\_crl\_dist\_points\_set**

**int gnutls\_x509\_crl\_dist\_points\_set** [Function]  
     (*gnutls\_x509\_crl\_dist\_points\_t* *cdp*, *gnutls\_x509\_subject\_alt\_name\_t* *type*,  
     *const gnutls\_datum\_t* \* *san*, *unsigned int* *reasons*)

*cdp*: The CRL distribution points structure

*type*: The type of the name (of `gnutls_subject_alt_names_t`)

*san*: The point name data

*reasons*: Revocation reasons. An Ored sequence of flags from `gnutls_x509_crl_reason_flags_t`.

This function will store the specified CRL distribution point value the *cdp* structure.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0), otherwise a negative error value.

**Since:** 3.3.0

### **gnutls\_x509\_crl\_export**

**int gnutls\_x509\_crl\_export** (*gnutls\_x509\_crl\_t* *crl*, [Function]  
     *gnutls\_x509\_crt\_fmt\_t* *format*, *void \***output\_data*, *size\_t* \*  
     *output\_data\_size*)

*crl*: Holds the revocation list

*format*: the format of output params. One of PEM or DER.

*output\_data*: will contain a private key PEM or DER encoded

*output\_data\_size*: holds the size of *output\_data* (and will be replaced by the actual size of parameters)

This function will export the revocation list to DER or PEM format.

If the buffer provided is not long enough to hold the output, then `GNUTLS_E_SHORT_MEMORY_BUFFER` will be returned.

If the structure is PEM encoded, it will have a header of "BEGIN X509 CRL".

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

## gnutls\_x509\_crl\_export2

`int gnutls_x509_crl_export2 (gnutls_x509_crl_t crl, [Function]  
                                 gnutls_x509_crt_fmt_t format, gnutls_datum_t * out)`

*crl*: Holds the revocation list

*format*: the format of output params. One of PEM or DER.

*out*: will contain a private key PEM or DER encoded

This function will export the revocation list to DER or PEM format.

The output buffer is allocated using `gnutls_malloc()` .

If the structure is PEM encoded, it will have a header of "BEGIN X509 CRL".

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

Since 3.1.3

## gnutls\_x509\_crl\_get\_authority\_key\_gn\_serial

`int gnutls_x509_crl_get_authority_key_gn_serial [Function]  
                                 (gnutls_x509_crl_t crl, unsigned int seq, void * alt, size_t * alt_size,  
                                 unsigned int * alt_type, void * serial, size_t * serial_size, unsigned int  
                                 * critical)`

*crl*: should contain a `gnutls_x509_crl_t` structure

*seq*: specifies the sequence number of the alt name (0 for the first one, 1 for the second etc.)

*alt*: is the place where the alternative name will be copied to

*alt\_size*: holds the size of alt.

*alt\_type*: holds the type of the alternative name (one of `gnutls_x509_subject_alt_name_t`).

*serial*: buffer to store the serial number (may be null)

*serial\_size*: Holds the size of the serial field (may be null)

*critical*: will be non-zero if the extension is marked as critical (may be null)

This function will return the X.509 authority key identifier when stored as a general name (authorityCertIssuer) and serial number.

Because more than one general names might be stored *seq* can be used as a counter to request them all until GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE is returned.

**Returns:** Returns 0 on success, or an error code.

Since: 3.0

## gnutls\_x509\_crl\_get\_authority\_key\_id

`int gnutls_x509_crl_get_authority_key_id (gnutls_x509_crl_t [Function]  
                                 crl, void * id, size_t * id_size, unsigned int * critical)`

*crl*: should contain a `gnutls_x509_crl_t` structure

*id*: The place where the identifier will be copied

*id\_size*: Holds the size of the result field.

*critical*: will be non-zero if the extension is marked as critical (may be null)

This function will return the CRL authority's key identifier. This is obtained by the X.509 Authority Key identifier extension field (2.5.29.35). Note that this function only returns the keyIdentifier field of the extension and GNUTLS\_E\_X509\_UNSUPPORTED\_EXTENSION, if the extension contains the name and serial number of the certificate. In that case `gnutls_x509_crl_get_authority_key_gn_serial()` may be used.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error code in case of an error.

**Since:** 2.8.0

## gnutls\_x509\_crl\_get\_crt\_count

`int gnutls_x509_crl_get_crt_count (gnutls_x509_crl_t crl)` [Function]

*crl*: should contain a `gnutls_x509_crl_t` structure

This function will return the number of revoked certificates in the given CRL.

**Returns:** number of certificates, a negative error code on failure.

## gnutls\_x509\_crl\_get\_crt\_serial

`int gnutls_x509_crl_get_crt_serial (gnutls_x509_crl_t crl, int indx, unsigned char * serial, size_t * serial_size, time_t * t)` [Function]

*crl*: should contain a `gnutls_x509_crl_t` structure

*indx*: the index of the certificate to extract (starting from 0)

*serial*: where the serial number will be copied

*serial\_size*: initially holds the size of serial

*t*: if non null, will hold the time this certificate was revoked

This function will retrieve the serial number of the specified, by the index, revoked certificate.

Note that this function will have performance issues in large sequences of revoked certificates. In that case use `gnutls_x509_crl_iter_crt_serial()`.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

## gnutls\_x509\_crl\_get\_dn\_oid

`int gnutls_x509_crl_get_dn_oid (gnutls_x509_crl_t crl, int indx, void * oid, size_t * sizeof_oid)` [Function]

*crl*: should contain a `gnutls_x509_crl_t` structure

*indx*: Specifies which DN OID to send. Use (0) to get the first one.

*oid*: a pointer to a structure to hold the name (may be null)

*sizeof\_oid*: initially holds the size of 'oid'

This function will extract the requested OID of the name of the CRL issuer, specified by the given index.

If oid is null then only the size will be filled.



**Returns:** GNUTLS\_E\_SHORT\_MEMORY\_BUFFER if the provided buffer is not long enough, and in that case the `sizeof_oid` will be updated with the required size. On success 0 is returned.

### **gnutls\_x509\_crl\_get\_extension\_data**

**int gnutls\_x509\_crl\_get\_extension\_data** (*gnutls\_x509\_crl\_t* *crl*, [Function]  
*int* *indx*, *void \***data*, *size\_t \***sizeof\_data*)

*crl*: should contain a *gnutls\_x509\_crl\_t* structure

*indx*: Specifies which extension OID to send. Use (0) to get the first one.

*data*: a pointer to a structure to hold the data (may be null)

*sizeof\_data*: initially holds the size of *oid*

This function will return the requested extension data in the CRL. The extension data will be stored as a string in the provided buffer.

Use *gnutls\_x509\_crl\_get\_extension\_info*() to extract the OID and critical flag. Use *gnutls\_x509\_crl\_get\_extension\_info*() instead, if you want to get data indexed by the extension OID rather than sequence.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error code in case of an error. If you have reached the last extension available GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE will be returned.

**Since:** 2.8.0

### **gnutls\_x509\_crl\_get\_extension\_data2**

**int gnutls\_x509\_crl\_get\_extension\_data2** (*gnutls\_x509\_crl\_t* *crl*, [Function]  
*unsigned* *indx*, *gnutls\_datum\_t \***data*)

*crl*: should contain a *gnutls\_x509\_crl\_t* structure

*indx*: Specifies which extension OID to read. Use (0) to get the first one.

*data*: will contain the extension DER-encoded data

This function will return the requested by the index extension data in the certificate revocation list. The extension data will be allocated using *gnutls\_malloc*() .

Use *gnutls\_x509\_crt\_get\_extension\_info*() to extract the OID.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error code is returned. If you have reached the last extension available GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE will be returned.

### **gnutls\_x509\_crl\_get\_extension\_info**

**int gnutls\_x509\_crl\_get\_extension\_info** (*gnutls\_x509\_crl\_t* *crl*, [Function]  
*int* *indx*, *void \***oid*, *size\_t \***sizeof\_oid*, *unsigned int \***critical*)

*crl*: should contain a *gnutls\_x509\_crl\_t* structure

*indx*: Specifies which extension OID to send, use (0) to get the first one.

*oid*: a pointer to a structure to hold the OID

*sizeof\_oid*: initially holds the maximum size of *oid* , on return holds actual size of *oid* .

*critical*: output variable with critical flag, may be NULL.

This function will return the requested extension OID in the CRL, and the critical flag for it. The extension OID will be stored as a string in the provided buffer. Use `gnutls_x509_crl_get_extension_data()` to extract the data.

If the buffer provided is not long enough to hold the output, then `* sizeof_oid` is updated and `GNUTLS_E_SHORT_MEMORY_BUFFER` will be returned.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code in case of an error. If you have reached the last extension available `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

**Since:** 2.8.0

### `gnutls_x509_crl_get_extension_oid`

```
int gnutls_x509_crl_get_extension_oid (gnutls_x509_crl_t crl, int      [Function]
                                     indx, void * oid, size_t * sizeof_oid)
```

*crl*: should contain a `gnutls_x509_crl_t` structure

*indx*: Specifies which extension OID to send, use (0) to get the first one.

*oid*: a pointer to a structure to hold the OID (may be null)

*sizeof\_oid*: initially holds the size of *oid*

This function will return the requested extension OID in the CRL. The extension OID will be stored as a string in the provided buffer.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code in case of an error. If you have reached the last extension available `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

**Since:** 2.8.0

### `gnutls_x509_crl_get_issuer_dn`

```
int gnutls_x509_crl_get_issuer_dn (const gnutls_x509_crl_t crl,      [Function]
                                   char * buf, size_t * sizeof_buf)
```

*crl*: should contain a `gnutls_x509_crl_t` structure

*buf*: a pointer to a structure to hold the peer's name (may be null)

*sizeof\_buf*: initially holds the size of *buf*

This function will copy the name of the CRL issuer in the provided buffer. The name will be in the form "C=xxxx,O=yyyy,CN=zzzz" as described in RFC4514. The output string will be ASCII or UTF-8 encoded, depending on the certificate data.

If *buf* is NULL then only the size will be filled.

**Returns:** `GNUTLS_E_SHORT_MEMORY_BUFFER` if the provided buffer is not long enough, and in that case the *sizeof\_buf* will be updated with the required size, and 0 on success.

### `gnutls_x509_crl_get_issuer_dn2`

```
int gnutls_x509_crl_get_issuer_dn2 (gnutls_x509_crl_t crl,          [Function]
                                     gnutls_datum_t * dn)
```

*crl*: should contain a `gnutls_x509_crl_t` structure

*dn*: a pointer to a structure to hold the name

This function will allocate buffer and copy the name of the CRL issuer. The name will be in the form "C=xxxx,O=yyyy,CN=zzzz" as described in RFC4514. The output string will be ASCII or UTF-8 encoded, depending on the certificate data.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.1.10

## gnutls\_x509\_crl\_get\_issuer\_dn\_by\_oid

```
int gnutls_x509_crl_get_issuer_dn_by_oid (gnutls_x509_crl_t [Function]
    crl, const char * oid, int indx, unsigned int raw_flag, void * buf, size_t *
    sizeof_buf)
```

*crl*: should contain a gnutls\_x509\_crl\_t structure

*oid*: holds an Object Identified in null terminated string

*indx*: In case multiple same OIDs exist in the RDN, this specifies which to send. Use (0) to get the first one.

*raw\_flag*: If non-zero returns the raw DER data of the DN part.

*buf*: a pointer to a structure to hold the peer's name (may be null)

*sizeof\_buf*: initially holds the size of buf

This function will extract the part of the name of the CRL issuer specified by the given OID. The output will be encoded as described in RFC4514. The output string will be ASCII or UTF-8 encoded, depending on the certificate data.

Some helper macros with popular OIDs can be found in gnutls/x509.h If raw flag is (0), this function will only return known OIDs as text. Other OIDs will be DER encoded, as described in RFC4514 – in hex format with a '#' prefix. You can check about known OIDs using gnutls\_x509\_dn\_oid\_known() .

If buf is null then only the size will be filled.

**Returns:** GNUTLS\_E\_SHORT\_MEMORY\_BUFFER if the provided buffer is not long enough, and in that case the sizeof\_buf will be updated with the required size, and 0 on success.

## gnutls\_x509\_crl\_get\_next\_update

```
time_t gnutls_x509_crl_get_next_update (gnutls_x509_crl_t crl) [Function]
    crl: should contain a gnutls_x509_crl_t structure
```

This function will return the time the next CRL will be issued. This field is optional in a CRL so it might be normal to get an error instead.

**Returns:** when the next CRL will be issued, or (time\_t)-1 on error.

## gnutls\_x509\_crl\_get\_number

```
int gnutls_x509_crl_get_number (gnutls_x509_crl_t crl, void * ret, [Function]
    size_t * ret_size, unsigned int * critical)
```

*crl*: should contain a gnutls\_x509\_crl\_t structure

*ret*: The place where the number will be copied

*ret\_size*: Holds the size of the result field.

*critical*: will be non-zero if the extension is marked as critical (may be null)

This function will return the CRL number extension. This is obtained by the CRL Number extension field (2.5.29.20).

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error code in case of an error.

**Since:** 2.8.0

### **gnutls\_x509\_crl\_get\_raw\_issuer\_dn**

**int gnutls\_x509\_crl\_get\_raw\_issuer\_dn** (*gnutls\_x509\_crl\_t* *crl*, [Function]  
*gnutls\_datum\_t* \* *dn*)

*crl*: should contain a gnutls\_x509\_crl\_t structure

*dn*: will hold the starting point of the DN

This function will return a pointer to the DER encoded DN structure and the length.

**Returns:** a negative error code on error, and (0) on success.

**Since:** 2.12.0

### **gnutls\_x509\_crl\_get\_signature**

**int gnutls\_x509\_crl\_get\_signature** (*gnutls\_x509\_crl\_t* *crl*, *char* \* [Function]  
*sig*, *size\_t* \* *sizeof\_sig*)

*crl*: should contain a gnutls\_x509\_crl\_t structure

*sig*: a pointer where the signature part will be copied (may be null).

*sizeof\_sig*: initially holds the size of *sig*

This function will extract the signature field of a CRL.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

### **gnutls\_x509\_crl\_get\_signature\_algorithm**

**int gnutls\_x509\_crl\_get\_signature\_algorithm** (*gnutls\_x509\_crl\_t* [Function]  
*crl*)

*crl*: should contain a gnutls\_x509\_crl\_t structure

This function will return a value of the *gnutls\_sign\_algorithm\_t* enumeration that is the signature algorithm.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

### **gnutls\_x509\_crl\_get\_this\_update**

**time\_t gnutls\_x509\_crl\_get\_this\_update** (*gnutls\_x509\_crl\_t* *crl*) [Function]

*crl*: should contain a gnutls\_x509\_crl\_t structure

This function will return the time this CRL was issued.

**Returns:** when the CRL was issued, or (time\_t)-1 on error.

**gnutls\_x509\_crl\_get\_version**

**int gnutls\_x509\_crl\_get\_version** (*gnutls\_x509\_crl\_t* *crl*) [Function]

*crl*: should contain a *gnutls\_x509\_crl\_t* structure

This function will return the version of the specified CRL.

**Returns:** The version number, or a negative error code on error.

**gnutls\_x509\_crl\_import**

**int gnutls\_x509\_crl\_import** (*gnutls\_x509\_crl\_t* *crl*, *const* [Function]

*gnutls\_datum\_t* \* *data*, *gnutls\_x509\_crt\_fmt\_t* *format*)

*crl*: The structure to store the parsed CRL.

*data*: The DER or PEM encoded CRL.

*format*: One of DER or PEM

This function will convert the given DER or PEM encoded CRL to the native *gnutls\_x509\_crl\_t* format. The output will be stored in '*crl*'.

If the CRL is PEM encoded it should have a header of "X509 CRL".

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**gnutls\_x509\_crl\_init**

**int gnutls\_x509\_crl\_init** (*gnutls\_x509\_crl\_t* \* *crl*) [Function]

*crl*: The structure to be initialized

This function will initialize a CRL structure. CRL stands for Certificate Revocation List. A revocation list usually contains lists of certificate serial numbers that have been revoked by an Authority. The revocation lists are always signed with the authority's private key.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**gnutls\_x509\_crl\_iter\_crt\_serial**

**int gnutls\_x509\_crl\_iter\_crt\_serial** (*gnutls\_x509\_crl\_t* *crl*, [Function]

*gnutls\_x509\_crl\_iter\_t* \* *iter*, *unsigned char* \* *serial*, *size\_t* \* *serial\_size*,  
*time\_t* \* *t*)

*crl*: should contain a *gnutls\_x509\_crl\_t* structure

*iter*: A pointer to an iterator (initially the iterator should be NULL )

*serial*: where the serial number will be copied

*serial\_size*: initially holds the size of serial

*t*: if non null, will hold the time this certificate was revoked

This function performs the same as *gnutls\_x509\_crl\_get\_crt\_serial()* , but reads sequentially and keeps state in the iterator between calls. That allows it to provide better performance in sequences with many elements (50000+).

When past the last element is accessed `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned and the iterator is reset.

After use, the iterator must be deinitialized using `gnutls_x509_crl_iter_deinit()`.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

## `gnutls_x509_crl_iter_deinit`

`void gnutls_x509_crl_iter_deinit (gnutls_x509_crl_iter_t iter)` [Function]  
*iter*: The iterator structure to be deinitialized

This function will deinitialize an iterator structure.

## `gnutls_x509_crl_list_import`

`int gnutls_x509_crl_list_import (gnutls_x509_crl_t *crls, [Function]  
 unsigned int *crl_max, const gnutls_datum_t *data, gnutls_x509_crt_fmt_t  
 format, unsigned int flags)`

*crls*: The structures to store the parsed CRLs. Must not be initialized.

*crl\_max*: Initially must hold the maximum number of crls. It will be updated with the number of crls available.

*data*: The PEM encoded CRLs

*format*: One of DER or PEM.

*flags*: must be (0) or an OR'd sequence of `gnutls_certificate_import_flags`.

This function will convert the given PEM encoded CRL list to the native `gnutls_x509_crl_t` format. The output will be stored in *crls*. They will be automatically initialized.

If the Certificate is PEM encoded it should have a header of "X509 CRL".

**Returns:** the number of certificates read or a negative error value.

**Since:** 3.0

## `gnutls_x509_crl_list_import2`

`int gnutls_x509_crl_list_import2 (gnutls_x509_crl_t **crls, [Function]  
 unsigned int *size, const gnutls_datum_t *data, gnutls_x509_crt_fmt_t  
 format, unsigned int flags)`

*crls*: The structures to store the parsed crl list. Must not be initialized.

*size*: It will contain the size of the list.

*data*: The PEM encoded CRL.

*format*: One of DER or PEM.

*flags*: must be (0) or an OR'd sequence of `gnutls_certificate_import_flags`.

This function will convert the given PEM encoded CRL list to the native `gnutls_x509_crl_t` format. The output will be stored in *crls*. They will be automatically initialized.

If the Certificate is PEM encoded it should have a header of "X509 CRL".

**Returns:** the number of certificates read or a negative error value.

**Since:** 3.0

## gnutls\_x509\_crl\_print

**int gnutls\_x509\_crl\_print** (*gnutls\_x509\_crl\_t* *crl*, [Function]  
*gnutls\_certificate\_print\_formats\_t* *format*, *gnutls\_datum\_t* \* *out*)

*crl*: The structure to be printed

*format*: Indicate the format to use

*out*: Newly allocated datum with null terminated string.

This function will pretty print a X.509 certificate revocation list, suitable for display to a human.

The output *out* needs to be deallocated using **gnutls\_free()** .

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

## gnutls\_x509\_crl\_set\_authority\_key\_id

**int gnutls\_x509\_crl\_set\_authority\_key\_id** (*gnutls\_x509\_crl\_t* *crl*, *const void* \* *id*, *size\_t* *id\_size*) [Function]

*crl*: a CRL of type *gnutls\_x509\_crl\_t*

*id*: The key ID

*id\_size*: Holds the size of the serial field.

This function will set the CRL's authority key ID extension. Only the keyIdentifier field can be set with this function. This may be used by an authority that holds multiple private keys, to distinguish the used key.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 2.8.0

## gnutls\_x509\_crl\_set\_crt

**int gnutls\_x509\_crl\_set\_crt** (*gnutls\_x509\_crl\_t* *crl*, *gnutls\_x509\_crt\_t* *crt*, *time\_t* *revocation\_time*) [Function]

*crl*: should contain a *gnutls\_x509\_crl\_t* structure

*crt*: a certificate of type *gnutls\_x509\_crt\_t* with the revoked certificate

*revocation\_time*: The time this certificate was revoked

This function will set a revoked certificate's serial number to the CRL.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**gnutls\_x509\_crl\_set\_crt\_serial**

**int gnutls\_x509\_crl\_set\_crt\_serial** (*gnutls\_x509\_crl\_t crl, const void \* serial, size\_t serial\_size, time\_t revocation\_time*) [Function]

*crl*: should contain a gnutls\_x509\_crl\_t structure

*serial*: The revoked certificate's serial number

*serial\_size*: Holds the size of the serial field.

*revocation\_time*: The time this certificate was revoked

This function will set a revoked certificate's serial number to the CRL.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**gnutls\_x509\_crl\_set\_next\_update**

**int gnutls\_x509\_crl\_set\_next\_update** (*gnutls\_x509\_crl\_t crl, time\_t exp\_time*) [Function]

*crl*: should contain a gnutls\_x509\_crl\_t structure

*exp\_time*: The actual time

This function will set the time this CRL will be updated.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**gnutls\_x509\_crl\_set\_number**

**int gnutls\_x509\_crl\_set\_number** (*gnutls\_x509\_crl\_t crl, const void \* nr, size\_t nr\_size*) [Function]

*crl*: a CRL of type gnutls\_x509\_crl\_t

*nr*: The CRL number

*nr\_size*: Holds the size of the nr field.

This function will set the CRL's number extension. This is to be used as a unique and monotonic number assigned to the CRL by the authority.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 2.8.0

**gnutls\_x509\_crl\_set\_this\_update**

**int gnutls\_x509\_crl\_set\_this\_update** (*gnutls\_x509\_crl\_t crl, time\_t act\_time*) [Function]

*crl*: should contain a gnutls\_x509\_crl\_t structure

*act\_time*: The actual time

This function will set the time this CRL was issued.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.



**gnutls\_x509\_crl\_set\_version**

**int gnutls\_x509\_crl\_set\_version** (*gnutls\_x509\_crl\_t* *crl*, *unsigned int* *version*) [Function]

*crl*: should contain a *gnutls\_x509\_crl\_t* structure

*version*: holds the version number. For CRLv1 crls must be 1.

This function will set the version of the CRL. This must be one for CRL version 1, and so on. The CRLs generated by gnutls should have a version number of 2.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**gnutls\_x509\_crl\_sign2**

**int gnutls\_x509\_crl\_sign2** (*gnutls\_x509\_crl\_t* *crl*, *gnutls\_x509\_cert\_t* *issuer*, *gnutls\_x509\_privkey\_t* *issuer\_key*, *gnutls\_digest\_algorithm\_t* *dig*, *unsigned int* *flags*) [Function]

*crl*: should contain a *gnutls\_x509\_crl\_t* structure

*issuer*: is the certificate of the certificate issuer

*issuer\_key*: holds the issuer's private key

*dig*: The message digest to use. GNUTLS\_DIG\_SHA1 is the safe choice unless you know what you're doing.

*flags*: must be 0

This function will sign the CRL with the issuer's private key, and will copy the issuer's information into the CRL.

This must be the last step in a certificate CRL since all the previously set parameters are now signed.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**gnutls\_x509\_crl\_verify**

**int gnutls\_x509\_crl\_verify** (*gnutls\_x509\_crl\_t* *crl*, *const gnutls\_x509\_cert\_t \***trusted\_cas*, *int* *tcas\_size*, *unsigned int* *flags*, *unsigned int \***verify*) [Function]

*crl*: is the crl to be verified

*trusted\_cas*: is a certificate list that is considered to be trusted one

*tcas\_size*: holds the number of CA certificates in *CA\_list*

*flags*: Flags that may be used to change the verification algorithm. Use OR of the *gnutls\_certificate\_verify\_flags* enumerations.

*verify*: will hold the crl verification output.

This function will try to verify the given crl and return its verification status. See *gnutls\_x509\_cert\_list\_verify()* for a detailed description of return values. Note that since GnuTLS 3.1.4 this function includes the time checks.

Note that value in *verify* is set only when the return value of this function is success (i.e, failure to trust a CRL a certificate does not imply a negative return value).

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

### **gnutls\_x509\_crq\_deinit**

**void gnutls\_x509\_crq\_deinit** (*gnutls\_x509\_crq\_t crq*) [Function]  
*crq*: The structure to be initialized

This function will deinitialize a PKCS10 certificate request structure.

### **gnutls\_x509\_crq\_export**

**int gnutls\_x509\_crq\_export** (*gnutls\_x509\_crq\_t crq*, [Function]  
*gnutls\_x509\_crq\_fmt\_t format*, *void \*output\_data*, *size\_t \*output\_data\_size*)

*crq*: should contain a `gnutls_x509_crq_t` structure

*format*: the format of output params. One of PEM or DER.

*output\_data*: will contain a certificate request PEM or DER encoded

*output\_data\_size*: holds the size of *output\_data* (and will be replaced by the actual size of parameters)

This function will export the certificate request to a PEM or DER encoded PKCS10 structure.

If the buffer provided is not long enough to hold the output, then `GNUTLS_E_SHORT_MEMORY_BUFFER` will be returned and *\*output\_data\_size* will be updated.

If the structure is PEM encoded, it will have a header of "BEGIN NEW CERTIFICATE REQUEST".

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

### **gnutls\_x509\_crq\_export2**

**int gnutls\_x509\_crq\_export2** (*gnutls\_x509\_crq\_t crq*, [Function]  
*gnutls\_x509\_crq\_fmt\_t format*, *gnutls\_datum\_t \*out*)

*crq*: should contain a `gnutls_x509_crq_t` structure

*format*: the format of output params. One of PEM or DER.

*out*: will contain a certificate request PEM or DER encoded

This function will export the certificate request to a PEM or DER encoded PKCS10 structure.

The output buffer is allocated using `gnutls_malloc()` .

If the structure is PEM encoded, it will have a header of "BEGIN NEW CERTIFICATE REQUEST".

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Since 3.1.3

## gnutls\_x509\_crq\_get\_attribute\_by\_oid

**int gnutls\_x509\_crq\_get\_attribute\_by\_oid** (*gnutls\_x509\_crq\_t crq, const char \* oid, int indx, void \* buf, size\_t \* buf\_size*) [Function]

*crq*: should contain a `gnutls_x509_crq_t` structure

*oid*: holds an Object Identifier in null-terminated string

*indx*: In case multiple same OIDs exist in the attribute list, this specifies which to get, use (0) to get the first one

*buf*: a pointer to a structure to hold the attribute data (may be NULL )

*buf\_size*: initially holds the size of *buf*

This function will return the attribute in the certificate request specified by the given Object ID. The attribute will be DER encoded.

Attributes in a certificate request is an optional set of data appended to the request. Their interpretation depends on the CA policy.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

## gnutls\_x509\_crq\_get\_attribute\_data

**int gnutls\_x509\_crq\_get\_attribute\_data** (*gnutls\_x509\_crq\_t crq, int indx, void \* data, size\_t \* sizeof\_data*) [Function]

*crq*: should contain a `gnutls_x509_crq_t` structure

*indx*: Specifies which attribute number to get. Use (0) to get the first one.

*data*: a pointer to a structure to hold the data (may be null)

*sizeof\_data*: initially holds the size of *oid*

This function will return the requested attribute data in the certificate request. The attribute data will be stored as a string in the provided buffer.

Use `gnutls_x509_crq_get_attribute_info()` to extract the OID. Use `gnutls_x509_crq_get_attribute_by_oid()` instead, if you want to get data indexed by the attribute OID rather than sequence.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code in case of an error. If your have reached the last extension available `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

**Since:** 2.8.0

## gnutls\_x509\_crq\_get\_attribute\_info

**int gnutls\_x509\_crq\_get\_attribute\_info** (*gnutls\_x509\_crq\_t crq, int indx, void \* oid, size\_t \* sizeof\_oid*) [Function]

*crq*: should contain a `gnutls_x509_crq_t` structure

*indx*: Specifies which attribute number to get. Use (0) to get the first one.

*oid*: a pointer to a structure to hold the OID

*sizeof\_oid*: initially holds the maximum size of *oid* , on return holds actual size of *oid* .

This function will return the requested attribute OID in the certificate, and the critical flag for it. The attribute OID will be stored as a string in the provided buffer. Use `gnutls_x509_crq_get_attribute_data()` to extract the data.

If the buffer provided is not long enough to hold the output, then `* sizeof_oid` is updated and `GNUTLS_E_SHORT_MEMORY_BUFFER` will be returned.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code in case of an error. If you have reached the last extension available `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

**Since:** 2.8.0

### `gnutls_x509_crq_get_basic_constraints`

`int gnutls_x509_crq_get_basic_constraints (gnutls_x509_crq_t crq, unsigned int * critical, unsigned int * ca, int * pathlen)` [Function]

*crq*: should contain a `gnutls_x509_crq_t` structure

*critical*: will be non-zero if the extension is marked as critical

*ca*: pointer to output integer indicating CA status, may be NULL, value is 1 if the certificate CA flag is set, 0 otherwise.

*pathlen*: pointer to output integer indicating path length (may be NULL), non-negative error codes indicate a present pathLenConstraint field and the actual value, -1 indicate that the field is absent.

This function will read the certificate's basic constraints, and return the certificates CA status. It reads the basicConstraints X.509 extension (2.5.29.19).

**Returns:** If the certificate is a CA a positive value will be returned, or (0) if the certificate does not have CA flag set. A negative error code may be returned in case of errors. If the certificate does not contain the basicConstraints extension `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

**Since:** 2.8.0

### `gnutls_x509_crq_get_challenge_password`

`int gnutls_x509_crq_get_challenge_password (gnutls_x509_crq_t crq, char * pass, size_t * pass_size)` [Function]

*crq*: should contain a `gnutls_x509_crq_t` structure

*pass*: will hold a (0)-terminated password string

*pass\_size*: Initially holds the size of *pass*.

This function will return the challenge password in the request. The challenge password is intended to be used for requesting a revocation of the certificate.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

### `gnutls_x509_crq_get_dn`

`int gnutls_x509_crq_get_dn (gnutls_x509_crq_t crq, char * buf, size_t * buf_size)` [Function]

*crq*: should contain a `gnutls_x509_crq_t` structure

*buf*: a pointer to a structure to hold the name (may be NULL )

*buf\_size*: initially holds the size of *buf*

This function will copy the name of the Certificate request subject to the provided buffer. The name will be in the form "C=xxxx,O=yyyy,CN=zzzz" as described in RFC 2253. The output string *buf* will be ASCII or UTF-8 encoded, depending on the certificate data.

**Returns:** GNUTLS\_E\_SHORT\_MEMORY\_BUFFER if the provided buffer is not long enough, and in that case the \* *buf\_size* will be updated with the required size. On success 0 is returned.

## gnutls\_x509\_crq\_get\_dn2

`int gnutls_x509_crq_get_dn2 (gnutls_x509_crq_t crq, gnutls_datum_t [Function]  
* dn)`

*crq*: should contain a `gnutls_x509_crq_t` structure

*dn*: a pointer to a structure to hold the name

This function will allocate buffer and copy the name of the Certificate request. The name will be in the form "C=xxxx,O=yyyy,CN=zzzz" as described in RFC4514. The output string will be ASCII or UTF-8 encoded, depending on the certificate data.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value. and a negative error code on error.

**Since:** 3.1.10

## gnutls\_x509\_crq\_get\_dn\_by\_oid

`int gnutls_x509_crq_get_dn_by_oid (gnutls_x509_crq_t crq, const [Function]  
char * oid, int indx, unsigned int raw_flag, void * buf, size_t * buf_size)`

*crq*: should contain a `gnutls_x509_crq_t` structure

*oid*: holds an Object Identifier in a null terminated string

*indx*: In case multiple same OIDs exist in the RDN, this specifies which to get. Use (0) to get the first one.

*raw\_flag*: If non-zero returns the raw DER data of the DN part.

*buf*: a pointer to a structure to hold the name (may be NULL )

*buf\_size*: initially holds the size of *buf*

This function will extract the part of the name of the Certificate request subject, specified by the given OID. The output will be encoded as described in RFC2253. The output string will be ASCII or UTF-8 encoded, depending on the certificate data.

Some helper macros with popular OIDs can be found in `gnutls/x509.h`. If raw flag is (0), this function will only return known OIDs as text. Other OIDs will be DER encoded, as described in RFC2253 – in hex format with a '\#' prefix. You can check about known OIDs using `gnutls_x509_dn_oid_known()` .

**Returns:** GNUTLS\_E\_SHORT\_MEMORY\_BUFFER if the provided buffer is not long enough, and in that case the \* *buf\_size* will be updated with the required size. On success 0 is returned.

**gnutls\_x509\_crq\_get\_dn\_oid**

**int gnutls\_x509\_crq\_get\_dn\_oid** (*gnutls\_x509\_crq\_t crq*, *int indx*, [Function]  
*void \* oid*, *size\_t \* sizeof\_oid*)

*crq*: should contain a *gnutls\_x509\_crq\_t* structure

*indx*: Specifies which DN OID to get. Use (0) to get the first one.

*oid*: a pointer to a structure to hold the name (may be NULL )

*sizeof\_oid*: initially holds the size of *oid*

This function will extract the requested OID of the name of the certificate request subject, specified by the given index.

**Returns:** GNUTLS\_E\_SHORT\_MEMORY\_BUFFER if the provided buffer is not long enough, and in that case the *\* sizeof\_oid* will be updated with the required size. On success 0 is returned.

**gnutls\_x509\_crq\_get\_extension\_by\_oid**

**int gnutls\_x509\_crq\_get\_extension\_by\_oid** (*gnutls\_x509\_crq\_t* [Function]  
*crq*, *const char \* oid*, *int indx*, *void \* buf*, *size\_t \* buf\_size*, *unsigned int \* critical*)

*crq*: should contain a *gnutls\_x509\_crq\_t* structure

*oid*: holds an Object Identifier in a null terminated string

*indx*: In case multiple same OIDs exist in the extensions, this specifies which to get. Use (0) to get the first one.

*buf*: a pointer to a structure to hold the name (may be null)

*buf\_size*: initially holds the size of *buf*

*critical*: will be non-zero if the extension is marked as critical

This function will return the extension specified by the OID in the certificate. The extensions will be returned as binary data DER encoded, in the provided buffer.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error code in case of an error. If the certificate does not contain the specified extension GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE will be returned.

**Since:** 2.8.0

**gnutls\_x509\_crq\_get\_extension\_by\_oid2**

**int gnutls\_x509\_crq\_get\_extension\_by\_oid2** (*gnutls\_x509\_crq\_t* [Function]  
*crq*, *const char \* oid*, *int indx*, *gnutls\_datum\_t \* output*, *unsigned int \* critical*)

*crq*: should contain a *gnutls\_x509\_crq\_t* structure

*oid*: holds an Object Identifier in a null terminated string

*indx*: In case multiple same OIDs exist in the extensions, this specifies which to get. Use (0) to get the first one.

*output*: will hold the allocated extension data

*critical*: will be non-zero if the extension is marked as critical

This function will return the extension specified by the OID in the certificate. The extensions will be returned as binary data DER encoded, in the provided buffer.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error code in case of an error. If the certificate does not contain the specified extension GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE will be returned.

**Since:** 3.3.8

### gnutls\_x509\_crq\_get\_extension\_data

`int gnutls_x509_crq_get_extension_data (gnutls_x509_crq_t crq, [Function]  
int indx, void * data, size_t * sizeof_data)`

*crq*: should contain a `gnutls_x509_crq_t` structure

*indx*: Specifies which extension number to get. Use (0) to get the first one.

*data*: a pointer to a structure to hold the data (may be null)

*sizeof\_data*: initially holds the size of oid

This function will return the requested extension data in the certificate. The extension data will be stored as a string in the provided buffer.

Use `gnutls_x509_crq_get_extension_info()` to extract the OID and critical flag.

Use `gnutls_x509_crq_get_extension_by_oid()` instead, if you want to get data indexed by the extension OID rather than sequence.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error code in case of an error. If you have reached the last extension available GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE will be returned.

**Since:** 2.8.0

### gnutls\_x509\_crq\_get\_extension\_data2

`int gnutls_x509_crq_get_extension_data2 (gnutls_x509_crq_t crq, [Function]  
unsigned indx, gnutls_datum_t * data)`

*crq*: should contain a `gnutls_x509_crq_t` structure

*indx*: Specifies which extension OID to read. Use (0) to get the first one.

*data*: will contain the extension DER-encoded data

This function will return the requested extension data in the certificate request. The extension data will be allocated using `gnutls_malloc()`.

Use `gnutls_x509_crq_get_extension_info()` to extract the OID.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error code is returned. If you have reached the last extension available GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE will be returned.

**Since:** 3.3.0

### gnutls\_x509\_crq\_get\_extension\_info

`int gnutls_x509_crq_get_extension_info (gnutls_x509_crq_t crq, [Function]  
int indx, void * oid, size_t * sizeof_oid, unsigned int * critical)`

*crq*: should contain a `gnutls_x509_crq_t` structure

*indx*: Specifies which extension number to get. Use (0) to get the first one.

*oid*: a pointer to a structure to hold the OID

*sizeof\_oid*: initially holds the maximum size of *oid* , on return holds actual size of *oid* .

*critical*: output variable with critical flag, may be NULL.

This function will return the requested extension OID in the certificate, and the critical flag for it. The extension OID will be stored as a string in the provided buffer. Use `gnutls_x509_crq_get_extension_data()` to extract the data.

If the buffer provided is not long enough to hold the output, then \* *sizeof\_oid* is updated and `GNUTLS_E_SHORT_MEMORY_BUFFER` will be returned.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code in case of an error. If your have reached the last extension available `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

**Since:** 2.8.0

## `gnutls_x509_crq_get_key_id`

```
int gnutls_x509_crq_get_key_id (gnutls_x509_crq_t crq, unsigned [Function]
                               int flags, unsigned char * output_data, size_t * output_data_size)
```

*crq*: a certificate of type `gnutls_x509_crq_t`

*flags*: should be 0 for now

*output\_data*: will contain the key ID

*output\_data\_size*: holds the size of *output\_data* (and will be replaced by the actual size of parameters)

This function will return a unique ID that depends on the public key parameters. This ID can be used in checking whether a certificate corresponds to the given private key.

If the buffer provided is not long enough to hold the output, then \* *output\_data\_size* is updated and `GNUTLS_E_SHORT_MEMORY_BUFFER` will be returned. The output will normally be a SHA-1 hash output, which is 20 bytes.

**Returns:** In case of failure a negative error code will be returned, and 0 on success.

**Since:** 2.8.0

## `gnutls_x509_crq_get_key_purpose_oid`

```
int gnutls_x509_crq_get_key_purpose_oid (gnutls_x509_crq_t crq, [Function]
                                       int indx, void * oid, size_t * sizeof_oid, unsigned int * critical)
```

*crq*: should contain a `gnutls_x509_crq_t` structure

*indx*: This specifies which OID to return, use (0) to get the first one

*oid*: a pointer to a buffer to hold the OID (may be NULL )

*sizeof\_oid*: initially holds the size of *oid*

*critical*: output variable with critical flag, may be NULL .

This function will extract the key purpose OIDs of the Certificate specified by the given index. These are stored in the Extended Key Usage extension (2.5.29.37). See the `GNUTLS_KP_*` definitions for human readable names.



**Returns:** GNUTLS\_E\_SHORT\_MEMORY\_BUFFER if the provided buffer is not long enough, and in that case the \* `sizeof_oid` will be updated with the required size. On success 0 is returned.

**Since:** 2.8.0

### **gnutls\_x509\_crq\_get\_key\_rsa\_raw**

**int gnutls\_x509\_crq\_get\_key\_rsa\_raw** (*gnutls\_x509\_crq\_t crq,* [Function]  
*gnutls\_datum\_t \* m, gnutls\_datum\_t \* e)*

*crq*: Holds the certificate

*m*: will hold the modulus

*e*: will hold the public exponent

This function will export the RSA public key's parameters found in the given structure. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 2.8.0

### **gnutls\_x509\_crq\_get\_key\_usage**

**int gnutls\_x509\_crq\_get\_key\_usage** (*gnutls\_x509\_crq\_t crq,* [Function]  
*unsigned int \* key\_usage, unsigned int \* critical)*

*crq*: should contain a `gnutls_x509_crq_t` structure

*key\_usage*: where the key usage bits will be stored

*critical*: will be non-zero if the extension is marked as critical

This function will return certificate's key usage, by reading the keyUsage X.509 extension (2.5.29.15). The key usage value will ORed values of the: GNUTLS\_KEY\_DIGITAL\_SIGNATURE , GNUTLS\_KEY\_NON\_REPUDIATION , GNUTLS\_KEY\_KEY\_ENCIPHERMENT , GNUTLS\_KEY\_DATA\_ENCIPHERMENT , GNUTLS\_KEY\_KEY\_AGREEMENT , GNUTLS\_KEY\_KEY\_CERT\_SIGN , GNUTLS\_KEY\_CRL\_SIGN , GNUTLS\_KEY\_ENCIPHER\_ONLY , GNUTLS\_KEY\_DECIPHER\_ONLY .

**Returns:** the certificate key usage, or a negative error code in case of parsing error. If the certificate does not contain the keyUsage extension GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE will be returned.

**Since:** 2.8.0

### **gnutls\_x509\_crq\_get\_pk\_algorithm**

**int gnutls\_x509\_crq\_get\_pk\_algorithm** (*gnutls\_x509\_crq\_t crq,* [Function]  
*unsigned int \* bits)*

*crq*: should contain a `gnutls_x509_crq_t` structure

*bits*: if bits is non-NULL it will hold the size of the parameters' in bits

This function will return the public key algorithm of a PKCS10 certificate request.

If `bits` is non-NULL, it should have enough size to hold the parameters size in bits. For RSA the bits returned is the modulus. For DSA the bits returned are of the public exponent.

**Returns:** a member of the `gnutls_pk_algorithm_t` enumeration on success, or a negative error code on error.

### **gnutls\_x509\_crq\_get\_private\_key\_usage\_period**

```
int gnutls_x509_crq_get_private_key_usage_period           [Function]
    (gnutls_x509_crq_t crq, time_t * activation, time_t * expiration,
     unsigned int * critical)
```

`crq`: should contain a `gnutls_x509_crq_t` structure

`activation`: The activation time

`expiration`: The expiration time

`critical`: the extension status

This function will return the expiration and activation times of the private key of the certificate.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if the extension is not present, otherwise a negative error value.

### **gnutls\_x509\_crq\_get\_subject\_alt\_name**

```
int gnutls_x509_crq_get_subject_alt_name (gnutls_x509_crq_t      [Function]
    crq, unsigned int seq, void * ret, size_t * ret_size, unsigned int *
    ret_type, unsigned int * critical)
```

`crq`: should contain a `gnutls_x509_crq_t` structure

`seq`: specifies the sequence number of the alt name, 0 for the first one, 1 for the second etc.

`ret`: is the place where the alternative name will be copied to

`ret_size`: holds the size of `ret`.

`ret_type`: holds the `gnutls_x509_subject_alt_name_t` name type

`critical`: will be non-zero if the extension is marked as critical (may be null)

This function will return the alternative names, contained in the given certificate. It is the same as `gnutls_x509_crq_get_subject_alt_name()` except for the fact that it will return the type of the alternative name in `ret_type` even if the function fails for some reason (i.e. the buffer provided is not enough).

**Returns:** the alternative subject name type on success, one of the enumerated `gnutls_x509_subject_alt_name_t`. It will return `GNUTLS_E_SHORT_MEMORY_BUFFER` if `ret_size` is not large enough to hold the value. In that case `ret_size` will be updated with the required size. If the certificate request does not have an Alternative name with the specified sequence number then `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned.

**Since:** 2.8.0

**gnutls\_x509\_crq\_get\_subject\_alt\_othername\_oid**

**int gnutls\_x509\_crq\_get\_subject\_alt\_othername\_oid** [Function]  
 (*gnutls\_x509\_crq\_t crq*, unsigned int *seq*, void \* *ret*, size\_t \* *ret\_size*)

*crq*: should contain a *gnutls\_x509\_crq\_t* structure

*seq*: specifies the sequence number of the alt name (0 for the first one, 1 for the second etc.)

*ret*: is the place where the otherName OID will be copied to

*ret\_size*: holds the size of *ret*.

This function will extract the type OID of an otherName Subject Alternative Name, contained in the given certificate, and return the type as an enumerated element.

This function is only useful if *gnutls\_x509\_crq\_get\_subject\_alt\_name()* returned *GNUTLS\_SAN\_OTHERNAME* .

**Returns:** the alternative subject name type on success, one of the enumerated *gnutls\_x509\_subject\_alt\_name\_t*. For supported OIDs, it will return one of the virtual (*GNUTLS\_SAN\_OTHERNAME\_\**) types, e.g. *GNUTLS\_SAN\_OTHERNAME\_XMPP* , and *GNUTLS\_SAN\_OTHERNAME* for unknown OIDs. It will return *GNUTLS\_E\_SHORT\_MEMORY\_BUFFER* if *ret\_size* is not large enough to hold the value. In that case *ret\_size* will be updated with the required size. If the certificate does not have an Alternative name with the specified sequence number and with the otherName type then *GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE* is returned.

**Since:** 2.8.0

**gnutls\_x509\_crq\_get\_version**

**int gnutls\_x509\_crq\_get\_version** (*gnutls\_x509\_crq\_t crq*) [Function]

*crq*: should contain a *gnutls\_x509\_crq\_t* structure

This function will return the version of the specified Certificate request.

**Returns:** version of certificate request, or a negative error code on error.

**gnutls\_x509\_crq\_import**

**int gnutls\_x509\_crq\_import** (*gnutls\_x509\_crq\_t crq*, const *gnutls\_datum\_t \* data*, *gnutls\_x509\_crt\_fmt\_t format*) [Function]

*crq*: The structure to store the parsed certificate request.

*data*: The DER or PEM encoded certificate.

*format*: One of DER or PEM

This function will convert the given DER or PEM encoded certificate request to a *gnutls\_x509\_crq\_t* structure. The output will be stored in *crq* .

If the Certificate is PEM encoded it should have a header of "NEW CERTIFICATE REQUEST".

**Returns:** On success, *GNUTLS\_E\_SUCCESS* (0) is returned, otherwise a negative error value.

**gnutls\_x509\_crq\_init**

**int gnutls\_x509\_crq\_init** (*gnutls\_x509\_crq\_t \* crq*) [Function]

*crq*: The structure to be initialized

This function will initialize a PKCS10 certificate request structure.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**gnutls\_x509\_crq\_print**

**int gnutls\_x509\_crq\_print** (*gnutls\_x509\_crq\_t crq*,  
*gnutls\_certificate\_print\_formats\_t format*, *gnutls\_datum\_t \* out*) [Function]

*crq*: The structure to be printed

*format*: Indicate the format to use

*out*: Newly allocated datum with null terminated string.

This function will pretty print a certificate request, suitable for display to a human.

The output *out* needs to be deallocated using `gnutls_free()` .

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 2.8.0

**gnutls\_x509\_crq\_set\_attribute\_by\_oid**

**int gnutls\_x509\_crq\_set\_attribute\_by\_oid** (*gnutls\_x509\_crq\_t crq*, *const char \* oid*, *void \* buf*, *size\_t buf\_size*) [Function]

*crq*: should contain a `gnutls_x509_crq_t` structure

*oid*: holds an Object Identifier in a null-terminated string

*buf*: a pointer to a structure that holds the attribute data

*buf\_size*: holds the size of *buf*

This function will set the attribute in the certificate request specified by the given Object ID. The provided attribute must be DER encoded.

Attributes in a certificate request is an optional set of data appended to the request. Their interpretation depends on the CA policy.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**gnutls\_x509\_crq\_set\_basic\_constraints**

**int gnutls\_x509\_crq\_set\_basic\_constraints** (*gnutls\_x509\_crq\_t crq*, *unsigned int ca*, *int pathLenConstraint*) [Function]

*crq*: a certificate request of type `gnutls_x509_crq_t`

*ca*: true(1) or false(0) depending on the Certificate authority status.

*pathLenConstraint*: non-negative error codes indicate maximum length of path, and negative error codes indicate that the `pathLenConstraints` field should not be present.

This function will set the `basicConstraints` certificate extension.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 2.8.0

## gnutls\_x509\_crq\_set\_challenge\_password

**int gnutls\_x509\_crq\_set\_challenge\_password** (*gnutls\_x509\_crq\_t crq, const char \* pass*) [Function]

*crq*: should contain a *gnutls\_x509\_crq\_t* structure

*pass*: holds a (0)-terminated password

This function will set a challenge password to be used when revoking the request.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

## gnutls\_x509\_crq\_set\_dn

**int gnutls\_x509\_crq\_set\_dn** (*gnutls\_x509\_crq\_t crq, const char \* dn, const char \*\* err*) [Function]

*crq*: a certificate of type *gnutls\_x509\_crq\_t*

*dn*: a comma separated DN string (RFC4514)

*err*: indicates the error position (if any)

This function will set the DN on the provided certificate. The input string should be plain ASCII or UTF-8 encoded.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

## gnutls\_x509\_crq\_set\_dn\_by\_oid

**int gnutls\_x509\_crq\_set\_dn\_by\_oid** (*gnutls\_x509\_crq\_t crq, const char \* oid, unsigned int raw\_flag, const void \* data, unsigned int sizeof\_data*) [Function]

*crq*: should contain a *gnutls\_x509\_crq\_t* structure

*oid*: holds an Object Identifier in a (0)-terminated string

*raw\_flag*: must be 0, or 1 if the data are DER encoded

*data*: a pointer to the input data

*sizeof\_data*: holds the size of *data*

This function will set the part of the name of the Certificate request subject, specified by the given OID. The input string should be ASCII or UTF-8 encoded.

Some helper macros with popular OIDs can be found in *gnutls/x509.h*. With this function you can only set the known OIDs. You can test for known OIDs using *gnutls\_x509\_dn\_oid\_known()*. For OIDs that are not known (by gnutls) you should properly DER encode your data, and call this function with *raw\_flag* set.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**gnutls\_x509\_crq\_set\_key**

**int gnutls\_x509\_crq\_set\_key** (*gnutls\_x509\_crq\_t crq,* [Function]  
*gnutls\_x509\_privkey\_t key*)

*crq*: should contain a **gnutls\_x509\_crq\_t** structure

*key*: holds a private key

This function will set the public parameters from the given private key to the request.

**Returns:** On success, **GNUTLS\_E\_SUCCESS** (0) is returned, otherwise a negative error value.

**gnutls\_x509\_crq\_set\_key\_purpose\_oid**

**int gnutls\_x509\_crq\_set\_key\_purpose\_oid** (*gnutls\_x509\_crq\_t crq,* [Function]  
*const void \*oid, unsigned int critical*)

*crq*: a certificate of type **gnutls\_x509\_crq\_t**

*oid*: a pointer to a (0)-terminated string that holds the OID

*critical*: Whether this extension will be critical or not

This function will set the key purpose OIDs of the Certificate. These are stored in the Extended Key Usage extension (2.5.29.37) See the **GNUTLS\_KP\_\*** definitions for human readable names.

Subsequent calls to this function will append OIDs to the OID list.

**Returns:** On success, **GNUTLS\_E\_SUCCESS** (0) is returned, otherwise a negative error value.

**Since:** 2.8.0

**gnutls\_x509\_crq\_set\_key\_rsa\_raw**

**int gnutls\_x509\_crq\_set\_key\_rsa\_raw** (*gnutls\_x509\_crq\_t crq,* [Function]  
*const gnutls\_datum\_t \*m, const gnutls\_datum\_t \*e*)

*crq*: should contain a **gnutls\_x509\_crq\_t** structure

*m*: holds the modulus

*e*: holds the public exponent

This function will set the public parameters from the given private key to the request. Only RSA keys are currently supported.

**Returns:** On success, **GNUTLS\_E\_SUCCESS** (0) is returned, otherwise a negative error value.

**Since:** 2.6.0

**gnutls\_x509\_crq\_set\_key\_usage**

**int gnutls\_x509\_crq\_set\_key\_usage** (*gnutls\_x509\_crq\_t crq,* [Function]  
*unsigned int usage*)

*crq*: a certificate request of type **gnutls\_x509\_crq\_t**

*usage*: an ORed sequence of the **GNUTLS\_KEY\_\*** elements.

This function will set the keyUsage certificate extension.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 2.8.0

### gnutls\_x509\_crq\_set\_private\_key\_usage\_period

**int gnutls\_x509\_crq\_set\_private\_key\_usage\_period** [Function]  
     (*gnutls\_x509\_crq\_t crq, time\_t activation, time\_t expiration*)

*crq*: a certificate of type `gnutls_x509_crq_t`

*activation*: The activation time

*expiration*: The expiration time

This function will set the private key usage period extension (2.5.29.16).

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

### gnutls\_x509\_crq\_set\_subject\_alt\_name

**int gnutls\_x509\_crq\_set\_subject\_alt\_name** (*gnutls\_x509\_crq\_t crq, gnutls\_x509\_subject\_alt\_name\_t nt, const void \* data, unsigned int data\_size, unsigned int flags*) [Function]

*crq*: a certificate request of type `gnutls_x509_crq_t`

*nt*: is one of the `gnutls_x509_subject_alt_name_t` enumerations

*data*: The data to be set

*data\_size*: The size of data to be set

*flags*: GNUTLS\_FSAN\_SET to clear previous data or GNUTLS\_FSAN\_APPEND to append.

This function will set the subject alternative name certificate extension. It can set the following types:

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 2.8.0

### gnutls\_x509\_crq\_set\_version

**int gnutls\_x509\_crq\_set\_version** (*gnutls\_x509\_crq\_t crq, unsigned int version*) [Function]

*crq*: should contain a `gnutls_x509_crq_t` structure

*version*: holds the version number, for v1 Requests must be 1

This function will set the version of the certificate request. For version 1 requests this must be one.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

## gnutls\_x509\_crq\_sign2

**int gnutls\_x509\_crq\_sign2** (*gnutls\_x509\_crq\_t crq*, [Function]  
*gnutls\_x509\_privkey\_t key*, *gnutls\_digest\_algorithm\_t dig*, unsigned int *flags*)  
*crq*: should contain a *gnutls\_x509\_crq\_t* structure

*key*: holds a private key

*dig*: The message digest to use, i.e., GNUTLS\_DIG\_SHA1

*flags*: must be 0

This function will sign the certificate request with a private key. This must be the same key as the one used in *gnutls\_x509 crt\_set\_key()* since a certificate request is self signed.

This must be the last step in a certificate request generation since all the previously set parameters are now signed.

**Returns:** GNUTLS\_E\_SUCCESS on success, otherwise a negative error code. GNUTLS\_E\_ASN1\_VALUE\_NOT\_FOUND is returned if you didn't set all information in the certificate request (e.g., the version using *gnutls\_x509\_crq\_set\_version()* ).

## gnutls\_x509\_crq\_verify

**int gnutls\_x509\_crq\_verify** (*gnutls\_x509\_crq\_t crq*, unsigned int [Function]  
*flags*)

*crq*: is the crq to be verified

*flags*: Flags that may be used to change the verification algorithm. Use OR of the *gnutls\_certificate\_verify\_flags* enumerations.

This function will verify self signature in the certificate request and return its status.

**Returns:** In case of a verification failure GNUTLS\_E\_PK\_SIG\_VERIFY\_FAILED is returned, and zero or positive code on success.

Since 2.12.0

## gnutls\_x509 crt\_check\_hostname

**int gnutls\_x509 crt\_check\_hostname** (*gnutls\_x509 crt\_t cert*, const [Function]  
*char \* hostname*)

*cert*: should contain an *gnutls\_x509 crt\_t* structure

*hostname*: A null terminated string that contains a DNS name

This function will check if the given certificate's subject matches the given hostname. This is a basic implementation of the matching described in RFC6125, and takes into account wildcards, and the DNSName/IPAddress subject alternative name PKIX extension.

For details see also *gnutls\_x509 crt\_check\_hostname2()* .

**Returns:** non-zero for a successful match, and zero on failure.



## gnutls\_x509\_cert\_check\_hostname2

**int gnutls\_x509\_cert\_check\_hostname2** (*gnutls\_x509\_cert\_t* **cert**, [Function]  
                                   *const char \****hostname**, *unsigned int* **flags**)

*cert*: should contain an *gnutls\_x509\_cert\_t* structure

*hostname*: A null terminated string that contains a DNS name

*flags*: *gnutls\_certificate\_verify\_flags*

This function will check if the given certificate's subject matches the given hostname. This is a basic implementation of the matching described in RFC6125, and takes into account wildcards, and the DNSName/IPAddress subject alternative name PKIX extension.

IPv4 addresses are accepted by this function in the dotted-decimal format (e.g, ddd.ddd.ddd.ddd), and IPv6 addresses in the hexadecimal x:x:x:x:x:x:x:x format. For them the IPAddress subject alternative name extension is consulted, as well as the DNSNames in case of a non-match. The latter fallback exists due to misconfiguration of many servers which place an IPAddress inside the DNSName extension.

The comparison of dns names may have false-negatives as it is done byte by byte in non-ascii names.

When the flag *GNUTLS\_VERIFY\_DO\_NOT\_ALLOW\_WILDCARDS* is specified no wildcards are considered. Otherwise they are only considered if the domain name consists of three components or more, and the wildcard starts at the leftmost position.

**Returns:** non-zero for a successful match, and zero on failure.

## gnutls\_x509\_cert\_check\_issuer

**int gnutls\_x509\_cert\_check\_issuer** (*gnutls\_x509\_cert\_t* **cert**, [Function]  
                                   *gnutls\_x509\_cert\_t* **issuer**)

*cert*: is the certificate to be checked

*issuer*: is the certificate of a possible issuer

This function will check if the given certificate was issued by the given issuer. It checks the DN fields and the authority key identifier and subject key identifier fields match.

If the same certificate is provided at the **cert** and **issuer** fields, it will check whether the certificate is self-signed.

**Returns:** It will return true (1) if the given certificate is issued by the given issuer, and false (0) if not.

## gnutls\_x509\_cert\_check\_revocation

**int gnutls\_x509\_cert\_check\_revocation** (*gnutls\_x509\_cert\_t* **cert**, [Function]  
                                   *const gnutls\_x509\_crl\_t \****crl\_list**, *int* **crl\_list\_length**)

*cert*: should contain a *gnutls\_x509\_cert\_t* structure

*crl\_list*: should contain a list of *gnutls\_x509\_crl\_t* structures

*crl\_list\_length*: the length of the *crl\_list*

This function will return check if the given certificate is revoked. It is assumed that the CRLs have been verified before.

**Returns:** 0 if the certificate is NOT revoked, and 1 if it is. A negative error code is returned on error.

## gnutls\_x509\_cert\_cpy\_crl\_dist\_points

**int gnutls\_x509\_cert\_cpy\_crl\_dist\_points** (*gnutls\_x509\_cert\_t dst*, [Function]  
*gnutls\_x509\_cert\_t src*)

*dst*: a certificate of type *gnutls\_x509\_cert\_t*

*src*: the certificate where the dist points will be copied from

This function will copy the CRL distribution points certificate extension, from the source to the destination certificate. This may be useful to copy from a CA certificate to issued ones.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

## gnutls\_x509\_cert\_deinit

**void gnutls\_x509\_cert\_deinit** (*gnutls\_x509\_cert\_t cert*) [Function]

*cert*: The structure to be deinitialized

This function will deinitialize a certificate structure.

## gnutls\_x509\_cert\_export

**int gnutls\_x509\_cert\_export** (*gnutls\_x509\_cert\_t cert*, [Function]  
*gnutls\_x509\_cert\_fmt\_t format*, void \* *output\_data*, size\_t \*  
*output\_data\_size*)

*cert*: Holds the certificate

*format*: the format of output params. One of PEM or DER.

*output\_data*: will contain a certificate PEM or DER encoded

*output\_data\_size*: holds the size of *output\_data* (and will be replaced by the actual size of parameters)

This function will export the certificate to DER or PEM format.

If the buffer provided is not long enough to hold the output, then \**output\_data\_size* is updated and GNUTLS\_E\_SHORT\_MEMORY\_BUFFER will be returned.

If the structure is PEM encoded, it will have a header of "BEGIN CERTIFICATE".

**Returns:** In case of failure a negative error code will be returned, and 0 on success.

## gnutls\_x509\_cert\_export2

**int gnutls\_x509\_cert\_export2** (*gnutls\_x509\_cert\_t cert*, [Function]  
*gnutls\_x509\_cert\_fmt\_t format*, *gnutls\_datum\_t \* out*)

*cert*: Holds the certificate

*format*: the format of output params. One of PEM or DER.

*out*: will contain a certificate PEM or DER encoded

This function will export the certificate to DER or PEM format. The output buffer is allocated using `gnutls_malloc()` .

If the structure is PEM encoded, it will have a header of "BEGIN CERTIFICATE".

**Returns:** In case of failure a negative error code will be returned, and 0 on success.

**Since:** 3.1.3

## **gnutls\_x509\_cert\_get\_activation\_time**

`time_t gnutls_x509_cert_get_activation_time (gnutls_x509_cert_t cert)` [Function]

*cert*: should contain a `gnutls_x509_cert_t` structure

This function will return the time this Certificate was or will be activated.

**Returns:** activation time, or (time\_t)-1 on error.

## **gnutls\_x509\_cert\_get\_authority\_info\_access**

`int gnutls_x509_cert_get_authority_info_access (gnutls_x509_cert_t cert, unsigned int seq, int what, gnutls_datum_t * data, unsigned int * critical)` [Function]

*cert*: Holds the certificate

*seq*: specifies the sequence number of the access descriptor (0 for the first one, 1 for the second etc.)

*what*: what data to get, a `gnutls_info_access_what_t` type.

*data*: output data to be freed with `gnutls_free()` .

*critical*: pointer to output integer that is set to non-zero if the extension is marked as critical (may be NULL )

Note that a simpler API to access the authority info data is provided by `gnutls_x509_aia_get()` and `gnutls_x509_ext_import_aia()` .

This function extracts the Authority Information Access (AIA) extension, see RFC 5280 section 4.2.2.1 for more information. The AIA extension holds a sequence of AccessDescription (AD) data.

The *seq* input parameter is used to indicate which member of the sequence the caller is interested in. The first member is 0, the second member 1 and so on. When the *seq* value is out of bounds, `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned.

The type of data returned in *data* is specified via *what* which should be `gnutls_info_access_what_t` values.

If *what* is `GNUTLS_IA_ACCESSMETHOD_OID` then *data* will hold the accessMethod OID (e.g., "1.3.6.1.5.5.7.48.1").

If *what* is `GNUTLS_IA_ACCESSLOCATION_GENERALNAME_TYPE` , *data* will hold the accessLocation GeneralName type (e.g., "uniformResourceIdentifier").

If *what* is `GNUTLS_IA_URI` , *data* will hold the accessLocation URI data. Requesting this *what* value leads to an error if the accessLocation is not of the "uniformResourceIdentifier" type.

If **what** is `GNUTLS_IA_OCSP_URI` , **data** will hold the OCSP URI. Requesting this **what** value leads to an error if the `accessMethod` is not 1.3.6.1.5.5.7.48.1 aka OSCP, or if `accessLocation` is not of the "uniformResourceIdentifier" type. In that case `GNUTLS_E_UNKNOWN_ALGORITHM` will be returned, and **seq** should be increased and this function called again.

If **what** is `GNUTLS_IA_CAISSEURS_URI` , **data** will hold the caIssuers URI. Requesting this **what** value leads to an error if the `accessMethod` is not 1.3.6.1.5.5.7.48.2 aka caIssuers, or if `accessLocation` is not of the "uniformResourceIdentifier" type. In that case handle as in `GNUTLS_IA_OCSP_URI` .

More **what** values may be allocated in the future as needed.

If **data** is `NULL`, the function does the same without storing the output data, that is, it will set **critical** and do error checking as usual.

The value of the critical flag is returned in `* critical` . Supply a `NULL critical` if you want the function to make sure the extension is non-critical, as required by RFC 5280.

**Returns:** `GNUTLS_E_SUCCESS` on success, `GNUTLS_E_INVALID_REQUEST` on invalid **crt** , `GNUTLS_E_CONSTRAINT_ERROR` if the extension is incorrectly marked as critical (use a non-`NULL critical` to override), `GNUTLS_E_UNKNOWN_ALGORITHM` if the requested OID does not match (e.g., when using `GNUTLS_IA_OCSP_URI` ), otherwise a negative error code.

**Since:** 3.0

## gnutls\_x509\_cert\_get\_authority\_key\_gn\_serial

```
int gnutls_x509_cert_get_authority_key_gn_serial           [Function]
    (gnutls_x509_cert_t cert, unsigned int seq, void * alt, size_t * alt_size,
     unsigned int * alt_type, void * serial, size_t * serial_size, unsigned int
     * critical)
```

*cert*: should contain a `gnutls_x509_cert_t` structure

*seq*: specifies the sequence number of the alt name (0 for the first one, 1 for the second etc.)

*alt*: is the place where the alternative name will be copied to

*alt\_size*: holds the size of alt.

*alt\_type*: holds the type of the alternative name (one of `gnutls_x509_subject_alt_name_t`).

*serial*: buffer to store the serial number (may be null)

*serial\_size*: Holds the size of the serial field (may be null)

*critical*: will be non-zero if the extension is marked as critical (may be null)

This function will return the X.509 authority key identifier when stored as a general name (`authorityCertIssuer`) and serial number.

Because more than one general names might be stored **seq** can be used as a counter to request them all until `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if the extension is not present, otherwise a negative error value.

**Since:** 3.0

## gnutls\_x509\_cert\_get\_authority\_key\_id

**int gnutls\_x509\_cert\_get\_authority\_key\_id** (*gnutls\_x509\_cert\_t* *cert*, *void \* id*, *size\_t \* id\_size*, *unsigned int \* critical*) [Function]

*cert*: should contain a *gnutls\_x509\_cert\_t* structure

*id*: The place where the identifier will be copied

*id\_size*: Holds the size of the id field.

*critical*: will be non-zero if the extension is marked as critical (may be null)

This function will return the X.509v3 certificate authority's key identifier. This is obtained by the X.509 Authority Key identifier extension field (2.5.29.35). Note that this function only returns the keyIdentifier field of the extension and *GNUTLS\_E\_X509\_UNSUPPORTED\_EXTENSION*, if the extension contains the name and serial number of the certificate. In that case *gnutls\_x509\_cert\_get\_authority\_key\_serial()* may be used.

**Returns:** On success, *GNUTLS\_E\_SUCCESS* (0) is returned, *GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE* if the extension is not present, otherwise a negative error value.

## gnutls\_x509\_cert\_get\_basic\_constraints

**int gnutls\_x509\_cert\_get\_basic\_constraints** (*gnutls\_x509\_cert\_t* *cert*, *unsigned int \* critical*, *unsigned int \* ca*, *int \* pathlen*) [Function]

*cert*: should contain a *gnutls\_x509\_cert\_t* structure

*critical*: will be non-zero if the extension is marked as critical

*ca*: pointer to output integer indicating CA status, may be NULL, value is 1 if the certificate CA flag is set, 0 otherwise.

*pathlen*: pointer to output integer indicating path length (may be NULL), non-negative error codes indicate a present pathLenConstraint field and the actual value, -1 indicate that the field is absent.

This function will read the certificate's basic constraints, and return the certificates CA status. It reads the basicConstraints X.509 extension (2.5.29.19).

**Returns:** If the certificate is a CA a positive value will be returned, or (0) if the certificate does not have CA flag set. A negative error code may be returned in case of errors. If the certificate does not contain the basicConstraints extension *GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE* will be returned.

## gnutls\_x509\_cert\_get\_ca\_status

**int gnutls\_x509\_cert\_get\_ca\_status** (*gnutls\_x509\_cert\_t* *cert*, *unsigned int \* critical*) [Function]

*cert*: should contain a *gnutls\_x509\_cert\_t* structure

*critical*: will be non-zero if the extension is marked as critical

This function will return certificates CA status, by reading the basicConstraints X.509 extension (2.5.29.19). If the certificate is a CA a positive value will be returned, or (0) if the certificate does not have CA flag set.

Use `gnutls_x509_cert_get_basic_constraints()` if you want to read the `pathLenConstraint` field too.

**Returns:** If the certificate is a CA a positive value will be returned, or (0) if the certificate does not have CA flag set. A negative error code may be returned in case of errors. If the certificate does not contain the `basicConstraints` extension `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

## `gnutls_x509_cert_get_crl_dist_points`

```
int gnutls_x509_cert_get_crl_dist_points (gnutls_x509_cert_t cert, unsigned int seq, void * san, size_t * san_size, unsigned int * reason_flags, unsigned int * critical) [Function]
```

*cert*: should contain a `gnutls_x509_cert_t` structure

*seq*: specifies the sequence number of the distribution point (0 for the first one, 1 for the second etc.)

*san*: is the place where the distribution point will be copied to

*san\_size*: holds the size of *ret*.

*reason\_flags*: Revocation reasons. An ORed sequence of flags from `gnutls_x509_crl_reason_flags_t`.

*critical*: will be non-zero if the extension is marked as critical (may be null)

This function retrieves the CRL distribution points (2.5.29.31), contained in the given certificate in the X509v3 Certificate Extensions.

**Returns:** `GNUTLS_E_SHORT_MEMORY_BUFFER` and updates *ret\_size* if *ret\_size* is not enough to hold the distribution point, or the type of the distribution point if everything was ok. The type is one of the enumerated `gnutls_x509_subject_alt_name_t`. If the certificate does not have an Alternative name with the specified sequence number then `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned.

## `gnutls_x509_cert_get_dn`

```
int gnutls_x509_cert_get_dn (gnutls_x509_cert_t cert, char * buf, size_t * buf_size) [Function]
```

*cert*: should contain a `gnutls_x509_cert_t` structure

*buf*: a pointer to a structure to hold the name (may be null)

*buf\_size*: initially holds the size of *buf*

This function will copy the name of the Certificate in the provided buffer. The name will be in the form "C=xxxx,O=yyyy,CN=zzzz" as described in RFC4514. The output string will be ASCII or UTF-8 encoded, depending on the certificate data.

If *buf* is null then only the size will be filled.

**Returns:** `GNUTLS_E_SHORT_MEMORY_BUFFER` if the provided buffer is not long enough, and in that case the *buf\_size* will be updated with the required size. On success 0 is returned.

## gnutls\_x509\_cert\_get\_dn2

`int gnutls_x509_cert_get_dn2 (gnutls_x509_cert_t cert, gnutls_datum_t * dn)` [Function]

*cert*: should contain a `gnutls_x509_cert_t` structure

*dn*: a pointer to a structure to hold the name

This function will allocate buffer and copy the name of the Certificate. The name will be in the form "C=xxxx,O=yyyy,CN=zzzz" as described in RFC4514. The output string will be ASCII or UTF-8 encoded, depending on the certificate data.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value. and a negative error code on error.

**Since:** 3.1.10

## gnutls\_x509\_cert\_get\_dn\_by\_oid

`int gnutls_x509_cert_get_dn_by_oid (gnutls_x509_cert_t cert, const char * oid, int indx, unsigned int raw_flag, void * buf, size_t * buf_size)` [Function]

*cert*: should contain a `gnutls_x509_cert_t` structure

*oid*: holds an Object Identified in null terminated string

*indx*: In case multiple same OIDs exist in the RDN, this specifies which to send. Use (0) to get the first one.

*raw\_flag*: If non-zero returns the raw DER data of the DN part.

*buf*: a pointer where the DN part will be copied (may be null).

*buf\_size*: initially holds the size of *buf*

This function will extract the part of the name of the Certificate subject specified by the given OID. The output, if the raw flag is not used, will be encoded as described in RFC4514. Thus a string that is ASCII or UTF-8 encoded, depending on the certificate data.

Some helper macros with popular OIDs can be found in `gnutls/x509.h`. If raw flag is (0), this function will only return known OIDs as text. Other OIDs will be DER encoded, as described in RFC4514 – in hex format with a '#' prefix. You can check about known OIDs using `gnutls_x509_dn_oid_known()`.

If *buf* is null then only the size will be filled. If the *raw\_flag* is not specified the output is always null terminated, although the *buf\_size* will not include the null character.

**Returns:** `GNUTLS_E_SHORT_MEMORY_BUFFER` if the provided buffer is not long enough, and in that case the *buf\_size* will be updated with the required size. `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if there are no data in the current index. On success 0 is returned.

## gnutls\_x509\_cert\_get\_dn\_oid

`int gnutls_x509_cert_get_dn_oid (gnutls_x509_cert_t cert, int indx, void * oid, size_t * oid_size)` [Function]

*cert*: should contain a `gnutls_x509_cert_t` structure

*indx*: This specifies which OID to return. Use (0) to get the first one.

*oid*: a pointer to a buffer to hold the OID (may be null)

*oid\_size*: initially holds the size of *oid*

This function will extract the OIDs of the name of the Certificate subject specified by the given index.

If *oid* is null then only the size will be filled. The *oid* returned will be null terminated, although *oid\_size* will not account for the trailing null.

**Returns:** GNUTLS\_E\_SHORT\_MEMORY\_BUFFER if the provided buffer is not long enough, and in that case the *buf\_size* will be updated with the required size. GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE if there are no data in the current index. On success 0 is returned.

### gnutls\_x509\_cert\_get\_expiration\_time

`time_t gnutls_x509_cert_get_expiration_time (gnutls_x509_cert_t cert)` [Function]

*cert*: should contain a `gnutls_x509_cert_t` structure

This function will return the time this Certificate was or will be expired.

The no well defined expiration time can be checked against with the GNUTLS\_X509\_NO\_WELL\_DEFINED\_EXPIRATION macro.

**Returns:** expiration time, or (time\_t)-1 on error.

### gnutls\_x509\_cert\_get\_extension\_by\_oid

`int gnutls_x509_cert_get_extension_by_oid (gnutls_x509_cert_t cert, const char * oid, int indx, void * buf, size_t * buf_size, unsigned int * critical)` [Function]

*cert*: should contain a `gnutls_x509_cert_t` structure

*oid*: holds an Object Identified in null terminated string

*indx*: In case multiple same OIDs exist in the extensions, this specifies which to send. Use (0) to get the first one.

*buf*: a pointer to a structure to hold the name (may be null)

*buf\_size*: initially holds the size of *buf*

*critical*: will be non-zero if the extension is marked as critical

This function will return the extension specified by the OID in the certificate. The extensions will be returned as binary data DER encoded, in the provided buffer.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error code is returned. If the certificate does not contain the specified extension GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE will be returned.

### gnutls\_x509\_cert\_get\_extension\_by\_oid2

`int gnutls_x509_cert_get_extension_by_oid2 (gnutls_x509_cert_t cert, const char * oid, int indx, gnutls_datum_t * output, unsigned int * critical)` [Function]

*cert*: should contain a `gnutls_x509_cert_t` structure



*oid*: holds an Object Identified in null terminated string

*indx*: In case multiple same OIDs exist in the extensions, this specifies which to send. Use (0) to get the first one.

*output*: will hold the allocated extension data

*critical*: will be non-zero if the extension is marked as critical

This function will return the extension specified by the OID in the certificate. The extensions will be returned as binary data DER encoded, in the provided buffer.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error code is returned. If the certificate does not contain the specified extension GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE will be returned.

**Since:** 3.3.8

## gnutls\_x509\_cert\_get\_extension\_data

```
int gnutls_x509_cert_get_extension_data (gnutls_x509_cert_t cert,      [Function]
                                         int indx, void * data, size_t * sizeof_data)
```

*cert*: should contain a gnutls\_x509\_cert\_t structure

*indx*: Specifies which extension OID to send. Use (0) to get the first one.

*data*: a pointer to a structure to hold the data (may be null)

*sizeof\_data*: initially holds the size of *data*

This function will return the requested extension data in the certificate. The extension data will be stored in the provided buffer.

Use gnutls\_x509\_cert\_get\_extension\_info() to extract the OID and critical flag. Use gnutls\_x509\_cert\_get\_extension\_by\_oid() instead, if you want to get data indexed by the extension OID rather than sequence.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error code is returned. If you have reached the last extension available GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE will be returned.

## gnutls\_x509\_cert\_get\_extension\_data2

```
int gnutls_x509_cert_get_extension_data2 (gnutls_x509_cert_t      [Function]
                                           cert, unsigned indx, gnutls_datum_t * data)
```

*cert*: should contain a gnutls\_x509\_cert\_t structure

*indx*: Specifies which extension OID to read. Use (0) to get the first one.

*data*: will contain the extension DER-encoded data

This function will return the requested by the index extension data in the certificate. The extension data will be allocated using gnutls\_malloc() .

Use gnutls\_x509\_cert\_get\_extension\_info() to extract the OID.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error code is returned. If you have reached the last extension available GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE will be returned.

**gnutls\_x509\_cert\_get\_extension\_info**

**int gnutls\_x509\_cert\_get\_extension\_info** (*gnutls\_x509\_cert\_t cert*, [Function]  
*int indx*, *void \* oid*, *size\_t \* oid\_size*, *unsigned int \* critical*)

*cert*: should contain a *gnutls\_x509\_cert\_t* structure

*indx*: Specifies which extension OID to send. Use (0) to get the first one.

*oid*: a pointer to a structure to hold the OID

*oid\_size*: initially holds the maximum size of *oid* , on return holds actual size of *oid*

.

*critical*: output variable with critical flag, may be NULL.

This function will return the requested extension OID in the certificate, and the critical flag for it. The extension OID will be stored as a string in the provided buffer. Use *gnutls\_x509\_cert\_get\_extension()* to extract the data.

If the buffer provided is not long enough to hold the output, then *oid\_size* is updated and *GNUTLS\_E\_SHORT\_MEMORY\_BUFFER* will be returned. The *oid* returned will be null terminated, although *oid\_size* will not account for the trailing null.

**Returns:** On success, *GNUTLS\_E\_SUCCESS* (0) is returned, otherwise a negative error code is returned. If you have reached the last extension available *GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE* will be returned.

**gnutls\_x509\_cert\_get\_extension\_oid**

**int gnutls\_x509\_cert\_get\_extension\_oid** (*gnutls\_x509\_cert\_t cert*, [Function]  
*int indx*, *void \* oid*, *size\_t \* oid\_size*)

*cert*: should contain a *gnutls\_x509\_cert\_t* structure

*indx*: Specifies which extension OID to send. Use (0) to get the first one.

*oid*: a pointer to a structure to hold the OID (may be null)

*oid\_size*: initially holds the size of *oid*

This function will return the requested extension OID in the certificate. The extension OID will be stored as a string in the provided buffer.

The *oid* returned will be null terminated, although *oid\_size* will not account for the trailing null.

**Returns:** On success, *GNUTLS\_E\_SUCCESS* (0) is returned, otherwise a negative error code is returned. If you have reached the last extension available *GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE* will be returned.

**gnutls\_x509\_cert\_get\_fingerprint**

**int gnutls\_x509\_cert\_get\_fingerprint** (*gnutls\_x509\_cert\_t cert*, [Function]  
*gnutls\_digest\_algorithm\_t algo*, *void \* buf*, *size\_t \* buf\_size*)

*cert*: should contain a *gnutls\_x509\_cert\_t* structure

*algo*: is a digest algorithm

*buf*: a pointer to a structure to hold the fingerprint (may be null)

*buf\_size*: initially holds the size of *buf*

This function will calculate and copy the certificate's fingerprint in the provided buffer. The fingerprint is a hash of the DER-encoded data of the certificate.

If the buffer is null then only the size will be filled.

**Returns:** GNUTLS\_E\_SHORT\_MEMORY\_BUFFER if the provided buffer is not long enough, and in that case the *\*buf.size* will be updated with the required size. On success 0 is returned.

### gnutls\_x509\_cert\_get\_issuer

`int gnutls_x509_cert_get_issuer (gnutls_x509_cert_t cert, [Function]  
gnutls_x509_dn_t * dn)`

*cert*: should contain a `gnutls_x509_cert_t` structure

*dn*: output variable with pointer to `uint8_t` DN

Return the Certificate's Issuer DN as a `gnutls_x509_dn_t` data type, that can be decoded using `gnutls_x509_dn_get_rdn_ava()`.

Note that *dn* should be treated as constant. Because it points into the *cert* object, you should not use *dn* after *cert* is deallocated.

**Returns:** Returns 0 on success, or an error code.

### gnutls\_x509\_cert\_get\_issuer\_alt\_name

`int gnutls_x509_cert_get_issuer_alt_name (gnutls_x509_cert_t [Function]  
cert, unsigned int seq, void * ian, size_t * ian_size, unsigned int *  
critical)`

*cert*: should contain a `gnutls_x509_cert_t` structure

*seq*: specifies the sequence number of the alt name (0 for the first one, 1 for the second etc.)

*ian*: is the place where the alternative name will be copied to

*ian\_size*: holds the size of *ian*.

*critical*: will be non-zero if the extension is marked as critical (may be null)

This function retrieves the Issuer Alternative Name (2.5.29.18), contained in the given certificate in the X509v3 Certificate Extensions.

When the SAN type is `otherName`, it will extract the data in the `otherName`'s value field, and `GNUTLS_SAN_OTHERNAME` is returned. You may use `gnutls_x509_cert_get_subject_alt_othername_oid()` to get the corresponding OID and the "virtual" SAN types (e.g., `GNUTLS_SAN_OTHERNAME_XMPP`).

If an `otherName` OID is known, the data will be decoded. Otherwise the returned data will be DER encoded, and you will have to decode it yourself. Currently, only the RFC 3920 `id-on-xmppAddr` Issuer AltName is recognized.

**Returns:** the alternative issuer name type on success, one of the enumerated `gnutls_x509_subject_alt_name_t`. It will return `GNUTLS_E_SHORT_MEMORY_BUFFER` if *ian\_size* is not large enough to hold the value. In that case *ian\_size* will be updated with the required size. If the certificate does not have an Alternative name with the specified sequence number then `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned.

**Since:** 2.10.0

## gnutls\_x509\_cert\_get\_issuer\_alt\_name2

`int gnutls_x509_cert_get_issuer_alt_name2 (gnutls_x509_cert_t [Function]  
   cert, unsigned int seq, void * ian, size_t * ian_size, unsigned int *  
   ian_type, unsigned int * critical)`

*cert*: should contain a `gnutls_x509_cert_t` structure

*seq*: specifies the sequence number of the alt name (0 for the first one, 1 for the second etc.)

*ian*: is the place where the alternative name will be copied to

*ian\_size*: holds the size of *ret*.

*ian\_type*: holds the type of the alternative name (one of `gnutls_x509_subject_alt_name_t`).

*critical*: will be non-zero if the extension is marked as critical (may be null)

This function will return the alternative names, contained in the given certificate. It is the same as `gnutls_x509_cert_get_issuer_alt_name()` except for the fact that it will return the type of the alternative name in *ian\_type* even if the function fails for some reason (i.e. the buffer provided is not enough).

**Returns:** the alternative issuer name type on success, one of the enumerated `gnutls_x509_subject_alt_name_t`. It will return `GNUTLS_E_SHORT_MEMORY_BUFFER` if *ian\_size* is not large enough to hold the value. In that case *ian\_size* will be updated with the required size. If the certificate does not have an Alternative name with the specified sequence number then `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned.

**Since:** 2.10.0

## gnutls\_x509\_cert\_get\_issuer\_alt\_othername\_oid

`int gnutls_x509_cert_get_issuer_alt_othername_oid [Function]  
   (gnutls_x509_cert_t cert, unsigned int seq, void * ret, size_t * ret_size)`

*cert*: should contain a `gnutls_x509_cert_t` structure

*seq*: specifies the sequence number of the alt name (0 for the first one, 1 for the second etc.)

*ret*: is the place where the otherName OID will be copied to

*ret\_size*: holds the size of *ret*.

This function will extract the type OID of an otherName Subject Alternative Name, contained in the given certificate, and return the type as an enumerated element.

If *oid* is null then only the size will be filled. The *oid* returned will be null terminated, although *oid\_size* will not account for the trailing null.

This function is only useful if `gnutls_x509_cert_get_issuer_alt_name()` returned `GNUTLS_SAN_OTHERNAME`.

**Returns:** the alternative issuer name type on success, one of the enumerated `gnutls_x509_subject_alt_name_t`. For supported OIDs, it will return one of the virtual (`GNUTLS_SAN_OTHERNAME_*`) types, e.g. `GNUTLS_SAN_OTHERNAME_XMPP`, and `GNUTLS_SAN_OTHERNAME` for unknown OIDs. It will return `GNUTLS_E_SHORT_MEMORY_BUFFER` if *ret\_size* is not large enough to hold the value. In that case

**ret\_size** will be updated with the required size. If the certificate does not have an Alternative name with the specified sequence number and with the otherName type then `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned.

**Since:** 2.10.0

### **gnutls\_x509\_cert\_get\_issuer\_dn**

```
int gnutls_x509_cert_get_issuer_dn (gnutls_x509_cert_t cert, char * buf, size_t * buf_size) [Function]
```

*cert*: should contain a `gnutls_x509_cert_t` structure

*buf*: a pointer to a structure to hold the name (may be null)

*buf\_size*: initially holds the size of *buf*

This function will copy the name of the Certificate issuer in the provided buffer. The name will be in the form "C=xxxx,O=yyyy,CN=zzzz" as described in RFC4514. The output string will be ASCII or UTF-8 encoded, depending on the certificate data.

If *buf* is null then only the size will be filled.

**Returns:** `GNUTLS_E_SHORT_MEMORY_BUFFER` if the provided buffer is not long enough, and in that case the *buf\_size* will be updated with the required size. On success 0 is returned.

### **gnutls\_x509\_cert\_get\_issuer\_dn2**

```
int gnutls_x509_cert_get_issuer_dn2 (gnutls_x509_cert_t cert, gnutls_datum_t * dn) [Function]
```

*cert*: should contain a `gnutls_x509_cert_t` structure

*dn*: a pointer to a structure to hold the name

This function will allocate buffer and copy the name of issuer of the Certificate. The name will be in the form "C=xxxx,O=yyyy,CN=zzzz" as described in RFC4514. The output string will be ASCII or UTF-8 encoded, depending on the certificate data.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value. and a negative error code on error.

**Since:** 3.1.10

### **gnutls\_x509\_cert\_get\_issuer\_dn\_by\_oid**

```
int gnutls_x509_cert_get_issuer_dn_by_oid (gnutls_x509_cert_t cert, const char * oid, int indx, unsigned int raw_flag, void * buf, size_t * buf_size) [Function]
```

*cert*: should contain a `gnutls_x509_cert_t` structure

*oid*: holds an Object Identified in null terminated string

*indx*: In case multiple same OIDs exist in the RDN, this specifies which to send. Use (0) to get the first one.

*raw\_flag*: If non-zero returns the raw DER data of the DN part.

*buf*: a pointer to a structure to hold the name (may be null)

*buf\_size*: initially holds the size of *buf*

This function will extract the part of the name of the Certificate issuer specified by the given OID. The output, if the raw flag is not used, will be encoded as described in RFC4514. Thus a string that is ASCII or UTF-8 encoded, depending on the certificate data.

Some helper macros with popular OIDs can be found in `gnutls/x509.h`. If raw flag is (0), this function will only return known OIDs as text. Other OIDs will be DER encoded, as described in RFC4514 – in hex format with a '#' prefix. You can check about known OIDs using `gnutls_x509_dn_oid_known()`.

If `buf` is null then only the size will be filled. If the `raw_flag` is not specified the output is always null terminated, although the `buf_size` will not include the null character.

**Returns:** `GNUTLS_E_SHORT_MEMORY_BUFFER` if the provided buffer is not long enough, and in that case the `buf_size` will be updated with the required size. `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if there are no data in the current index. On success 0 is returned.

### `gnutls_x509_cert_get_issuer_dn_oid`

```
int gnutls_x509_cert_get_issuer_dn_oid (gnutls_x509_cert_t cert,      [Function]
                                       int indx, void * oid, size_t * oid_size)
```

*cert*: should contain a `gnutls_x509_cert_t` structure

*indx*: This specifies which OID to return. Use (0) to get the first one.

*oid*: a pointer to a buffer to hold the OID (may be null)

*oid\_size*: initially holds the size of *oid*

This function will extract the OIDs of the name of the Certificate issuer specified by the given index.

If *oid* is null then only the size will be filled. The *oid* returned will be null terminated, although *oid\_size* will not account for the trailing null.

**Returns:** `GNUTLS_E_SHORT_MEMORY_BUFFER` if the provided buffer is not long enough, and in that case the `buf_size` will be updated with the required size. `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if there are no data in the current index. On success 0 is returned.

### `gnutls_x509_cert_get_issuer_unique_id`

```
int gnutls_x509_cert_get_issuer_unique_id (gnutls_x509_cert_t      [Function]
                                           crt, char * buf, size_t * buf_size)
```

*crt*: Holds the certificate

*buf*: user allocated memory buffer, will hold the unique id

*buf\_size*: size of user allocated memory buffer (on input), will hold actual size of the unique ID on return.

This function will extract the issuerUniqueID value (if present) for the given certificate.

If the user allocated memory buffer is not large enough to hold the full subjectUniqueID, then a `GNUTLS_E_SHORT_MEMORY_BUFFER` error will be returned, and `buf_size` will be set to the actual length.

**Returns:** GNUTLS\_E\_SUCCESS on success, otherwise a negative error code.

**Since:** 2.12.0

## gnutls\_x509\_cert\_get\_key\_id

```
int gnutls_x509_cert_get_key_id (gnutls_x509_cert_t cert, unsigned int flags, unsigned char * output_data, size_t * output_data_size) [Function]
```

*cert*: Holds the certificate

*flags*: should be 0 for now

*output\_data*: will contain the key ID

*output\_data\_size*: holds the size of *output\_data* (and will be replaced by the actual size of parameters)

This function will return a unique ID that depends on the public key parameters. This ID can be used in checking whether a certificate corresponds to the given private key.

If the buffer provided is not long enough to hold the output, then *\*output\_data\_size* is updated and GNUTLS\_E\_SHORT\_MEMORY\_BUFFER will be returned. The output will normally be a SHA-1 hash output, which is 20 bytes.

**Returns:** In case of failure a negative error code will be returned, and 0 on success.

## gnutls\_x509\_cert\_get\_key\_purpose\_oid

```
int gnutls_x509_cert_get_key_purpose_oid (gnutls_x509_cert_t cert, int indx, void * oid, size_t * oid_size, unsigned int * critical) [Function]
```

*cert*: should contain a *gnutls\_x509\_cert\_t* structure

*indx*: This specifies which OID to return. Use (0) to get the first one.

*oid*: a pointer to a buffer to hold the OID (may be null)

*oid\_size*: initially holds the size of *oid*

*critical*: output flag to indicate criticality of extension

This function will extract the key purpose OIDs of the Certificate specified by the given index. These are stored in the Extended Key Usage extension (2.5.29.37) See the GNUTLS\_KP\_\* definitions for human readable names.

If *oid* is null then only the size will be filled. The *oid* returned will be null terminated, although *oid\_size* will not account for the trailing null.

**Returns:** GNUTLS\_E\_SHORT\_MEMORY\_BUFFER if the provided buffer is not long enough, and in that case the *\*oid\_size* will be updated with the required size. On success 0 is returned.

## gnutls\_x509\_cert\_get\_key\_usage

```
int gnutls_x509_cert_get_key_usage (gnutls_x509_cert_t cert, unsigned int * key_usage, unsigned int * critical) [Function]
```

*cert*: should contain a *gnutls\_x509\_cert\_t* structure

*key\_usage*: where the key usage bits will be stored

*critical*: will be non-zero if the extension is marked as critical

This function will return certificate's key usage, by reading the keyUsage X.509 extension (2.5.29.15). The key usage value will ORed values of the: GNUTLS\_KEY\_DIGITAL\_SIGNATURE , GNUTLS\_KEY\_NON\_REPUDIATION , GNUTLS\_KEY\_KEY\_ENCIPHERMENT , GNUTLS\_KEY\_DATA\_ENCIPHERMENT , GNUTLS\_KEY\_KEY\_AGREEMENT , GNUTLS\_KEY\_KEY\_CERT\_SIGN , GNUTLS\_KEY\_CRL\_SIGN , GNUTLS\_KEY\_ENCIPHER\_ONLY , GNUTLS\_KEY\_DECIPHER\_ONLY .

**Returns:** the certificate key usage, or a negative error code in case of parsing error. If the certificate does not contain the keyUsage extension GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE will be returned.

### gnutls\_x509\_cert\_get\_name\_constraints

```
int gnutls_x509_cert_get_name_constraints (gnutls_x509_cert_t cert,      [Function]
                                           gnutls_x509_name_constraints_t nc, unsigned int flags, unsigned int *
                                           critical)
```

*cert*: should contain a gnutls\_x509\_cert\_t structure

*nc*: The nameconstraints intermediate structure

*flags*: zero or GNUTLS\_NAME\_CONSTRAINTS\_FLAG\_APPEND

*critical*: the extension status

This function will return an intermediate structure containing the name constraints of the provided CA certificate. That structure can be used in combination with gnutls\_x509\_name\_constraints\_check() to verify whether a server's name is in accordance with the constraints.

When the *flags* is set to GNUTLS\_NAME\_CONSTRAINTS\_FLAG\_APPEND , then if the *nc* structure is empty this function will behave identically as if the flag was not set. Otherwise if there are elements in the *nc* structure then only the excluded constraints will be appended to the constraints.

Note that *nc* must be initialized prior to calling this function.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE if the extension is not present, otherwise a negative error value.

**Since:** 3.3.0

### gnutls\_x509\_cert\_get\_pk\_algorithm

```
int gnutls_x509_cert_get_pk_algorithm (gnutls_x509_cert_t cert,      [Function]
                                         unsigned int * bits)
```

*cert*: should contain a gnutls\_x509\_cert\_t structure

*bits*: if bits is non null it will hold the size of the parameters' in bits

This function will return the public key algorithm of an X.509 certificate.

If bits is non null, it should have enough size to hold the parameters size in bits. For RSA the bits returned is the modulus. For DSA the bits returned are of the public exponent.

**Returns:** a member of the gnutls\_pk\_algorithm\_t enumeration on success, or a negative error code on error.



**gnutls\_x509\_cert\_get\_pk\_dsa\_raw**

**int gnutls\_x509\_cert\_get\_pk\_dsa\_raw** (*gnutls\_x509\_cert\_t crt*, [Function]  
*gnutls\_datum\_t \* p*, *gnutls\_datum\_t \* q*, *gnutls\_datum\_t \* g*, *gnutls\_datum\_t \* y*)

*crt*: Holds the certificate

*p*: will hold the p

*q*: will hold the q

*g*: will hold the g

*y*: will hold the y

This function will export the DSA public key's parameters found in the given certificate. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

**Returns:** GNUTLS\_E\_SUCCESS on success, otherwise a negative error code.

**gnutls\_x509\_cert\_get\_pk\_rsa\_raw**

**int gnutls\_x509\_cert\_get\_pk\_rsa\_raw** (*gnutls\_x509\_cert\_t crt*, [Function]  
*gnutls\_datum\_t \* m*, *gnutls\_datum\_t \* e*)

*crt*: Holds the certificate

*m*: will hold the modulus

*e*: will hold the public exponent

This function will export the RSA public key's parameters found in the given structure. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

**Returns:** GNUTLS\_E\_SUCCESS on success, otherwise a negative error code.

**gnutls\_x509\_cert\_get\_policy**

**int gnutls\_x509\_cert\_get\_policy** (*gnutls\_x509\_cert\_t crt*, *int indx*, [Function]  
*struct gnutls\_x509\_policy\_st \* policy*, *unsigned int \* critical*)

*crt*: should contain a `gnutls_x509_cert_t` structure

*indx*: This specifies which policy to return. Use (0) to get the first one.

*policy*: A pointer to a policy structure.

*critical*: will be non-zero if the extension is marked as critical

This function will extract the certificate policy (extension 2.5.29.32) specified by the given index.

The policy returned by this function must be deinitialized by using `gnutls_x509_policy_release()`.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE if the extension is not present, otherwise a negative error value.

**Since:** 3.1.5

**gnutls\_x509\_cert\_get\_private\_key\_usage\_period**

**int gnutls\_x509\_cert\_get\_private\_key\_usage\_period** [Function]  
 (*gnutls\_x509\_cert\_t cert, time\_t \* activation, time\_t \* expiration,*  
*unsigned int \* critical*)

*cert*: should contain a *gnutls\_x509\_cert\_t* structure

*activation*: The activation time

*expiration*: The expiration time

*critical*: the extension status

This function will return the expiration and activation times of the private key of the certificate. It relies on the PKIX extension 2.5.29.16 being present.

**Returns:** On success, *GNUTLS\_E\_SUCCESS* (0) is returned, *GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE* if the extension is not present, otherwise a negative error value.

**gnutls\_x509\_cert\_get\_proxy**

**int gnutls\_x509\_cert\_get\_proxy** (*gnutls\_x509\_cert\_t cert, unsigned* [Function]  
*int \* critical, int \* pathlen, char \*\* policyLanguage, char \*\* policy,*  
*size\_t \* sizeof\_policy*)

*cert*: should contain a *gnutls\_x509\_cert\_t* structure

*critical*: will be non-zero if the extension is marked as critical

*pathlen*: pointer to output integer indicating path length (may be NULL), non-negative error codes indicate a present *pCPathLenConstraint* field and the actual value, -1 indicate that the field is absent.

*policyLanguage*: output variable with OID of policy language

*policy*: output variable with policy data

*sizeof\_policy*: output variable size of policy data

This function will get information from a proxy certificate. It reads the *ProxyCertInfo* X.509 extension (1.3.6.1.5.5.7.1.14).

**Returns:** On success, *GNUTLS\_E\_SUCCESS* (0) is returned, otherwise a negative error code is returned.

**gnutls\_x509\_cert\_get\_raw\_dn**

**int gnutls\_x509\_cert\_get\_raw\_dn** (*gnutls\_x509\_cert\_t cert,* [Function]  
*gnutls\_datum\_t \* dn*)

*cert*: should contain a *gnutls\_x509\_cert\_t* structure

*dn*: will hold the starting point of the DN

This function will return a pointer to the DER encoded DN structure and the length. This points to allocated data that must be free'd using *gnutls\_free()* .

**Returns:** On success, *GNUTLS\_E\_SUCCESS* (0) is returned, otherwise a negative error value. or a negative error code on error.

**gnutls\_x509\_cert\_get\_raw\_issuer\_dn**

**int gnutls\_x509\_cert\_get\_raw\_issuer\_dn** (*gnutls\_x509\_cert\_t cert*, [Function]  
*gnutls\_datum\_t \* dn*)

*cert*: should contain a **gnutls\_x509\_cert\_t** structure

*dn*: will hold the starting point of the DN

This function will return a pointer to the DER encoded DN structure and the length. This points to allocated data that must be free'd using **gnutls\_free()** .

**Returns:** On success, **GNUTLS\_E\_SUCCESS** (0) is returned, otherwise a negative error value or a negative error code on error.

**gnutls\_x509\_cert\_get\_serial**

**int gnutls\_x509\_cert\_get\_serial** (*gnutls\_x509\_cert\_t cert*, void \* [Function]  
*result*, *size\_t \* result\_size*)

*cert*: should contain a **gnutls\_x509\_cert\_t** structure

*result*: The place where the serial number will be copied

*result\_size*: Holds the size of the result field.

This function will return the X.509 certificate's serial number. This is obtained by the X509 Certificate serialNumber field. Serial is not always a 32 or 64bit number. Some CAs use large serial numbers, thus it may be wise to handle it as something **uint8\_t**.

**Returns:** On success, **GNUTLS\_E\_SUCCESS** (0) is returned, otherwise a negative error value.

**gnutls\_x509\_cert\_get\_signature**

**int gnutls\_x509\_cert\_get\_signature** (*gnutls\_x509\_cert\_t cert*, char \* [Function]  
*sig*, *size\_t \* sig\_size*)

*cert*: should contain a **gnutls\_x509\_cert\_t** structure

*sig*: a pointer where the signature part will be copied (may be null).

*sig\_size*: initially holds the size of **sig**

This function will extract the signature field of a certificate.

**Returns:** On success, **GNUTLS\_E\_SUCCESS** (0) is returned, otherwise a negative error value. and a negative error code on error.

**gnutls\_x509\_cert\_get\_signature\_algorithm**

**int gnutls\_x509\_cert\_get\_signature\_algorithm** (*gnutls\_x509\_cert\_t* [Function]  
*cert*)

*cert*: should contain a **gnutls\_x509\_cert\_t** structure

This function will return a value of the **gnutls\_sign\_algorithm\_t** enumeration that is the signature algorithm that has been used to sign this certificate.

**Returns:** a **gnutls\_sign\_algorithm\_t** value, or a negative error code on error.

## gnutls\_x509\_cert\_get\_subject

```
int gnutls_x509_cert_get_subject (gnutls_x509_cert_t cert,          [Function]
                                gnutls_x509_dn_t * dn)
```

*cert*: should contain a `gnutls_x509_cert_t` structure

*dn*: output variable with pointer to `uint8_t` DN.

Return the Certificate's Subject DN as a `gnutls_x509_dn_t` data type, that can be decoded using `gnutls_x509_dn_get_rdn_ava()` .

Note that *dn* should be treated as constant. Because it points into the *cert* object, you should not use *dn* after *cert* is deallocated.

**Returns:** Returns 0 on success, or an error code.

## gnutls\_x509\_cert\_get\_subject\_alt\_name

```
int gnutls_x509_cert_get_subject_alt_name (gnutls_x509_cert_t      [Function]
                                            cert, unsigned int seq, void * san, size_t * san_size, unsigned int *
                                            critical)
```

*cert*: should contain a `gnutls_x509_cert_t` structure

*seq*: specifies the sequence number of the alt name (0 for the first one, 1 for the second etc.)

*san*: is the place where the alternative name will be copied to

*san\_size*: holds the size of *san*.

*critical*: will be non-zero if the extension is marked as critical (may be null)

This function retrieves the Alternative Name (2.5.29.17), contained in the given certificate in the X509v3 Certificate Extensions.

When the SAN type is `otherName`, it will extract the data in the `otherName`'s value field, and `GNUTLS_SAN_OTHERNAME` is returned. You may use `gnutls_x509_cert_get_subject_alt_othername_oid()` to get the corresponding OID and the "virtual" SAN types (e.g., `GNUTLS_SAN_OTHERNAME_XMPP` ).

If an `otherName` OID is known, the data will be decoded. Otherwise the returned data will be DER encoded, and you will have to decode it yourself. Currently, only the RFC 3920 `id-on-xmppAddr` SAN is recognized.

**Returns:** the alternative subject name type on success, one of the enumerated `gnutls_x509_subject_alt_name_t` . It will return `GNUTLS_E_SHORT_MEMORY_BUFFER` if *san\_size* is not large enough to hold the value. In that case *san\_size* will be updated with the required size. If the certificate does not have an Alternative name with the specified sequence number then `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned.

## gnutls\_x509\_cert\_get\_subject\_alt\_name2

```
int gnutls_x509_cert_get_subject_alt_name2 (gnutls_x509_cert_t      [Function]
                                              cert, unsigned int seq, void * san, size_t * san_size, unsigned int *
                                              san_type, unsigned int * critical)
```

*cert*: should contain a `gnutls_x509_cert_t` structure

*seq*: specifies the sequence number of the alt name (0 for the first one, 1 for the second etc.)

*san*: is the place where the alternative name will be copied to

*san\_size*: holds the size of ret.

*san\_type*: holds the type of the alternative name (one of `gnutls_x509_subject_alt_name_t`).

*critical*: will be non-zero if the extension is marked as critical (may be null)

This function will return the alternative names, contained in the given certificate. It is the same as `gnutls_x509_cert_get_subject_alt_name()` except for the fact that it will return the type of the alternative name in *san\_type* even if the function fails for some reason (i.e. the buffer provided is not enough).

**Returns:** the alternative subject name type on success, one of the enumerated `gnutls_x509_subject_alt_name_t`. It will return `GNUTLS_E_SHORT_MEMORY_BUFFER` if *san\_size* is not large enough to hold the value. In that case *san\_size* will be updated with the required size. If the certificate does not have an Alternative name with the specified sequence number then `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned.

## `gnutls_x509_cert_get_subject_alt_othername_oid`

`int gnutls_x509_cert_get_subject_alt_othername_oid` [Function]

(*gnutls\_x509\_cert\_t cert*, *unsigned int seq*, *void \* oid*, *size\_t \* oid\_size*)

*cert*: should contain a `gnutls_x509_cert_t` structure

*seq*: specifies the sequence number of the alt name (0 for the first one, 1 for the second etc.)

*oid*: is the place where the otherName OID will be copied to

*oid\_size*: holds the size of ret.

This function will extract the type OID of an otherName Subject Alternative Name, contained in the given certificate, and return the type as an enumerated element.

This function is only useful if `gnutls_x509_cert_get_subject_alt_name()` returned `GNUTLS_SAN_OTHERNAME`.

If *oid* is null then only the size will be filled. The *oid* returned will be null terminated, although *oid\_size* will not account for the trailing null.

**Returns:** the alternative subject name type on success, one of the enumerated `gnutls_x509_subject_alt_name_t`. For supported OIDs, it will return one of the virtual (`GNUTLS_SAN_OTHERNAME_*`) types, e.g. `GNUTLS_SAN_OTHERNAME_XMPP`, and `GNUTLS_SAN_OTHERNAME` for unknown OIDs. It will return `GNUTLS_E_SHORT_MEMORY_BUFFER` if *oid\_size* is not large enough to hold the value. In that case *oid\_size* will be updated with the required size. If the certificate does not have an Alternative name with the specified sequence number and with the otherName type then `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned.

## `gnutls_x509_cert_get_subject_key_id`

`int gnutls_x509_cert_get_subject_key_id` (*gnutls\_x509\_cert\_t cert*, [Function]

*void \* ret*, *size\_t \* ret\_size*, *unsigned int \* critical*)

*cert*: should contain a `gnutls_x509_cert_t` structure

*ret*: The place where the identifier will be copied

*ret\_size*: Holds the size of the result field.

*critical*: will be non-zero if the extension is marked as critical (may be null)

This function will return the X.509v3 certificate's subject key identifier. This is obtained by the X.509 Subject Key identifier extension field (2.5.29.14).

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE if the extension is not present, otherwise a negative error value.

## gnutls\_x509\_cert\_get\_subject\_unique\_id

**int gnutls\_x509\_cert\_get\_subject\_unique\_id** (*gnutls\_x509\_cert\_t crt*, *char \* buf*, *size\_t \* buf\_size*) [Function]

*crt*: Holds the certificate

*buf*: user allocated memory buffer, will hold the unique id

*buf\_size*: size of user allocated memory buffer (on input), will hold actual size of the unique ID on return.

This function will extract the subjectUniqueID value (if present) for the given certificate.

If the user allocated memory buffer is not large enough to hold the full subjectUniqueID, then a GNUTLS\_E\_SHORT\_MEMORY\_BUFFER error will be returned, and *buf\_size* will be set to the actual length.

**Returns:** GNUTLS\_E\_SUCCESS on success, otherwise a negative error code.

## gnutls\_x509\_cert\_get\_version

**int gnutls\_x509\_cert\_get\_version** (*gnutls\_x509\_cert\_t cert*) [Function]

*cert*: should contain a gnutls\_x509\_cert\_t structure

This function will return the version of the specified Certificate.

**Returns:** version of certificate, or a negative error code on error.

## gnutls\_x509\_cert\_import

**int gnutls\_x509\_cert\_import** (*gnutls\_x509\_cert\_t cert*, *const gnutls\_datum\_t \* data*, *gnutls\_x509\_cert\_fmt\_t format*) [Function]

*cert*: The structure to store the parsed certificate.

*data*: The DER or PEM encoded certificate.

*format*: One of DER or PEM

This function will convert the given DER or PEM encoded Certificate to the native gnutls\_x509\_cert\_t format. The output will be stored in *cert*.

If the Certificate is PEM encoded it should have a header of "X509 CERTIFICATE", or "CERTIFICATE".

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

## gnutls\_x509\_cert\_init

**int gnutls\_x509\_cert\_init** (*gnutls\_x509\_cert\_t \* cert*) [Function]

*cert*: The structure to be initialized

This function will initialize an X.509 certificate structure.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

## gnutls\_x509\_cert\_list\_import

**int gnutls\_x509\_cert\_list\_import** (*gnutls\_x509\_cert\_t \* certs*, [Function]  
*unsigned int \* cert\_max*, *const gnutls\_datum\_t \* data*, *gnutls\_x509\_cert\_fmt\_t format*, *unsigned int flags*)

*certs*: The structures to store the parsed certificate. Must not be initialized.

*cert\_max*: Initially must hold the maximum number of certs. It will be updated with the number of certs available.

*data*: The PEM encoded certificate.

*format*: One of DER or PEM.

*flags*: must be (0) or an OR'd sequence of gnutls\_certificate\_import\_flags.

This function will convert the given PEM encoded certificate list to the native gnutls\_x509\_cert\_t format. The output will be stored in *certs*. They will be automatically initialized.

The flag GNUTLS\_X509\_CERT\_LIST\_IMPORT\_FAIL\_IF\_EXCEED will cause import to fail if the certificates in the provided buffer are more than the available structures. The GNUTLS\_X509\_CERT\_LIST\_IMPORT\_FAIL\_IF\_UNSORTED flag will cause the function to fail if the provided list is not sorted from subject to issuer.

If the Certificate is PEM encoded it should have a header of "X509 CERTIFICATE", or "CERTIFICATE".

**Returns:** the number of certificates read or a negative error value.

## gnutls\_x509\_cert\_list\_import2

**int gnutls\_x509\_cert\_list\_import2** (*gnutls\_x509\_cert\_t \*\* certs*, [Function]  
*unsigned int \* size*, *const gnutls\_datum\_t \* data*, *gnutls\_x509\_cert\_fmt\_t format*, *unsigned int flags*)

*certs*: The structures to store the parsed certificate. Must not be initialized.

*size*: It will contain the size of the list.

*data*: The PEM encoded certificate.

*format*: One of DER or PEM.

*flags*: must be (0) or an OR'd sequence of gnutls\_certificate\_import\_flags.

This function will convert the given PEM encoded certificate list to the native gnutls\_x509\_cert\_t format. The output will be stored in *certs* which will be allocated and initialized.

If the Certificate is PEM encoded it should have a header of "X509 CERTIFICATE", or "CERTIFICATE".

To deinitialize `certs`, you need to deinitialize each `crt` structure independently, and use `gnutls_free()` at

**Returns:** the number of certificates read or a negative error value.

**Since:** 3.0

## **gnutls\_x509\_cert\_list\_verify**

```
int gnutls_x509_cert_list_verify (const gnutls_x509_cert_t *      [Function]
    cert_list, int cert_list_length, const gnutls_x509_cert_t * CA_list, int
    CA_list_length, const gnutls_x509_crl_t * CRL_list, int
    CRL_list_length, unsigned int flags, unsigned int * verify)
```

*cert\_list*: is the certificate list to be verified

*cert\_list\_length*: holds the number of certificate in *cert\_list*

*CA\_list*: is the CA list which will be used in verification

*CA\_list\_length*: holds the number of CA certificate in *CA\_list*

*CRL\_list*: holds a list of CRLs.

*CRL\_list\_length*: the length of CRL list.

*flags*: Flags that may be used to change the verification algorithm. Use OR of the `gnutls_certificate_verify_flags` enumerations.

*verify*: will hold the certificate verification output.

This function will try to verify the given certificate list and return its status. If no flags are specified (0), this function will use the basicConstraints (2.5.29.19) PKIX extension. This means that only a certificate authority is allowed to sign a certificate. You must also check the peer's name in order to check if the verified certificate belongs to the actual peer.

The certificate verification output will be put in *verify* and will be one or more of the `gnutls_certificate_status_t` enumerated elements bitwise or'd. For a more detailed verification status use `gnutls_x509_cert_verify()` per list element.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

## **gnutls\_x509\_cert\_print**

```
int gnutls_x509_cert_print (gnutls_x509_cert_t cert,          [Function]
    gnutls_certificate_print_formats_t format, gnutls_datum_t * out)
```

*cert*: The structure to be printed

*format*: Indicate the format to use

*out*: Newly allocated datum with null terminated string.

This function will pretty print a X.509 certificate, suitable for display to a human.

If the format is `GNUTLS_CERT_PRINT_FULL` then all fields of the certificate will be output, on multiple lines. The `GNUTLS_CERT_PRINT_ONELINE` format will generate one line with some selected fields, which is useful for logging purposes.

The output *out* needs to be deallocated using `gnutls_free()`.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.



### gnutls\_x509\_cert\_set\_activation\_time

`int gnutls_x509_cert_set_activation_time (gnutls_x509_cert_t cert, time_t act_time)` [Function]

*cert*: a certificate of type `gnutls_x509_cert_t`

*act\_time*: The actual time

This function will set the time this Certificate was or will be activated.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

### gnutls\_x509\_cert\_set\_authority\_info\_access

`int gnutls_x509_cert_set_authority_info_access (gnutls_x509_cert_t crt, int what, gnutls_datum_t * data)` [Function]

*crt*: Holds the certificate

*what*: what data to get, a `gnutls_info_access_what_t` type.

*data*: output data to be freed with `gnutls_free()` .

This function sets the Authority Information Access (AIA) extension, see RFC 5280 section 4.2.2.1 for more information.

The type of data stored in *data* is specified via *what* which should be `gnutls_info_access_what_t` values.

If *what* is `GNUTLS_IA_OCSP_URI` , *data* will hold the OCSP URI. If *what* is `GNUTLS_IA_CAISSUERS_URI` , *data* will hold the caIssuers URI.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.0

### gnutls\_x509\_cert\_set\_authority\_key\_id

`int gnutls_x509_cert_set_authority_key_id (gnutls_x509_cert_t cert, const void * id, size_t id_size)` [Function]

*cert*: a certificate of type `gnutls_x509_cert_t`

*id*: The key ID

*id\_size*: Holds the size of the key ID field.

This function will set the X.509 certificate's authority key ID extension. Only the `keyIdentifier` field can be set with this function.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

### gnutls\_x509\_cert\_set\_basic\_constraints

`int gnutls_x509_cert_set_basic_constraints (gnutls_x509_cert_t crt, unsigned int ca, int pathLenConstraint)` [Function]

*crt*: a certificate of type `gnutls_x509_cert_t`

*ca*: true(1) or false(0). Depending on the Certificate authority status.

*pathLenConstraint*: non-negative error codes indicate maximum length of path, and negative error codes indicate that the *pathLenConstraints* field should not be present. This function will set the *basicConstraints* certificate extension.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

### **gnutls\_x509\_cert\_set\_ca\_status**

`int gnutls_x509_cert_set_ca_status (gnutls_x509_cert_t crt, [Function]  
unsigned int ca)`

*crt*: a certificate of type `gnutls_x509_cert_t`

*ca*: true(1) or false(0). Depending on the Certificate authority status.

This function will set the *basicConstraints* certificate extension. Use `gnutls_x509_cert_set_basic_constraints()` if you want to control the *pathLenConstraint* field too.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

### **gnutls\_x509\_cert\_set\_crl\_dist\_points**

`int gnutls_x509_cert_set_crl_dist_points (gnutls_x509_cert_t crt, [Function]  
gnutls_x509_subject_alt_name_t type, const void * data_string, unsigned int  
reason_flags)`

*crt*: a certificate of type `gnutls_x509_cert_t`

*type*: is one of the `gnutls_x509_subject_alt_name_t` enumerations

*data\_string*: The data to be set

*reason\_flags*: revocation reasons

This function will set the CRL distribution points certificate extension.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

### **gnutls\_x509\_cert\_set\_crl\_dist\_points2**

`int gnutls_x509_cert_set_crl_dist_points2 (gnutls_x509_cert_t [Function]  
crt, gnutls_x509_subject_alt_name_t type, const void * data, unsigned int  
data_size, unsigned int reason_flags)`

*crt*: a certificate of type `gnutls_x509_cert_t`

*type*: is one of the `gnutls_x509_subject_alt_name_t` enumerations

*data*: The data to be set

*data\_size*: The data size

*reason\_flags*: revocation reasons

This function will set the CRL distribution points certificate extension.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 2.6.0

**gnutls\_x509\_cert\_set\_crq**

**int gnutls\_x509\_cert\_set\_crq** (*gnutls\_x509\_cert\_t crt*, [Function]  
*gnutls\_x509\_crq\_t crq*)

*crt*: a certificate of type `gnutls_x509_cert_t`

*crq*: holds a certificate request

This function will set the name and public parameters as well as the extensions from the given certificate request to the certificate. Only RSA keys are currently supported.

Note that this function will only set the `crq` if it is self signed and the signature is correct. See `gnutls_x509_crq_sign2()`.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**gnutls\_x509\_cert\_set\_crq\_extensions**

**int gnutls\_x509\_cert\_set\_crq\_extensions** (*gnutls\_x509\_cert\_t crt*, [Function]  
*gnutls\_x509\_crq\_t crq*)

*crt*: a certificate of type `gnutls_x509_cert_t`

*crq*: holds a certificate request

This function will set extensions from the given request to the certificate.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 2.8.0

**gnutls\_x509\_cert\_set\_dn**

**int gnutls\_x509\_cert\_set\_dn** (*gnutls\_x509\_cert\_t crt*, *const char \* dn*, [Function]  
*const char \*\* err*)

*crt*: a certificate of type `gnutls_x509_cert_t`

*dn*: a comma separated DN string (RFC4514)

*err*: indicates the error position (if any)

This function will set the DN on the provided certificate. The input string should be plain ASCII or UTF-8 encoded.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**gnutls\_x509\_cert\_set\_dn\_by\_oid**

**int gnutls\_x509\_cert\_set\_dn\_by\_oid** (*gnutls\_x509\_cert\_t crt*, *const* [Function]  
*char \* oid*, *unsigned int raw\_flag*, *const void \* name*, *unsigned int*  
*sizeof\_name*)

*crt*: a certificate of type `gnutls_x509_cert_t`

*oid*: holds an Object Identifier in a null terminated string

*raw\_flag*: must be 0, or 1 if the data are DER encoded

*name*: a pointer to the name

*sizeof\_name*: holds the size of **name**

This function will set the part of the name of the Certificate subject, specified by the given OID. The input string should be ASCII or UTF-8 encoded.

Some helper macros with popular OIDs can be found in `gnutls/x509.h`. With this function you can only set the known OIDs. You can test for known OIDs using `gnutls_x509_dn_oid_known()`. For OIDs that are not known (by gnutls) you should properly DER encode your data, and call this function with `raw_flag` set.

**Returns:** On success, `GNUTLS_E_SUCCESS (0)` is returned, otherwise a negative error value.

### **gnutls\_x509\_cert\_set\_expiration\_time**

```
int gnutls_x509_cert_set_expiration_time (gnutls_x509_cert_t cert, time_t exp_time) [Function]
```

*cert*: a certificate of type `gnutls_x509_cert_t`

*exp\_time*: The actual time

This function will set the time this Certificate will expire. Setting an expiration time to `(time_t)-1` or to `GNUTLS_X509_NO_WELL_DEFINED_EXPIRATION` will set to the no well-defined expiration date value.

**Returns:** On success, `GNUTLS_E_SUCCESS (0)` is returned, otherwise a negative error value.

### **gnutls\_x509\_cert\_set\_extension\_by\_oid**

```
int gnutls_x509_cert_set_extension_by_oid (gnutls_x509_cert_t cert, const char * oid, const void * buf, size_t sizeof_buf, unsigned int critical) [Function]
```

*cert*: a certificate of type `gnutls_x509_cert_t`

*oid*: holds an Object Identified in null terminated string

*buf*: a pointer to a DER encoded data

*sizeof\_buf*: holds the size of *buf*

*critical*: should be non-zero if the extension is to be marked as critical

This function will set an the extension, by the specified OID, in the certificate. The extension data should be binary data DER encoded.

**Returns:** On success, `GNUTLS_E_SUCCESS (0)` is returned, otherwise a negative error value.

### **gnutls\_x509\_cert\_set\_issuer\_alt\_name**

```
int gnutls_x509_cert_set_issuer_alt_name (gnutls_x509_cert_t cert, gnutls_x509_subject_alt_name_t type, const void * data, unsigned int data_size, unsigned int flags) [Function]
```

*cert*: a certificate of type `gnutls_x509_cert_t`

*type*: is one of the `gnutls_x509_subject_alt_name_t` enumerations

*data*: The data to be set

*data\_size*: The size of data to be set

*flags*: GNUTLS\_FSAN\_SET to clear previous data or GNUTLS\_FSAN\_APPEND to append.

This function will set the issuer alternative name certificate extension. It can set the same types as `gnutls_x509_cert_set_subject_alt_name()` .

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

## gnutls\_x509\_cert\_set\_issuer\_dn

```
int gnutls_x509_cert_set_issuer_dn (gnutls_x509_cert_t crt, const [Function]
    char * dn, const char ** err)
```

*crt*: a certificate of type `gnutls_x509_cert_t`

*dn*: a comma separated DN string (RFC4514)

*err*: indicates the error position (if any)

This function will set the DN on the provided certificate. The input string should be plain ASCII or UTF-8 encoded.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

## gnutls\_x509\_cert\_set\_issuer\_dn\_by\_oid

```
int gnutls_x509_cert_set_issuer_dn_by_oid (gnutls_x509_cert_t [Function]
    crt, const char * oid, unsigned int raw_flag, const void * name, unsigned int
    sizeof_name)
```

*crt*: a certificate of type `gnutls_x509_cert_t`

*oid*: holds an Object Identifier in a null terminated string

*raw\_flag*: must be 0, or 1 if the data are DER encoded

*name*: a pointer to the name

*sizeof\_name*: holds the size of *name*

This function will set the part of the name of the Certificate issuer, specified by the given OID. The input string should be ASCII or UTF-8 encoded.

Some helper macros with popular OIDs can be found in `gnutls/x509.h` With this function you can only set the known OIDs. You can test for known OIDs using `gnutls_x509_dn_oid_known()` . For OIDs that are not known (by gnutls) you should properly DER encode your data, and call this function with *raw\_flag* set.

Normally you do not need to call this function, since the signing operation will copy the signer's name as the issuer of the certificate.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**gnutls\_x509\_cert\_set\_key**

**int gnutls\_x509\_cert\_set\_key** (*gnutls\_x509\_cert\_t crt,* [Function]  
*gnutls\_x509\_privkey\_t key*)

*crt*: a certificate of type **gnutls\_x509\_cert\_t**

*key*: holds a private key

This function will set the public parameters from the given private key to the certificate. Only RSA keys are currently supported.

**Returns:** On success, **GNUTLS\_E\_SUCCESS** (0) is returned, otherwise a negative error value.

**gnutls\_x509\_cert\_set\_key\_purpose\_oid**

**int gnutls\_x509\_cert\_set\_key\_purpose\_oid** (*gnutls\_x509\_cert\_t* [Function]  
*cert, const void \* oid, unsigned int critical*)

*cert*: a certificate of type **gnutls\_x509\_cert\_t**

*oid*: a pointer to a null terminated string that holds the OID

*critical*: Whether this extension will be critical or not

This function will set the key purpose OIDs of the Certificate. These are stored in the Extended Key Usage extension (2.5.29.37) See the **GNUTLS\_KP\_\*** definitions for human readable names.

Subsequent calls to this function will append OIDs to the OID list.

**Returns:** On success, **GNUTLS\_E\_SUCCESS** (0) is returned, otherwise a negative error code is returned.

**gnutls\_x509\_cert\_set\_key\_usage**

**int gnutls\_x509\_cert\_set\_key\_usage** (*gnutls\_x509\_cert\_t crt,* [Function]  
*unsigned int usage*)

*crt*: a certificate of type **gnutls\_x509\_cert\_t**

*usage*: an ORed sequence of the **GNUTLS\_KEY\_\*** elements.

This function will set the keyUsage certificate extension.

**Returns:** On success, **GNUTLS\_E\_SUCCESS** (0) is returned, otherwise a negative error value.

**gnutls\_x509\_cert\_set\_name\_constraints**

**int gnutls\_x509\_cert\_set\_name\_constraints** (*gnutls\_x509\_cert\_t* [Function]  
*cert, gnutls\_x509\_name\_constraints\_t nc, unsigned int critical*)

*cert*: The certificate structure

*nc*: The nameconstraints structure

*critical*: whether this extension will be critical

This function will set the provided name constraints to the certificate extension list. This extension is always marked as critical.

**Returns:** On success, **GNUTLS\_E\_SUCCESS** (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

## gnutls\_x509\_cert\_set\_pin\_function

**void gnutls\_x509\_cert\_set\_pin\_function** (*gnutls\_x509\_cert\_t crt*, [Function]  
*gnutls\_pin\_callback\_t fn*, void \* *userdata*)

*crt*: The certificate structure

*fn*: the callback

*userdata*: data associated with the callback

This function will set a callback function to be used when it is required to access a protected object. This function overrides the global function set using `gnutls_pkcs11_set_pin_function()`.

Note that this callback is currently used only during the import of a PKCS 11 certificate with `gnutls_x509_cert_import_pkcs11_url()`.

**Since:** 3.1.0

## gnutls\_x509\_cert\_set\_policy

**int gnutls\_x509\_cert\_set\_policy** (*gnutls\_x509\_cert\_t crt*, const struct [Function]  
*gnutls\_x509\_policy\_st \*policy*, unsigned int *critical*)

*crt*: should contain a `gnutls_x509_cert_t` structure

*policy*: A pointer to a policy structure.

*critical*: use non-zero if the extension is marked as critical

This function will set the certificate policy extension (2.5.29.32). Multiple calls to this function append a new policy.

Note the maximum text size for the qualifier `GNUTLS_X509_QUALIFIER_NOTICE` is 200 characters. This function will fail with `GNUTLS_E_INVALID_REQUEST` if this is exceeded.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.1.5

## gnutls\_x509\_cert\_set\_private\_key\_usage\_period

**int gnutls\_x509\_cert\_set\_private\_key\_usage\_period** [Function]  
(*gnutls\_x509\_cert\_t crt*, time\_t *activation*, time\_t *expiration*)

*crt*: a certificate of type `gnutls_x509_cert_t`

*activation*: The activation time

*expiration*: The expiration time

This function will set the private key usage period extension (2.5.29.16).

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**gnutls\_x509\_cert\_set\_proxy**

```
int gnutls_x509_cert_set_proxy (gnutls_x509_cert_t cert, int [Function]
                               pathLenConstraint, const char * policyLanguage, const char * policy,
                               size_t sizeof_policy)
```

*cert*: a certificate of type `gnutls_x509_cert_t`

*pathLenConstraint*: non-negative error codes indicate maximum length of path, and negative error codes indicate that the `pathLenConstraints` field should not be present.

*policyLanguage*: OID describing the language of *policy* .

*policy*: `uint8_t` byte array with policy language, can be NULL

*sizeof\_policy*: size of *policy* .

This function will set the `proxyCertInfo` extension.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**gnutls\_x509\_cert\_set\_proxy\_dn**

```
int gnutls_x509_cert_set_proxy_dn (gnutls_x509_cert_t cert, [Function]
                                   gnutls_x509_cert_t eecert, unsigned int raw_flag, const void * name, unsigned
                                   int sizeof_name)
```

*cert*: a `gnutls_x509_cert_t` structure with the new proxy cert

*eecert*: the end entity certificate that will be issuing the proxy

*raw\_flag*: must be 0, or 1 if the CN is DER encoded

*name*: a pointer to the CN name, may be NULL (but MUST then be added later)

*sizeof\_name*: holds the size of *name*

This function will set the subject in *cert* to the end entity's *eecert* subject name, and add a single Common Name component *name* of size *sizeof\_name* . This corresponds to the required proxy certificate naming style. Note that if *name* is NULL , you MUST set it later by using `gnutls_x509_cert_set_dn_by_oid()` or similar.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**gnutls\_x509\_cert\_set\_serial**

```
int gnutls_x509_cert_set_serial (gnutls_x509_cert_t cert, const void [Function]
                                 * serial, size_t serial_size)
```

*cert*: a certificate of type `gnutls_x509_cert_t`

*serial*: The serial number

*serial\_size*: Holds the size of the serial field.

This function will set the X.509 certificate's serial number. While the serial number is an integer, it is often handled as an opaque field by several CAs. For this reason this function accepts any kind of data as a serial number. To be consistent with the X.509/PKIX specifications the provided *serial* should be a big-endian positive number (i.e. it's leftmost bit should be zero).

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.



**gnutls\_x509\_cert\_set\_subject\_alt\_name**

**int gnutls\_x509\_cert\_set\_subject\_alt\_name** (*gnutls\_x509\_cert\_t* *cert*, *gnutls\_x509\_subject\_alt\_name\_t* *type*, *const void \***data*, *unsigned int* *data\_size*, *unsigned int* *flags*) [Function]

*cert*: a certificate of type *gnutls\_x509\_cert\_t*

*type*: is one of the *gnutls\_x509\_subject\_alt\_name\_t* enumerations

*data*: The data to be set

*data\_size*: The size of data to be set

*flags*: GNUTLS\_FSAN\_SET to clear previous data or GNUTLS\_FSAN\_APPEND to append.

This function will set the subject alternative name certificate extension. It can set the following types:

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 2.6.0

**gnutls\_x509\_cert\_set\_subject\_alternative\_name**

**int gnutls\_x509\_cert\_set\_subject\_alternative\_name** (*gnutls\_x509\_cert\_t* *cert*, *gnutls\_x509\_subject\_alt\_name\_t* *type*, *const char \***data\_string*) [Function]

*cert*: a certificate of type *gnutls\_x509\_cert\_t*

*type*: is one of the *gnutls\_x509\_subject\_alt\_name\_t* enumerations

*data\_string*: The data to be set, a (0) terminated string

This function will set the subject alternative name certificate extension. This function assumes that data can be expressed as a null terminated string.

The name of the function is unfortunate since it is inconsistent with *gnutls\_x509\_cert\_get\_subject\_alt\_name()*.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**gnutls\_x509\_cert\_set\_subject\_key\_id**

**int gnutls\_x509\_cert\_set\_subject\_key\_id** (*gnutls\_x509\_cert\_t* *cert*, *const void \***id*, *size\_t* *id\_size*) [Function]

*cert*: a certificate of type *gnutls\_x509\_cert\_t*

*id*: The key ID

*id\_size*: Holds the size of the subject key ID field.

This function will set the X.509 certificate's subject key ID extension.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**gnutls\_x509\_cert\_set\_version**

**int gnutls\_x509\_cert\_set\_version** (*gnutls\_x509\_cert\_t crt, unsigned int version*) [Function]

*crt*: a certificate of type `gnutls_x509_cert_t`

*version*: holds the version number. For X.509v1 certificates must be 1.

This function will set the version of the certificate. This must be one for X.509 version 1, and so on. Plain certificates without extensions must have version set to one.

To create well-formed certificates, you must specify version 3 if you use any certificate extensions. Extensions are created by functions such as `gnutls_x509_cert_set_subject_alt_name()` or `gnutls_x509_cert_set_key_usage()`.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**gnutls\_x509\_cert\_sign**

**int gnutls\_x509\_cert\_sign** (*gnutls\_x509\_cert\_t crt, gnutls\_x509\_cert\_t issuer, gnutls\_x509\_privkey\_t issuer\_key*) [Function]

*crt*: a certificate of type `gnutls_x509_cert_t`

*issuer*: is the certificate of the certificate issuer

*issuer\_key*: holds the issuer's private key

This function is the same as `gnutls_x509_cert_sign2()` with no flags, and SHA1 as the hash algorithm.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**gnutls\_x509\_cert\_sign2**

**int gnutls\_x509\_cert\_sign2** (*gnutls\_x509\_cert\_t crt, gnutls\_x509\_cert\_t issuer, gnutls\_x509\_privkey\_t issuer\_key, gnutls\_digest\_algorithm\_t dig, unsigned int flags*) [Function]

*crt*: a certificate of type `gnutls_x509_cert_t`

*issuer*: is the certificate of the certificate issuer

*issuer\_key*: holds the issuer's private key

*dig*: The message digest to use, `GNUTLS_DIG_SHA1` is a safe choice

*flags*: must be 0

This function will sign the certificate with the issuer's private key, and will copy the issuer's information into the certificate.

This must be the last step in a certificate generation since all the previously set parameters are now signed.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

## gnutls\_x509\_cert\_verify

```
int gnutls_x509_cert_verify (gnutls_x509_cert_t cert, const [Function]
                             gnutls_x509_cert_t * CA_list, int CA_list_length, unsigned int flags,
                             unsigned int * verify)
```

*cert*: is the certificate to be verified

*CA\_list*: is one certificate that is considered to be trusted one

*CA\_list\_length*: holds the number of CA certificate in *CA\_list*

*flags*: Flags that may be used to change the verification algorithm. Use OR of the `gnutls_certificate_verify_flags` enumerations.

*verify*: will hold the certificate verification output.

This function will try to verify the given certificate and return its status. Note that a verification error does not imply a negative return status. In that case the `verify` status is set.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

## gnutls\_x509\_dn\_deinit

```
void gnutls_x509_dn_deinit (gnutls_x509_dn_t dn) [Function]
dn: a DN uint8_t object pointer.
```

This function deallocates the DN object as returned by `gnutls_x509_dn_import()`.

**Since:** 2.4.0

## gnutls\_x509\_dn\_export

```
int gnutls_x509_dn_export (gnutls_x509_dn_t dn, [Function]
                           gnutls_x509_cert_fmt_t format, void * output_data, size_t *
                           output_data_size)
```

*dn*: Holds the uint8\_t DN object

*format*: the format of output params. One of PEM or DER.

*output\_data*: will contain a DN PEM or DER encoded

*output\_data\_size*: holds the size of *output\_data* (and will be replaced by the actual size of parameters)

This function will export the DN to DER or PEM format.

If the buffer provided is not long enough to hold the output, then `*output_data_size` is updated and `GNUTLS_E_SHORT_MEMORY_BUFFER` will be returned.

If the structure is PEM encoded, it will have a header of "BEGIN NAME".

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**gnutls\_x509\_dn\_export2**

**int gnutls\_x509\_dn\_export2** (*gnutls\_x509\_dn\_t dn*, [Function]  
*gnutls\_x509\_cert\_fmt\_t format*, *gnutls\_datum\_t \* out*)

*dn*: Holds the uint8\_t DN object

*format*: the format of output params. One of PEM or DER.

*out*: will contain a DN PEM or DER encoded

This function will export the DN to DER or PEM format.

The output buffer is allocated using `gnutls_malloc()` .

If the structure is PEM encoded, it will have a header of "BEGIN NAME".

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.1.3

**gnutls\_x509\_dn\_get\_rdn\_ava**

**int gnutls\_x509\_dn\_get\_rdn\_ava** (*gnutls\_x509\_dn\_t dn*, *int irdn*, *int* [Function]  
*iava*, *gnutls\_x509\_ava\_st \* ava*)

*dn*: a pointer to DN

*irdn*: index of RDN

*iava*: index of AVA.

*ava*: Pointer to structure which will hold output information.

Get pointers to data within the DN. The format of the `ava` structure is shown below.

```
struct gnutls_x509_ava_st { gnutls_datum_t oid; gnutls_datum_t value; unsigned long
value_tag; };
```

The X.509 distinguished name is a sequence of sequences of strings and this is what the `irdn` and `iava` indexes model.

Note that `ava` will contain pointers into the `dn` structure which in turns points to the original certificate. Thus you should not modify any data or deallocate any of those.

This is a low-level function that requires the caller to do the value conversions when necessary (e.g. from UCS-2).

**Returns:** Returns 0 on success, or an error code.

**gnutls\_x509\_dn\_import**

**int gnutls\_x509\_dn\_import** (*gnutls\_x509\_dn\_t dn*, *const* [Function]  
*gnutls\_datum\_t \* data*)

*dn*: the structure that will hold the imported DN

*data*: should contain a DER encoded RDN sequence

This function parses an RDN sequence and stores the result to a `gnutls_x509_dn_t` structure. The structure must have been initialized with `gnutls_x509_dn_init()` .

You may use `gnutls_x509_dn_get_rdn_ava()` to decode the DN.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 2.4.0

**gnutls\_x509\_dn\_init**

**int gnutls\_x509\_dn\_init** (*gnutls\_x509\_dn\_t \* dn*) [Function]

*dn*: the object to be initialized

This function initializes a **gnutls\_x509\_dn\_t** structure.

The object returned must be deallocated using **gnutls\_x509\_dn\_deinit()** .

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 2.4.0

**gnutls\_x509\_dn\_oid\_known**

**int gnutls\_x509\_dn\_oid\_known** (*const char \* oid*) [Function]

*oid*: holds an Object Identifier in a null terminated string

This function will inform about known DN OIDs. This is useful since functions like **gnutls\_x509\_cert\_set\_dn\_by\_oid()** use the information on known OIDs to properly encode their input. Object Identifiers that are not known are not encoded by these functions, and their input is stored directly into the ASN.1 structure. In that case of unknown OIDs, you have the responsibility of DER encoding your data.

**Returns:** 1 on known OIDs and 0 otherwise.

**gnutls\_x509\_dn\_oid\_name**

**const char \* gnutls\_x509\_dn\_oid\_name** (*const char \* oid, unsigned int flags*) [Function]

*oid*: holds an Object Identifier in a null terminated string

*flags*: 0 or GNUTLS\_X509\_DN\_OID\_\*

This function will return the name of a known DN OID. If GNUTLS\_X509\_DN\_OID\_RETURN\_OID is specified this function will return the given OID if no descriptive name has been found.

**Returns:** A null terminated string or NULL otherwise.

**Since:** 3.0

**gnutls\_x509\_ext\_deinit**

**void gnutls\_x509\_ext\_deinit** (*gnutls\_x509\_ext\_st \* ext*) [Function]

*ext*: The extensions structure

This function will deinitialize an extensions structure.

**Since:** 3.3.8

**gnutls\_x509\_ext\_export\_aia**

**int gnutls\_x509\_ext\_export\_aia** (*gnutls\_x509\_aia\_t aia, gnutls\_datum\_t \* ext*) [Function]

*aia*: The authority info access structure

*ext*: The DER-encoded extension data; must be freed using **gnutls\_free()** .

This function will DER encode the Authority Information Access (AIA) extension; see RFC 5280 section 4.2.2.1 for more information on the extension.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

### `gnutls_x509_ext_export_authority_key_id`

`int gnutls_x509_ext_export_authority_key_id (gnutls_x509_aki_t aki, gnutls_datum_t * ext)` [Function]

*aki*: An initialized authority key identifier structure

*ext*: The DER-encoded extension data; must be freed using `gnutls_free()` .

This function will convert the provided key identifier to a DER-encoded PKIX AuthorityKeyIdentifier extension. The output data in *ext* will be allocated using `gnutls_malloc()` .

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

### `gnutls_x509_ext_export_basic_constraints`

`int gnutls_x509_ext_export_basic_constraints (unsigned int ca, int pathlen, gnutls_datum_t * ext)` [Function]

*ca*: non-zero for a CA

*pathlen*: The path length constraint (set to -1 for no constraint)

*ext*: The DER-encoded extension data; must be freed using `gnutls_free()` .

This function will convert the parameters provided to a basic constraints DER encoded extension (2.5.29.19). The *ext* data will be allocated using `gnutls_malloc()` .

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

### `gnutls_x509_ext_export_crl_dist_points`

`int gnutls_x509_ext_export_crl_dist_points (gnutls_x509_crl_dist_points_t cdp, gnutls_datum_t * ext)` [Function]

*cdp*: A pointer to an initialized CRL distribution points structure.

*ext*: The DER-encoded extension data; must be freed using `gnutls_free()` .

This function will convert the provided policies, to a certificate policy DER encoded extension (2.5.29.31).

The *ext* data will be allocated using `gnutls_malloc()` .

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

**gnutls\_x509\_ext\_export\_key\_purposes**

**int gnutls\_x509\_ext\_export\_key\_purposes** [Function]  
     (*gnutls\_x509\_key\_purposes\_t* *p*, *gnutls\_datum\_t* \* *ext*)

*p*: The key purposes structure

*ext*: The DER-encoded extension data; must be freed using **gnutls\_free()** .

This function will convert the key purposes structure to a DER-encoded PKIX ExtKeyUsageSyntax (2.5.29.37) extension. The output data in *ext* will be allocated using **gnutls\_malloc()** .

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

**gnutls\_x509\_ext\_export\_key\_usage**

**int gnutls\_x509\_ext\_export\_key\_usage** (*unsigned int* *usage*, [Function]  
     *gnutls\_datum\_t* \* *ext*)

*usage*: an ORed sequence of the GNUTLS\_KEY\_\* elements.

*ext*: The DER-encoded extension data; must be freed using **gnutls\_free()** .

This function will convert the keyUsage bit string to a DER encoded PKIX extension. The *ext* data will be allocated using **gnutls\_malloc()** .

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

**gnutls\_x509\_ext\_export\_name\_constraints**

**int gnutls\_x509\_ext\_export\_name\_constraints** [Function]  
     (*gnutls\_x509\_name\_constraints\_t* *nc*, *gnutls\_datum\_t* \* *ext*)

*nc*: The nameconstraints structure

*ext*: The DER-encoded extension data; must be freed using **gnutls\_free()** .

This function will convert the provided name constraints structure to a DER-encoded PKIX NameConstraints (2.5.29.30) extension. The output data in *ext* will be allocated using **gnutls\_malloc()** .

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

**gnutls\_x509\_ext\_export\_policies**

**int gnutls\_x509\_ext\_export\_policies** (*gnutls\_x509\_policies\_t* [Function]  
     *policies*, *gnutls\_datum\_t* \* *ext*)

*policies*: A pointer to an initialized policies structure.

*ext*: The DER-encoded extension data; must be freed using **gnutls\_free()** .

This function will convert the provided policies, to a certificate policy DER encoded extension (2.5.29.32).

The `ext` data will be allocated using `gnutls_malloc()` .

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

### **gnutls\_x509\_ext\_export\_private\_key\_usage\_period**

`int gnutls_x509_ext_export_private_key_usage_period (time_t [Function]  
activation, time_t expiration, gnutls_datum_t * ext)`

*activation*: The activation time

*expiration*: The expiration time

*ext*: The DER-encoded extension data; must be freed using `gnutls_free()` .

This function will convert the periods provided to a private key usage DER encoded extension (2.5.29.16). The `ext` data will be allocated using `gnutls_malloc()` .

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

### **gnutls\_x509\_ext\_export\_proxy**

`int gnutls_x509_ext_export_proxy (int pathLenConstraint, const [Function]  
char * policyLanguage, const char * policy, size_t sizeof_policy,  
gnutls_datum_t * ext)`

*pathLenConstraint*: non-negative error codes indicate maximum length of path, and negative error codes indicate that the *pathLenConstraints* field should not be present.

*policyLanguage*: OID describing the language of *policy* .

*policy*: `uint8_t` byte array with policy language, can be `NULL`

*sizeof\_policy*: size of *policy* .

*ext*: The DER-encoded extension data; must be freed using `gnutls_free()` .

This function will convert the parameters provided to a `proxyCertInfo` extension.

The `ext` data will be allocated using `gnutls_malloc()` .

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

### **gnutls\_x509\_ext\_export\_subject\_alt\_names**

`int gnutls_x509_ext_export_subject_alt_names [Function]  
(gnutls_subject_alt_names_t sans, gnutls_datum_t * ext)`

*sans*: The alternative names structure

*ext*: The DER-encoded extension data; must be freed using `gnutls_free()` .

This function will convert the provided alternative names structure to a DER-encoded `SubjectAltName` PKIX extension. The output data in `ext` will be allocated using `gnutls_malloc()` .



**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

### gnutls\_x509\_ext\_export\_subject\_key\_id

int gnutls\_x509\_ext\_export\_subject\_key\_id (const gnutls\_datum\_t \* id, gnutls\_datum\_t \* ext) [Function]

*id*: The key identifier

*ext*: The DER-encoded extension data; must be freed using `gnutls_free()` .

This function will convert the provided key identifier to a DER-encoded PKIX SubjectKeyIdentifier extension. The output data in *ext* will be allocated using `gnutls_malloc()` .

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

### gnutls\_x509\_ext\_import\_aia

int gnutls\_x509\_ext\_import\_aia (const gnutls\_datum\_t \* ext, gnutls\_x509\_aia\_t aia, unsigned int flags) [Function]

*ext*: The DER-encoded extension data

*aia*: The authority info access structure

*flags*: should be zero

This function extracts the Authority Information Access (AIA) extension from the provided DER-encoded data; see RFC 5280 section 4.2.2.1 for more information on the extension. The AIA extension holds a sequence of AccessDescription (AD) data.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

### gnutls\_x509\_ext\_import\_authority\_key\_id

int gnutls\_x509\_ext\_import\_authority\_key\_id (const gnutls\_datum\_t \* ext, gnutls\_x509\_aki\_t aki, unsigned int flags) [Function]

*ext*: a DER encoded extension

*aki*: An initialized authority key identifier structure

*flags*: should be zero

This function will return the subject key ID stored in the provided AuthorityKeyIdentifier extension.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE if the extension is not present, otherwise a negative error value.

**Since:** 3.3.0

**gnutls\_x509\_ext\_import\_basic\_constraints**

**int gnutls\_x509\_ext\_import\_basic\_constraints** (*const* [Function]  
           *gnutls\_datum\_t \* ext, unsigned int \* ca, int \* pathlen*)

*ext*: the DER encoded extension data

*ca*: will be non zero if the CA status is true

*pathlen*: the path length constraint; will be set to -1 for no limit

This function will return the CA status and path length constraint as written in the PKIX extension 2.5.29.19.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

**gnutls\_x509\_ext\_import\_crl\_dist\_points**

**int gnutls\_x509\_ext\_import\_crl\_dist\_points** (*const* [Function]  
           *gnutls\_datum\_t \* ext, gnutls\_x509\_crl\_dist\_points\_t cdp, unsigned int flags*)

*ext*: the DER encoded extension data

*cdp*: A pointer to an initialized CRL distribution points structure.

*flags*: should be zero

This function will extract the CRL distribution points extension (2.5.29.31) and store it into the provided structure.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

**gnutls\_x509\_ext\_import\_key\_purposes**

**int gnutls\_x509\_ext\_import\_key\_purposes** (*const gnutls\_datum\_t \* ext, gnutls\_x509\_key\_purposes\_t p, unsigned int flags*) [Function]

*ext*: The DER-encoded extension data

*p*: The key purposes structure

*flags*: should be zero

This function will extract the key purposes in the provided DER-encoded ExtKeyUsageSyntax PKIX extension, to a `gnutls_x509_key_purposes_t` structure. The structure must be initialized.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

**gnutls\_x509\_ext\_import\_key\_usage**

**int gnutls\_x509\_ext\_import\_key\_usage** (*const gnutls\_datum\_t \* ext, unsigned int \* key\_usage*) [Function]

*ext*: the DER encoded extension data

*key\_usage*: where the key usage bits will be stored

This function will return certificate's key usage, by reading the DER data of the keyUsage X.509 extension (2.5.29.15). The key usage value will ORed values of the: GNUTLS\_KEY\_DIGITAL\_SIGNATURE , GNUTLS\_KEY\_NON\_REPUDIATION , GNUTLS\_KEY\_KEY\_ENCIPHERMENT , GNUTLS\_KEY\_DATA\_ENCIPHERMENT , GNUTLS\_KEY\_KEY\_AGREEMENT , GNUTLS\_KEY\_KEY\_CERT\_SIGN , GNUTLS\_KEY\_CRL\_SIGN , GNUTLS\_KEY\_ENCIPHER\_ONLY , GNUTLS\_KEY\_DECIPHER\_ONLY .

**Returns:** the certificate key usage, or a negative error code in case of parsing error. If the certificate does not contain the keyUsage extension GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE will be returned.

**Since:** 3.3.0

### gnutls\_x509\_ext\_import\_name\_constraints

```
int gnutls_x509_ext_import_name_constraints (const gnutls_datum_t * ext, gnutls_x509_name_constraints_t nc, unsigned int flags) [Function]
```

*ext*: a DER encoded extension

*nc*: The nameconstraints intermediate structure

*flags*: zero or GNUTLS\_NAME\_CONSTRAINTS\_FLAG\_APPEND

This function will return an intermediate structure containing the name constraints of the provided NameConstraints extension. That structure can be used in combination with `gnutls_x509_name_constraints_check()` to verify whether a server's name is in accordance with the constraints.

When the *flags* is set to GNUTLS\_NAME\_CONSTRAINTS\_FLAG\_APPEND , then if the *nc* structure is empty this function will behave identically as if the flag was not set. Otherwise if there are elements in the *nc* structure then only the excluded constraints will be appended to the constraints.

Note that *nc* must be initialized prior to calling this function.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE if the extension is not present, otherwise a negative error value.

**Since:** 3.3.0

### gnutls\_x509\_ext\_import\_policies

```
int gnutls_x509_ext_import_policies (const gnutls_datum_t * ext, gnutls_x509_policies_t policies, unsigned int flags) [Function]
```

*ext*: the DER encoded extension data

*policies*: A pointer to an initialized policies structures.

*flags*: should be zero

This function will extract the certificate policy extension (2.5.29.32) and store it the provided structure.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

**gnutls\_x509\_ext\_import\_private\_key\_usage\_period**

```
int gnutls_x509_ext_import_private_key_usage_period (const [Function]
    gnutls_datum_t * ext, time_t * activation, time_t * expiration)
```

*ext*: the DER encoded extension data

*activation*: Will hold the activation time

*expiration*: Will hold the expiration time

This function will return the expiration and activation times of the private key as written in the PKIX extension 2.5.29.16.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

**gnutls\_x509\_ext\_import\_proxy**

```
int gnutls_x509_ext_import_proxy (const gnutls_datum_t * ext, int [Function]
    * pathlen, char ** policyLanguage, char ** policy, size_t *
    sizeof_policy)
```

*ext*: the DER encoded extension data

*pathlen*: pointer to output integer indicating path length (may be NULL), non-negative error codes indicate a present pCPathLenConstraint field and the actual value, -1 indicate that the field is absent.

*policyLanguage*: output variable with OID of policy language

*policy*: output variable with policy data

*sizeof\_policy*: output variable size of policy data

This function will return the information from a proxy certificate extension. It reads the ProxyCertInfo X.509 extension (1.3.6.1.5.5.7.1.14). The *policyLanguage* and *policy* values must be deinitialized using *gnutls\_free()* after use.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

**gnutls\_x509\_ext\_import\_subject\_alt\_names**

```
int gnutls_x509_ext_import_subject_alt_names (const [Function]
    gnutls_datum_t * ext, gnutls_subject_alt_names_t sans, unsigned int flags)
```

*ext*: The DER-encoded extension data

*sans*: The alternative names structure

*flags*: should be zero

This function will export the alternative names in the provided DER-encoded SubjectAltName PKIX extension, to a *gnutls\_subject\_alt\_names\_t* structure. The structure must have been initialized.

This function will succeed even if there no subject alternative names in the structure.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

**gnutls\_x509\_ext\_import\_subject\_key\_id**

**int gnutls\_x509\_ext\_import\_subject\_key\_id** (*const* [Function]  
           *gnutls\_datum\_t \* ext, gnutls\_datum\_t \* id*)

*ext*: a DER encoded extension

*id*: will contain the subject key ID

This function will return the subject key ID stored in the provided SubjectKeyIdentifier extension. The ID will be allocated using `gnutls_malloc()` .

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if the extension is not present, otherwise a negative error value.

**Since:** 3.3.0

**gnutls\_x509\_ext\_print**

**int gnutls\_x509\_ext\_print** (*gnutls\_x509\_ext\_st \* exts, unsigned int* [Function]  
           *exts\_size, gnutls\_certificate\_print\_formats\_t format, gnutls\_datum\_t \* out*)

*exts*: The structures to be printed

*exts\_size*: the number of available structures

*format*: Indicate the format to use

*out*: Newly allocated datum with null terminated string.

This function will pretty print X.509 certificate extensions, suitable for display to a human.

The output *out* needs to be deallocated using `gnutls_free()` .

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**gnutls\_x509\_key\_purpose\_deinit**

**void gnutls\_x509\_key\_purpose\_deinit** (*gnutls\_x509\_key\_purposes\_t* [Function]  
           *p*)

*p*: The key purposes structure

This function will deinitialize an alternative names structure.

**Since:** 3.3.0

**gnutls\_x509\_key\_purpose\_get**

**int gnutls\_x509\_key\_purpose\_get** (*gnutls\_x509\_key\_purposes\_t p,* [Function]  
           *unsigned idx, gnutls\_datum\_t \* oid*)

*p*: The key purposes structure

*idx*: The index of the key purpose to retrieve

*oid*: Will hold the object identifier of the key purpose (to be treated as constant)

This function will retrieve the specified by the index key purpose in the purposes structure. The object identifier will be a null terminated string.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if the index is out of bounds, otherwise a negative error value.

**Since:** 3.3.0

**gnutls\_x509\_key\_purpose\_init**

**int gnutls\_x509\_key\_purpose\_init** (*gnutls\_x509\_key\_purposes\_t \*p*) [Function]

*p*: The key purposes structure

This function will initialize an alternative names structure.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

**gnutls\_x509\_key\_purpose\_set**

**int gnutls\_x509\_key\_purpose\_set** (*gnutls\_x509\_key\_purposes\_t p*, [Function]  
*const char \*oid*)

*p*: The key purposes structure

*oid*: The object identifier of the key purpose

This function will store the specified key purpose in the purposes structure.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0), otherwise a negative error value.

**Since:** 3.3.0

**gnutls\_x509\_name\_constraints\_add\_excluded**

**int gnutls\_x509\_name\_constraints\_add\_excluded** [Function]  
(*gnutls\_x509\_name\_constraints\_t nc*, *gnutls\_x509\_subject\_alt\_name\_t type*,  
*const gnutls\_datum\_t \*name*)

*nc*: The nameconstraints structure

*type*: The type of the constraints

*name*: The data of the constraints

This function will add a name constraint to the list of excluded constraints.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

**gnutls\_x509\_name\_constraints\_add\_permitted**

**int gnutls\_x509\_name\_constraints\_add\_permitted** [Function]  
(*gnutls\_x509\_name\_constraints\_t nc*, *gnutls\_x509\_subject\_alt\_name\_t type*,  
*const gnutls\_datum\_t \*name*)

*nc*: The nameconstraints structure

*type*: The type of the constraints

*name*: The data of the constraints

This function will add a name constraint to the list of permitted constraints.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

## gnutls\_x509\_name\_constraints\_check

unsigned gnutls\_x509\_name\_constraints\_check [Function]

(gnutls\_x509\_name\_constraints\_t *nc*, gnutls\_x509\_subject\_alt\_name\_t *type*,  
const gnutls\_datum\_t \* *name*)

*nc*: the extracted name constraints structure

*type*: the type of the constraint to check (of type gnutls\_x509\_subject\_alt\_name\_t)

*name*: the name to be checked

This function will check the provided name against the constraints in *nc* using the RFC5280 rules. Currently this function is limited to DNS names and emails (of type GNUTLS\_SAN\_DNSNAME and GNUTLS\_SAN\_RFC822NAME ).

**Returns:** zero if the provided name is not acceptable, and non-zero otherwise.

**Since:** 3.3.0

## gnutls\_x509\_name\_constraints\_check\_cert

unsigned gnutls\_x509\_name\_constraints\_check\_cert [Function]

(gnutls\_x509\_name\_constraints\_t *nc*, gnutls\_x509\_subject\_alt\_name\_t *type*,  
gnutls\_x509\_cert\_t *cert*)

*nc*: the extracted name constraints structure

*type*: the type of the constraint to check (of type gnutls\_x509\_subject\_alt\_name\_t)

*cert*: the certificate to be checked

This function will check the provided certificate names against the constraints in *nc* using the RFC5280 rules. It will traverse all the certificate's names and alternative names.

Currently this function is limited to DNS names and emails (of type GNUTLS\_SAN\_DNSNAME and GNUTLS\_SAN\_RFC822NAME ).

**Returns:** zero if the provided name is not acceptable, and non-zero otherwise.

**Since:** 3.3.0

## gnutls\_x509\_name\_constraints\_deinit

void gnutls\_x509\_name\_constraints\_deinit [Function]

(gnutls\_x509\_name\_constraints\_t *nc*)

*nc*: The nameconstraints structure

This function will deinitialize a name constraints structure.

**Since:** 3.3.0

## gnutls\_x509\_name\_constraints\_get\_excluded

int gnutls\_x509\_name\_constraints\_get\_excluded [Function]

(gnutls\_x509\_name\_constraints\_t *nc*, unsigned *idx*, unsigned \* *type*,  
gnutls\_datum\_t \* *name*)

*nc*: the extracted name constraints structure

*idx*: the index of the constraint

*type*: the type of the constraint (of type `gnutls_x509_subject_alt_name_t`)

*name*: the name in the constraint (of the specific type)

This function will return an intermediate structure containing the name constraints of the provided CA certificate. That structure can be used in combination with `gnutls_x509_name_constraints_check()` to verify whether a server's name is in accordance with the constraints.

The name should be treated as constant and valid for the lifetime of `nc`.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if the extension is not present, otherwise a negative error value.

**Since:** 3.3.0

### `gnutls_x509_name_constraints_get_permitted`

```
int gnutls_x509_name_constraints_get_permitted [Function]
    (gnutls_x509_name_constraints_t nc, unsigned idx, unsigned * type,
     gnutls_datum_t * name)
```

*nc*: the extracted name constraints structure

*idx*: the index of the constraint

*type*: the type of the constraint (of type `gnutls_x509_subject_alt_name_t`)

*name*: the name in the constraint (of the specific type)

This function will return an intermediate structure containing the name constraints of the provided CA certificate. That structure can be used in combination with `gnutls_x509_name_constraints_check()` to verify whether a server's name is in accordance with the constraints.

The name should be treated as constant and valid for the lifetime of `nc`.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if the extension is not present, otherwise a negative error value.

**Since:** 3.3.0

### `gnutls_x509_name_constraints_init`

```
int gnutls_x509_name_constraints_init [Function]
    (gnutls_x509_name_constraints_t * nc)
```

*nc*: The nameconstraints structure

This function will initialize a name constraints structure.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

### `gnutls_x509_othername_to_virtual`

```
int gnutls_x509_othername_to_virtual (const char * oid, const [Function]
    gnutls_datum_t * othername, unsigned int * virt_type, gnutls_datum_t *
    virt)
```

*oid*: The othername object identifier



*othername*: – undescribed –

*virt\_type*: GNUTLS\_SAN\_OTHERNAME\_XXX

*virt*: allocated printable data

This function will parse and convert the *othername* data to a virtual type supported by gnutls.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.3.8

## gnutls\_x509\_policies\_deinit

```
void gnutls_x509_policies_deinit (gnutls_x509_policies_t policies) [Function]
```

*policies*: The authority key identifier structure

This function will deinitialize an authority key identifier structure.

**Since:** 3.3.0

## gnutls\_x509\_policies\_get

```
int gnutls_x509_policies_get (gnutls_x509_policies_t policies, unsigned int seq, struct gnutls_x509_policy_st *policy) [Function]
```

*policies*: The policies structure

*seq*: The index of the name to get

*policy*: Will hold the policy

This function will return a specific policy as stored in the *policies* structure. The returned values should be treated as constant and valid for the lifetime of *policies*.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE if the index is out of bounds, otherwise a negative error value.

**Since:** 3.3.0

## gnutls\_x509\_policies\_init

```
int gnutls_x509_policies_init (gnutls_x509_policies_t *policies) [Function]
```

*policies*: The authority key ID structure

This function will initialize an authority key ID structure.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

## gnutls\_x509\_policies\_set

```
int gnutls_x509_policies_set (gnutls_x509_policies_t policies, const struct gnutls_x509_policy_st *policy) [Function]
```

*policies*: An initialized policies structure

*policy*: Contains the policy to set

This function will store the specified policy in the provided `policies` structure.

**Returns:** On success, `GNUTLS_E_SUCCESS (0)`, otherwise a negative error value.

**Since:** 3.3.0

## **gnutls\_x509\_policy\_release**

```
void gnutls_x509_policy_release (struct gnutls_x509_policy_st *      [Function]
                                policy)
```

*policy*: a certificate policy

This function will deinitialize all memory associated with the provided `policy`. The policy is allocated using `gnutls_x509_cert_get_policy()`.

**Since:** 3.1.5

## **gnutls\_x509\_privkey\_cpy**

```
int gnutls_x509_privkey_cpy (gnutls_x509_privkey_t dst,           [Function]
                             gnutls_x509_privkey_t src)
```

*dst*: The destination key, which should be initialized.

*src*: The source key

This function will copy a private key from source to destination key. Destination has to be initialized.

**Returns:** On success, `GNUTLS_E_SUCCESS (0)` is returned, otherwise a negative error value.

## **gnutls\_x509\_privkey\_deinit**

```
void gnutls_x509_privkey_deinit (gnutls_x509_privkey_t key)      [Function]
```

*key*: The structure to be deinitialized

This function will deinitialize a private key structure.

## **gnutls\_x509\_privkey\_export**

```
int gnutls_x509_privkey_export (gnutls_x509_privkey_t key,       [Function]
                                gnutls_x509_cert_fmt_t format, void * output_data, size_t *
                                output_data_size)
```

*key*: Holds the key

*format*: the format of output params. One of PEM or DER.

*output\_data*: will contain a private key PEM or DER encoded

*output\_data\_size*: holds the size of *output\_data* (and will be replaced by the actual size of parameters)

This function will export the private key to a PKCS1 structure for RSA keys, or an integer sequence for DSA keys. The DSA keys are in the same format with the parameters used by openssl.

If the buffer provided is not long enough to hold the output, then `* output_data_size` is updated and `GNUTLS_E_SHORT_MEMORY_BUFFER` will be returned.

If the structure is PEM encoded, it will have a header of "BEGIN RSA PRIVATE KEY".

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

## gnutls\_x509\_privkey\_export2

```
int gnutls_x509_privkey_export2 (gnutls_x509_privkey_t key,          [Function]
                                gnutls_x509_crt_fmt_t format, gnutls_datum_t * out)
```

*key*: Holds the key

*format*: the format of output params. One of PEM or DER.

*out*: will contain a private key PEM or DER encoded

This function will export the private key to a PKCS1 structure for RSA keys, or an integer sequence for DSA keys. The DSA keys are in the same format with the parameters used by openssl.

The output buffer is allocated using `gnutls_malloc()` .

If the structure is PEM encoded, it will have a header of "BEGIN RSA PRIVATE KEY".

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

Since 3.1.3

## gnutls\_x509\_privkey\_export2\_pkcs8

```
int gnutls_x509_privkey_export2_pkcs8 (gnutls_x509_privkey_t      [Function]
                                         key, gnutls_x509_crt_fmt_t format, const char * password, unsigned int
                                         flags, gnutls_datum_t * out)
```

*key*: Holds the key

*format*: the format of output params. One of PEM or DER.

*password*: the password that will be used to encrypt the key.

*flags*: an ORed sequence of `gnutls_pkcs_encrypt_flags_t`

*out*: will contain a private key PEM or DER encoded

This function will export the private key to a PKCS8 structure. Both RSA and DSA keys can be exported. For DSA keys we use PKCS 11 definitions. If the flags do not specify the encryption cipher, then the default 3DES (PBES2) will be used.

The `password` can be either ASCII or UTF-8 in the default PBES2 encryption schemas, or ASCII for the PKCS12 schemas.

The output buffer is allocated using `gnutls_malloc()` .

If the structure is PEM encoded, it will have a header of "BEGIN ENCRYPTED PRIVATE KEY" or "BEGIN PRIVATE KEY" if encryption is not used.

**Returns:** In case of failure a negative error code will be returned, and 0 on success.

Since 3.1.3

**gnutls\_x509\_privkey\_export\_dsa\_raw**

**int gnutls\_x509\_privkey\_export\_dsa\_raw** (*gnutls\_x509\_privkey\_t* *key*, *gnutls\_datum\_t* \* *p*, *gnutls\_datum\_t* \* *q*, *gnutls\_datum\_t* \* *g*, *gnutls\_datum\_t* \* *y*, *gnutls\_datum\_t* \* *x*) [Function]

*key*: a structure that holds the DSA parameters

*p*: will hold the p

*q*: will hold the q

*g*: will hold the g

*y*: will hold the y

*x*: will hold the x

This function will export the DSA private key's parameters found in the given structure. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**gnutls\_x509\_privkey\_export\_ecc\_raw**

**int gnutls\_x509\_privkey\_export\_ecc\_raw** (*gnutls\_x509\_privkey\_t* *key*, *gnutls\_ecc\_curve\_t* \* *curve*, *gnutls\_datum\_t* \* *x*, *gnutls\_datum\_t* \* *y*, *gnutls\_datum\_t* \* *k*) [Function]

*key*: a structure that holds the rsa parameters

*curve*: will hold the curve

*x*: will hold the x coordinate

*y*: will hold the y coordinate

*k*: will hold the private key

This function will export the ECC private key's parameters found in the given structure. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.0

**gnutls\_x509\_privkey\_export\_pkcs8**

**int gnutls\_x509\_privkey\_export\_pkcs8** (*gnutls\_x509\_privkey\_t* *key*, *gnutls\_x509\_crt\_fmt\_t* *format*, *const char* \* *password*, *unsigned int* *flags*, *void* \* *output\_data*, *size\_t* \* *output\_data\_size*) [Function]

*key*: Holds the key

*format*: the format of output params. One of PEM or DER.

*password*: the password that will be used to encrypt the key.

*flags*: an ORed sequence of `gnutls_pkcs_encrypt_flags_t`

*output\_data*: will contain a private key PEM or DER encoded

*output\_data\_size*: holds the size of *output\_data* (and will be replaced by the actual size of parameters)

This function will export the private key to a PKCS8 structure. Both RSA and DSA keys can be exported. For DSA keys we use PKCS 11 definitions. If the flags do not specify the encryption cipher, then the default 3DES (PBES2) will be used.

The *password* can be either ASCII or UTF-8 in the default PBES2 encryption schemas, or ASCII for the PKCS12 schemas.

If the buffer provided is not long enough to hold the output, then *\*output\_data\_size* is updated and `GNUTLS_E_SHORT_MEMORY_BUFFER` will be returned.

If the structure is PEM encoded, it will have a header of "BEGIN ENCRYPTED PRIVATE KEY" or "BEGIN PRIVATE KEY" if encryption is not used.

**Returns:** In case of failure a negative error code will be returned, and 0 on success.

### **gnutls\_x509\_privkey\_export\_rsa\_raw**

```
int gnutls_x509_privkey_export_rsa_raw (gnutls_x509_privkey_t      [Function]
    key, gnutls_datum_t * m, gnutls_datum_t * e, gnutls_datum_t * d,
    gnutls_datum_t * p, gnutls_datum_t * q, gnutls_datum_t * u)
```

*key*: a structure that holds the rsa parameters

*m*: will hold the modulus

*e*: will hold the public exponent

*d*: will hold the private exponent

*p*: will hold the first prime (p)

*q*: will hold the second prime (q)

*u*: will hold the coefficient

This function will export the RSA private key's parameters found in the given structure. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

### **gnutls\_x509\_privkey\_export\_rsa\_raw2**

```
int gnutls_x509_privkey_export_rsa_raw2 (gnutls_x509_privkey_t      [Function]
    key, gnutls_datum_t * m, gnutls_datum_t * e, gnutls_datum_t * d,
    gnutls_datum_t * p, gnutls_datum_t * q, gnutls_datum_t * u, gnutls_datum_t *
    e1, gnutls_datum_t * e2)
```

*key*: a structure that holds the rsa parameters

*m*: will hold the modulus

*e*: will hold the public exponent

*d*: will hold the private exponent

*p*: will hold the first prime (p)

*q*: will hold the second prime (q)

*u*: will hold the coefficient  
*e1*: will hold  $e1 = d \bmod (p-1)$   
*e2*: will hold  $e2 = d \bmod (q-1)$

This function will export the RSA private key's parameters found in the given structure. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

**Returns:** On success, `GNUTLS_E_SUCCESS (0)` is returned, otherwise a negative error value.

**Since:** 2.12.0

## `gnutls_x509_privkey_fix`

`int gnutls_x509_privkey_fix (gnutls_x509_privkey_t key)` [Function]  
*key*: Holds the key

This function will recalculate the secondary parameters in a key. In RSA keys, this can be the coefficient and exponent1,2.

**Returns:** On success, `GNUTLS_E_SUCCESS (0)` is returned, otherwise a negative error value.

## `gnutls_x509_privkey_generate`

`int gnutls_x509_privkey_generate (gnutls_x509_privkey_t key, [Function]  
 gnutls_pk_algorithm_t algo, unsigned int bits, unsigned int flags)`

*key*: should contain a `gnutls_x509_privkey_t` structure

*algo*: is one of the algorithms in `gnutls_pk_algorithm_t` .

*bits*: the size of the modulus

*flags*: unused for now. Must be 0.

This function will generate a random private key. Note that this function must be called on an empty private key.

Note that when generating an elliptic curve key, the curve can be substituted in the place of the bits parameter using the `GNUTLS_CURVE_TO_BITS()` macro.

For DSA keys, if the subgroup size needs to be specified check the `GNUTLS_SUBGROUP_TO_BITS()` macro.

Do not set the number of bits directly, use `gnutls_sec_param_to_pk_bits()` .

**Returns:** On success, `GNUTLS_E_SUCCESS (0)` is returned, otherwise a negative error value.

## `gnutls_x509_privkey_get_key_id`

`int gnutls_x509_privkey_get_key_id (gnutls_x509_privkey_t key, [Function]  
 unsigned int flags, unsigned char * output_data, size_t *  
 output_data_size)`

*key*: Holds the key

*flags*: should be 0 for now

*output\_data*: will contain the key ID

*output\_data\_size*: holds the size of *output\_data* (and will be replaced by the actual size of parameters)

This function will return a unique ID that depends on the public key parameters. This ID can be used in checking whether a certificate corresponds to the given key.

If the buffer provided is not long enough to hold the output, then \* *output\_data\_size* is updated and `GNUTLS_E_SHORT_MEMORY_BUFFER` will be returned. The output will normally be a SHA-1 hash output, which is 20 bytes.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

### **gnutls\_x509\_privkey\_get\_pk\_algorithm**

```
int gnutls_x509_privkey_get_pk_algorithm (gnutls_x509_privkey_t key) [Function]
```

*key*: should contain a `gnutls_x509_privkey_t` structure

This function will return the public key algorithm of a private key.

**Returns:** a member of the `gnutls_pk_algorithm_t` enumeration on success, or a negative error code on error.

### **gnutls\_x509\_privkey\_get\_pk\_algorithm2**

```
int gnutls_x509_privkey_get_pk_algorithm2 (gnutls_x509_privkey_t key, unsigned int * bits) [Function]
```

*key*: should contain a `gnutls_x509_privkey_t` structure

*bits*: The number of bits in the public key algorithm

This function will return the public key algorithm of a private key.

**Returns:** a member of the `gnutls_pk_algorithm_t` enumeration on success, or a negative error code on error.

### **gnutls\_x509\_privkey\_import**

```
int gnutls_x509_privkey_import (gnutls_x509_privkey_t key, const gnutls_datum_t * data, gnutls_x509_crt_fmt_t format) [Function]
```

*key*: The structure to store the parsed key

*data*: The DER or PEM encoded certificate.

*format*: One of DER or PEM

This function will convert the given DER or PEM encoded key to the native `gnutls_x509_privkey_t` format. The output will be stored in *key*.

If the key is PEM encoded it should have a header that contains "PRIVATE KEY". Note that this function falls back to PKCS 8 decoding without password, if the default format fails to import.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**gnutls\_x509\_privkey\_import2**

**int gnutls\_x509\_privkey\_import2** (*gnutls\_x509\_privkey\_t* **key**, *const gnutls\_datum\_t \* data*, *gnutls\_x509\_crt\_fmt\_t format*, *const char \* password*, *unsigned int flags*) [Function]

*key*: The structure to store the parsed key

*data*: The DER or PEM encoded key.

*format*: One of DER or PEM

*password*: A password (optional)

*flags*: an ORed sequence of *gnutls\_pkcs\_encrypt\_flags\_t*

This function will import the given DER or PEM encoded key, to the native **gnutls\_x509\_privkey\_t** format, irrespective of the input format. The input format is auto-detected.

The supported formats are basic unencrypted key, PKCS8, PKCS12, and the openssl format.

If the provided key is encrypted but no password was given, then **GNUTLS\_E\_DECRYPTION\_FAILED** is returned.

**Returns:** On success, **GNUTLS\_E\_SUCCESS** (0) is returned, otherwise a negative error value.

**gnutls\_x509\_privkey\_import\_dsa\_raw**

**int gnutls\_x509\_privkey\_import\_dsa\_raw** (*gnutls\_x509\_privkey\_t* **key**, *const gnutls\_datum\_t \* p*, *const gnutls\_datum\_t \* q*, *const gnutls\_datum\_t \* g*, *const gnutls\_datum\_t \* y*, *const gnutls\_datum\_t \* x*) [Function]

*key*: The structure to store the parsed key

*p*: holds the p

*q*: holds the q

*g*: holds the g

*y*: holds the y

*x*: holds the x

This function will convert the given DSA raw parameters to the native **gnutls\_x509\_privkey\_t** format. The output will be stored in **key**.

**Returns:** On success, **GNUTLS\_E\_SUCCESS** (0) is returned, otherwise a negative error value.

**gnutls\_x509\_privkey\_import\_ecc\_raw**

**int gnutls\_x509\_privkey\_import\_ecc\_raw** (*gnutls\_x509\_privkey\_t* **key**, *gnutls\_ecc\_curve\_t curve*, *const gnutls\_datum\_t \* x*, *const gnutls\_datum\_t \* y*, *const gnutls\_datum\_t \* k*) [Function]

*key*: The structure to store the parsed key

*curve*: holds the curve

*x*: holds the x



*y*: holds the *y*

*k*: holds the *k*

This function will convert the given elliptic curve parameters to the native `gnutls_x509_privkey_t` format. The output will be stored in `key`.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.0

## `gnutls_x509_privkey_import_openssl`

`int gnutls_x509_privkey_import_openssl (gnutls_x509_privkey_t key, const gnutls_datum_t * data, const char * password)` [Function]

*key*: The structure to store the parsed key

*data*: The DER or PEM encoded key.

*password*: the password to decrypt the key (if it is encrypted).

This function will convert the given PEM encrypted to the native `gnutls_x509_privkey_t` format. The output will be stored in `key`.

The `password` should be in ASCII. If the password is not provided or wrong then `GNUTLS_E_DECRYPTION_FAILED` will be returned.

If the Certificate is PEM encoded it should have a header of "PRIVATE KEY" and the "DEK-Info" header.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

## `gnutls_x509_privkey_import_pkcs8`

`int gnutls_x509_privkey_import_pkcs8 (gnutls_x509_privkey_t key, const gnutls_datum_t * data, gnutls_x509_crt_fmt_t format, const char * password, unsigned int flags)` [Function]

*key*: The structure to store the parsed key

*data*: The DER or PEM encoded key.

*format*: One of DER or PEM

*password*: the password to decrypt the key (if it is encrypted).

*flags*: 0 if encrypted or `GNUTLS_PKCS_PLAIN` if not encrypted.

This function will convert the given DER or PEM encoded PKCS8 2.0 encrypted key to the native `gnutls_x509_privkey_t` format. The output will be stored in `key`. Both RSA and DSA keys can be imported, and flags can only be used to indicate an unencrypted key.

The `password` can be either ASCII or UTF-8 in the default PBES2 encryption schemas, or ASCII for the PKCS12 schemas.

If the Certificate is PEM encoded it should have a header of "ENCRYPTED PRIVATE KEY", or "PRIVATE KEY". You only need to specify the flags if the key is DER encoded, since in that case the encryption status cannot be auto-detected.

If the `GNUTLS_PKCS_PLAIN` flag is specified and the supplied data are encrypted then `GNUTLS_E_DECRYPTION_FAILED` is returned.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

### `gnutls_x509_privkey_import_rsa_raw`

```
int gnutls_x509_privkey_import_rsa_raw (gnutls_x509_privkey_t [Function]
    key, const gnutls_datum_t * m, const gnutls_datum_t * e, const gnutls_datum_t
    * d, const gnutls_datum_t * p, const gnutls_datum_t * q, const gnutls_datum_t
    * u)
```

*key*: The structure to store the parsed key

*m*: holds the modulus

*e*: holds the public exponent

*d*: holds the private exponent

*p*: holds the first prime (p)

*q*: holds the second prime (q)

*u*: holds the coefficient

This function will convert the given RSA raw parameters to the native `gnutls_x509_privkey_t` format. The output will be stored in *key*.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

### `gnutls_x509_privkey_import_rsa_raw2`

```
int gnutls_x509_privkey_import_rsa_raw2 (gnutls_x509_privkey_t [Function]
    key, const gnutls_datum_t * m, const gnutls_datum_t * e, const gnutls_datum_t
    * d, const gnutls_datum_t * p, const gnutls_datum_t * q, const gnutls_datum_t
    * u, const gnutls_datum_t * e1, const gnutls_datum_t * e2)
```

*key*: The structure to store the parsed key

*m*: holds the modulus

*e*: holds the public exponent

*d*: holds the private exponent

*p*: holds the first prime (p)

*q*: holds the second prime (q)

*u*: holds the coefficient (optional)

*e1*: holds  $e1 = d \bmod (p-1)$  (optional)

*e2*: holds  $e2 = d \bmod (q-1)$  (optional)

This function will convert the given RSA raw parameters to the native `gnutls_x509_privkey_t` format. The output will be stored in *key*.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**gnutls\_x509\_privkey\_init**

**int gnutls\_x509\_privkey\_init** (*gnutls\_x509\_privkey\_t \* key*) [Function]

*key*: The structure to be initialized

This function will initialize an private key structure.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**gnutls\_x509\_privkey\_sec\_param**

**gnutls\_sec\_param\_t gnutls\_x509\_privkey\_sec\_param** [Function]  
(*gnutls\_x509\_privkey\_t key*)

*key*: a key structure

This function will return the security parameter appropriate with this private key.

**Returns:** On success, a valid security parameter is returned otherwise GNUTLS\_SEC\_PARAM\_UNKNOWN is returned.

**Since:** 2.12.0

**gnutls\_x509\_privkey\_verify\_params**

**int gnutls\_x509\_privkey\_verify\_params** (*gnutls\_x509\_privkey\_t key*) [Function]

*key*: should contain a **gnutls\_x509\_privkey\_t** structure

This function will verify the private key parameters.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**gnutls\_x509\_rdn\_get**

**int gnutls\_x509\_rdn\_get** (*const gnutls\_datum\_t \* idn, char \* buf, size\_t \* buf\_size*) [Function]

*idn*: should contain a DER encoded RDN sequence

*buf*: a pointer to a structure to hold the peer's name

*buf\_size*: holds the size of *buf*

This function will return the name of the given RDN sequence. The name will be in the form "C=xxxx,O=yyyy,CN=zzzz" as described in RFC4514.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, or GNUTLS\_E\_SHORT\_MEMORY\_BUFFER is returned and *\* buf\_size* is updated if the provided buffer is not long enough, otherwise a negative error value.

**gnutls\_x509\_rdn\_get\_by\_oid**

**int gnutls\_x509\_rdn\_get\_by\_oid** (*const gnutls\_datum\_t \* idn, const char \* oid, int indx, unsigned int raw\_flag, void \* buf, size\_t \* buf\_size*) [Function]

*idn*: should contain a DER encoded RDN sequence

*oid*: an Object Identifier

*indx*: In case multiple same OIDs exist in the RDN indicates which to send. Use 0 for the first one.

*raw\_flag*: If non-zero then the raw DER data are returned.

*buf*: a pointer to a structure to hold the peer's name

*buf\_size*: holds the size of *buf*

This function will return the name of the given Object identifier, of the RDN sequence. The name will be encoded using the rules from RFC4514.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, or `GNUTLS_E_SHORT_MEMORY_BUFFER` is returned and *\* buf\_size* is updated if the provided buffer is not long enough, otherwise a negative error value.

### `gnutls_x509_rdn_get_oid`

```
int gnutls_x509_rdn_get_oid (const gnutls_datum_t * idn, int indx,    [Function]
                             void * buf, size_t * buf_size)
```

*idn*: should contain a DER encoded RDN sequence

*indx*: Indicates which OID to return. Use 0 for the first one.

*buf*: a pointer to a structure to hold the peer's name OID

*buf\_size*: holds the size of *buf*

This function will return the specified Object identifier, of the RDN sequence.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, or `GNUTLS_E_SHORT_MEMORY_BUFFER` is returned and *\* buf\_size* is updated if the provided buffer is not long enough, otherwise a negative error value.

**Since:** 2.4.0

### `gnutls_x509_trust_list_add_cas`

```
int gnutls_x509_trust_list_add_cas (gnutls_x509_trust_list_t    [Function]
                                     list, const gnutls_x509_crt_t * clist, unsigned clist_size, unsigned int
                                     flags)
```

*list*: The structure of the list

*clist*: A list of CAs

*clist\_size*: The length of the CA list

*flags*: should be 0 or an or'ed sequence of `GNUTLS_TL` options.

This function will add the given certificate authorities to the trusted list. The list of CAs must not be deinitialized during this structure's lifetime.

If the flag `GNUTLS_TL_NO_DUPLICATES` is specified, then the provided *clist* entries that are duplicates will not be added to the list and will be deinitialized.

**Returns:** The number of added elements is returned.

**Since:** 3.0.0

**gnutls\_x509\_trust\_list\_add\_crls**

```
int gnutls_x509_trust_list_add_crls (gnutls_x509_trust_list_t [Function]
    list, const gnutls_x509_crl_t *crl_list, int crl_size, unsigned int flags,
    unsigned int verification_flags)
```

*list*: The structure of the list

*crl\_list*: A list of CRLs

*crl\_size*: The length of the CRL list

*flags*: if GNUTLS\_TL\_VERIFY\_CRL is given the CRLs will be verified before being added.

*verification\_flags*: gnutls\_certificate\_verify\_flags if flags specifies GNUTLS\_TL\_VERIFY\_CRL

This function will add the given certificate revocation lists to the trusted list. The list of CRLs must not be deinitialized during this structure's lifetime.

This function must be called after `gnutls_x509_trust_list_add_cas()` to allow verifying the CRLs for validity. If the flag `GNUTLS_TL_NO_DUPLICATES` is given, then any provided CRLs that are a duplicate, will be deinitialized and not added to the list (that assumes that `gnutls_x509_trust_list_deinit()` will be called with `all=1`).

**Returns:** The number of added elements is returned.

**Since:** 3.0

**gnutls\_x509\_trust\_list\_add\_named\_cert**

```
int gnutls_x509_trust_list_add_named_cert [Function]
    (gnutls_x509_trust_list_t list, gnutls_x509_cert_t cert, const void *name,
    size_t name_size, unsigned int flags)
```

*list*: The structure of the list

*cert*: A certificate

*name*: An identifier for the certificate

*name\_size*: The size of the identifier

*flags*: should be 0.

This function will add the given certificate to the trusted list and associate it with a name. The certificate will not be used for verification with `gnutls_x509_trust_list_verify_cert()` but only with `gnutls_x509_trust_list_verify_named_cert()`.

In principle this function can be used to set individual "server" certificates that are trusted by the user for that specific server but for no other purposes.

The certificate must not be deinitialized during the lifetime of the trusted list.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.0.0

**gnutls\_x509\_trust\_list\_add\_system\_trust**

**int gnutls\_x509\_trust\_list\_add\_system\_trust** [Function]

(*gnutls\_x509\_trust\_list\_t list*, unsigned int *tl\_flags*, unsigned int *tl\_vflags*)

*list*: The structure of the list

*tl\_flags*: GNUTLS\_TL\_\*

*tl\_vflags*: gnutls\_certificate\_verify\_flags if flags specifies GNUTLS\_TL\_VERIFY\_CRL

This function adds the system's default trusted certificate authorities to the trusted list. Note that on unsupported systems this function returns GNUTLS\_E\_UNIMPLEMENTED\_FEATURE .

This function implies the flag GNUTLS\_TL\_NO\_DUPLICATES .

**Returns:** The number of added elements or a negative error code on error.

**Since:** 3.1

**gnutls\_x509\_trust\_list\_add\_trust\_dir**

**int gnutls\_x509\_trust\_list\_add\_trust\_dir** [Function]

(*gnutls\_x509\_trust\_list\_t list*, const char \* *ca\_dir*, const char \* *crl\_dir*,  
gnutls\_x509\_cert\_fmt\_t *type*, unsigned int *tl\_flags*, unsigned int *tl\_vflags*)

*list*: The structure of the list

*ca\_dir*: A directory containing the CAs (optional)

*crl\_dir*: A directory containing a list of CRLs (optional)

*type*: The format of the certificates

*tl\_flags*: GNUTLS\_TL\_\*

*tl\_vflags*: gnutls\_certificate\_verify\_flags if flags specifies GNUTLS\_TL\_VERIFY\_CRL

This function will add the given certificate authorities to the trusted list. Only directories are accepted by this function.

**Returns:** The number of added elements is returned.

**Since:** 3.3.6

**gnutls\_x509\_trust\_list\_add\_trust\_file**

**int gnutls\_x509\_trust\_list\_add\_trust\_file** [Function]

(*gnutls\_x509\_trust\_list\_t list*, const char \* *ca\_file*, const char \* *crl\_file*,  
gnutls\_x509\_cert\_fmt\_t *type*, unsigned int *tl\_flags*, unsigned int *tl\_vflags*)

*list*: The structure of the list

*ca\_file*: A file containing a list of CAs (optional)

*crl\_file*: A file containing a list of CRLs (optional)

*type*: The format of the certificates

*tl\_flags*: GNUTLS\_TL\_\*

*tl\_vflags*: gnutls\_certificate\_verify\_flags if flags specifies GNUTLS\_TL\_VERIFY\_CRL

This function will add the given certificate authorities to the trusted list. PKCS 11 URLs are also accepted, instead of files, by this function. A PKCS 11 URL implies

a trust database (a specially marked module in p11-kit); the URL "pkcs11:" implies all trust databases in the system. Only a single URL specifying trust databases can be set; they cannot be stacked with multiple calls.

**Returns:** The number of added elements is returned.

**Since:** 3.1

### gnutls\_x509\_trust\_list\_add\_trust\_mem

```
int gnutls_x509_trust_list_add_trust_mem [Function]
    (gnutls_x509_trust_list_t list, const gnutls_datum_t * cas, const
     gnutls_datum_t * crls, gnutls_x509_cert_fmt_t type, unsigned int tl_flags,
     unsigned int tl_vflags)
```

*list*: The structure of the list

*cas*: A buffer containing a list of CAs (optional)

*crls*: A buffer containing a list of CRLs (optional)

*type*: The format of the certificates

*tl\_flags*: GNUTLS\_TL\_\*

*tl\_vflags*: gnutls\_certificate\_verify\_flags if flags specifies GNUTLS\_TL\_VERIFY\_CRL

This function will add the given certificate authorities to the trusted list.

**Returns:** The number of added elements is returned.

**Since:** 3.1

### gnutls\_x509\_trust\_list\_deinit

```
void gnutls_x509_trust_list_deinit (gnutls_x509_trust_list_t [Function]
    list, unsigned int all)
```

*list*: The structure to be deinitialized

*all*: if non-zero it will deinitialize all the certificates and CRLs contained in the structure.

This function will deinitialize a trust list. Note that the *all* flag should be typically non-zero unless you have specified your certificates using `gnutls_x509_trust_list_add_cas()` and you want to prevent them from being deinitialized by this function.

**Since:** 3.0.0

### gnutls\_x509\_trust\_list\_get\_issuer

```
int gnutls_x509_trust_list_get_issuer (gnutls_x509_trust_list_t [Function]
    list, gnutls_x509_cert_t cert, gnutls_x509_cert_t * issuer, unsigned int
    flags)
```

*list*: The structure of the list

*cert*: is the certificate to find issuer for

*issuer*: Will hold the issuer if any. Should be treated as constant.

*flags*: Use zero or GNUTLS\_TL\_GET\_COPY

This function will find the issuer of the given certificate. If the flag GNUTLS\_TL\_GET\_COPY is specified a copy of the issuer will be returned which must be freed using

`gnutls_x509_cert_deinit()` . Note that the flag `GNUTLS_TL_GET_COPY` is required for this function to work with PKCS 11 trust lists in a thread-safe way.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.0

## `gnutls_x509_trust_list_init`

`int gnutls_x509_trust_list_init (gnutls_x509_trust_list_t * list, [Function]  
                                  unsigned int size)`

*list*: The structure to be initialized

*size*: The size of the internal hash table. Use (0) for default size.

This function will initialize an X.509 trust list structure.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.0.0

## `gnutls_x509_trust_list_remove_cas`

`int gnutls_x509_trust_list_remove_cas (gnutls_x509_trust_list_t [Function]  
                                  list, const gnutls_x509_cert_t * clist, int clist_size)`

*list*: The structure of the list

*clist*: A list of CAs

*clist\_size*: The length of the CA list

This function will remove the given certificate authorities from the trusted list.

Note that this function can accept certificates and authorities not yet known. In that case they will be kept in a separate black list that will be used during certificate verification. Unlike `gnutls_x509_trust_list_add_cas()` there is no deinitialization restriction for certificate list provided in this function.

**Returns:** The number of removed elements is returned.

**Since:** 3.1.10

## `gnutls_x509_trust_list_remove_trust_file`

`int gnutls_x509_trust_list_remove_trust_file [Function]  
                                  (gnutls_x509_trust_list_t list, const char * ca_file, gnutls_x509_cert_fmt_t  
                                  type)`

*list*: The structure of the list

*ca\_file*: A file containing a list of CAs

*type*: The format of the certificates

This function will remove the given certificate authorities from the trusted list, and add them into a black list when needed. PKCS 11 URLs are also accepted, instead of files, by this function.

See also `gnutls_x509_trust_list_remove_cas()` .

**Returns:** The number of added elements is returned.

**Since:** 3.1.10



**gnutls\_x509\_trust\_list\_remove\_trust\_mem**

**int gnutls\_x509\_trust\_list\_remove\_trust\_mem** [Function]

(*gnutls\_x509\_trust\_list\_t list*, *const gnutls\_datum\_t \* cas*,  
*gnutls\_x509\_crt\_fmt\_t type*)

*list*: The structure of the list

*cas*: A buffer containing a list of CAs (optional)

*type*: The format of the certificates

This function will remove the provided certificate authorities from the trusted list, and add them into a black list when needed.

See also `gnutls_x509_trust_list_remove_cas()` .

**Returns:** The number of removed elements is returned.

**Since:** 3.1.10

**gnutls\_x509\_trust\_list\_verify\_cert**

**int gnutls\_x509\_trust\_list\_verify\_cert** (*gnutls\_x509\_trust\_list\_t* [Function]

*list*, *gnutls\_x509\_crt\_t \* cert\_list*, *unsigned int cert\_list\_size*,  
*unsigned int flags*, *unsigned int \* voutput*, *gnutls\_verify\_output\_function*  
*func*)

*list*: The structure of the list

*cert\_list*: is the certificate list to be verified

*cert\_list\_size*: is the certificate list size

*flags*: Flags that may be used to change the verification algorithm. Use OR of the `gnutls_certificate_verify_flags` enumerations.

*voutput*: will hold the certificate verification output.

*func*: If non-null will be called on each chain element verification with the output.

This function will try to verify the given certificate and return its status. The `verify` parameter will hold an OR'ed sequence of `gnutls_certificate_status_t` flags.

Additionally a certificate verification profile can be specified from the ones in `gnutls_certificate_verification_profiles_t` by ORing the result of `GNUTLS_PROFILE_TO_VFLAGS()` to the verification flags.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.0

**gnutls\_x509\_trust\_list\_verify\_cert2**

**int gnutls\_x509\_trust\_list\_verify\_cert2** (*gnutls\_x509\_trust\_list\_t* [Function]

*list*, *gnutls\_x509\_crt\_t \* cert\_list*, *unsigned int cert\_list\_size*,  
*gnutls\_typed\_vdata\_st \* data*, *unsigned int elements*, *unsigned int flags*,  
*unsigned int \* voutput*, *gnutls\_verify\_output\_function func*)

*list*: The structure of the list

*cert\_list*: is the certificate list to be verified

*cert\_list\_size*: is the certificate list size

*data*: an array of typed data

*elements*: the number of data elements

*flags*: Flags that may be used to change the verification algorithm. Use OR of the `gnutls_certificate_verify_flags` enumerations.

*voutput*: will hold the certificate verification output.

*func*: If non-null will be called on each chain element verification with the output.

This function will try to verify the given certificate and return its status. The `verify` parameter will hold an OR'ed sequence of `gnutls_certificate_status_t` flags.

Additionally a certificate verification profile can be specified from the ones in `gnutls_certificate_verification_profiles_t` by ORing the result of `GNUTLS_PROFILE_TO_VFLAGS()` to the verification flags.

The acceptable `data` types are `GNUTLS_DT_DNS_HOSTNAME` and `GNUTLS_DT_KEY_PURPOSE_OID`. The former accepts as data a null-terminated hostname, and the latter a null-terminated object identifier (e.g., `GNUTLS_KP_TLS_WWW_SERVER`). If a DNS hostname is provided then this function will compare the hostname in the certificate against the given. If names do not match the `GNUTLS_CERT_UNEXPECTED_OWNER` status flag will be set. If a key purpose OID is provided and the end-certificate contains the extended key usage PKIX extension, it will be required to have the provided key purpose or be marked for any purpose, otherwise verification will fail with `GNUTLS_CERT_SIGNER_CONSTRAINTS_FAILURE` status.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value. Note that verification failure will not result to an error code, only `voutput` will be updated.

**Since:** 3.3.8

## `gnutls_x509_trust_list_verify_named_cert`

```
int gnutls_x509_trust_list_verify_named_cert           [Function]
    (gnutls_x509_trust_list_t list, gnutls_x509_cert_t cert, const void * name,
     size_t name_size, unsigned int flags, unsigned int * voutput,
     gnutls_verify_output_function func)
```

*list*: The structure of the list

*cert*: is the certificate to be verified

*name*: is the certificate's name

*name\_size*: is the certificate's name size

*flags*: Flags that may be used to change the verification algorithm. Use OR of the `gnutls_certificate_verify_flags` enumerations.

*voutput*: will hold the certificate verification output.

*func*: If non-null will be called on each chain element verification with the output.

This function will try to find a certificate that is associated with the provided name – see `gnutls_x509_trust_list_add_named_cert()`. If a match is found the certificate is considered valid. In addition to that this function will also check CRLs. The

voutput parameter will hold an OR'ed sequence of `gnutls_certificate_status_t` flags.

Additionally a certificate verification profile can be specified from the ones in `gnutls_certificate_verification_profiles_t` by ORing the result of `GNUTLS_PROFILE_TO_VFLAGS()` to the verification flags.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.0.0

## E.4 OCSP API

The following functions are for OCSP certificate status checking. Their prototypes lie in `gnutls/ocsp.h`.

### `gnutls_ocsp_req_add_cert`

```
int gnutls_ocsp_req_add_cert (gnutls_ocsp_req_t req, [Function]
                             gnutls_digest_algorithm_t digest, gnutls_x509_crt_t issuer,
                             gnutls_x509_crt_t cert)
```

*req*: should contain a `gnutls_ocsp_req_t` structure

*digest*: hash algorithm, a `gnutls_digest_algorithm_t` value

*issuer*: issuer of **subject** certificate

*cert*: certificate to request status for

This function will add another request to the OCSP request for a particular certificate. The issuer name hash, issuer key hash, and serial number fields is populated as follows. The issuer name and the serial number is taken from **cert** . The issuer key is taken from **issuer** . The hashed values will be hashed using the **digest** algorithm, normally `GNUTLS_DIG_SHA1` .

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

### `gnutls_ocsp_req_add_cert_id`

```
int gnutls_ocsp_req_add_cert_id (gnutls_ocsp_req_t req, [Function]
                                 gnutls_digest_algorithm_t digest, const gnutls_datum_t *
                                 issuer_name_hash, const gnutls_datum_t * issuer_key_hash, const
                                 gnutls_datum_t * serial_number)
```

*req*: should contain a `gnutls_ocsp_req_t` structure

*digest*: hash algorithm, a `gnutls_digest_algorithm_t` value

*issuer\_name\_hash*: hash of issuer's DN

*issuer\_key\_hash*: hash of issuer's public key

*serial\_number*: serial number of certificate to check

This function will add another request to the OCSP request for a particular certificate having the issuer name hash of **issuer\_name\_hash** and issuer key hash of **issuer\_key\_hash** (both hashed using **digest** ) and serial number **serial\_number** .

The information needed corresponds to the CertID structure:

```
<informalexample><programlisting> CertID ::= SEQUENCE { hashAlgorithm Algo-
rithmIdentifier, issuerNameHash OCTET STRING, – Hash of Issuer’s DN issuerKey-
Hash OCTET STRING, – Hash of Issuers public key serialNumber CertificateSerial-
Number } </programlisting></informalexample>
```

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error code is returned.

## gnutls\_ocsp\_req\_deinit

```
void gnutls_ocsp_req_deinit (gnutls_ocsp_req_t req) [Function]
```

*req*: The structure to be deinitialized

This function will deinitialize a OCSP request structure.

## gnutls\_ocsp\_req\_export

```
int gnutls_ocsp_req_export (gnutls_ocsp_req_t req, gnutls_datum_t [Function]
    * data)
```

*req*: Holds the OCSP request

*data*: newly allocate buffer holding DER encoded OCSP request

This function will export the OCSP request to DER format.

**Returns:** In case of failure a negative error code will be returned, and 0 on success.

## gnutls\_ocsp\_req\_get\_cert\_id

```
int gnutls_ocsp_req_get_cert_id (gnutls_ocsp_req_t req, unsigned [Function]
    indx, gnutls_digest_algorithm_t * digest, gnutls_datum_t *
    issuer_name_hash, gnutls_datum_t * issuer_key_hash, gnutls_datum_t *
    serial_number)
```

*req*: should contain a gnutls\_ocsp\_req\_t structure

*indx*: Specifies which extension OID to get. Use (0) to get the first one.

*digest*: output variable with gnutls\_digest\_algorithm\_t hash algorithm

*issuer\_name\_hash*: output buffer with hash of issuer’s DN

*issuer\_key\_hash*: output buffer with hash of issuer’s public key

*serial\_number*: output buffer with serial number of certificate to check

This function will return the certificate information of the *indx* 'ed request in the OCSP request. The information returned corresponds to the CertID structure:

```
<informalexample><programlisting> CertID ::= SEQUENCE { hashAlgorithm Algo-
rithmIdentifier, issuerNameHash OCTET STRING, – Hash of Issuer’s DN issuerKey-
Hash OCTET STRING, – Hash of Issuers public key serialNumber CertificateSerial-
Number } </programlisting></informalexample>
```

Each of the pointers to output variables may be NULL to indicate that the caller is not interested in that value.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error code is returned. If you have reached the last CertID available GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE will be returned.

## gnutls\_ocsp\_req\_get\_extension

```
int gnutls_ocsp_req_get_extension (gnutls_ocsp_req_t req,          [Function]
                                   unsigned indx, gnutls_datum_t * oid, unsigned int * critical,
                                   gnutls_datum_t * data)
```

*req*: should contain a `gnutls_ocsp_req_t` structure

*indx*: Specifies which extension OID to get. Use (0) to get the first one.

*oid*: will hold newly allocated buffer with OID of extension, may be NULL

*critical*: output variable with critical flag, may be NULL.

*data*: will hold newly allocated buffer with extension data, may be NULL

This function will return all information about the requested extension in the OCSF request. The information returned is the OID, the critical flag, and the data itself. The extension OID will be stored as a string. Any of *oid*, *critical*, and *data* may be NULL which means that the caller is not interested in getting that information back.

The caller needs to deallocate memory by calling `gnutls_free()` on *oid* ->data and *data* ->data.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned. If you have reached the last extension available `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

## gnutls\_ocsp\_req\_get\_nonce

```
int gnutls_ocsp_req_get_nonce (gnutls_ocsp_req_t req, unsigned int  [Function]
                               * critical, gnutls_datum_t * nonce)
```

*req*: should contain a `gnutls_ocsp_req_t` structure

*critical*: whether nonce extension is marked critical, or NULL

*nonce*: will hold newly allocated buffer with nonce data

This function will return the OCSF request nonce extension data.

The caller needs to deallocate memory by calling `gnutls_free()` on *nonce* ->data.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

## gnutls\_ocsp\_req\_get\_version

```
int gnutls_ocsp_req_get_version (gnutls_ocsp_req_t req)          [Function]
    req: should contain a gnutls_ocsp_req_t structure
```

This function will return the version of the OCSF request. Typically this is always 1 indicating version 1.

**Returns:** version of OCSF request, or a negative error code on error.

**gnutls\_ocsp\_req\_import**

**int gnutls\_ocsp\_req\_import** (*gnutls\_ocsp\_req\_t req, const gnutls\_datum\_t \* data*) [Function]

*req*: The structure to store the parsed request.

*data*: DER encoded OCSP request.

This function will convert the given DER encoded OCSP request to the native `gnutls_ocsp_req_t` format. The output will be stored in *req*.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**gnutls\_ocsp\_req\_init**

**int gnutls\_ocsp\_req\_init** (*gnutls\_ocsp\_req\_t \* req*) [Function]

*req*: The structure to be initialized

This function will initialize an OCSP request structure.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**gnutls\_ocsp\_req\_print**

**int gnutls\_ocsp\_req\_print** (*gnutls\_ocsp\_req\_t req, gnutls\_ocsp\_print\_formats\_t format, gnutls\_datum\_t \* out*) [Function]

*req*: The structure to be printed

*format*: Indicate the format to use

*out*: Newly allocated datum with (0) terminated string.

This function will pretty print a OCSP request, suitable for display to a human.

If the format is `GNUTLS_OCSP_PRINT_FULL` then all fields of the request will be output, on multiple lines.

The output *out* ->data needs to be deallocate using `gnutls_free()`.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**gnutls\_ocsp\_req\_randomize\_nonce**

**int gnutls\_ocsp\_req\_randomize\_nonce** (*gnutls\_ocsp\_req\_t req*) [Function]

*req*: should contain a `gnutls_ocsp_req_t` structure

This function will add or update an nonce extension to the OCSP request with a newly generated random value.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

## gnutls\_ocsp\_req\_set\_extension

**int gnutls\_ocsp\_req\_set\_extension** (*gnutls\_ocsp\_req\_t req*, *const char \*oid*, *unsigned int critical*, *const gnutls\_datum\_t \*data*) [Function]

*req*: should contain a `gnutls_ocsp_req_t` structure

*oid*: buffer with OID of extension as a string.

*critical*: critical flag, normally false.

*data*: the extension data

This function will add an extension to the OCSF request. Calling this function multiple times for the same OID will overwrite values from earlier calls.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

## gnutls\_ocsp\_req\_set\_nonce

**int gnutls\_ocsp\_req\_set\_nonce** (*gnutls\_ocsp\_req\_t req*, *unsigned int critical*, *const gnutls\_datum\_t \*nonce*) [Function]

*req*: should contain a `gnutls_ocsp_req_t` structure

*critical*: critical flag, normally false.

*nonce*: the nonce data

This function will add an nonce extension to the OCSF request. Calling this function multiple times will overwrite values from earlier calls.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

## gnutls\_ocsp\_resp\_check\_cert

**int gnutls\_ocsp\_resp\_check\_cert** (*gnutls\_ocsp\_resp\_t resp*, *unsigned int indx*, *gnutls\_x509\_cert\_t crt*) [Function]

*resp*: should contain a `gnutls_ocsp_resp_t` structure

*indx*: Specifies response number to get. Use (0) to get the first one.

*crt*: The certificate to check

This function will check whether the OCSF response is about the provided certificate.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned.

**Since:** 3.1.3

## gnutls\_ocsp\_resp\_deinit

**void gnutls\_ocsp\_resp\_deinit** (*gnutls\_ocsp\_resp\_t resp*) [Function]

*resp*: The structure to be deinitialized

This function will deinitialize a OCSF response structure.

## gnutls\_ocsp\_resp\_export

**int gnutls\_ocsp\_resp\_export** (*gnutls\_ocsp\_resp\_t resp*, [Function]  
*gnutls\_datum\_t \* data*)

*resp*: Holds the OCSP response

*data*: newly allocate buffer holding DER encoded OCSP response

This function will export the OCSP response to DER format.

**Returns:** In case of failure a negative error code will be returned, and 0 on success.

## gnutls\_ocsp\_resp\_get\_certs

**int gnutls\_ocsp\_resp\_get\_certs** (*gnutls\_ocsp\_resp\_t resp*, [Function]  
*gnutls\_x509\_cert\_t \*\* certs*, *size\_t \* ncerts*)

*resp*: should contain a *gnutls\_ocsp\_resp\_t* structure

*certs*: newly allocated array with *gnutls\_x509\_cert\_t* certificates

*ncerts*: output variable with number of allocated certs.

This function will extract the X.509 certificates found in the Basic OCSP Response. The *certs* output variable will hold a newly allocated zero-terminated array with X.509 certificates.

Every certificate in the array needs to be de-allocated with *gnutls\_x509\_cert\_deinit()* and the array itself must be freed using *gnutls\_free()*.

Both the *certs* and *ncerts* variables may be NULL. Then the function will work as normal but will not return the NULL:d information. This can be used to get the number of certificates only, or to just get the certificate array without its size.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

## gnutls\_ocsp\_resp\_get\_extension

**int gnutls\_ocsp\_resp\_get\_extension** (*gnutls\_ocsp\_resp\_t resp*, [Function]  
*unsigned indx*, *gnutls\_datum\_t \* oid*, *unsigned int \* critical*,  
*gnutls\_datum\_t \* data*)

*resp*: should contain a *gnutls\_ocsp\_resp\_t* structure

*indx*: Specifies which extension OID to get. Use (0) to get the first one.

*oid*: will hold newly allocated buffer with OID of extension, may be NULL

*critical*: output variable with critical flag, may be NULL.

*data*: will hold newly allocated buffer with extension data, may be NULL

This function will return all information about the requested extension in the OCSP response. The information returned is the OID, the critical flag, and the data itself. The extension OID will be stored as a string. Any of *oid*, *critical*, and *data* may be NULL which means that the caller is not interested in getting that information back.

The caller needs to deallocate memory by calling *gnutls\_free()* on *oid ->data* and *data ->data*.



**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error code is returned. If you have reached the last extension available GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE will be returned.

### gnutls\_ocsp\_resp\_get\_nonce

**int gnutls\_ocsp\_resp\_get\_nonce** (*gnutls\_ocsp\_resp\_t resp*, unsigned [Function]  
*int \*critical*, *gnutls\_datum\_t \*nonce*)

*resp*: should contain a *gnutls\_ocsp\_resp\_t* structure

*critical*: whether nonce extension is marked critical

*nonce*: will hold newly allocated buffer with nonce data

This function will return the Basic OCSP Response nonce extension data.

The caller needs to deallocate memory by calling *gnutls\_free()* on *nonce* ->data.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error code is returned.

### gnutls\_ocsp\_resp\_get\_produced

**time\_t gnutls\_ocsp\_resp\_get\_produced** (*gnutls\_ocsp\_resp\_t resp*) [Function]  
*resp*: should contain a *gnutls\_ocsp\_resp\_t* structure

This function will return the time when the OCSP response was signed.

**Returns:** signing time, or (time\_t)-1 on error.

### gnutls\_ocsp\_resp\_get\_responder

**int gnutls\_ocsp\_resp\_get\_responder** (*gnutls\_ocsp\_resp\_t resp*, [Function]  
*gnutls\_datum\_t \*dn*)

*resp*: should contain a *gnutls\_ocsp\_resp\_t* structure

*dn*: newly allocated buffer with name

This function will extract the name of the Basic OCSP Response in the provided buffer. The name will be in the form "C=xxxx,O=yyyy,CN=zzzz" as described in RFC2253. The output string will be ASCII or UTF-8 encoded, depending on the certificate data.

The caller needs to deallocate memory by calling *gnutls\_free()* on *dn* ->data.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error code is returned.

### gnutls\_ocsp\_resp\_get\_response

**int gnutls\_ocsp\_resp\_get\_response** (*gnutls\_ocsp\_resp\_t resp*, [Function]  
*gnutls\_datum\_t \*response\_type\_oid*, *gnutls\_datum\_t \*response*)

*resp*: should contain a *gnutls\_ocsp\_resp\_t* structure

*response\_type\_oid*: newly allocated output buffer with response type OID

*response*: newly allocated output buffer with DER encoded response

This function will extract the response type OID in and the response data from an OCSP response. Normally the *response\_type\_oid* is always "1.3.6.1.5.5.7.48.1.1"

which means the **response** should be decoded as a Basic OCSP Response, but technically other response types could be used.

This function is typically only useful when you want to extract the response type OID of an response for diagnostic purposes. Otherwise `gnutls_ocsp_resp_import()` will decode the basic OCSP response part and the caller need not worry about that aspect.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

### **gnutls\_ocsp\_resp\_get\_signature**

```
int gnutls_ocsp_resp_get_signature (gnutls_ocsp_resp_t resp,      [Function]
                                   gnutls_datum_t * sig)
```

*resp*: should contain a `gnutls_ocsp_resp_t` structure

*sig*: newly allocated output buffer with signature data

This function will extract the signature field of a OCSP response.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

### **gnutls\_ocsp\_resp\_get\_signature\_algorithm**

```
int gnutls_ocsp_resp_get_signature_algorithm (gnutls_ocsp_resp_t resp) [Function]
```

*resp*: should contain a `gnutls_ocsp_resp_t` structure

This function will return a value of the `gnutls_sign_algorithm_t` enumeration that is the signature algorithm that has been used to sign the OCSP response.

**Returns:** a `gnutls_sign_algorithm_t` value, or a negative error code on error.

### **gnutls\_ocsp\_resp\_get\_single**

```
int gnutls_ocsp_resp_get_single (gnutls_ocsp_resp_t resp,      [Function]
                                 unsigned indx, gnutls_digest_algorithm_t * digest, gnutls_datum_t *
                                 issuer_name_hash, gnutls_datum_t * issuer_key_hash, gnutls_datum_t *
                                 serial_number, unsigned int * cert_status, time_t * this_update,
                                 time_t * next_update, time_t * revocation_time, unsigned int *
                                 revocation_reason)
```

*resp*: should contain a `gnutls_ocsp_resp_t` structure

*indx*: Specifies response number to get. Use (0) to get the first one.

*digest*: output variable with `gnutls_digest_algorithm_t` hash algorithm

*issuer\_name\_hash*: output buffer with hash of issuer's DN

*issuer\_key\_hash*: output buffer with hash of issuer's public key

*serial\_number*: output buffer with serial number of certificate to check

*cert\_status*: a certificate status, a `gnutls_ocsp_cert_status_t` enum.

*this\_update*: time at which the status is known to be correct.

*next\_update*: when newer information will be available, or (time\_t)-1 if unspecified

*revocation\_time*: when `cert_status` is `GNUTLS_OCSP_CERT_REVOKED` , holds time of revocation.

*revocation\_reason*: revocation reason, a `gnutls_x509_crl_reason_t` enum.

This function will return the certificate information of the `indx` 'ed response in the Basic OCSP Response `resp` . The information returned corresponds to the OCSP SingleResponse structure except the final singleExtensions.

Each of the pointers to output variables may be `NULL` to indicate that the caller is not interested in that value.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error code is returned. If you have reached the last CertID available `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

## gnutls\_ocsp\_resp\_get\_status

`int gnutls_ocsp_resp_get_status (gnutls_ocsp_resp_t resp)` [Function]

*resp*: should contain a `gnutls_ocsp_resp_t` structure

This function will return the status of a OCSP response, an `gnutls_ocsp_resp_status_t` enumeration.

**Returns:** status of OCSP request as a `gnutls_ocsp_resp_status_t` , or a negative error code on error.

## gnutls\_ocsp\_resp\_get\_version

`int gnutls_ocsp_resp_get_version (gnutls_ocsp_resp_t resp)` [Function]

*resp*: should contain a `gnutls_ocsp_resp_t` structure

This function will return the version of the Basic OCSP Response. Typically this is always 1 indicating version 1.

**Returns:** version of Basic OCSP response, or a negative error code on error.

## gnutls\_ocsp\_resp\_import

`int gnutls_ocsp_resp_import (gnutls_ocsp_resp_t resp, const gnutls_datum_t * data)` [Function]

*resp*: The structure to store the parsed response.

*data*: DER encoded OCSP response.

This function will convert the given DER encoded OCSP response to the native `gnutls_ocsp_resp_t` format. It also decodes the Basic OCSP Response part, if any. The output will be stored in `resp` .

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

## gnutls\_ocsp\_resp\_init

`int gnutls_ocsp_resp_init (gnutls_ocsp_resp_t * resp)` [Function]

*resp*: The structure to be initialized

This function will initialize an OCSP response structure.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

## gnutls\_ocsp\_resp\_print

```
int gnutls_ocsp_resp_print (gnutls_ocsp_resp_t resp, [Function]
                           gnutls_ocsp_print_formats_t format, gnutls_datum_t * out)
```

*resp*: The structure to be printed

*format*: Indicate the format to use

*out*: Newly allocated datum with (0) terminated string.

This function will pretty print a OCSP response, suitable for display to a human.

If the format is GNUTLS\_OCSP\_PRINT\_FULL then all fields of the response will be output, on multiple lines.

The output *out* ->data needs to be deallocate using `gnutls_free()` .

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

## gnutls\_ocsp\_resp\_verify

```
int gnutls_ocsp_resp_verify (gnutls_ocsp_resp_t resp, [Function]
                             gnutls_x509_trust_list_t trustlist, unsigned int * verify, unsigned int
                             flags)
```

*resp*: should contain a `gnutls_ocsp_resp_t` structure

*trustlist*: trust anchors as a `gnutls_x509_trust_list_t` structure

*verify*: output variable with verification status, an `gnutls_ocsp_cert_status_t`

*flags*: verification flags, 0 for now.

Verify signature of the Basic OCSP Response against the public key in the certificate of a trusted signer. The `trustlist` should be populated with trust anchors. The function will extract the signer certificate from the Basic OCSP Response and will verify it against the `trustlist` . A trusted signer is a certificate that is either in `trustlist` , or it is signed directly by a certificate in `trustlist` and has the id-ad-ocspSigning Extended Key Usage bit set.

The output `verify` variable will hold verification status codes (e.g., GNUTLS\_OCSP\_VERIFY\_SIGNER\_NOT\_FOUND , GNUTLS\_OCSP\_VERIFY\_INSECURE\_ALGORITHM ) which are only valid if the function returned GNUTLS\_E\_SUCCESS .

Note that the function returns GNUTLS\_E\_SUCCESS even when verification failed. The caller must always inspect the `verify` variable to find out the verification status.

The `flags` variable should be 0 for now.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

## gnutls\_ocsp\_resp\_verify\_direct

**int** gnutls\_ocsp\_resp\_verify\_direct (*gnutls\_ocsp\_resp\_t resp*, [Function]  
*gnutls\_x509\_cert\_t issuer*, *unsigned int \* verify*, *unsigned int flags*)

*resp*: should contain a *gnutls\_ocsp\_resp\_t* structure

*issuer*: certificate believed to have signed the response

*verify*: output variable with verification status, an *gnutls\_ocsp\_cert\_status\_t*

*flags*: verification flags, 0 for now.

Verify signature of the Basic OCSP Response against the public key in the *issuer* certificate.

The output *verify* variable will hold verification status codes (e.g., *GNUTLS\_OCSP\_VERIFY\_SIGNER\_NOT\_FOUND* , *GNUTLS\_OCSP\_VERIFY\_INSECURE\_ALGORITHM* ) which are only valid if the function returned *GNUTLS\_E\_SUCCESS* .

Note that the function returns *GNUTLS\_E\_SUCCESS* even when verification failed. The caller must always inspect the *verify* variable to find out the verification status.

The *flags* variable should be 0 for now.

**Returns:** On success, *GNUTLS\_E\_SUCCESS* (0) is returned, otherwise a negative error value.

## E.5 OpenPGP API

The following functions are to be used for OpenPGP certificate handling. Their prototypes lie in *gnutls/openpgp.h*.

### gnutls\_certificate\_set\_openpgp\_key

**int** gnutls\_certificate\_set\_openpgp\_key [Function]  
(*gnutls\_certificate\_credentials\_t res*, *gnutls\_openpgp\_cert\_t crt*,  
*gnutls\_openpgp\_privkey\_t pkey*)

*res*: is a *gnutls\_certificate\_credentials\_t* structure.

*crt*: contains an openpgp public key

*pkey*: is an openpgp private key

This function sets a certificate/private key pair in the *gnutls\_certificate\_credentials\_t* structure. This function may be called more than once (in case multiple keys/certificates exist for the server).

Note that this function requires that the preferred key ids have been set and be used. See *gnutls\_openpgp\_cert\_set\_preferred\_key\_id()* . Otherwise the master key will be used.

**Returns:** On success, *GNUTLS\_E\_SUCCESS* (0) is returned, otherwise a negative error code is returned.

### gnutls\_certificate\_set\_openpgp\_key\_file

**int** gnutls\_certificate\_set\_openpgp\_key\_file [Function]  
(*gnutls\_certificate\_credentials\_t res*, *const char \* certfile*, *const char \**  
*keyfile*, *gnutls\_openpgp\_cert\_fmt\_t format*)

*res*: the destination context to save the data.

*certfile*: the file that contains the public key.

*keyfile*: the file that contains the secret key.

*format*: the format of the keys

This function is used to load OpenPGP keys into the GnuTLS credentials structure. The file should contain at least one valid non encrypted subkey.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

## gnutls\_certificate\_set\_openpgp\_key\_file2

```
int gnutls_certificate_set_openpgp_key_file2           [Function]
    (gnutls_certificate_credentials_t res, const char * certfile, const char *
    keyfile, const char * subkey_id, gnutls_openpgp_cert_fmt_t format)
```

*res*: the destination context to save the data.

*certfile*: the file that contains the public key.

*keyfile*: the file that contains the secret key.

*subkey\_id*: a hex encoded subkey id

*format*: the format of the keys

This function is used to load OpenPGP keys into the GnuTLS credential structure. The file should contain at least one valid non encrypted subkey.

The special keyword "auto" is also accepted as *subkey\_id*. In that case the `gnutls_openpgp_cert_get_auth_subkey()` will be used to retrieve the subkey.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 2.4.0

## gnutls\_certificate\_set\_openpgp\_key\_mem

```
int gnutls_certificate_set_openpgp_key_mem           [Function]
    (gnutls_certificate_credentials_t res, const gnutls_datum_t * cert, const
    gnutls_datum_t * key, gnutls_openpgp_cert_fmt_t format)
```

*res*: the destination context to save the data.

*cert*: the datum that contains the public key.

*key*: the datum that contains the secret key.

*format*: the format of the keys

This function is used to load OpenPGP keys into the GnuTLS credential structure. The datum should contain at least one valid non encrypted subkey.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**gnutls\_certificate\_set\_openpgp\_key\_mem2**

**int gnutls\_certificate\_set\_openpgp\_key\_mem2** [Function]  
 (*gnutls\_certificate\_credentials\_t* **res**, *const gnutls\_datum\_t* \* **cert**, *const gnutls\_datum\_t* \* **key**, *const char* \* **subkey\_id**, *gnutls\_openpgp\_cert\_fmt\_t* **format**)

**res**: the destination context to save the data.

**cert**: the datum that contains the public key.

**key**: the datum that contains the secret key.

**subkey\_id**: a hex encoded subkey id

**format**: the format of the keys

This function is used to load OpenPGP keys into the GnuTLS credentials structure. The datum should contain at least one valid non encrypted subkey.

The special keyword "auto" is also accepted as **subkey\_id**. In that case the **gnutls\_openpgp\_cert\_get\_auth\_subkey()** will be used to retrieve the subkey.

**Returns**: On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since**: 2.4.0

**gnutls\_certificate\_set\_openpgp\_keyring\_file**

**int gnutls\_certificate\_set\_openpgp\_keyring\_file** [Function]  
 (*gnutls\_certificate\_credentials\_t* **c**, *const char* \* **file**, *gnutls\_openpgp\_cert\_fmt\_t* **format**)

**c**: A certificate credentials structure

**file**: filename of the keyring.

**format**: format of keyring.

The function is used to set keyrings that will be used internally by various OpenPGP functions. For example to find a key when it is needed for an operations. The keyring will also be used at the verification functions.

**Returns**: On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**gnutls\_certificate\_set\_openpgp\_keyring\_mem**

**int gnutls\_certificate\_set\_openpgp\_keyring\_mem** [Function]  
 (*gnutls\_certificate\_credentials\_t* **c**, *const uint8\_t* \* **data**, *size\_t* **dlen**, *gnutls\_openpgp\_cert\_fmt\_t* **format**)

**c**: A certificate credentials structure

**data**: buffer with keyring data.

**dlen**: length of data buffer.

**format**: the format of the keyring

The function is used to set keyrings that will be used internally by various OpenPGP functions. For example to find a key when it is needed for an operations. The keyring will also be used at the verification functions.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

## gnutls\_openpgp\_cert\_check\_hostname

**int gnutls\_openpgp\_cert\_check\_hostname** (*gnutls\_openpgp\_cert\_t* *key*, *const char \*hostname*) [Function]

*key*: should contain a *gnutls\_openpgp\_cert\_t* structure

*hostname*: A null terminated string that contains a DNS name

This function will check if the given key's owner matches the given hostname. This is a basic implementation of the matching described in RFC2818 (HTTPS), which takes into account wildcards.

**Returns:** non-zero for a successful match, and zero on failure.

## gnutls\_openpgp\_cert\_check\_hostname2

**int gnutls\_openpgp\_cert\_check\_hostname2** (*gnutls\_openpgp\_cert\_t* *key*, *const char \*hostname*, *unsigned flags*) [Function]

*key*: should contain a *gnutls\_openpgp\_cert\_t* structure

*hostname*: A null terminated string that contains a DNS name

*flags*: *gnutls\_certificate\_verify\_flags*

This function will check if the given key's owner matches the given hostname.

Unless, the flag GNUTLS\_VERIFY\_DO\_NOT\_ALLOW\_WILDCARDS is specified, wildcards are only considered if the domain name consists of three components or more, and the wildcard starts at the leftmost position.

**Returns:** non-zero for a successful match, and zero on failure.

## gnutls\_openpgp\_cert\_deinit

**void gnutls\_openpgp\_cert\_deinit** (*gnutls\_openpgp\_cert\_t* *key*) [Function]

*key*: The structure to be initialized

This function will deinitialize a key structure.

## gnutls\_openpgp\_cert\_export

**int gnutls\_openpgp\_cert\_export** (*gnutls\_openpgp\_cert\_t* *key*, *gnutls\_openpgp\_cert\_fmt\_t* *format*, *void \*output\_data*, *size\_t \*output\_data\_size*) [Function]

*key*: Holds the key.

*format*: One of *gnutls\_openpgp\_cert\_fmt\_t* elements.

*output\_data*: will contain the raw or base64 encoded key

*output\_data\_size*: holds the size of *output\_data* (and will be replaced by the actual size of parameters)

This function will convert the given key to RAW or Base64 format. If the buffer provided is not long enough to hold the output, then GNUTLS\_E\_SHORT\_MEMORY\_BUFFER will be returned.

**Returns:** GNUTLS\_E\_SUCCESS on success, or an error code.



**gnutls\_openpgp\_cert\_export2**

**int** gnutls\_openpgp\_cert\_export2 (*gnutls\_openpgp\_cert\_t* *key*, [Function]  
*gnutls\_openpgp\_cert\_fmt\_t* *format*, *gnutls\_datum\_t* \* *out*)

*key*: Holds the key.

*format*: One of gnutls\_openpgp\_cert\_fmt\_t elements.

*out*: will contain the raw or base64 encoded key

This function will convert the given key to RAW or Base64 format. The output buffer is allocated using `gnutls_malloc()` .

**Returns:** GNUTLS\_E\_SUCCESS on success, or an error code.

**Since:** 3.1.3

**gnutls\_openpgp\_cert\_get\_auth\_subkey**

**int** gnutls\_openpgp\_cert\_get\_auth\_subkey (*gnutls\_openpgp\_cert\_t* [Function]  
*cert*, *gnutls\_openpgp\_keyid\_t* *keyid*, *unsigned int* *flag*)

*cert*: the structure that contains the OpenPGP public key.

*keyid*: the struct to save the keyid.

*flag*: Non-zero indicates that a valid subkey is always returned.

Returns the 64-bit keyID of the first valid OpenPGP subkey marked for authentication. If flag is non-zero and no authentication subkey exists, then a valid subkey will be returned even if it is not marked for authentication.

**Returns:** GNUTLS\_E\_SUCCESS on success, or an error code.

**gnutls\_openpgp\_cert\_get\_creation\_time**

**time\_t** gnutls\_openpgp\_cert\_get\_creation\_time [Function]  
(*gnutls\_openpgp\_cert\_t* *key*)

*key*: the structure that contains the OpenPGP public key.

Get key creation time.

**Returns:** the timestamp when the OpenPGP key was created.

**gnutls\_openpgp\_cert\_get\_expiration\_time**

**time\_t** gnutls\_openpgp\_cert\_get\_expiration\_time [Function]  
(*gnutls\_openpgp\_cert\_t* *key*)

*key*: the structure that contains the OpenPGP public key.

Get key expiration time. A value of '0' means that the key doesn't expire at all.

**Returns:** the time when the OpenPGP key expires.

**gnutls\_openpgp\_cert\_get\_fingerprint**

**int** gnutls\_openpgp\_cert\_get\_fingerprint (*gnutls\_openpgp\_cert\_t* [Function]  
*key*, *void* \* *fpr*, *size\_t* \* *fprlen*)

*key*: the raw data that contains the OpenPGP public key.

*fpr*: the buffer to save the fingerprint, must hold at least 20 bytes.

*fprlen*: the integer to save the length of the fingerprint.

Get key fingerprint. Depending on the algorithm, the fingerprint can be 16 or 20 bytes.

**Returns:** On success, 0 is returned. Otherwise, an error code.

### **gnutls\_openpgp\_cert\_get\_key\_id**

**int gnutls\_openpgp\_cert\_get\_key\_id** (*gnutls\_openpgp\_cert\_t key*, [Function]  
*gnutls\_openpgp\_keyid\_t keyid*)

*key*: the structure that contains the OpenPGP public key.

*keyid*: the buffer to save the keyid.

Get key id string.

**Returns:** the 64-bit keyID of the OpenPGP key.

**Since:** 2.4.0

### **gnutls\_openpgp\_cert\_get\_key\_usage**

**int gnutls\_openpgp\_cert\_get\_key\_usage** (*gnutls\_openpgp\_cert\_t key*, [Function]  
*unsigned int \*key\_usage*)

*key*: should contain a gnutls\_openpgp\_cert\_t structure

*key\_usage*: where the key usage bits will be stored

This function will return certificate's key usage, by checking the key algorithm. The key usage value will ORed values of the: GNUTLS\_KEY\_DIGITAL\_SIGNATURE , GNUTLS\_KEY\_KEY\_ENCIPHERMENT .

**Returns:** GNUTLS\_E\_SUCCESS on success, or an error code.

### **gnutls\_openpgp\_cert\_get\_name**

**int gnutls\_openpgp\_cert\_get\_name** (*gnutls\_openpgp\_cert\_t key*, *int* [Function]  
*idx*, *char \*buf*, *size\_t \*sizeof\_buf*)

*key*: the structure that contains the OpenPGP public key.

*idx*: the index of the ID to extract

*buf*: a pointer to a structure to hold the name, may be NULL to only get the **sizeof\_buf** .

*sizeof\_buf*: holds the maximum size of *buf* , on return hold the actual/required size of *buf* .

Extracts the userID from the parsed OpenPGP key.

**Returns:** GNUTLS\_E\_SUCCESS on success, and if the index of the ID does not exist GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE , or an error code.

### **gnutls\_openpgp\_cert\_get\_pk\_algorithm**

**gnutls\_pk\_algorithm\_t gnutls\_openpgp\_cert\_get\_pk\_algorithm** [Function]  
(*gnutls\_openpgp\_cert\_t key*, *unsigned int \*bits*)

*key*: is an OpenPGP key

*bits*: if *bits* is non null it will hold the size of the parameters' in bits

This function will return the public key algorithm of an OpenPGP certificate.

If *bits* is non null, it should have enough size to hold the parameters size in bits. For RSA the bits returned is the modulus. For DSA the bits returned are of the public exponent.

**Returns:** a member of the `gnutls_pk_algorithm_t` enumeration on success, or `GNUTLS_PK_UNKNOWN` on error.

### `gnutls_openpgp_cert_get_pk_dsa_raw`

`int gnutls_openpgp_cert_get_pk_dsa_raw (gnutls_openpgp_cert_t crt, gnutls_datum_t * p, gnutls_datum_t * q, gnutls_datum_t * g, gnutls_datum_t * y)` [Function]

*crt*: Holds the certificate

*p*: will hold the p

*q*: will hold the q

*g*: will hold the g

*y*: will hold the y

This function will export the DSA public key's parameters found in the given certificate. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

**Returns:** `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

**Since:** 2.4.0

### `gnutls_openpgp_cert_get_pk_rsa_raw`

`int gnutls_openpgp_cert_get_pk_rsa_raw (gnutls_openpgp_cert_t crt, gnutls_datum_t * m, gnutls_datum_t * e)` [Function]

*crt*: Holds the certificate

*m*: will hold the modulus

*e*: will hold the public exponent

This function will export the RSA public key's parameters found in the given structure. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

**Returns:** `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

**Since:** 2.4.0

### `gnutls_openpgp_cert_get_preferred_key_id`

`int gnutls_openpgp_cert_get_preferred_key_id (gnutls_openpgp_cert_t key, gnutls_openpgp_keyid_t keyid)` [Function]

*key*: the structure that contains the OpenPGP public key.

*keyid*: the struct to save the keyid.

Get preferred key id. If it hasn't been set it returns `GNUTLS_E_INVALID_REQUEST`.

**Returns:** the 64-bit preferred keyID of the OpenPGP key.

**gnutls\_openpgp\_cert\_get\_revoked\_status**

**int gnutls\_openpgp\_cert\_get\_revoked\_status** [Function]

(*gnutls\_openpgp\_cert\_t key*)

*key*: the structure that contains the OpenPGP public key.

Get revocation status of key.

**Returns:** true (1) if the key has been revoked, or false (0) if it has not.

**Since:** 2.4.0

**gnutls\_openpgp\_cert\_get\_subkey\_count**

**int gnutls\_openpgp\_cert\_get\_subkey\_count** (*gnutls\_openpgp\_cert\_t key*) [Function]

*key*: is an OpenPGP key

This function will return the number of subkeys present in the given OpenPGP certificate.

**Returns:** the number of subkeys, or a negative error code on error.

**Since:** 2.4.0

**gnutls\_openpgp\_cert\_get\_subkey\_creation\_time**

**time\_t gnutls\_openpgp\_cert\_get\_subkey\_creation\_time** (*gnutls\_openpgp\_cert\_t key, unsigned int idx*) [Function]

*key*: the structure that contains the OpenPGP public key.

*idx*: the subkey index

Get subkey creation time.

**Returns:** the timestamp when the OpenPGP sub-key was created.

**Since:** 2.4.0

**gnutls\_openpgp\_cert\_get\_subkey\_expiration\_time**

**time\_t gnutls\_openpgp\_cert\_get\_subkey\_expiration\_time** (*gnutls\_openpgp\_cert\_t key, unsigned int idx*) [Function]

*key*: the structure that contains the OpenPGP public key.

*idx*: the subkey index

Get subkey expiration time. A value of '0' means that the key doesn't expire at all.

**Returns:** the time when the OpenPGP key expires.

**Since:** 2.4.0

**gnutls\_openpgp\_cert\_get\_subkey\_fingerprint**

**int gnutls\_openpgp\_cert\_get\_subkey\_fingerprint** (*gnutls\_openpgp\_cert\_t key, unsigned int idx, void \* fpr, size\_t \* fprlen*) [Function]

*key*: the raw data that contains the OpenPGP public key.

*idx*: the subkey index

*fpr*: the buffer to save the fingerprint, must hold at least 20 bytes.

*fprlen*: the integer to save the length of the fingerprint.

Get key fingerprint of a subkey. Depending on the algorithm, the fingerprint can be 16 or 20 bytes.

**Returns:** On success, 0 is returned. Otherwise, an error code.

**Since:** 2.4.0

## **gnutls\_openpgp\_cert\_get\_subkey\_id**

**int** gnutls\_openpgp\_cert\_get\_subkey\_id (*gnutls\_openpgp\_cert\_t* **key**, [Function]  
   *unsigned int* **idx**, *gnutls\_openpgp\_keyid\_t* **keyid**)

*key*: the structure that contains the OpenPGP public key.

*idx*: the subkey index

*keyid*: the buffer to save the keyid.

Get the subkey's key-id.

**Returns:** the 64-bit keyID of the OpenPGP key.

## **gnutls\_openpgp\_cert\_get\_subkey\_idx**

**int** gnutls\_openpgp\_cert\_get\_subkey\_idx (*gnutls\_openpgp\_cert\_t* [Function]  
   **key**, *const gnutls\_openpgp\_keyid\_t* **keyid**)

*key*: the structure that contains the OpenPGP public key.

*keyid*: the keyid.

Get subkey's index.

**Returns:** the index of the subkey or a negative error value.

**Since:** 2.4.0

## **gnutls\_openpgp\_cert\_get\_subkey\_pk\_algorithm**

*gnutls\_pk\_algorithm\_t* [Function]  
   gnutls\_openpgp\_cert\_get\_subkey\_pk\_algorithm (*gnutls\_openpgp\_cert\_t*  
   **key**, *unsigned int* **idx**, *unsigned int \** **bits**)

*key*: is an OpenPGP key

*idx*: is the subkey index

*bits*: if bits is non null it will hold the size of the parameters' in bits

This function will return the public key algorithm of a subkey of an OpenPGP certificate.

If bits is non null, it should have enough size to hold the parameters size in bits. For RSA the bits returned is the modulus. For DSA the bits returned are of the public exponent.

**Returns:** a member of the *gnutls\_pk\_algorithm\_t* enumeration on success, or GNUTLS\_PK\_UNKNOWN on error.

**Since:** 2.4.0

**gnutls\_openpgp\_cert\_get\_subkey\_pk\_dsa\_raw**

**int gnutls\_openpgp\_cert\_get\_subkey\_pk\_dsa\_raw** [Function]

(*gnutls\_openpgp\_cert\_t crt*, unsigned int *idx*, *gnutls\_datum\_t* \* *p*,  
*gnutls\_datum\_t* \* *q*, *gnutls\_datum\_t* \* *g*, *gnutls\_datum\_t* \* *y*)

*crt*: Holds the certificate

*idx*: Is the subkey index

*p*: will hold the p

*q*: will hold the q

*g*: will hold the g

*y*: will hold the y

This function will export the DSA public key's parameters found in the given certificate. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

**Returns:** GNUTLS\_E\_SUCCESS on success, otherwise a negative error code.

**Since:** 2.4.0

**gnutls\_openpgp\_cert\_get\_subkey\_pk\_rsa\_raw**

**int gnutls\_openpgp\_cert\_get\_subkey\_pk\_rsa\_raw** [Function]

(*gnutls\_openpgp\_cert\_t crt*, unsigned int *idx*, *gnutls\_datum\_t* \* *m*,  
*gnutls\_datum\_t* \* *e*)

*crt*: Holds the certificate

*idx*: Is the subkey index

*m*: will hold the modulus

*e*: will hold the public exponent

This function will export the RSA public key's parameters found in the given structure. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

**Returns:** GNUTLS\_E\_SUCCESS on success, otherwise a negative error code.

**Since:** 2.4.0

**gnutls\_openpgp\_cert\_get\_subkey\_revoked\_status**

**int gnutls\_openpgp\_cert\_get\_subkey\_revoked\_status** [Function]

(*gnutls\_openpgp\_cert\_t key*, unsigned int *idx*)

*key*: the structure that contains the OpenPGP public key.

*idx*: is the subkey index

Get subkey revocation status. A negative error code indicates an error.

**Returns:** true (1) if the key has been revoked, or false (0) if it has not.

**Since:** 2.4.0

**gnutls\_openpgp\_cert\_get\_subkey\_usage**

**int gnutls\_openpgp\_cert\_get\_subkey\_usage** (*gnutls\_openpgp\_cert\_t key, unsigned int idx, unsigned int \* key\_usage*) [Function]

*key*: should contain a gnutls\_openpgp\_cert\_t structure

*idx*: the subkey index

*key\_usage*: where the key usage bits will be stored

This function will return certificate's key usage, by checking the key algorithm. The key usage value will ORed values of GNUTLS\_KEY\_DIGITAL\_SIGNATURE or GNUTLS\_KEY\_KEY\_ENCIIPHERMENT .

A negative error code may be returned in case of parsing error.

**Returns:** key usage value.

**Since:** 2.4.0

**gnutls\_openpgp\_cert\_get\_version**

**int gnutls\_openpgp\_cert\_get\_version** (*gnutls\_openpgp\_cert\_t key*) [Function]

*key*: the structure that contains the OpenPGP public key.

Extract the version of the OpenPGP key.

**Returns:** the version number is returned, or a negative error code on errors.

**gnutls\_openpgp\_cert\_import**

**int gnutls\_openpgp\_cert\_import** (*gnutls\_openpgp\_cert\_t key, const gnutls\_datum\_t \* data, gnutls\_openpgp\_cert\_fmt\_t format*) [Function]

*key*: The structure to store the parsed key.

*data*: The RAW or BASE64 encoded key.

*format*: One of gnutls\_openpgp\_cert\_fmt\_t elements.

This function will convert the given RAW or Base64 encoded key to the native gnutls\_openpgp\_cert\_t format. The output will be stored in 'key'.

**Returns:** GNUTLS\_E\_SUCCESS on success, or an error code.

**gnutls\_openpgp\_cert\_init**

**int gnutls\_openpgp\_cert\_init** (*gnutls\_openpgp\_cert\_t \* key*) [Function]

*key*: The structure to be initialized

This function will initialize an OpenPGP key structure.

**Returns:** GNUTLS\_E\_SUCCESS on success, or an error code.

**gnutls\_openpgp\_cert\_print**

**int gnutls\_openpgp\_cert\_print** (*gnutls\_openpgp\_cert\_t cert, gnutls\_certificate\_print\_formats\_t format, gnutls\_datum\_t \* out*) [Function]

*cert*: The structure to be printed

*format*: Indicate the format to use

*out*: Newly allocated datum with (0) terminated string.

This function will pretty print an OpenPGP certificate, suitable for display to a human.

The format should be (0) for future compatibility.

The output *out* needs to be deallocate using `gnutls_free()` .

**Returns:** GNUTLS\_E\_SUCCESS on success, or an error code.

## **gnutls\_openpgp\_cert\_set\_preferred\_key\_id**

**int gnutls\_openpgp\_cert\_set\_preferred\_key\_id** [Function]  
     (*gnutls\_openpgp\_cert\_t key*, *const gnutls\_openpgp\_keyid\_t keyid*)

*key*: the structure that contains the OpenPGP public key.

*keyid*: the selected keyid

This allows setting a preferred key id for the given certificate. This key will be used by functions that involve key handling.

If the provided *keyid* is NULL then the master key is set as preferred.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error code is returned.

## **gnutls\_openpgp\_cert\_verify\_ring**

**int gnutls\_openpgp\_cert\_verify\_ring** (*gnutls\_openpgp\_cert\_t key*, [Function]  
     *gnutls\_openpgp\_keyring\_t keyring*, *unsigned int flags*, *unsigned int \* verify*)

*key*: the structure that holds the key.

*keyring*: holds the keyring to check against

*flags*: unused (should be 0)

*verify*: will hold the certificate verification output.

Verify all signatures in the key, using the given set of keys (*keyring*).

The key verification output will be put in *verify* and will be one or more of the `gnutls_certificate_status_t` enumerated elements bitwise or'd.

Note that this function does not verify using any "web of trust". You may use GnuPG for that purpose, or any other external PGP application.

**Returns:** GNUTLS\_E\_SUCCESS on success, or an error code.

## **gnutls\_openpgp\_cert\_verify\_self**

**int gnutls\_openpgp\_cert\_verify\_self** (*gnutls\_openpgp\_cert\_t key*, [Function]  
     *unsigned int flags*, *unsigned int \* verify*)

*key*: the structure that holds the key.

*flags*: unused (should be 0)

*verify*: will hold the key verification output.

Verifies the self signature in the key. The key verification output will be put in *verify* and will be one or more of the `gnutls_certificate_status_t` enumerated elements bitwise or'd.

**Returns:** GNUTLS\_E\_SUCCESS on success, or an error code.



**gnutls\_openpgp\_keyring\_check\_id**

**int gnutls\_openpgp\_keyring\_check\_id** (*gnutls\_openpgp\_keyring\_t* *ring*, *const gnutls\_openpgp\_keyid\_t* *keyid*, *unsigned int* *flags*) [Function]

*ring*: holds the keyring to check against

*keyid*: will hold the keyid to check for.

*flags*: unused (should be 0)

Check if a given key ID exists in the keyring.

**Returns:** GNUTLS\_E\_SUCCESS on success (if keyid exists) and a negative error code on failure.

**gnutls\_openpgp\_keyring\_deinit**

**void gnutls\_openpgp\_keyring\_deinit** (*gnutls\_openpgp\_keyring\_t* *keyring*) [Function]

*keyring*: The structure to be initialized

This function will deinitialize a keyring structure.

**gnutls\_openpgp\_keyring\_get\_cert**

**int gnutls\_openpgp\_keyring\_get\_cert** (*gnutls\_openpgp\_keyring\_t* *ring*, *unsigned int* *idx*, *gnutls\_openpgp\_cert\_t* \* *cert*) [Function]

*ring*: Holds the keyring.

*idx*: the index of the certificate to export

*cert*: An uninitialized **gnutls\_openpgp\_cert\_t** structure

This function will extract an OpenPGP certificate from the given keyring. If the index given is out of range GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE will be returned. The returned structure needs to be deinit.

**Returns:** GNUTLS\_E\_SUCCESS on success, or an error code.

**gnutls\_openpgp\_keyring\_get\_cert\_count**

**int gnutls\_openpgp\_keyring\_get\_cert\_count** (*gnutls\_openpgp\_keyring\_t* *ring*) [Function]

*ring*: is an OpenPGP key ring

This function will return the number of OpenPGP certificates present in the given keyring.

**Returns:** the number of subkeys, or a negative error code on error.

**gnutls\_openpgp\_keyring\_import**

**int gnutls\_openpgp\_keyring\_import** (*gnutls\_openpgp\_keyring\_t* *keyring*, *const gnutls\_datum\_t* \* *data*, *gnutls\_openpgp\_cert\_fmt\_t* *format*) [Function]

*keyring*: The structure to store the parsed key.

*data*: The RAW or BASE64 encoded keyring.

*format*: One of **gnutls\_openpgp\_keyring\_fmt** elements.

This function will convert the given RAW or Base64 encoded keyring to the native `gnutls_openpgp_keyring_t` format. The output will be stored in 'keyring'.

**Returns:** GNUTLS\_E\_SUCCESS on success, or an error code.

### **gnutls\_openpgp\_keyring\_init**

```
int gnutls_openpgp_keyring_init (gnutls_openpgp_keyring_t *      [Function]
                                keyring)
```

*keyring*: The structure to be initialized

This function will initialize an keyring structure.

**Returns:** GNUTLS\_E\_SUCCESS on success, or an error code.

### **gnutls\_openpgp\_privkey\_deinit**

```
void gnutls_openpgp_privkey_deinit (gnutls_openpgp_privkey_t    [Function]
                                     key)
```

*key*: The structure to be initialized

This function will deinitialize a key structure.

### **gnutls\_openpgp\_privkey\_export**

```
int gnutls_openpgp_privkey_export (gnutls_openpgp_privkey_t key,  [Function]
                                   gnutls_openpgp_cert_fmt_t format, const char * password, unsigned int
                                   flags, void * output_data, size_t * output_data_size)
```

*key*: Holds the key.

*format*: One of `gnutls_openpgp_cert_fmt_t` elements.

*password*: the password that will be used to encrypt the key. (unused for now)

*flags*: (0) for future compatibility

*output\_data*: will contain the key base64 encoded or raw

*output\_data\_size*: holds the size of *output\_data* (and will be replaced by the actual size of parameters)

This function will convert the given key to RAW or Base64 format. If the buffer provided is not long enough to hold the output, then GNUTLS\_E\_SHORT\_MEMORY\_BUFFER will be returned.

**Returns:** GNUTLS\_E\_SUCCESS on success, or an error code.

**Since:** 2.4.0

### **gnutls\_openpgp\_privkey\_export2**

```
int gnutls_openpgp_privkey_export2 (gnutls_openpgp_privkey_t    [Function]
                                     key, gnutls_openpgp_cert_fmt_t format, const char * password, unsigned int
                                     flags, gnutls_datum_t * out)
```

*key*: Holds the key.

*format*: One of `gnutls_openpgp_cert_fmt_t` elements.

*password*: the password that will be used to encrypt the key. (unused for now)

*flags*: (0) for future compatibility

*out*: will contain the raw or based64 encoded key

This function will convert the given key to RAW or Base64 format. The output buffer is allocated using `gnutls_malloc()` .

**Returns:** GNUTLS\_E\_SUCCESS on success, or an error code.

**Since:** 3.1.3

### **gnutls\_openpgp\_privkey\_export\_dsa\_raw**

```
int gnutls_openpgp_privkey_export_dsa_raw [Function]
    (gnutls_openpgp_privkey_t pkey, gnutls_datum_t * p, gnutls_datum_t * q,
     gnutls_datum_t * g, gnutls_datum_t * y, gnutls_datum_t * x)
```

*pkey*: Holds the certificate

*p*: will hold the p

*q*: will hold the q

*g*: will hold the g

*y*: will hold the y

*x*: will hold the x

This function will export the DSA private key's parameters found in the given certificate. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

**Returns:** GNUTLS\_E\_SUCCESS on success, otherwise a negative error code.

**Since:** 2.4.0

### **gnutls\_openpgp\_privkey\_export\_rsa\_raw**

```
int gnutls_openpgp_privkey_export_rsa_raw [Function]
    (gnutls_openpgp_privkey_t pkey, gnutls_datum_t * m, gnutls_datum_t * e,
     gnutls_datum_t * d, gnutls_datum_t * p, gnutls_datum_t * q, gnutls_datum_t *
     u)
```

*pkey*: Holds the certificate

*m*: will hold the modulus

*e*: will hold the public exponent

*d*: will hold the private exponent

*p*: will hold the first prime (p)

*q*: will hold the second prime (q)

*u*: will hold the coefficient

This function will export the RSA private key's parameters found in the given structure. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

**Returns:** GNUTLS\_E\_SUCCESS on success, otherwise a negative error code.

**Since:** 2.4.0

**gnutls\_openpgp\_privkey\_export\_subkey\_dsa\_raw**

```
int gnutls_openpgp_privkey_export_subkey_dsa_raw [Function]
    (gnutls_openpgp_privkey_t pkey, unsigned int idx, gnutls_datum_t * p,
     gnutls_datum_t * q, gnutls_datum_t * g, gnutls_datum_t * y, gnutls_datum_t *
     x)
```

*pkey*: Holds the certificate

*idx*: Is the subkey index

*p*: will hold the p

*q*: will hold the q

*g*: will hold the g

*y*: will hold the y

*x*: will hold the x

This function will export the DSA private key's parameters found in the given certificate. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

**Returns:** GNUTLS\_E\_SUCCESS on success, otherwise a negative error code.

**Since:** 2.4.0

**gnutls\_openpgp\_privkey\_export\_subkey\_rsa\_raw**

```
int gnutls_openpgp_privkey_export_subkey_rsa_raw [Function]
    (gnutls_openpgp_privkey_t pkey, unsigned int idx, gnutls_datum_t * m,
     gnutls_datum_t * e, gnutls_datum_t * d, gnutls_datum_t * p, gnutls_datum_t *
     q, gnutls_datum_t * u)
```

*pkey*: Holds the certificate

*idx*: Is the subkey index

*m*: will hold the modulus

*e*: will hold the public exponent

*d*: will hold the private exponent

*p*: will hold the first prime (p)

*q*: will hold the second prime (q)

*u*: will hold the coefficient

This function will export the RSA private key's parameters found in the given structure. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

**Returns:** GNUTLS\_E\_SUCCESS on success, otherwise a negative error code.

**Since:** 2.4.0

**gnutls\_openpgp\_privkey\_get\_fingerprint**

**int gnutls\_openpgp\_privkey\_get\_fingerprint** [Function]

(*gnutls\_openpgp\_privkey\_t* *key*, *void \*fpr*, *size\_t \*fprlen*)

*key*: the raw data that contains the OpenPGP secret key.

*fpr*: the buffer to save the fingerprint, must hold at least 20 bytes.

*fprlen*: the integer to save the length of the fingerprint.

Get the fingerprint of the OpenPGP key. Depends on the algorithm, the fingerprint can be 16 or 20 bytes.

**Returns:** On success, 0 is returned, or an error code.

**Since:** 2.4.0

**gnutls\_openpgp\_privkey\_get\_key\_id**

**int gnutls\_openpgp\_privkey\_get\_key\_id** (*gnutls\_openpgp\_privkey\_t* [Function]

*key*, *gnutls\_openpgp\_keyid\_t* *keyid*)

*key*: the structure that contains the OpenPGP secret key.

*keyid*: the buffer to save the keyid.

Get key-id.

**Returns:** the 64-bit keyID of the OpenPGP key.

**Since:** 2.4.0

**gnutls\_openpgp\_privkey\_get\_pk\_algorithm**

**gnutls\_pk\_algorithm\_t** [Function]

**gnutls\_openpgp\_privkey\_get\_pk\_algorithm** (*gnutls\_openpgp\_privkey\_t*  
*key*, *unsigned int \*bits*)

*key*: is an OpenPGP key

*bits*: if bits is non null it will hold the size of the parameters' in bits

This function will return the public key algorithm of an OpenPGP certificate.

If bits is non null, it should have enough size to hold the parameters size in bits. For RSA the bits returned is the modulus. For DSA the bits returned are of the public exponent.

**Returns:** a member of the *gnutls\_pk\_algorithm\_t* enumeration on success, or a negative error code on error.

**Since:** 2.4.0

**gnutls\_openpgp\_privkey\_get\_preferred\_key\_id**

**int gnutls\_openpgp\_privkey\_get\_preferred\_key\_id** [Function]

(*gnutls\_openpgp\_privkey\_t* *key*, *gnutls\_openpgp\_keyid\_t* *keyid*)

*key*: the structure that contains the OpenPGP public key.

*keyid*: the struct to save the keyid.

Get the preferred key-id for the key.

**Returns:** the 64-bit preferred keyID of the OpenPGP key, or if it hasn't been set it returns *GNUTLS\_E\_INVALID\_REQUEST* .

## **gnutls\_openpgp\_privkey\_get\_revoked\_status**

**int gnutls\_openpgp\_privkey\_get\_revoked\_status** [Function]

(*gnutls\_openpgp\_privkey\_t key*)

*key*: the structure that contains the OpenPGP private key.

Get revocation status of key.

**Returns:** true (1) if the key has been revoked, or false (0) if it has not, or a negative error code indicates an error.

**Since:** 2.4.0

## **gnutls\_openpgp\_privkey\_get\_subkey\_count**

**int gnutls\_openpgp\_privkey\_get\_subkey\_count** [Function]

(*gnutls\_openpgp\_privkey\_t key*)

*key*: is an OpenPGP key

This function will return the number of subkeys present in the given OpenPGP certificate.

**Returns:** the number of subkeys, or a negative error code on error.

**Since:** 2.4.0

## **gnutls\_openpgp\_privkey\_get\_subkey\_creation\_time**

**time\_t gnutls\_openpgp\_privkey\_get\_subkey\_creation\_time** [Function]

(*gnutls\_openpgp\_privkey\_t key*, *unsigned int idx*)

*key*: the structure that contains the OpenPGP private key.

*idx*: the subkey index

Get subkey creation time.

**Returns:** the timestamp when the OpenPGP key was created.

**Since:** 2.4.0

## **gnutls\_openpgp\_privkey\_get\_subkey\_expiration\_time**

**time\_t gnutls\_openpgp\_privkey\_get\_subkey\_expiration\_time** [Function]

(*gnutls\_openpgp\_privkey\_t key*, *unsigned int idx*)

*key*: the structure that contains the OpenPGP private key.

*idx*: the subkey index

Get subkey expiration time. A value of '0' means that the key doesn't expire at all.

**Returns:** the time when the OpenPGP key expires.

**Since:** 2.4.0

## **gnutls\_openpgp\_privkey\_get\_subkey\_fingerprint**

**int gnutls\_openpgp\_privkey\_get\_subkey\_fingerprint** [Function]

(*gnutls\_openpgp\_privkey\_t key*, *unsigned int idx*, *void \* fpr*, *size\_t \* fprlen*)

*key*: the raw data that contains the OpenPGP secret key.

*idx*: the subkey index

*fpr*: the buffer to save the fingerprint, must hold at least 20 bytes.

*fprlen*: the integer to save the length of the fingerprint.

Get the fingerprint of an OpenPGP subkey. Depends on the algorithm, the fingerprint can be 16 or 20 bytes.

**Returns:** On success, 0 is returned, or an error code.

**Since:** 2.4.0

## **gnutls\_openpgp\_privkey\_get\_subkey\_id**

```
int gnutls_openpgp_privkey_get_subkey_id [Function]
    (gnutls_openpgp_privkey_t key, unsigned int idx, gnutls_openpgp_keyid_t
    keyid)
```

*key*: the structure that contains the OpenPGP secret key.

*idx*: the subkey index

*keyid*: the buffer to save the keyid.

Get the key-id for the subkey.

**Returns:** the 64-bit keyID of the OpenPGP key.

**Since:** 2.4.0

## **gnutls\_openpgp\_privkey\_get\_subkey\_idx**

```
int gnutls_openpgp_privkey_get_subkey_idx [Function]
    (gnutls_openpgp_privkey_t key, const gnutls_openpgp_keyid_t keyid)
```

*key*: the structure that contains the OpenPGP private key.

*keyid*: the keyid.

Get index of subkey.

**Returns:** the index of the subkey or a negative error value.

**Since:** 2.4.0

## **gnutls\_openpgp\_privkey\_get\_subkey\_pk\_algorithm**

```
gnutls_pk_algorithm_t gnutls_openpgp_privkey_get_subkey_pk_algorithm [Function]
    (gnutls_openpgp_privkey_t key, unsigned int idx, unsigned int * bits)
```

*key*: is an OpenPGP key

*idx*: is the subkey index

*bits*: if bits is non null it will hold the size of the parameters' in bits

This function will return the public key algorithm of a subkey of an OpenPGP certificate.

If bits is non null, it should have enough size to hold the parameters size in bits. For RSA the bits returned is the modulus. For DSA the bits returned are of the public exponent.

**Returns:** a member of the `gnutls_pk_algorithm_t` enumeration on success, or a negative error code on error.

**Since:** 2.4.0

**gnutls\_openpgp\_privkey\_get\_subkey\_revoked\_status**

**int gnutls\_openpgp\_privkey\_get\_subkey\_revoked\_status** [Function]  
 (*gnutls\_openpgp\_privkey\_t* **key**, *unsigned int* **idx**)

**key**: the structure that contains the OpenPGP private key.

**idx**: is the subkey index

Get revocation status of key.

**Returns:** true (1) if the key has been revoked, or false (0) if it has not, or a negative error code indicates an error.

**Since:** 2.4.0

**gnutls\_openpgp\_privkey\_import**

**int gnutls\_openpgp\_privkey\_import** (*gnutls\_openpgp\_privkey\_t* **key**, [Function]  
*const gnutls\_datum\_t* \* **data**, *gnutls\_openpgp\_crt\_fmt\_t* **format**, *const char* \*  
**password**, *unsigned int* **flags**)

**key**: The structure to store the parsed key.

**data**: The RAW or BASE64 encoded key.

**format**: One of *gnutls\_openpgp\_crt\_fmt\_t* elements.

**password**: not used for now

**flags**: should be (0)

This function will convert the given RAW or Base64 encoded key to the native *gnutls\_openpgp\_privkey\_t* format. The output will be stored in 'key'.

**Returns:** GNUTLS\_E\_SUCCESS on success, or an error code.

**gnutls\_openpgp\_privkey\_init**

**int gnutls\_openpgp\_privkey\_init** (*gnutls\_openpgp\_privkey\_t* \* **key**) [Function]  
**key**: The structure to be initialized

This function will initialize an OpenPGP key structure.

**Returns:** GNUTLS\_E\_SUCCESS on success, or an error code.

**gnutls\_openpgp\_privkey\_sec\_param**

**gnutls\_sec\_param\_t gnutls\_openpgp\_privkey\_sec\_param** [Function]  
 (*gnutls\_openpgp\_privkey\_t* **key**)

**key**: a key structure

This function will return the security parameter appropriate with this private key.

**Returns:** On success, a valid security parameter is returned otherwise GNUTLS\_SEC\_PARAM\_UNKNOWN is returned.

**Since:** 2.12.0



**gnutls\_openpgp\_privkey\_set\_preferred\_key\_id**

**int gnutls\_openpgp\_privkey\_set\_preferred\_key\_id** [Function]

(*gnutls\_openpgp\_privkey\_t* **key**, *const gnutls\_openpgp\_keyid\_t* **keyid**)

*key*: the structure that contains the OpenPGP public key.

*keyid*: the selected keyid

This allows setting a preferred key id for the given certificate. This key will be used by functions that involve key handling.

If the provided **keyid** is NULL then the master key is set as preferred.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error code is returned.

**gnutls\_openpgp\_set\_recv\_key\_function**

**void gnutls\_openpgp\_set\_recv\_key\_function** (*gnutls\_session\_t* [Function]

*session*, *gnutls\_openpgp\_recv\_key\_func* **func**)

*session*: a TLS session

*func*: the callback

This function will set a key retrieval function for OpenPGP keys. This callback is only useful in server side, and will be used if the peer sent a key fingerprint instead of a full key.

The retrieved key must be allocated using **gnutls\_malloc()** .

**E.6 PKCS 12 API**

The following functions are to be used for PKCS 12 handling. Their prototypes lie in **gnutls/pkcs12.h**.

**gnutls\_pkcs12\_bag\_decrypt**

**int gnutls\_pkcs12\_bag\_decrypt** (*gnutls\_pkcs12\_bag\_t* **bag**, *const char* [Function]  
\* **pass**)

*bag*: The bag

*pass*: The password used for encryption, must be ASCII.

This function will decrypt the given encrypted bag and return 0 on success.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error code is returned.

**gnutls\_pkcs12\_bag\_deinit**

**void gnutls\_pkcs12\_bag\_deinit** (*gnutls\_pkcs12\_bag\_t* **bag**) [Function]

*bag*: The structure to be initialized

This function will deinitialize a PKCS12 Bag structure.

**gnutls\_pkcs12\_bag\_encrypt**

**int gnutls\_pkcs12\_bag\_encrypt** (*gnutls\_pkcs12\_bag\_t bag*, *const char \*pass*, *unsigned int flags*) [Function]

*bag*: The bag

*pass*: The password used for encryption, must be ASCII

*flags*: should be one of **gnutls\_pkcs\_encrypt\_flags\_t** elements bitwise or'd

This function will encrypt the given bag.

**Returns:** On success, **GNUTLS\_E\_SUCCESS** (0) is returned, otherwise a negative error code is returned.

**gnutls\_pkcs12\_bag\_get\_count**

**int gnutls\_pkcs12\_bag\_get\_count** (*gnutls\_pkcs12\_bag\_t bag*) [Function]

*bag*: The bag

This function will return the number of the elements withing the bag.

**Returns:** Number of elements in bag, or an negative error code on error.

**gnutls\_pkcs12\_bag\_get\_data**

**int gnutls\_pkcs12\_bag\_get\_data** (*gnutls\_pkcs12\_bag\_t bag*, *int indx*, *gnutls\_datum\_t \*data*) [Function]

*bag*: The bag

*indx*: The element of the bag to get the data from

*data*: where the bag's data will be. Should be treated as constant.

This function will return the bag's data. The data is a constant that is stored into the bag. Should not be accessed after the bag is deleted.

**Returns:** On success, **GNUTLS\_E\_SUCCESS** (0) is returned, otherwise a negative error value.

**gnutls\_pkcs12\_bag\_get\_friendly\_name**

**int gnutls\_pkcs12\_bag\_get\_friendly\_name** (*gnutls\_pkcs12\_bag\_t bag*, *int indx*, *char \*\*name*) [Function]

*bag*: The bag

*indx*: The bag's element to add the id

*name*: will hold a pointer to the name (to be treated as const)

This function will return the friendly name, of the specified bag element. The key ID is usually used to distinguish the local private key and the certificate pair.

**Returns:** On success, **GNUTLS\_E\_SUCCESS** (0) is returned, otherwise a negative error value. or a negative error code on error.

**gnutls\_pkcs12\_bag\_get\_key\_id**

**int gnutls\_pkcs12\_bag\_get\_key\_id** (*gnutls\_pkcs12\_bag\_t bag*, *int indx*, *gnutls\_datum\_t \* id*) [Function]

*bag*: The bag

*indx*: The bag's element to add the id

*id*: where the ID will be copied (to be treated as const)

This function will return the key ID, of the specified bag element. The key ID is usually used to distinguish the local private key and the certificate pair.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value. or a negative error code on error.

**gnutls\_pkcs12\_bag\_get\_type**

**gnutls\_pkcs12\_bag\_type\_t gnutls\_pkcs12\_bag\_get\_type** (*gnutls\_pkcs12\_bag\_t bag*, *int indx*) [Function]

*bag*: The bag

*indx*: The element of the bag to get the type

This function will return the bag's type.

**Returns:** One of the *gnutls\_pkcs12\_bag\_type\_t* enumerations.

**gnutls\_pkcs12\_bag\_init**

**int gnutls\_pkcs12\_bag\_init** (*gnutls\_pkcs12\_bag\_t \* bag*) [Function]

*bag*: The structure to be initialized

This function will initialize a PKCS12 bag structure. PKCS12 Bags usually contain private keys, lists of X.509 Certificates and X.509 Certificate revocation lists.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**gnutls\_pkcs12\_bag\_set\_crl**

**int gnutls\_pkcs12\_bag\_set\_crl** (*gnutls\_pkcs12\_bag\_t bag*, *gnutls\_x509\_crl\_t crl*) [Function]

*bag*: The bag

*crl*: the CRL to be copied.

This function will insert the given CRL into the bag. This is just a wrapper over *gnutls\_pkcs12\_bag\_set\_data()* .

**Returns:** the index of the added bag on success, or a negative error code on failure.

**gnutls\_pkcs12\_bag\_set\_cert**

**int gnutls\_pkcs12\_bag\_set\_cert** (*gnutls\_pkcs12\_bag\_t bag*, *gnutls\_x509\_cert\_t crt*) [Function]

*bag*: The bag

*crt*: the certificate to be copied.

This function will insert the given certificate into the bag. This is just a wrapper over `gnutls_pkcs12_bag_set_data()` .

**Returns:** the index of the added bag on success, or a negative value on failure.

### **gnutls\_pkcs12\_bag\_set\_data**

**int gnutls\_pkcs12\_bag\_set\_data** (*gnutls\_pkcs12\_bag\_t bag*, [Function]  
*gnutls\_pkcs12\_bag\_type\_t type, const gnutls\_datum\_t \* data*)

*bag*: The bag

*type*: The data's type

*data*: the data to be copied.

This function will insert the given data of the given type into the bag.

**Returns:** the index of the added bag on success, or a negative value on error.

### **gnutls\_pkcs12\_bag\_set\_friendly\_name**

**int gnutls\_pkcs12\_bag\_set\_friendly\_name** (*gnutls\_pkcs12\_bag\_t bag*, [Function]  
*int indx, const char \* name*)

*bag*: The bag

*indx*: The bag's element to add the id

*name*: the name

This function will add the given key friendly name, to the specified, by the index, bag element. The name will be encoded as a 'Friendly name' bag attribute, which is usually used to set a user name to the local private key and the certificate pair.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value. or a negative error code on error.

### **gnutls\_pkcs12\_bag\_set\_key\_id**

**int gnutls\_pkcs12\_bag\_set\_key\_id** (*gnutls\_pkcs12\_bag\_t bag*, [Function]  
*int indx, const gnutls\_datum\_t \* id*)

*bag*: The bag

*indx*: The bag's element to add the id

*id*: the ID

This function will add the given key ID, to the specified, by the index, bag element. The key ID will be encoded as a 'Local key identifier' bag attribute, which is usually used to distinguish the local private key and the certificate pair.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value. or a negative error code on error.

### **gnutls\_pkcs12\_deinit**

**void gnutls\_pkcs12\_deinit** (*gnutls\_pkcs12\_t pkcs12*) [Function]  
*pkcs12*: The structure to be initialized

This function will deinitialize a PKCS12 structure.

## gnutls\_pkcs12\_export

**int gnutls\_pkcs12\_export** (*gnutls\_pkcs12\_t pkcs12*, [Function]  
*gnutls\_x509\_crt\_fmt\_t format*, *void \* output\_data*, *size\_t \* output\_data\_size*)

*pkcs12*: Holds the pkcs12 structure

*format*: the format of output params. One of PEM or DER.

*output\_data*: will contain a structure PEM or DER encoded

*output\_data\_size*: holds the size of output\_data (and will be replaced by the actual size of parameters)

This function will export the pkcs12 structure to DER or PEM format.

If the buffer provided is not long enough to hold the output, then \*output\_data\_size will be updated and GNUTLS\_E\_SHORT\_MEMORY\_BUFFER will be returned.

If the structure is PEM encoded, it will have a header of "BEGIN PKCS12".

**Returns:** In case of failure a negative error code will be returned, and 0 on success.

## gnutls\_pkcs12\_export2

**int gnutls\_pkcs12\_export2** (*gnutls\_pkcs12\_t pkcs12*, [Function]  
*gnutls\_x509\_crt\_fmt\_t format*, *gnutls\_datum\_t \* out*)

*pkcs12*: Holds the pkcs12 structure

*format*: the format of output params. One of PEM or DER.

*out*: will contain a structure PEM or DER encoded

This function will export the pkcs12 structure to DER or PEM format.

The output buffer is allocated using `gnutls_malloc()` .

If the structure is PEM encoded, it will have a header of "BEGIN PKCS12".

**Returns:** In case of failure a negative error code will be returned, and 0 on success.

**Since:** 3.1.3

## gnutls\_pkcs12\_generate\_mac

**int gnutls\_pkcs12\_generate\_mac** (*gnutls\_pkcs12\_t pkcs12*, *const* [Function]  
*char \* pass*)

*pkcs12*: should contain a gnutls\_pkcs12\_t structure

*pass*: The password for the MAC

This function will generate a MAC for the PKCS12 structure.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

## gnutls\_pkcs12\_get\_bag

**int gnutls\_pkcs12\_get\_bag** (*gnutls\_pkcs12\_t pkcs12*, *int indx*, [Function]  
*gnutls\_pkcs12\_bag\_t bag*)

*pkcs12*: should contain a gnutls\_pkcs12\_t structure

*indx*: contains the index of the bag to extract

*bag*: An initialized bag, where the contents of the bag will be copied

This function will return a Bag from the PKCS12 structure.

After the last Bag has been read GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE will be returned.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

## gnutls\_pkcs12\_import

```
int gnutls_pkcs12_import (gnutls_pkcs12_t pkcs12, const [Function]
                        gnutls_datum_t * data, gnutls_x509_crt_fmt_t format, unsigned int flags)
pkcs12: The structure to store the parsed PKCS12.
```

*data*: The DER or PEM encoded PKCS12.

*format*: One of DER or PEM

*flags*: an ORed sequence of gnutls\_privkey\_pkcs8\_flags

This function will convert the given DER or PEM encoded PKCS12 to the native gnutls\_pkcs12\_t format. The output will be stored in 'pkcs12'.

If the PKCS12 is PEM encoded it should have a header of "PKCS12".

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

## gnutls\_pkcs12\_init

```
int gnutls_pkcs12_init (gnutls_pkcs12_t * pkcs12) [Function]
pkcs12: The structure to be initialized
```

This function will initialize a PKCS12 structure. PKCS12 structures usually contain lists of X.509 Certificates and X.509 Certificate revocation lists.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

## gnutls\_pkcs12\_set\_bag

```
int gnutls_pkcs12_set_bag (gnutls_pkcs12_t pkcs12, [Function]
                        gnutls_pkcs12_bag_t bag)
pkcs12: should contain a gnutls_pkcs12_t structure
```

*bag*: An initialized bag

This function will insert a Bag into the PKCS12 structure.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

## gnutls\_pkcs12\_simple\_parse

```
int gnutls_pkcs12_simple_parse (gnutls_pkcs12_t p12, const char * [Function]
    password, gnutls_x509_privkey_t * key, gnutls_x509_crt_t ** chain, unsigned
    int * chain_len, gnutls_x509_crt_t ** extra_certs, unsigned int *
    extra_certs_len, gnutls_x509_crl_t * crl, unsigned int flags)
```

*p12*: should contain a gnutls\_pkcs12\_t structure

*password*: optional password used to decrypt the structure, bags and keys.

*key*: a structure to store the parsed private key.

*chain*: the corresponding to key certificate chain (may be NULL )

*chain\_len*: will be updated with the number of additional (may be NULL )

*extra\_certs*: optional pointer to receive an array of additional certificates found in the PKCS12 structure (may be NULL ).

*extra\_certs\_len*: will be updated with the number of additional certs (may be NULL ).

*crl*: an optional structure to store the parsed CRL (may be NULL ).

*flags*: should be zero or one of GNUTLS\_PKCS12\_SP\_\*

This function parses a PKCS12 structure in *pkcs12* and extracts the private key, the corresponding certificate chain, any additional certificates and a CRL.

The *extra\_certs* and *extra\_certs\_len* parameters are optional and both may be set to NULL . If either is non-NULL , then both must be set. The value for *extra\_certs* is allocated using *gnutls\_malloc()* .

Encrypted PKCS12 bags and PKCS8 private keys are supported, but only with password based security and the same password for all operations.

Note that a PKCS12 structure may contain many keys and/or certificates, and there is no way to identify which key/certificate pair you want. For this reason this function is useful for PKCS12 files that contain only one key/certificate pair and/or one CRL.

If the provided structure has encrypted fields but no password is provided then this function returns *GNUTLS\_E\_DECRYPTION\_FAILED* .

Note that normally the chain constructed does not include self signed certificates, to comply with TLS' requirements. If, however, the flag *GNUTLS\_PKCS12\_SP\_INCLUDE\_SELF\_SIGNED* is specified then self signed certificates will be included in the chain.

Prior to using this function the PKCS 12 structure integrity must be verified using *gnutls\_pkcs12\_verify\_mac()* .

**Returns:** On success, *GNUTLS\_E\_SUCCESS* (0) is returned, otherwise a negative error value.

**Since:** 3.1.0

## gnutls\_pkcs12\_verify\_mac

```
int gnutls_pkcs12_verify_mac (gnutls_pkcs12_t pkcs12, const char * [Function]
    pass)
```

*pkcs12*: should contain a gnutls\_pkcs12\_t structure

*pass*: The password for the MAC

This function will verify the MAC for the PKCS12 structure.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

## E.7 Hardware token via PKCS 11 API

The following functions are to be used for PKCS 11 handling. Their prototypes lie in `gnutls/pkcs11.h`.

### `gnutls_pkcs11_add_provider`

`int gnutls_pkcs11_add_provider (const char * name, const char * params)` [Function]

*name*: The filename of the module

*params*: should be NULL

This function will load and add a PKCS 11 module to the module list used in gnutls. After this function is called the module will be used for PKCS 11 operations.

When loading a module to be used for certificate verification, use the string 'trusted' as *params*.

Note that this function is not thread safe.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

### `gnutls_pkcs11_copy_secret_key`

`int gnutls_pkcs11_copy_secret_key (const char * token_url, gnutls_datum_t * key, const char * label, unsigned int key_usage, unsigned int flags)` [Function]

*token\_url*: A PKCS 11 URL specifying a token

*key*: The raw key

*label*: A name to be used for the stored data

*key\_usage*: One of GNUTLS\_KEY\_\*

*flags*: One of GNUTLS\_PKCS11\_OBJ\_FLAG\_\*

This function will copy a raw secret (symmetric) key into a PKCS 11 token specified by a URL. The key can be marked as sensitive or not.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

### `gnutls_pkcs11_copy_x509_cert`

`int gnutls_pkcs11_copy_x509_cert (const char * token_url, gnutls_x509_cert_t crt, const char * label, unsigned int flags)` [Function]

*token\_url*: A PKCS 11 URL specifying a token



*crt*: A certificate

*label*: A name to be used for the stored data

*flags*: One of GNUTLS\_PKCS11\_OBJ\_FLAG\_\*

This function will copy a certificate into a PKCS 11 token specified by a URL. The certificate can be marked as trusted or not.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

### gnutls\_pkcs11\_copy\_x509\_privkey

```
int gnutls_pkcs11_copy_x509_privkey (const char * token_url,      [Function]
                                     gnutls_x509_privkey_t key, const char * label, unsigned int key_usage,
                                     unsigned int flags)
```

*token\_url*: A PKCS 11 URL specifying a token

*key*: A private key

*label*: A name to be used for the stored data

*key\_usage*: One of GNUTLS\_KEY\_\*

*flags*: One of GNUTLS\_PKCS11\_OBJ\_\* flags

This function will copy a private key into a PKCS 11 token specified by a URL. It is highly recommended flags to contain GNUTLS\_PKCS11\_OBJ\_FLAG\_MARK\_SENSITIVE unless there is a strong reason not to.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

### gnutls\_pkcs11\_cert\_is\_known

```
int gnutls_pkcs11_cert_is_known (const char * url,              [Function]
                                  gnutls_x509_cert_t cert, unsigned int flags)
```

*url*: A PKCS 11 url identifying a token

*cert*: is the certificate to find issuer for

*flags*: Use zero or flags from GNUTLS\_PKCS11\_OBJ\_FLAG .

This function will check whether the provided certificate is stored in the specified token. This is useful in combination with GNUTLS\_PKCS11\_OBJ\_FLAG\_RETRIEVE\_TRUSTED or GNUTLS\_PKCS11\_OBJ\_FLAG\_RETRIEVE\_DISTRUSTED , to check whether a CA is present or a certificate is blacklisted in a trust PKCS 11 module.

This function can be used with a url of "pkcs11:", and in that case all modules will be searched. To restrict the modules to the marked as trusted in p11-kit use the GNUTLS\_PKCS11\_OBJ\_FLAG\_PRESENT\_IN\_TRUSTED\_MODULE flag.

Note that the flag GNUTLS\_PKCS11\_OBJ\_FLAG\_RETRIEVE\_DISTRUSTED is specific to p11-kit trust modules.

**Returns:** If the certificate exists non-zero is returned, otherwise zero.

**Since:** 3.3.0

## gnutls\_pkcs11\_deinit

`void gnutls_pkcs11_deinit ( void)` [Function]

This function will deinitialize the PKCS 11 subsystem in gnutls. This function is only needed if you need to deinitialize the subsystem without calling `gnutls_global_deinit()` .

**Since:** 2.12.0

## gnutls\_pkcs11\_delete\_url

`int gnutls_pkcs11_delete_url (const char * object_url, unsigned int flags)` [Function]

*object\_url*: The URL of the object to delete.

*flags*: One of GNUTLS\_PKCS11\_OBJ\_\* flags

This function will delete objects matching the given URL. Note that not all tokens support the delete operation.

**Returns:** On success, the number of objects deleted is returned, otherwise a negative error value.

**Since:** 2.12.0

## gnutls\_pkcs11\_get\_pin\_function

`gnutls_pin_callback_t gnutls_pkcs11_get_pin_function (void ** userdata)` [Function]

*userdata*: data to be supplied to callback

This function will return the callback function set using `gnutls_pkcs11_set_pin_function()` .

**Returns:** The function set or NULL otherwise.

**Since:** 3.1.0

## gnutls\_pkcs11\_get\_raw\_issuer

`int gnutls_pkcs11_get_raw_issuer (const char * url, gnutls_x509_cert_t cert, gnutls_datum_t * issuer, gnutls_x509_cert_fmt_t fmt, unsigned int flags)` [Function]

*url*: A PKCS 11 url identifying a token

*cert*: is the certificate to find issuer for

*issuer*: Will hold the issuer if any in an allocated buffer.

*fmt*: The format of the exported issuer.

*flags*: Use zero or flags from GNUTLS\_PKCS11\_OBJ\_FLAG .

This function will return the issuer of a given certificate, if it is stored in the token. By default only marked as trusted issuers are returned. If any issuer should be returned specify GNUTLS\_PKCS11\_OBJ\_FLAG\_RETRIEVE\_ANY in *flags* .

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.2.7

## gnutls\_pkcs11\_init

**int gnutls\_pkcs11\_init** (*unsigned int flags, const char \* deprecated\_config\_file*) [Function]

*flags*: An ORed sequence of GNUTLS\_PKCS11\_FLAG\_\*

*deprecated\_config\_file*: either NULL or the location of a deprecated configuration file

This function will initialize the PKCS 11 subsystem in gnutls. It will read configuration files if GNUTLS\_PKCS11\_FLAG\_AUTO is used or allow you to independently load PKCS 11 modules using gnutls\_pkcs11\_add\_provider() if GNUTLS\_PKCS11\_FLAG\_MANUAL is specified.

Normally you don't need to call this function since it is being called when the first PKCS 11 operation is requested using the GNUTLS\_PKCS11\_FLAG\_AUTO flag. If another flags are required then it must be called independently prior to any PKCS 11 operation.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

## gnutls\_pkcs11\_obj\_deinit

**void gnutls\_pkcs11\_obj\_deinit** (*gnutls\_pkcs11\_obj\_t obj*) [Function]

*obj*: The structure to be initialized

This function will deinitialize a certificate structure.

**Since:** 2.12.0

## gnutls\_pkcs11\_obj\_export

**int gnutls\_pkcs11\_obj\_export** (*gnutls\_pkcs11\_obj\_t obj, void \* output\_data, size\_t \* output\_data\_size*) [Function]

*obj*: Holds the object

*output\_data*: will contain the object data

*output\_data\_size*: holds the size of output\_data (and will be replaced by the actual size of parameters)

This function will export the PKCS11 object data. It is normal for data to be inaccessible and in that case GNUTLS\_E\_INVALID\_REQUEST will be returned.

If the buffer provided is not long enough to hold the output, then \*output\_data\_size is updated and GNUTLS\_E\_SHORT\_MEMORY\_BUFFER will be returned.

**Returns:** In case of failure a negative error code will be returned, and GNUTLS\_E\_SUCCESS (0) on success.

**Since:** 2.12.0

## gnutls\_pkcs11\_obj\_export2

**int gnutls\_pkcs11\_obj\_export2** (*gnutls\_pkcs11\_obj\_t obj, gnutls\_datum\_t \* out*) [Function]

*obj*: Holds the object

*out*: will contain the object data

This function will export the PKCS11 object data. It is normal for data to be inaccessible and in that case `GNUTLS_E_INVALID_REQUEST` will be returned.

The output buffer is allocated using `gnutls_malloc()` .

**Returns:** In case of failure a negative error code will be returned, and `GNUTLS_E_SUCCESS` (0) on success.

**Since:** 3.1.3

### **gnutls\_pkcs11\_obj\_export3**

`int gnutls_pkcs11_obj_export3 (gnutls_pkcs11_obj_t obj, [Function]  
gnutls_x509_cert_fmt_t fmt, gnutls_datum_t * out)`

*obj*: Holds the object

*fmt*: The format of the exported data

*out*: will contain the object data

This function will export the PKCS11 object data. It is normal for data to be inaccessible and in that case `GNUTLS_E_INVALID_REQUEST` will be returned.

The output buffer is allocated using `gnutls_malloc()` .

**Returns:** In case of failure a negative error code will be returned, and `GNUTLS_E_SUCCESS` (0) on success.

**Since:** 3.2.7

### **gnutls\_pkcs11\_obj\_export\_url**

`int gnutls_pkcs11_obj_export_url (gnutls_pkcs11_obj_t obj, [Function]  
gnutls_pkcs11_url_type_t detailed, char ** url)`

*obj*: Holds the PKCS 11 certificate

*detailed*: non zero if a detailed URL is required

*url*: will contain an allocated url

This function will export a URL identifying the given certificate.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

### **gnutls\_pkcs11\_obj\_flags\_get\_str**

`char * gnutls_pkcs11_obj_flags_get_str (unsigned int flags) [Function]  
flags: holds the flags`

This function given an or-sequence of `GNUTLS_PKCS11_OBJ_FLAG_MARK` , will return an allocated string with its description. The string needs to be deallocated using `gnutls_free()` .

**Returns:** If flags is zero NULL is returned, otherwise an allocated string.

**Since:** 3.3.7

## gnutls\_pkcs11\_obj\_get\_exts

**int gnutls\_pkcs11\_obj\_get\_exts** (*gnutls\_pkcs11\_obj\_t obj*, [Function]  
*gnutls\_x509\_ext\_st \*\* exts*, *unsigned int \* exts\_size*, *unsigned int flags*)

*obj*: should contain a *gnutls\_pkcs11\_obj\_t* structure

*exts*: an allocated list of pointers to *gnutls\_x509\_ext\_st*

*exts\_size*: the number of *exts*

*flags*: Or sequence of *GNUTLS\_PKCS11\_OBJ\_ \** flags

This function will return information about attached extensions that associate to the provided object (which should be a certificate). The extensions are the attached p11-kit trust module extensions.

**Returns:** *GNUTLS\_E\_SUCCESS* (0) on success or a negative error code on error.

**Since:** 3.3.8

## gnutls\_pkcs11\_obj\_get\_flags

**int gnutls\_pkcs11\_obj\_get\_flags** (*gnutls\_pkcs11\_obj\_t obj*, [Function]  
*unsigned int \* oflags*)

*obj*: The structure that holds the object

*oflags*: Will hold the output flags

This function will return the flags of the object being stored in the structure. The *oflags* are the *GNUTLS\_PKCS11\_OBJ\_FLAG\_MARK* flags.

**Returns:** On success, *GNUTLS\_E\_SUCCESS* (0) is returned, otherwise a negative error value.

**Since:** 3.3.7

## gnutls\_pkcs11\_obj\_get\_info

**int gnutls\_pkcs11\_obj\_get\_info** (*gnutls\_pkcs11\_obj\_t obj*, [Function]  
*gnutls\_pkcs11\_obj\_info\_t itype*, *void \* output*, *size\_t \* output\_size*)

*obj*: should contain a *gnutls\_pkcs11\_obj\_t* structure

*itype*: Denotes the type of information requested

*output*: where output will be stored

*output\_size*: contains the maximum size of the output and will be overwritten with actual

This function will return information about the PKCS11 certificate such as the label, id as well as token information where the key is stored. When output is text it returns null terminated string although *output\_size* contains the size of the actual data only.

**Returns:** *GNUTLS\_E\_SUCCESS* (0) on success or a negative error code on error.

**Since:** 2.12.0

## gnutls\_pkcs11\_obj\_get\_type

`gnutls_pkcs11_obj_type_t gnutls_pkcs11_obj_get_type` [Function]  
 (*gnutls\_pkcs11\_obj\_t obj*)

*obj*: Holds the PKCS 11 object

This function will return the type of the object being stored in the structure.

**Returns:** The type of the object

**Since:** 2.12.0

## gnutls\_pkcs11\_obj\_import\_url

`int gnutls_pkcs11_obj_import_url` (*gnutls\_pkcs11\_obj\_t obj, const* [Function]  
*char \* url, unsigned int flags*)

*obj*: The structure to store the object

*url*: a PKCS 11 url identifying the key

*flags*: Or sequence of GNUTLS\_PKCS11\_OBJ\_\* flags

This function will "import" a PKCS 11 URL identifying an object (e.g. certificate) to the `gnutls_pkcs11_obj_t` structure. This does not involve any parsing (such as X.509 or OpenPGP) since the `gnutls_pkcs11_obj_t` is format agnostic. Only data are transferred.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

## gnutls\_pkcs11\_obj\_init

`int gnutls_pkcs11_obj_init` (*gnutls\_pkcs11\_obj\_t \* obj*) [Function]

*obj*: The structure to be initialized

This function will initialize a pkcs11 certificate structure.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

## gnutls\_pkcs11\_obj\_list\_import\_url

`int gnutls_pkcs11_obj_list_import_url` (*gnutls\_pkcs11\_obj\_t \** [Function]  
*p\_list, unsigned int \* n\_list, const char \* url, gnutls\_pkcs11\_obj\_attr\_t*  
*attrs, unsigned int flags*)

*p\_list*: An uninitialized object list (may be NULL)

*n\_list*: initially should hold the maximum size of the list. Will contain the actual size.

*url*: A PKCS 11 url identifying a set of objects

*attrs*: Attributes of type `gnutls_pkcs11_obj_attr_t` that can be used to limit output

*flags*: Or sequence of GNUTLS\_PKCS11\_OBJ\_\* flags

This function will initialize and set values to an object list by using all objects identified by a PKCS 11 URL.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

## gnutls\_pkcs11\_obj\_list\_import\_url2

```
int gnutls_pkcs11_obj_list_import_url2 (gnutls_pkcs11_obj_t **      [Function]
    p_list, unsigned int * n_list, const char * url, gnutls_pkcs11_obj_attr_t
    attrs, unsigned int flags)
```

*p\_list*: An uninitialized object list (may be NULL)

*n\_list*: It will contain the size of the list.

*url*: A PKCS 11 url identifying a set of objects

*attrs*: Attributes of type `gnutls_pkcs11_obj_attr_t` that can be used to limit output

*flags*: Or sequence of GNUTLS\_PKCS11\_OBJ\_\* flags

This function will initialize and set values to an object list by using all objects identified by the PKCS 11 URL. The output is stored in `p_list`, which will be initialized.

All returned objects must be deinitialized using `gnutls_pkcs11_obj_deinit()`, and `p_list` must be free'd using `gnutls_free()`.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.1.0

## gnutls\_pkcs11\_obj\_set\_pin\_function

```
void gnutls_pkcs11_obj_set_pin_function (gnutls_pkcs11_obj_t      [Function]
    obj, gnutls_pin_callback_t fn, void * userdata)
```

*obj*: The object structure

*fn*: the callback

*userdata*: data associated with the callback

This function will set a callback function to be used when required to access the object. This function overrides the global set using `gnutls_pkcs11_set_pin_function()`.

**Since:** 3.1.0

## gnutls\_pkcs11\_privkey\_deinit

```
void gnutls_pkcs11_privkey_deinit (gnutls_pkcs11_privkey_t key)  [Function]
    key: The structure to be initialized
```

This function will deinitialize a private key structure.

**gnutls\_pkcs11\_privkey\_export\_pubkey**

**int gnutls\_pkcs11\_privkey\_export\_pubkey** [Function]  
 (*gnutls\_pkcs11\_privkey\_t* **pkey**, *gnutls\_x509\_cert\_fmt\_t* **fmt**, *gnutls\_datum\_t* \*  
**data**, unsigned int **flags**)

*pkey*: The private key

*fmt*: the format of output params. PEM or DER.

*data*: will hold the public key

*flags*: should be zero

This function will extract the public key (modulus and public exponent) from the private key specified by the *url* private key. This public key will be stored in **pubkey** in the format specified by **fmt**. **pubkey** should be deinitialized using **gnutls\_free()**.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.3.7

**gnutls\_pkcs11\_privkey\_export\_url**

**int gnutls\_pkcs11\_privkey\_export\_url** (*gnutls\_pkcs11\_privkey\_t* [Function]  
**key**, *gnutls\_pkcs11\_url\_type\_t* **detailed**, char \*\* **url**)

*key*: Holds the PKCS 11 key

*detailed*: non zero if a detailed URL is required

*url*: will contain an allocated url

This function will export a URL identifying the given key.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**gnutls\_pkcs11\_privkey\_generate**

**int gnutls\_pkcs11\_privkey\_generate** (*const char \****url**, [Function]  
*gnutls\_pk\_algorithm\_t* **pk**, unsigned int **bits**, *const char \****label**, unsigned int  
**flags**)

*url*: a token URL

*pk*: the public key algorithm

*bits*: the security bits

*label*: a label

*flags*: should be zero

This function will generate a private key in the specified by the *url* token. The private key will be generate within the token and will not be exportable.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.0



## gnutls\_pkcs11\_privkey\_generate2

```
int gnutls_pkcs11_privkey_generate2 (const char * url, [Function]
                                     gnutls_pk_algorithm_t pk, unsigned int bits, const char * label,
                                     gnutls_x509_crt_fmt_t fmt, gnutls_datum_t * pubkey, unsigned int flags)
```

*url*: a token URL

*pk*: the public key algorithm

*bits*: the security bits

*label*: a label

*fmt*: the format of output params. PEM or DER.

*pubkey*: will hold the public key (may be NULL )

*flags*: zero or an OR'ed sequence of GNUTLS\_PKCS11\_OBJ\_FLAGS

This function will generate a private key in the specified by the *url* token. The private key will be generate within the token and will not be exportable. This function will store the DER-encoded public key in the SubjectPublicKeyInfo format in *pubkey* . The *pubkey* should be deinitialized using `gnutls_free()` .

Note that when generating an elliptic curve key, the curve can be substituted in the place of the bits parameter using the `GNUTLS_CURVE_TO_BITS()` macro.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.1.5

## gnutls\_pkcs11\_privkey\_get\_info

```
int gnutls_pkcs11_privkey_get_info (gnutls_pkcs11_privkey_t [Function]
                                     pkey, gnutls_pkcs11_obj_info_t itype, void * output, size_t * output_size)
```

*pkey*: should contain a `gnutls_pkcs11_privkey_t` structure

*itype*: Denotes the type of information requested

*output*: where output will be stored

*output\_size*: contains the maximum size of the output and will be overwritten with actual

This function will return information about the PKCS 11 private key such as the label, id as well as token information where the key is stored. When output is text it returns null terminated string although *output\_size* contains the size of the actual data only.

**Returns:** `GNUTLS_E_SUCCESS` (0) on success or a negative error code on error.

## gnutls\_pkcs11\_privkey\_get\_pk\_algorithm

```
int gnutls_pkcs11_privkey_get_pk_algorithm [Function]
      (gnutls_pkcs11_privkey_t key, unsigned int * bits)
```

*key*: should contain a `gnutls_pkcs11_privkey_t` structure

*bits*: if bits is non null it will hold the size of the parameters' in bits

This function will return the public key algorithm of a private key.

**Returns:** a member of the `gnutls_pk_algorithm_t` enumeration on success, or a negative error code on error.

## gnutls\_pkcs11\_privkey\_import\_url

**int gnutls\_pkcs11\_privkey\_import\_url** (*gnutls\_pkcs11\_privkey\_t* *pkey*, *const char \* url*, *unsigned int flags*) [Function]

*pkey*: The structure to store the parsed key

*url*: a PKCS 11 url identifying the key

*flags*: Or sequence of GNUTLS\_PKCS11\_OBJ\_\* flags

This function will "import" a PKCS 11 URL identifying a private key to the **gnutls\_pkcs11\_privkey\_t** structure. In reality since in most cases keys cannot be exported, the private key structure is being associated with the available operations on the token.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

## gnutls\_pkcs11\_privkey\_init

**int gnutls\_pkcs11\_privkey\_init** (*gnutls\_pkcs11\_privkey\_t \* key*) [Function]

*key*: The structure to be initialized

This function will initialize an private key structure.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

## gnutls\_pkcs11\_privkey\_set\_pin\_function

**void gnutls\_pkcs11\_privkey\_set\_pin\_function** (*gnutls\_pkcs11\_privkey\_t key*, *gnutls\_pin\_callback\_t fn*, *void \* userdata*) [Function]

*key*: The private key

*fn*: the callback

*userdata*: data associated with the callback

This function will set a callback function to be used when required to access the object. This function overrides the global set using **gnutls\_pkcs11\_set\_pin\_function()** .

**Since:** 3.1.0

## gnutls\_pkcs11\_privkey\_status

**int gnutls\_pkcs11\_privkey\_status** (*gnutls\_pkcs11\_privkey\_t key*) [Function]

*key*: Holds the key

Checks the status of the private key token.

**Returns:** this function will return non-zero if the token holding the private key is still available (inserted), and zero otherwise.

**Since:** 3.1.9

## gnutls\_pkcs11\_reinit

**int gnutls\_pkcs11\_reinit ( void) [Function]**

This function will reinitialize the PKCS 11 subsystem in gnutls. This is required by PKCS 11 when an application uses `fork()` . The reinitialization function must be called on the child.

Note that since GnuTLS 3.3.0, the reinitialization of the PKCS 11 subsystem occurs automatically after `fork`.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.0

## gnutls\_pkcs11\_set\_pin\_function

**void gnutls\_pkcs11\_set\_pin\_function (gnutls\_pin\_callback\_t fn, void \* userdata) [Function]**

*fn*: The PIN callback, a `gnutls_pin_callback_t()` function.

*userdata*: data to be supplied to callback

This function will set a callback function to be used when a PIN is required for PKCS 11 operations. See `gnutls_pin_callback_t()` on how the callback should behave.

**Since:** 2.12.0

## gnutls\_pkcs11\_set\_token\_function

**void gnutls\_pkcs11\_set\_token\_function (gnutls\_pkcs11\_token\_callback\_t fn, void \* userdata) [Function]**

*fn*: The token callback

*userdata*: data to be supplied to callback

This function will set a callback function to be used when a token needs to be inserted to continue PKCS 11 operations.

**Since:** 2.12.0

## gnutls\_pkcs11\_token\_get\_flags

**int gnutls\_pkcs11\_token\_get\_flags (const char \* url, unsigned int \* flags) [Function]**

*url*: should contain a PKCS 11 URL

*flags*: The output flags (`GNUTLS_PKCS11_TOKEN_*`)

This function will return information about the PKCS 11 token flags.

The supported flags are: `GNUTLS_PKCS11_TOKEN_HW` and `GNUTLS_PKCS11_TOKEN_TRUSTED` .

**Returns:** `GNUTLS_E_SUCCESS` (0) on success or a negative error code on error.

**Since:** 2.12.0

**gnutls\_pkcs11\_token\_get\_info**

**int gnutls\_pkcs11\_token\_get\_info** (*const char \* url*, [Function]  
*gnutls\_pkcs11\_token\_info\_t ttype*, *void \* output*, *size\_t \* output\_size*)

*url*: should contain a PKCS 11 URL

*ttype*: Denotes the type of information requested

*output*: where output will be stored

*output\_size*: contains the maximum size of the output and will be overwritten with actual

This function will return information about the PKCS 11 token such as the label, id, etc.

**Returns:** GNUTLS\_E\_SUCCESS (0) on success or a negative error code on error.

**Since:** 2.12.0

**gnutls\_pkcs11\_token\_get\_mechanism**

**int gnutls\_pkcs11\_token\_get\_mechanism** (*const char \* url*, [Function]  
*unsigned int idx*, *unsigned long \* mechanism*)

*url*: should contain a PKCS 11 URL

*idx*: The index of the mechanism

*mechanism*: The PKCS 11 mechanism ID

This function will return the names of the supported mechanisms by the token. It should be called with an increasing index until it return GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE.

**Returns:** GNUTLS\_E\_SUCCESS (0) on success or a negative error code on error.

**Since:** 2.12.0

**gnutls\_pkcs11\_token\_get\_random**

**int gnutls\_pkcs11\_token\_get\_random** (*const char \* token\_url*, [Function]  
*void \* rnddata*, *size\_t len*)

*token\_url*: A PKCS 11 URL specifying a token

*rnddata*: A pointer to the memory area to be filled with random data

*len*: The number of bytes of randomness to request

This function will get random data from the given token. It will store *rnddata* and fill the memory pointed to by *rnddata* with *len* random bytes from the token.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**gnutls\_pkcs11\_token\_get\_url**

**int gnutls\_pkcs11\_token\_get\_url** (*unsigned int seq*, [Function]  
*gnutls\_pkcs11\_url\_type\_t detailed*, *char \*\* url*)

*seq*: sequence number starting from 0

*detailed*: non zero if a detailed URL is required

*url*: will contain an allocated url

This function will return the URL for each token available in system. The url has to be released using `gnutls_free()`

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` if the sequence number exceeds the available tokens, otherwise a negative error value.

**Since:** 2.12.0

## gnutls\_pkcs11\_token\_init

`int gnutls_pkcs11_token_init (const char * token_url, const char * so_pin, const char * label)` [Function]

*token\_url*: A PKCS 11 URL specifying a token

*so\_pin*: Security Officer's PIN

*label*: A name to be used for the token

This function will initialize (format) a token. If the token is at a factory defaults state the security officer's PIN given will be set to be the default. Otherwise it should match the officer's PIN.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

## gnutls\_pkcs11\_token\_set\_pin

`int gnutls_pkcs11_token_set_pin (const char * token_url, const char * oldpin, const char * newpin, unsigned int flags)` [Function]

*token\_url*: A PKCS 11 URL specifying a token

*oldpin*: old user's PIN

*newpin*: new user's PIN

*flags*: one of `gnutls_pin_flag_t` .

This function will modify or set a user's PIN for the given token. If it is called to set a user pin for first time the oldpin must be NULL.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

## gnutls\_pkcs11\_type\_get\_name

`const char * gnutls_pkcs11_type_get_name (gnutls_pkcs11_obj_type_t type)` [Function]

*type*: Holds the PKCS 11 object type, a `gnutls_pkcs11_obj_type_t` .

This function will return a human readable description of the PKCS11 object type `obj` . It will return "Unknown" for unknown types.

**Returns:** human readable string labeling the PKCS11 object type `type` .

**Since:** 2.12.0

**gnutls\_x509\_cert\_import\_pkcs11**

**int gnutls\_x509\_cert\_import\_pkcs11** (*gnutls\_x509\_cert\_t crt*, [Function]  
*gnutls\_pkcs11\_obj\_t pkcs11\_cert*)

*crt*: A certificate of type *gnutls\_x509\_cert\_t*

*pkcs11\_cert*: A PKCS 11 object that contains a certificate

This function will import a PKCS 11 certificate to a *gnutls\_x509\_cert\_t* structure.

**Returns:** On success, *GNUTLS\_E\_SUCCESS* (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

**gnutls\_x509\_cert\_import\_pkcs11\_url**

**int gnutls\_x509\_cert\_import\_pkcs11\_url** (*gnutls\_x509\_cert\_t crt*, [Function]  
*const char \* url*, *unsigned int flags*)

*crt*: A certificate of type *gnutls\_x509\_cert\_t*

*url*: A PKCS 11 url

*flags*: One of *GNUTLS\_PKCS11\_OBJ\_\** flags

This function will import a PKCS 11 certificate directly from a token without involving the *gnutls\_pkcs11\_obj\_t* structure. This function will fail if the certificate stored is not of X.509 type.

**Returns:** On success, *GNUTLS\_E\_SUCCESS* (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

**gnutls\_x509\_cert\_list\_import\_pkcs11**

**int gnutls\_x509\_cert\_list\_import\_pkcs11** (*gnutls\_x509\_cert\_t \** [Function]  
*certs*, *unsigned int cert\_max*, *gnutls\_pkcs11\_obj\_t \* const objs*, *unsigned int flags*)

*certs*: A list of certificates of type *gnutls\_x509\_cert\_t*

*cert\_max*: The maximum size of the list

*objs*: A list of PKCS 11 objects

*flags*: 0 for now

This function will import a PKCS 11 certificate list to a list of *gnutls\_x509\_cert\_t* structure. These must not be initialized.

**Returns:** On success, *GNUTLS\_E\_SUCCESS* (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

**E.8 TPM API**

The following functions are to be used for TPM handling. Their prototypes lie in *gnutls/tpm.h*.

## gnutls\_tpm\_get\_registered

**int gnutls\_tpm\_get\_registered** (*gnutls\_tpm\_key\_list\_t \* list*) [Function]

*list*: a list to store the keys

This function will get a list of stored keys in the TPM. The uuid of those keys

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.1.0

## gnutls\_tpm\_key\_list\_deinit

**void gnutls\_tpm\_key\_list\_deinit** (*gnutls\_tpm\_key\_list\_t list*) [Function]

*list*: a list of the keys

This function will deinitialize the list of stored keys in the TPM.

**Since:** 3.1.0

## gnutls\_tpm\_key\_list\_get\_url

**int gnutls\_tpm\_key\_list\_get\_url** (*gnutls\_tpm\_key\_list\_t list*, [Function]  
*unsigned int idx, char \*\* url, unsigned int flags*)

*list*: a list of the keys

*idx*: The index of the key (starting from zero)

*url*: The URL to be returned

*flags*: should be zero

This function will return for each given index a URL of the corresponding key. If the provided index is out of bounds then GNUTLS\_E\_REQUESTED\_DATA\_NOT\_AVAILABLE is returned.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.1.0

## gnutls\_tpm\_privkey\_delete

**int gnutls\_tpm\_privkey\_delete** (*const char \* url, const char \** [Function]  
*srk\_password*)

*url*: the URL describing the key

*srk\_password*: a password for the SRK key

This function will unregister the private key from the TPM chip.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.1.0

## gnutls\_tpm\_privkey\_generate

```
int gnutls_tpm_privkey_generate (gnutls_pk_algorithm_t pk, [Function]
                                unsigned int bits, const char *srk_password, const char *key_password,
                                gnutls_tpmkey_fmt_t format, gnutls_x509_crt_fmt_t pub_format,
                                gnutls_datum_t *privkey, gnutls_datum_t *pubkey, unsigned int flags)
```

*pk*: the public key algorithm

*bits*: the security bits

*srk\_password*: a password to protect the exported key (optional)

*key\_password*: the password for the TPM (optional)

*format*: the format of the private key

*pub\_format*: the format of the public key

*privkey*: the generated key

*pubkey*: the corresponding public key (may be null)

*flags*: should be a list of GNUTLS\_TPM\_\* flags

This function will generate a private key in the TPM chip. The private key will be generated within the chip and will be exported in a wrapped with TPM's master key form. Furthermore the wrapped key can be protected with the provided `password`.

Note that bits in TPM is quantized value. If the input value is not one of the allowed values, then it will be quantized to one of 512, 1024, 2048, 4096, 8192 and 16384.

Allowed flags are:

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.1.0

## E.9 Abstract key API

The following functions are to be used for abstract key handling. Their prototypes lie in `gnutls/abstract.h`.

### gnutls\_certificate\_set\_key

```
int gnutls_certificate_set_key (gnutls_certificate_credentials_t [Function]
                                res, const char **names, int names_size, gnutls_pcert_st *pcert_list, int
                                pcert_list_size, gnutls_privkey_t key)
```

*res*: is a `gnutls_certificate_credentials_t` structure.

*names*: is an array of DNS name of the certificate (NULL if none)

*names\_size*: holds the size of the names list

*pcert\_list*: contains a certificate list (path) for the specified private key

*pcert\_list\_size*: holds the size of the certificate list

*key*: is a `gnutls_privkey_t` key

This function sets a certificate/private key pair in the `gnutls_certificate_credentials_t` structure. This function may be called more than once, in case multiple keys/certificates exist for the server. For clients that wants to send more than its



own end entity certificate (e.g., also an intermediate CA cert) then put the certificate chain in `pcert_list` .

Note that the `pcert_list` and `key` will become part of the credentials structure and must not be deallocated. They will be automatically deallocated when the `res` structure is deinitialized.

If that function fails to load the `res` structure is at an undefined state, it must not be reused to load other keys or certificates.

**Returns:** GNUTLS\_E\_SUCCESS (0) on success, or a negative error code.

**Since:** 3.0

## gnutls\_certificate\_set\_retrieve\_function2

```
void gnutls_certificate_set_retrieve_function2 [Function]
      (gnutls_certificate_credentials_t cred, gnutls_certificate_retrieve_function2 *
      func)
```

*cred*: is a `gnutls_certificate_credentials_t` structure.

*func*: is the callback function

This function sets a callback to be called in order to retrieve the certificate to be used in the handshake.

The callback's function prototype is: `int (*callback)(gnutls_session_t, const gnutls_datum_t* req_ca_dn, int nreqs, const gnutls_pk_algorithm_t* pk_algos, int pk_algos_length, gnutls_pcert_st** pcert, unsigned int *pcert_length, gnutls_privkey_t * pkey);`

`req_ca_dn` is only used in X.509 certificates. Contains a list with the CA names that the server considers trusted. Normally we should send a certificate that is signed by one of these CAs. These names are DER encoded. To get a more meaningful value use the function `gnutls_x509_rdn_get()` .

`pk_algos` contains a list with server's acceptable signature algorithms. The certificate returned should support the server's given algorithms.

`pcert` should contain a single certificate and public key or a list of them.

`pcert_length` is the size of the previous list.

`pkey` is the private key.

If the callback function is provided then gnutls will call it, in the handshake, after the certificate request message has been received. All the provided by the callback values will not be released or modified by gnutls.

In server side `pk_algos` and `req_ca_dn` are NULL.

The callback function should set the certificate list to be sent, and return 0 on success. If no certificate was selected then the number of certificates should be set to zero. The value (-1) indicates error and the handshake will be terminated.

**Since:** 3.0

**gnutls\_pcert\_deinit**

**void gnutls\_pcert\_deinit** (*gnutls\_pcert\_st* \**pcert*) [Function]

*pcert*: The structure to be deinitialized

This function will deinitialize a pcert structure.

**Since:** 3.0

**gnutls\_pcert\_import\_openpgp**

**int gnutls\_pcert\_import\_openpgp** (*gnutls\_pcert\_st* \**pcert*, [Function]

*gnutls\_openpgp\_cert\_t* *crt*, unsigned int *flags*)

*pcert*: The pcert structure

*crt*: The raw certificate to be imported

*flags*: zero for now

This convenience function will import the given certificate to a **gnutls\_pcert\_st** structure. The structure must be deinitialized afterwards using **gnutls\_pcert\_deinit()** ;

**Returns:** On success, **GNUTLS\_E\_SUCCESS** (0) is returned, otherwise a negative error value.

**Since:** 3.0

**gnutls\_pcert\_import\_openpgp\_raw**

**int gnutls\_pcert\_import\_openpgp\_raw** (*gnutls\_pcert\_st* \**pcert*, [Function]

const *gnutls\_datum\_t* \**cert*, *gnutls\_openpgp\_cert\_fmt\_t* *format*,

*gnutls\_openpgp\_keyid\_t* *keyid*, unsigned int *flags*)

*pcert*: The pcert structure

*cert*: The raw certificate to be imported

*format*: The format of the certificate

*keyid*: The key ID to use (NULL for the master key)

*flags*: zero for now

This convenience function will import the given certificate to a **gnutls\_pcert\_st** structure. The structure must be deinitialized afterwards using **gnutls\_pcert\_deinit()** ;

**Returns:** On success, **GNUTLS\_E\_SUCCESS** (0) is returned, otherwise a negative error value.

**Since:** 3.0

**gnutls\_pcert\_import\_x509**

**int gnutls\_pcert\_import\_x509** (*gnutls\_pcert\_st* \**pcert*, [Function]

*gnutls\_x509\_cert\_t* *crt*, unsigned int *flags*)

*pcert*: The pcert structure

*crt*: The raw certificate to be imported

*flags*: zero for now

This convenience function will import the given certificate to a `gnutls_pcert_st` structure. The structure must be deinitialized afterwards using `gnutls_pcert_deinit()` ;

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.0

### `gnutls_pcert_import_x509_raw`

```
int gnutls_pcert_import_x509_raw (gnutls_pcert_st *pcert, const [Function]
                                gnutls_datum_t *cert, gnutls_x509_crt_fmt_t format, unsigned int flags)
pcert: The pcert structure
```

*cert*: The raw certificate to be imported

*format*: The format of the certificate

*flags*: zero for now

This convenience function will import the given certificate to a `gnutls_pcert_st` structure. The structure must be deinitialized afterwards using `gnutls_pcert_deinit()` ;

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.0

### `gnutls_pcert_list_import_x509_raw`

```
int gnutls_pcert_list_import_x509_raw (gnutls_pcert_st * [Function]
                                        pcerts, unsigned int *pcert_max, const gnutls_datum_t *data,
                                        gnutls_x509_crt_fmt_t format, unsigned int flags)
```

*pcerts*: The structures to store the parsed certificate. Must not be initialized.

*pcert\_max*: Initially must hold the maximum number of certs. It will be updated with the number of certs available.

*data*: The certificates.

*format*: One of DER or PEM.

*flags*: must be (0) or an OR'd sequence of `gnutls_certificate_import_flags`.

This function will convert the given PEM encoded certificate list to the native `gnutls_x509_crt_t` format. The output will be stored in `certs` . They will be automatically initialized.

If the Certificate is PEM encoded it should have a header of "X509 CERTIFICATE", or "CERTIFICATE".

**Returns:** the number of certificates read or a negative error value.

**Since:** 3.0

**gnutls\_privkey\_decrypt\_data**

**int gnutls\_privkey\_decrypt\_data** (*gnutls\_privkey\_t* **key**, *unsigned* [Function]  
*int* **flags**, *const gnutls\_datum\_t \** **ciphertext**, *gnutls\_datum\_t \**  
*plaintext*)

*key*: Holds the key

*flags*: zero for now

*ciphertext*: holds the data to be decrypted

*plaintext*: will contain the decrypted data, allocated with **gnutls\_malloc()**

This function will decrypt the given data using the algorithm supported by the private key.

**Returns:** On success, **GNUTLS\_E\_SUCCESS** (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

**gnutls\_privkey\_deinit**

**void gnutls\_privkey\_deinit** (*gnutls\_privkey\_t* **key**) [Function]

*key*: The structure to be deinitialized

This function will deinitialize a private key structure.

**Since:** 2.12.0

**gnutls\_privkey\_export\_dsa\_raw**

**int gnutls\_privkey\_export\_dsa\_raw** (*gnutls\_privkey\_t* **key**, [Function]  
*gnutls\_datum\_t \** **p**, *gnutls\_datum\_t \** **q**, *gnutls\_datum\_t \** **g**, *gnutls\_datum\_t \**  
*y*, *gnutls\_datum\_t \** **x**)

*key*: Holds the public key

*p*: will hold the p

*q*: will hold the q

*g*: will hold the g

*y*: will hold the y

*x*: will hold the x

This function will export the DSA private key's parameters found in the given structure. The new parameters will be allocated using **gnutls\_malloc()** and will be stored in the appropriate datum.

**Returns:** **GNUTLS\_E\_SUCCESS** on success, otherwise a negative error code.

**Since:** 3.3.0

**gnutls\_privkey\_export\_ecc\_raw**

**int gnutls\_privkey\_export\_ecc\_raw** (*gnutls\_privkey\_t* **key**, [Function]  
*gnutls\_ecc\_curve\_t \** **curve**, *gnutls\_datum\_t \** **x**, *gnutls\_datum\_t \** **y**,  
*gnutls\_datum\_t \** **k**)

*key*: Holds the public key

*curve*: will hold the curve  
*x*: will hold the x coordinate  
*y*: will hold the y coordinate  
*k*: will hold the private key

This function will export the ECC private key's parameters found in the given structure. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

**Returns:** GNUTLS\_E\_SUCCESS on success, otherwise a negative error code.

**Since:** 3.3.0

### **gnutls\_privkey\_export\_rsa\_raw**

**int gnutls\_privkey\_export\_rsa\_raw** (*gnutls\_privkey\_t key*, [Function]  
*gnutls\_datum\_t \* m*, *gnutls\_datum\_t \* e*, *gnutls\_datum\_t \* d*, *gnutls\_datum\_t \* p*,  
*gnutls\_datum\_t \* q*, *gnutls\_datum\_t \* u*, *gnutls\_datum\_t \* e1*,  
*gnutls\_datum\_t \* e2*)

*key*: Holds the certificate  
*m*: will hold the modulus  
*e*: will hold the public exponent  
*d*: will hold the private exponent  
*p*: will hold the first prime (p)  
*q*: will hold the second prime (q)  
*u*: will hold the coefficient  
*e1*: will hold  $e1 = d \bmod (p-1)$   
*e2*: will hold  $e2 = d \bmod (q-1)$

This function will export the RSA private key's parameters found in the given structure. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

**Returns:** GNUTLS\_E\_SUCCESS on success, otherwise a negative error code.

**Since:** 3.3.0

### **gnutls\_privkey\_generate**

**int gnutls\_privkey\_generate** (*gnutls\_privkey\_t pkey*, [Function]  
*gnutls\_pk\_algorithm\_t algo*, *unsigned int bits*, *unsigned int flags*)

*pkey*: The private key  
*algo*: is one of the algorithms in `gnutls_pk_algorithm_t`.  
*bits*: the size of the modulus  
*flags*: unused for now. Must be 0.

This function will generate a random private key. Note that this function must be called on an empty private key.

Note that when generating an elliptic curve key, the curve can be substituted in the place of the bits parameter using the `GNUTLS_CURVE_TO_BITS()` macro.

Do not set the number of bits directly, use `gnutls_sec_param_to_pk_bits()` .

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

## `gnutls_privkey_get_pk_algorithm`

`int gnutls_privkey_get_pk_algorithm (gnutls_privkey_t key, [Function]  
unsigned int * bits)`

*key*: should contain a `gnutls_privkey_t` structure

*bits*: If set will return the number of bits of the parameters (may be NULL)

This function will return the public key algorithm of a private key and if possible will return a number of bits that indicates the security parameter of the key.

**Returns:** a member of the `gnutls_pk_algorithm_t` enumeration on success, or a negative error code on error.

**Since:** 2.12.0

## `gnutls_privkey_get_type`

`gnutls_privkey_type_t gnutls_privkey_get_type [Function]  
(gnutls_privkey_t key)`

*key*: should contain a `gnutls_privkey_t` structure

This function will return the type of the private key. This is actually the type of the subsystem used to set this private key.

**Returns:** a member of the `gnutls_privkey_type_t` enumeration on success, or a negative error code on error.

**Since:** 2.12.0

## `gnutls_privkey_import_dsa_raw`

`int gnutls_privkey_import_dsa_raw (gnutls_privkey_t key, const [Function]  
gnutls_datum_t * p, const gnutls_datum_t * q, const gnutls_datum_t * g, const  
gnutls_datum_t * y, const gnutls_datum_t * x)`

*key*: The structure to store the parsed key

*p*: holds the p

*q*: holds the q

*g*: holds the g

*y*: holds the y

*x*: holds the x

This function will convert the given DSA raw parameters to the native `gnutls_privkey_t` format. The output will be stored in *key* .

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**gnutls\_privkey\_import\_ecc\_raw**

**int** gnutls\_privkey\_import\_ecc\_raw (*gnutls\_privkey\_t* **key**, [Function]  
           *gnutls\_ecc\_curve\_t* **curve**, *const gnutls\_datum\_t* \* **x**, *const gnutls\_datum\_t* \* **y**,  
           *const gnutls\_datum\_t* \* **k**)

**key**: The structure to store the parsed key

**curve**: holds the curve

**x**: holds the x

**y**: holds the y

**k**: holds the k

This function will convert the given elliptic curve parameters to the native **gnutls\_privkey\_t** format. The output will be stored in **key** .

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.0

**gnutls\_privkey\_import\_ext**

**int** gnutls\_privkey\_import\_ext (*gnutls\_privkey\_t* **pkey**, [Function]  
           *gnutls\_pk\_algorithm\_t* **pk**, *void \** **userdata**, *gnutls\_privkey\_sign\_func*  
           **sign\_func**, *gnutls\_privkey\_decrypt\_func* **decrypt\_func**, *unsigned int* **flags**)

**pkey**: The private key

**pk**: The public key algorithm

**userdata**: private data to be provided to the callbacks

**sign\_func**: callback for signature operations

**decrypt\_func**: callback for decryption operations

**flags**: Flags for the import

This function will associate the given callbacks with the **gnutls\_privkey\_t** structure. At least one of the two callbacks must be non-null.

See also **gnutls\_privkey\_import\_ext2()** .

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.0

**gnutls\_privkey\_import\_ext2**

**int** gnutls\_privkey\_import\_ext2 (*gnutls\_privkey\_t* **pkey**, [Function]  
           *gnutls\_pk\_algorithm\_t* **pk**, *void \** **userdata**, *gnutls\_privkey\_sign\_func*  
           **sign\_func**, *gnutls\_privkey\_decrypt\_func* **decrypt\_func**,  
           *gnutls\_privkey\_deinit\_func* **deinit\_func**, *unsigned int* **flags**)

**pkey**: The private key

**pk**: The public key algorithm

**userdata**: private data to be provided to the callbacks

**sign\_func**: callback for signature operations

*decrypt\_func*: callback for decryption operations

*deinit\_func*: a deinitialization function

*flags*: Flags for the import

This function will associate the given callbacks with the `gnutls_privkey_t` structure. At least one of the two callbacks must be non-null. If a deinitialization function is provided then flags is assumed to contain `GNUTLS_PRIVKEY_IMPORT_AUTO_RELEASE`. Note that the signing function is supposed to "raw" sign data, i.e., without any hashing or preprocessing. In case of RSA the `DigestInfo` will be provided, and the signing function is expected to do the PKCS 1 1.5 padding and the exponentiation.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.1

## `gnutls_privkey_import_openpgp`

`int gnutls_privkey_import_openpgp (gnutls_privkey_t pkey, [Function]  
gnutls_openpgp_privkey_t key, unsigned int flags)`

*pkey*: The private key

*key*: The private key to be imported

*flags*: Flags for the import

This function will import the given private key to the abstract `gnutls_privkey_t` structure.

The `gnutls_openpgp_privkey_t` object must not be deallocated during the lifetime of this structure. The subkey set as preferred will be used, or the master key otherwise. *flags* might be zero or one of `GNUTLS_PRIVKEY_IMPORT_AUTO_RELEASE` and `GNUTLS_PRIVKEY_IMPORT_COPY`.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

## `gnutls_privkey_import_openpgp_raw`

`int gnutls_privkey_import_openpgp_raw (gnutls_privkey_t pkey, [Function]  
const gnutls_datum_t * data, gnutls_openpgp_crt_fmt_t format, const  
gnutls_openpgp_keyid_t keyid, const char * password)`

*pkey*: The private key

*data*: The private key data to be imported

*format*: The format of the private key

*keyid*: The key id to use (optional)

*password*: A password (optional)

This function will import the given private key to the abstract `gnutls_privkey_t` structure.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.1.0



**gnutls\_privkey\_import\_pkcs11**

**int gnutls\_privkey\_import\_pkcs11** (*gnutls\_privkey\_t pkey*, [Function]  
*gnutls\_pkcs11\_privkey\_t key*, unsigned int *flags*)

*pkey*: The private key

*key*: The private key to be imported

*flags*: Flags for the import

This function will import the given private key to the abstract `gnutls_privkey_t` structure.

The `gnutls_pkcs11_privkey_t` object must not be deallocated during the lifetime of this structure.

*flags* might be zero or one of `GNUTLS_PRIVKEY_IMPORT_AUTO_RELEASE` and `GNUTLS_PRIVKEY_IMPORT_COPY`.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

**gnutls\_privkey\_import\_pkcs11\_url**

**int gnutls\_privkey\_import\_pkcs11\_url** (*gnutls\_privkey\_t key*, [Function]  
 const char \* *url*)

*key*: A key of type `gnutls_pubkey_t`

*url*: A PKCS 11 url

This function will import a PKCS 11 private key to a `gnutls_private_key_t` structure.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.1.0

**gnutls\_privkey\_import\_rsa\_raw**

**int gnutls\_privkey\_import\_rsa\_raw** (*gnutls\_privkey\_t key*, const [Function]  
*gnutls\_datum\_t \* m*, const *gnutls\_datum\_t \* e*, const *gnutls\_datum\_t \* d*, const  
*gnutls\_datum\_t \* p*, const *gnutls\_datum\_t \* q*, const *gnutls\_datum\_t \* u*, const  
*gnutls\_datum\_t \* e1*, const *gnutls\_datum\_t \* e2*)

*key*: The structure to store the parsed key

*m*: holds the modulus

*e*: holds the public exponent

*d*: holds the private exponent

*p*: holds the first prime (p)

*q*: holds the second prime (q)

*u*: holds the coefficient (optional)

*e1*: holds  $e1 = d \bmod (p-1)$  (optional)

*e2*: holds  $e2 = d \bmod (q-1)$  (optional)

This function will convert the given RSA raw parameters to the native `gnutls_privkey_t` format. The output will be stored in `key`.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

### `gnutls_privkey_import_tpm_raw`

```
int gnutls_privkey_import_tpm_raw (gnutls_privkey_t pkey, const [Function]
    gnutls_datum_t * fdata, gnutls_tpmkey_fmt_t format, const char *
    srk_password, const char * key_password, unsigned int flags)
```

*pkey*: The private key

*fdata*: The TPM key to be imported

*format*: The format of the private key

*srk\_password*: The password for the SRK key (optional)

*key\_password*: A password for the key (optional)

*flags*: should be zero

This function will import the given private key to the abstract `gnutls_privkey_t` structure.

With respect to passwords the same as in `gnutls_privkey_import_tpm_url()` apply.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.1.0

### `gnutls_privkey_import_tpm_url`

```
int gnutls_privkey_import_tpm_url (gnutls_privkey_t pkey, const [Function]
    char * url, const char * srk_password, const char * key_password,
    unsigned int flags)
```

*pkey*: The private key

*url*: The URL of the TPM key to be imported

*srk\_password*: The password for the SRK key (optional)

*key\_password*: A password for the key (optional)

*flags*: One of the `GNUTLS_PRIVKEY_*` flags

This function will import the given private key to the abstract `gnutls_privkey_t` structure.

Note that unless `GNUTLS_PRIVKEY_DISABLE_CALLBACKS` is specified, if incorrect (or NULL) passwords are given the PKCS11 callback functions will be used to obtain the correct passwords. Otherwise if the SRK password is wrong `GNUTLS_E_TPM_SRK_PASSWORD_ERROR` is returned and if the key password is wrong or not provided then `GNUTLS_E_TPM_KEY_PASSWORD_ERROR` is returned.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.1.0

**gnutls\_privkey\_import\_url**

**int gnutls\_privkey\_import\_url** (*gnutls\_privkey\_t key, const char \* url, unsigned int flags*) [Function]

*key*: A key of type `gnutls_privkey_t`

*url*: A PKCS 11 url

*flags*: should be zero

This function will import a PKCS11 or TPM URL as a private key. The supported URL types can be checked using `gnutls_url_is_supported()` .

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.1.0

**gnutls\_privkey\_import\_x509**

**int gnutls\_privkey\_import\_x509** (*gnutls\_privkey\_t pkey, gnutls\_x509\_privkey\_t key, unsigned int flags*) [Function]

*pkey*: The private key

*key*: The private key to be imported

*flags*: Flags for the import

This function will import the given private key to the abstract `gnutls_privkey_t` structure.

The `gnutls_x509_privkey_t` object must not be deallocated during the lifetime of this structure.

*flags* might be zero or one of `GNUTLS_PRIVKEY_IMPORT_AUTO_RELEASE` and `GNUTLS_PRIVKEY_IMPORT_COPY` .

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

**gnutls\_privkey\_import\_x509\_raw**

**int gnutls\_privkey\_import\_x509\_raw** (*gnutls\_privkey\_t pkey, const gnutls\_datum\_t \* data, gnutls\_x509\_crt\_fmt\_t format, const char \* password, unsigned int flags*) [Function]

*pkey*: The private key

*data*: The private key data to be imported

*format*: The format of the private key

*password*: A password (optional)

*flags*: an ORed sequence of `gnutls_pkcs_encrypt_flags_t`

This function will import the given private key to the abstract `gnutls_privkey_t` structure.

The supported formats are basic unencrypted key, PKCS8, PKCS12, and the openssl format.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.1.0

## gnutls\_privkey\_init

`int gnutls_privkey_init (gnutls_privkey_t * key)` [Function]

*key*: The structure to be initialized

This function will initialize an private key structure.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

## gnutls\_privkey\_set\_pin\_function

`void gnutls_privkey_set_pin_function (gnutls_privkey_t key, gnutls_pin_callback_t fn, void * userdata)` [Function]

*key*: A key of type `gnutls_privkey_t`

*fn*: the callback

*userdata*: data associated with the callback

This function will set a callback function to be used when required to access the object. This function overrides any other global PIN functions.

Note that this function must be called right after initialization to have effect.

**Since:** 3.1.0

## gnutls\_privkey\_sign\_data

`int gnutls_privkey_sign_data (gnutls_privkey_t signer, gnutls_digest_algorithm_t hash, unsigned int flags, const gnutls_datum_t * data, gnutls_datum_t * signature)` [Function]

*signer*: Holds the key

*hash*: should be a digest algorithm

*flags*: Zero or one of `gnutls_privkey_flags_t`

*data*: holds the data to be signed

*signature*: will contain the signature allocate with `gnutls_malloc()`

This function will sign the given data using a signature algorithm supported by the private key. Signature algorithms are always used together with a hash functions. Different hash functions may be used for the RSA algorithm, but only the SHA family for the DSA keys.

You may use `gnutls_pubkey_get_preferred_hash_algorithm()` to determine the hash algorithm.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

## gnutls\_privkey\_sign\_hash

```
int gnutls_privkey_sign_hash (gnutls_privkey_t signer, [Function]
                             gnutls_digest_algorithm_t hash_algo, unsigned int flags, const
                             gnutls_datum_t * hash_data, gnutls_datum_t * signature)
```

*signer*: Holds the signer's key

*hash\_algo*: The hash algorithm used

*flags*: Zero or one of `gnutls_privkey_flags_t`

*hash\_data*: holds the data to be signed

*signature*: will contain newly allocated signature

This function will sign the given hashed data using a signature algorithm supported by the private key. Signature algorithms are always used together with a hash functions. Different hash functions may be used for the RSA algorithm, but only SHA-XXX for the DSA keys.

You may use `gnutls_pubkey_get_preferred_hash_algorithm()` to determine the hash algorithm.

Note that if `GNUTLS_PRIVKEY_SIGN_FLAG_TLS1_RSA` flag is specified this function will ignore *hash\_algo* and perform a raw PKCS1 signature.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

## gnutls\_privkey\_status

```
int gnutls_privkey_status (gnutls_privkey_t key) [Function]
key: Holds the key
```

Checks the status of the private key token. This function is an actual wrapper over `gnutls_pkcs11_privkey_status()`, and if the private key is a PKCS 11 token it will check whether it is inserted or not.

**Returns:** this function will return non-zero if the token holding the private key is still available (inserted), and zero otherwise.

**Since:** 3.1.10

## gnutls\_privkey\_verify\_params

```
int gnutls_privkey_verify_params (gnutls_privkey_t key) [Function]
key: should contain a gnutls_privkey_t structure
```

This function will verify the private key parameters.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

**gnutls\_pubkey\_deinit**

**void gnutls\_pubkey\_deinit** (*gnutls\_pubkey\_t key*) [Function]

*key*: The structure to be deinitialized

This function will deinitialize a public key structure.

**Since:** 2.12.0

**gnutls\_pubkey\_encrypt\_data**

**int gnutls\_pubkey\_encrypt\_data** (*gnutls\_pubkey\_t key, unsigned int flags, const gnutls\_datum\_t \*plaintext, gnutls\_datum\_t \*ciphertext*) [Function]

*key*: Holds the public key

*flags*: should be 0 for now

*plaintext*: The data to be encrypted

*ciphertext*: contains the encrypted data

This function will encrypt the given data, using the public key.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.0

**gnutls\_pubkey\_export**

**int gnutls\_pubkey\_export** (*gnutls\_pubkey\_t key, gnutls\_x509\_crt\_fmt\_t format, void \*output\_data, size\_t \*output\_data\_size*) [Function]

*key*: Holds the certificate

*format*: the format of output params. One of PEM or DER.

*output\_data*: will contain a certificate PEM or DER encoded

*output\_data\_size*: holds the size of output\_data (and will be replaced by the actual size of parameters)

This function will export the public key to DER or PEM format. The contents of the exported data is the SubjectPublicKeyInfo X.509 structure.

If the buffer provided is not long enough to hold the output, then \*output\_data\_size is updated and GNUTLS\_E\_SHORT\_MEMORY\_BUFFER will be returned.

If the structure is PEM encoded, it will have a header of "BEGIN CERTIFICATE".

**Returns:** In case of failure a negative error code will be returned, and 0 on success.

**Since:** 2.12.0

**gnutls\_pubkey\_export2**

**int gnutls\_pubkey\_export2** (*gnutls\_pubkey\_t key, gnutls\_x509\_crt\_fmt\_t format, gnutls\_datum\_t \*out*) [Function]

*key*: Holds the certificate

*format*: the format of output params. One of PEM or DER.

*out*: will contain a certificate PEM or DER encoded

This function will export the public key to DER or PEM format. The contents of the exported data is the SubjectPublicKeyInfo X.509 structure.

The output buffer will be allocated using `gnutls_malloc()` .

If the structure is PEM encoded, it will have a header of "BEGIN CERTIFICATE".

**Returns:** In case of failure a negative error code will be returned, and 0 on success.

**Since:** 3.1.3

### `gnutls_pubkey_export_dsa_raw`

```
int gnutls_pubkey_export_dsa_raw (gnutls_pubkey_t key, [Function]
                                gnutls_datum_t * p, gnutls_datum_t * q, gnutls_datum_t * g, gnutls_datum_t *
                                y)
```

*key*: Holds the public key

*p*: will hold the p

*q*: will hold the q

*g*: will hold the g

*y*: will hold the y

This function will export the DSA public key's parameters found in the given certificate. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

**Returns:** GNUTLS\_E\_SUCCESS on success, otherwise a negative error code.

**Since:** 3.3.0

### `gnutls_pubkey_export_ecc_raw`

```
int gnutls_pubkey_export_ecc_raw (gnutls_pubkey_t key, [Function]
                                gnutls_ecc_curve_t * curve, gnutls_datum_t * x, gnutls_datum_t * y)
```

*key*: Holds the public key

*curve*: will hold the curve

*x*: will hold x

*y*: will hold y

This function will export the ECC public key's parameters found in the given certificate. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

**Returns:** GNUTLS\_E\_SUCCESS on success, otherwise a negative error code.

**Since:** 3.0

### `gnutls_pubkey_export_ecc_x962`

```
int gnutls_pubkey_export_ecc_x962 (gnutls_pubkey_t key, [Function]
                                gnutls_datum_t * parameters, gnutls_datum_t * ecpoint)
```

*key*: Holds the public key

*parameters*: DER encoding of an ANSI X9.62 parameters

*ecpoint*: DER encoding of ANSI X9.62 ECPoint

This function will export the ECC public key's parameters found in the given certificate. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

**Returns:** GNUTLS\_E\_SUCCESS on success, otherwise a negative error code.

**Since:** 3.3.0

## gnutls\_pubkey\_export\_rsa\_raw

`int gnutls_pubkey_export_rsa_raw (gnutls_pubkey_t key, [Function]  
gnutls_datum_t * m, gnutls_datum_t * e)`

*key*: Holds the certificate

*m*: will hold the modulus

*e*: will hold the public exponent

This function will export the RSA public key's parameters found in the given structure. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

**Returns:** GNUTLS\_E\_SUCCESS on success, otherwise a negative error code.

**Since:** 3.3.0

## gnutls\_pubkey\_get\_key\_id

`int gnutls_pubkey_get_key_id (gnutls_pubkey_t key, unsigned int [Function]  
flags, unsigned char * output_data, size_t * output_data_size)`

*key*: Holds the public key

*flags*: should be 0 for now

*output\_data*: will contain the key ID

*output\_data\_size*: holds the size of output\_data (and will be replaced by the actual size of parameters)

This function will return a unique ID that depends on the public key parameters. This ID can be used in checking whether a certificate corresponds to the given public key.

If the buffer provided is not long enough to hold the output, then `*output_data_size` is updated and GNUTLS\_E\_SHORT\_MEMORY\_BUFFER will be returned. The output will normally be a SHA-1 hash output, which is 20 bytes.

**Returns:** In case of failure a negative error code will be returned, and 0 on success.

**Since:** 2.12.0

## gnutls\_pubkey\_get\_key\_usage

`int gnutls_pubkey_get_key_usage (gnutls_pubkey_t key, unsigned [Function]  
int * usage)`

*key*: should contain a `gnutls_pubkey_t` structure

*usage*: If set will return the number of bits of the parameters (may be NULL)



This function will return the key usage of the public key.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

## gnutls\_pubkey\_get\_openpgp\_key\_id

```
int gnutls_pubkey_get_openpgp_key_id (gnutls_pubkey_t key,           [Function]
                                     unsigned int flags, unsigned char * output_data, size_t *
                                     output_data_size, unsigned int * subkey)
```

*key*: Holds the public key

*flags*: should be 0 or GNUTLS\_PUBKEY\_GET\_OPENPGP\_FINGERPRINT

*output\_data*: will contain the key ID

*output\_data\_size*: holds the size of *output\_data* (and will be replaced by the actual size of parameters)

*subkey*: Will be non zero if the key ID corresponds to a subkey

This function returns the OpenPGP key ID of the corresponding key. The key is a unique ID that depends on the public key parameters.

If the flag GNUTLS\_PUBKEY\_GET\_OPENPGP\_FINGERPRINT is specified this function returns the fingerprint of the master key.

If the buffer provided is not long enough to hold the output, then *\*output\_data\_size* is updated and GNUTLS\_E\_SHORT\_MEMORY\_BUFFER will be returned. The output is GNUTLS\_OPENPGP\_KEYID\_SIZE bytes long.

**Returns:** In case of failure a negative error code will be returned, and 0 on success.

**Since:** 3.0

## gnutls\_pubkey\_get\_pk\_algorithm

```
int gnutls_pubkey_get_pk_algorithm (gnutls_pubkey_t key,           [Function]
                                   unsigned int * bits)
```

*key*: should contain a *gnutls\_pubkey\_t* structure

*bits*: If set will return the number of bits of the parameters (may be NULL)

This function will return the public key algorithm of a public key and if possible will return a number of bits that indicates the security parameter of the key.

**Returns:** a member of the *gnutls\_pk\_algorithm\_t* enumeration on success, or a negative error code on error.

**Since:** 2.12.0

## gnutls\_pubkey\_get\_preferred\_hash\_algorithm

```
int gnutls_pubkey_get_preferred_hash_algorithm (gnutls_pubkey_t key, [Function]
                                                gnutls_digest_algorithm_t * hash, unsigned int * mand)
```

*key*: Holds the certificate

*hash*: The result of the call with the hash algorithm used for signature

*mand*: If non zero it means that the algorithm MUST use this hash. May be NULL. This function will read the certificate and return the appropriate digest algorithm to use for signing with this certificate. Some certificates (i.e. DSA might not be able to sign without the preferred algorithm).

To get the signature algorithm instead of just the hash use `gnutls_pk_to_sign()` with the algorithm of the certificate/key and the provided `hash`.

**Returns:** the 0 if the hash algorithm is found. A negative error code is returned on error.

**Since:** 2.12.0

## gnutls\_pubkey\_get\_verify\_algorithm

```
int gnutls_pubkey_get_verify_algorithm (gnutls_pubkey_t key,          [Function]
                                       const gnutls_datum_t * signature, gnutls_digest_algorithm_t * hash)
```

*key*: Holds the certificate

*signature*: contains the signature

*hash*: The result of the call with the hash algorithm used for signature

This function will read the certificate and the signed data to determine the hash algorithm used to generate the signature.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

## gnutls\_pubkey\_import

```
int gnutls_pubkey_import (gnutls_pubkey_t key, const                [Function]
                          gnutls_datum_t * data, gnutls_x509_crt_fmt_t format)
```

*key*: The structure to store the parsed public key.

*data*: The DER or PEM encoded certificate.

*format*: One of DER or PEM

This function will import the provided public key in a SubjectPublicKeyInfo X.509 structure to a native `gnutls_pubkey_t` structure. The output will be stored in `key`. If the public key is PEM encoded it should have a header of "PUBLIC KEY".

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

## gnutls\_pubkey\_import\_dsa\_raw

```
int gnutls_pubkey_import_dsa_raw (gnutls_pubkey_t key, const        [Function]
                                   gnutls_datum_t * p, const gnutls_datum_t * q, const gnutls_datum_t * g, const
                                   gnutls_datum_t * y)
```

*key*: The structure to store the parsed key

*p*: holds the p

*q*: holds the q

*g*: holds the *g*

*y*: holds the *y*

This function will convert the given DSA raw parameters to the native `gnutls_pubkey_t` format. The output will be stored in `key`.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

## `gnutls_pubkey_import_ecc_raw`

```
int gnutls_pubkey_import_ecc_raw (gnutls_pubkey_t key, [Function]
                                gnutls_ecc_curve_t curve, const gnutls_datum_t * x, const gnutls_datum_t * y)
```

*key*: The structure to store the parsed key

*curve*: holds the curve

*x*: holds the *x*

*y*: holds the *y*

This function will convert the given elliptic curve parameters to a `gnutls_pubkey_t`. The output will be stored in `key`.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.0

## `gnutls_pubkey_import_ecc_x962`

```
int gnutls_pubkey_import_ecc_x962 (gnutls_pubkey_t key, const [Function]
                                gnutls_datum_t * parameters, const gnutls_datum_t * ecpoint)
```

*key*: The structure to store the parsed key

*parameters*: DER encoding of an ANSI X9.62 parameters

*ecpoint*: DER encoding of ANSI X9.62 ECPoint

This function will convert the given elliptic curve parameters to a `gnutls_pubkey_t`. The output will be stored in `key`.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.0

## `gnutls_pubkey_import_openpgp`

```
int gnutls_pubkey_import_openpgp (gnutls_pubkey_t key, [Function]
                                gnutls_openpgp_cert_t crt, unsigned int flags)
```

*key*: The public key

*crt*: The certificate to be imported

*flags*: should be zero

Imports a public key from an openpgp key. This function will import the given public key to the abstract `gnutls_pubkey_t` structure. The subkey set as preferred will be imported or the master key otherwise.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

### gnutls\_pubkey\_import\_openpgp\_raw

```
int gnutls_pubkey_import_openpgp_raw (gnutls_pubkey_t pkey,          [Function]
                                       const gnutls_datum_t *data, gnutls_openpgp_crt_fmt_t format, const
                                       gnutls_openpgp_keyid_t keyid, unsigned int flags)
```

*pkey*: The public key

*data*: The public key data to be imported

*format*: The format of the public key

*keyid*: The key id to use (optional)

*flags*: Should be zero

This function will import the given public key to the abstract `gnutls_pubkey_t` structure.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.1.3

### gnutls\_pubkey\_import\_pkcs11

```
int gnutls_pubkey_import_pkcs11 (gnutls_pubkey_t key,              [Function]
                                  gnutls_pkcs11_obj_t obj, unsigned int flags)
```

*key*: The public key

*obj*: The parameters to be imported

*flags*: should be zero

Imports a public key from a pkcs11 key. This function will import the given public key to the abstract `gnutls_pubkey_t` structure.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

### gnutls\_pubkey\_import\_pkcs11\_url

```
int gnutls_pubkey_import_pkcs11_url (gnutls_pubkey_t key, const   [Function]
                                       char *url, unsigned int flags)
```

*key*: A key of type `gnutls_pubkey_t`

*url*: A PKCS 11 url

*flags*: One of GNUTLS\_PKCS11\_OBJ\_\* flags

This function will import a PKCS 11 certificate to a `gnutls_pubkey_t` structure.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

## gnutls\_pubkey\_import\_privkey

`int gnutls_pubkey_import_privkey (gnutls_pubkey_t key, [Function]  
                                   gnutls_privkey_t pkey, unsigned int usage, unsigned int flags)`

*key*: The public key

*pkey*: The private key

*usage*: GNUTLS\_KEY\_\* key usage flags.

*flags*: should be zero

Imports the public key from a private. This function will import the given public key to the abstract `gnutls_pubkey_t` structure.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

## gnutls\_pubkey\_import\_rsa\_raw

`int gnutls_pubkey_import_rsa_raw (gnutls_pubkey_t key, const [Function]  
                                   gnutls_datum_t * m, const gnutls_datum_t * e)`

*key*: Is a structure will hold the parameters

*m*: holds the modulus

*e*: holds the public exponent

This function will replace the parameters in the given structure. The new parameters should be stored in the appropriate `gnutls_datum`.

**Returns:** GNUTLS\_E\_SUCCESS on success, or an negative error code.

**Since:** 2.12.0

## gnutls\_pubkey\_import\_tpm\_raw

`int gnutls_pubkey_import_tpm_raw (gnutls_pubkey_t pkey, const [Function]  
                                   gnutls_datum_t * fdata, gnutls_tpmkey_fmt_t format, const char *  
                                   srk_password, unsigned int flags)`

*pkey*: The public key

*fdata*: The TPM key to be imported

*format*: The format of the private key

*srk\_password*: The password for the SRK key (optional)

*flags*: One of the GNUTLS\_PUBKEY\_\* flags

This function will import the public key from the provided TPM key structure.

With respect to passwords the same as in `gnutls_pubkey_import_tpm_url()` apply.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.1.0

## gnutls\_pubkey\_import\_tpm\_url

**int gnutls\_pubkey\_import\_tpm\_url** (*gnutls\_pubkey\_t pkey, const char \* url, const char \* srk\_password, unsigned int flags*) [Function]

*pkey*: The public key

*url*: The URL of the TPM key to be imported

*srk\_password*: The password for the SRK key (optional)

*flags*: should be zero

This function will import the given private key to the abstract `gnutls_privkey_t` structure.

Note that unless `GNUTLS_PUBKEY_DISABLE_CALLBACKS` is specified, if incorrect (or NULL) passwords are given the PKCS11 callback functions will be used to obtain the correct passwords. Otherwise if the SRK password is wrong `GNUTLS_E_TPM_SRK_PASSWORD_ERROR` is returned.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.1.0

## gnutls\_pubkey\_import\_url

**int gnutls\_pubkey\_import\_url** (*gnutls\_pubkey\_t key, const char \* url, unsigned int flags*) [Function]

*key*: A key of type `gnutls_pubkey_t`

*url*: A PKCS 11 url

*flags*: One of `GNUTLS_PKCS11_OBJ_*` flags

This function will import a PKCS11 certificate or a TPM key as a public key.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.1.0

## gnutls\_pubkey\_import\_x509

**int gnutls\_pubkey\_import\_x509** (*gnutls\_pubkey\_t key, gnutls\_x509\_crt\_t crt, unsigned int flags*) [Function]

*key*: The public key

*crt*: The certificate to be imported

*flags*: should be zero

This function will import the given public key to the abstract `gnutls_pubkey_t` structure.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

**gnutls\_pubkey\_import\_x509\_crq**

**int** gnutls\_pubkey\_import\_x509\_crq (*gnutls\_pubkey\_t* *key*, [Function]  
*gnutls\_x509\_crq\_t* *crq*, unsigned int *flags*)

*key*: The public key

*crq*: The certificate to be imported

*flags*: should be zero

This function will import the given public key to the abstract *gnutls\_pubkey\_t* structure.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.1.5

**gnutls\_pubkey\_import\_x509\_raw**

**int** gnutls\_pubkey\_import\_x509\_raw (*gnutls\_pubkey\_t* *pkey*, const [Function]  
*gnutls\_datum\_t* \* *data*, *gnutls\_x509\_crt\_fmt\_t* *format*, unsigned int *flags*)

*pkey*: The public key

*data*: The public key data to be imported

*format*: The format of the public key

*flags*: should be zero

This function will import the given public key to the abstract *gnutls\_pubkey\_t* structure.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 3.1.3

**gnutls\_pubkey\_init**

**int** gnutls\_pubkey\_init (*gnutls\_pubkey\_t* \* *key*) [Function]

*key*: The structure to be initialized

This function will initialize an public key structure.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

**gnutls\_pubkey\_print**

**int** gnutls\_pubkey\_print (*gnutls\_pubkey\_t* *pubkey*, [Function]  
*gnutls\_certificate\_print\_formats\_t* *format*, *gnutls\_datum\_t* \* *out*)

*pubkey*: The structure to be printed

*format*: Indicate the format to use

*out*: Newly allocated datum with null terminated string.

This function will pretty print public key information, suitable for display to a human.

Only `GNUTLS_CRT_PRINT_FULL` and `GNUTLS_CRT_PRINT_FULL_NUMBERS` are implemented.

The output `out` needs to be deallocated using `gnutls_free()`.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.1.5

## `gnutls_pubkey_set_key_usage`

`int gnutls_pubkey_set_key_usage (gnutls_pubkey_t key, unsigned int usage)` [Function]

*key*: a certificate of type `gnutls_x509_crt_t`

*usage*: an ORed sequence of the `GNUTLS_KEY_*` elements.

This function will set the key usage flags of the public key. This is only useful if the key is to be exported to a certificate or certificate request.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

## `gnutls_pubkey_set_pin_function`

`void gnutls_pubkey_set_pin_function (gnutls_pubkey_t key, gnutls_pin_callback_t fn, void *userdata)` [Function]

*key*: A key of type `gnutls_pubkey_t`

*fn*: the callback

*userdata*: data associated with the callback

This function will set a callback function to be used when required to access the object. This function overrides any other global PIN functions.

Note that this function must be called right after initialization to have effect.

**Since:** 3.1.0

## `gnutls_pubkey_verify_data`

`int gnutls_pubkey_verify_data (gnutls_pubkey_t pubkey, unsigned int flags, const gnutls_datum_t *data, const gnutls_datum_t *signature)` [Function]

*pubkey*: Holds the public key

*flags*: Zero or one of `gnutls_pubkey_flags_t`

*data*: holds the signed data

*signature*: contains the signature

This function will verify the given signed data, using the parameters from the certificate.

Deprecated. This function cannot be easily used securely. Use `gnutls_pubkey_verify_data2()` instead.

**Returns:** In case of a verification failure `GNUTLS_E_PK_SIG_VERIFY_FAILED` is returned, and zero or positive code on success.

**Since:** 2.12.0



## gnutls\_pubkey\_verify\_data2

```
int gnutls_pubkey_verify_data2 (gnutls_pubkey_t pubkey, [Function]
                               gnutls_sign_algorithm_t algo, unsigned int flags, const gnutls_datum_t *
                               data, const gnutls_datum_t * signature)
```

*pubkey*: Holds the public key

*algo*: The signature algorithm used

*flags*: Zero or one of `gnutls_pubkey_flags_t`

*data*: holds the signed data

*signature*: contains the signature

This function will verify the given signed data, using the parameters from the certificate.

**Returns:** In case of a verification failure `GNUTLS_E_PK_SIG_VERIFY_FAILED` is returned, and zero or positive code on success.

**Since:** 3.0

## gnutls\_pubkey\_verify\_hash

```
int gnutls_pubkey_verify_hash (gnutls_pubkey_t key, unsigned int [Function]
                               flags, const gnutls_datum_t * hash, const gnutls_datum_t * signature)
```

*key*: Holds the public key

*flags*: Zero or one of `gnutls_pubkey_flags_t`

*hash*: holds the hash digest to be verified

*signature*: contains the signature

This function will verify the given signed digest, using the parameters from the public key.

Deprecated. This function cannot be easily used securely. Use `gnutls_pubkey_verify_hash2()` instead.

**Returns:** In case of a verification failure `GNUTLS_E_PK_SIG_VERIFY_FAILED` is returned, and zero or positive code on success.

**Since:** 2.12.0

## gnutls\_pubkey\_verify\_hash2

```
int gnutls_pubkey_verify_hash2 (gnutls_pubkey_t key, [Function]
                                gnutls_sign_algorithm_t algo, unsigned int flags, const gnutls_datum_t *
                                hash, const gnutls_datum_t * signature)
```

*key*: Holds the public key

*algo*: The signature algorithm used

*flags*: Zero or one of `gnutls_pubkey_flags_t`

*hash*: holds the hash digest to be verified

*signature*: contains the signature

This function will verify the given signed digest, using the parameters from the public key. Note that unlike `gnutls_privkey_sign_hash()`, this function accepts a signature algorithm instead of a digest algorithm. You can use `gnutls_pk_to_sign()` to get the appropriate value.

**Returns:** In case of a verification failure `GNUTLS_E_PK_SIG_VERIFY_FAILED` is returned, and zero or positive code on success.

**Since:** 3.0

## gnutls\_pubkey\_verify\_params

`int gnutls_pubkey_verify_params (gnutls_pubkey_t key)` [Function]  
*key*: should contain a `gnutls_pubkey_t` structure

This function will verify the private key parameters.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.3.0

## gnutls\_x509\_crl\_privkey\_sign

`int gnutls_x509_crl_privkey_sign (gnutls_x509_crl_t crl,` [Function]  
`gnutls_x509_crt_t issuer, gnutls_privkey_t issuer_key,`  
`gnutls_digest_algorithm_t dig, unsigned int flags)`

*crl*: should contain a `gnutls_x509_crl_t` structure

*issuer*: is the certificate of the certificate issuer

*issuer\_key*: holds the issuer's private key

*dig*: The message digest to use. `GNUTLS_DIG_SHA1` is the safe choice unless you know what you're doing.

*flags*: must be 0

This function will sign the CRL with the issuer's private key, and will copy the issuer's information into the CRL.

This must be the last step in a certificate CRL since all the previously set parameters are now signed.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

## gnutls\_x509\_crq\_privkey\_sign

`int gnutls_x509_crq_privkey_sign (gnutls_x509_crq_t crq,` [Function]  
`gnutls_privkey_t key, gnutls_digest_algorithm_t dig, unsigned int flags)`

*crq*: should contain a `gnutls_x509_crq_t` structure

*key*: holds a private key

*dig*: The message digest to use, i.e., `GNUTLS_DIG_SHA1`

*flags*: must be 0

This function will sign the certificate request with a private key. This must be the same key as the one used in `gnutls_x509 crt_set_key()` since a certificate request is self signed.

This must be the last step in a certificate request generation since all the previously set parameters are now signed.

**Returns:** `GNUTLS_E_SUCCESS` on success, otherwise a negative error code. `GNUTLS_E_ASN1_VALUE_NOT_FOUND` is returned if you didn't set all information in the certificate request (e.g., the version using `gnutls_x509_crq_set_version()` ).

**Since:** 2.12.0

## `gnutls_x509_crq_set_pubkey`

`int gnutls_x509_crq_set_pubkey (gnutls_x509_crq_t crq, [Function]  
gnutls_pubkey_t key)`

*crq*: should contain a `gnutls_x509_crq_t` structure

*key*: holds a public key

This function will set the public parameters from the given public key to the request.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

## `gnutls_x509 crt_privkey_sign`

`int gnutls_x509 crt_privkey_sign (gnutls_x509 crt_t crt, [Function]  
gnutls_x509 crt_t issuer, gnutls_privkey_t issuer_key,  
gnutls_digest_algorithm_t dig, unsigned int flags)`

*crt*: a certificate of type `gnutls_x509 crt_t`

*issuer*: is the certificate of the certificate issuer

*issuer\_key*: holds the issuer's private key

*dig*: The message digest to use, `GNUTLS_DIG_SHA1` is a safe choice

*flags*: must be 0

This function will sign the certificate with the issuer's private key, and will copy the issuer's information into the certificate.

This must be the last step in a certificate generation since all the previously set parameters are now signed.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

## `gnutls_x509 crt_set_pubkey`

`int gnutls_x509 crt_set_pubkey (gnutls_x509 crt_t crt, [Function]  
gnutls_pubkey_t key)`

*crt*: should contain a `gnutls_x509 crt_t` structure

*key*: holds a public key

This function will set the public parameters from the given public key to the request.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 2.12.0

## E.10 DANE API

The following functions are to be used for DANE certificate verification. Their prototypes lie in `gnutls/dane.h`. Note that you need to link with the `libgnutls-dane` library to use them.

### `dane_cert_type_name`

`const char * dane_cert_type_name (dane_cert_type_t type)` [Function]

*type*: is a DANE match type

Convert a `dane_cert_type_t` value to a string.

**Returns:** a string that contains the name of the specified type, or `NULL`.

### `dane_cert_usage_name`

`const char * dane_cert_usage_name (dane_cert_usage_t usage)` [Function]

*usage*: – undescribed –

Convert a `dane_cert_usage_t` value to a string.

**Returns:** a string that contains the name of the specified type, or `NULL`.

### `dane_match_type_name`

`const char * dane_match_type_name (dane_match_type_t type)` [Function]

*type*: is a DANE match type

Convert a `dane_match_type_t` value to a string.

**Returns:** a string that contains the name of the specified type, or `NULL`.

### `dane_query_data`

`int dane_query_data (dane_query_t q, unsigned int idx, unsigned int * usage, unsigned int * type, unsigned int * match, gnutls_datum_t * data)` [Function]

*q*: The query result structure

*idx*: The index of the query response.

*usage*: The certificate usage (see `dane_cert_usage_t`)

*type*: The certificate type (see `dane_cert_type_t`)

*match*: The DANE matching type (see `dane_match_type_t`)

*data*: The DANE data.

This function will provide the DANE data from the query response.

**Returns:** On success, `DANE_E_SUCCESS` (0) is returned, otherwise a negative error value.

**dane\_query\_deinit**

**void** dane\_query\_deinit (*dane\_query\_t* *q*) [Function]

*q*: The structure to be deinitialized

This function will deinitialize a DANE query result structure.

**dane\_query\_entries**

**unsigned int** dane\_query\_entries (*dane\_query\_t* *q*) [Function]

*q*: The query result structure

This function will return the number of entries in a query.

**Returns:** The number of entries.

**dane\_query\_status**

**dane\_query\_status\_t** dane\_query\_status (*dane\_query\_t* *q*) [Function]

*q*: The query result structure

This function will return the status of the query response. See **dane\_query\_status\_t** for the possible types.

**Returns:** The status type.

**dane\_query\_tlsa**

**int** dane\_query\_tlsa (*dane\_state\_t* *s*, *dane\_query\_t* \* *r*, *const char* \* *host*, *const char* \* *proto*, *unsigned int* *port*) [Function]

*s*: The DANE state structure

*r*: A structure to place the result

*host*: The host name to resolve.

*proto*: The protocol type (tcp, udp, etc.)

*port*: The service port number (eg. 443).

This function will query the DNS server for the TLSA (DANE) data for the given host.

**Returns:** On success, **DANE\_E\_SUCCESS** (0) is returned, otherwise a negative error value.

**dane\_query\_to\_raw\_tlsa**

**int** dane\_query\_to\_raw\_tlsa (*dane\_query\_t* *q*, *unsigned int* \* *data\_entries*, *char* \*\*\* *dane\_data*, *int* \*\* *dane\_data\_len*, *int* \* *secure*, *int* \* *bogus*) [Function]

*q*: The query result structure

*data\_entries*: Pointer set to the number of entries in the query

*dane\_data*: Pointer to contain an array of DNS rdata items, terminated with a NULL pointer; caller must guarantee that the referenced data remains valid until **dane\_query\_deinit()** is called.

*dane\_data\_len*: Pointer to contain the length n bytes of the *dane\_data* items

*secure*: Pointer set true if the result is validated securely, false if validation failed or the domain queried has no security info

*bogus*: Pointer set true if the result was not secure due to a security failure

This function will provide the DANE data from the query response.

The pointers `dane_data` and `dane_data_len` are allocated with `gnutls_malloc()` to contain the data from the query result structure (individual `dane_data` items simply point to the original data and are not allocated separately). The returned `dane_data` are only valid during the lifetime of `q`.

**Returns:** On success, `DANE_E_SUCCESS` (0) is returned, otherwise a negative error value.

## dane\_raw\_tlsa

`int dane_raw_tlsa (dane_state_t s, dane_query_t * r, char *const * dane_data, const int * dane_data_len, int secure, int bogus)` [Function]

*s*: The DANE state structure

*r*: A structure to place the result

*dane\_data*: array of DNS rdata items, terminated with a NULL pointer; caller must guarantee that the referenced data remains valid until `dane_query_deinit()` is called.

*dane\_data\_len*: the length *n* bytes of the `dane_data` items

*secure*: true if the result is validated securely, false if validation failed or the domain queried has no security info

*bogus*: if the result was not secure (`secure = 0`) due to a security failure, and the result is due to a security failure, *bogus* is true.

This function will fill in the TLSA (DANE) structure from the given raw DNS record data. The `dane_data` must be valid during the lifetime of the query.

**Returns:** On success, `DANE_E_SUCCESS` (0) is returned, otherwise a negative error value.

## dane\_state\_deinit

`void dane_state_deinit (dane_state_t s)` [Function]

*s*: The structure to be deinitialized

This function will deinitialize a DANE query structure.

## dane\_state\_init

`int dane_state_init (dane_state_t * s, unsigned int flags)` [Function]

*s*: The structure to be initialized

*flags*: flags from the `dane_state_flags` enumeration

This function will initialize a DANE query structure.

**Returns:** On success, `DANE_E_SUCCESS` (0) is returned, otherwise a negative error value.

**dane\_state\_set\_dlv\_file**

**int dane\_state\_set\_dlv\_file** (*dane\_state\_t s, const char \* file*) [Function]

*s*: The structure to be deinitialized

*file*: The file holding the DLV keys.

This function will set a file with trusted keys for DLV (DNSSEC Lookaside Validation).

**dane\_strerror**

**const char \* dane\_strerror** (*int error*) [Function]

*error*: is a DANE error code, a negative error code

This function is similar to `strerror`. The difference is that it accepts an error number returned by a `gnutls` function; In case of an unknown error a descriptive string is sent instead of `NULL`.

Error codes are always a negative error code.

**Returns:** A string explaining the DANE error message.

**dane\_verification\_status\_print**

**int dane\_verification\_status\_print** (*unsigned int status, gnutls\_datum\_t \* out, unsigned int flags*) [Function]

*status*: The status flags to be printed

*out*: Newly allocated datum with (0) terminated string.

*flags*: should be zero

This function will pretty print the status of a verification process – eg. the one obtained by `dane_verify_cert()`.

The output *out* needs to be deallocated using `gnutls_free()`.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**dane\_verify\_cert**

**int dane\_verify\_cert** (*dane\_state\_t s, const gnutls\_datum\_t \* chain, unsigned chain\_size, gnutls\_certificate\_type\_t chain\_type, const char \* hostname, const char \* proto, unsigned int port, unsigned int sflags, unsigned int vflags, unsigned int \* verify*) [Function]

*s*: A DANE state structure (may be `NULL`)

*chain*: A certificate chain

*chain\_size*: The size of the chain

*chain\_type*: The type of the certificate chain

*hostname*: The hostname associated with the chain

*proto*: The protocol of the service connecting (e.g. `tcp`)

*port*: The port of the service connecting (e.g. `443`)

*sflags*: Flags for the the initialization of *s* (if `NULL`)

*vflags*: Verification flags; an OR'ed list of `dane_verify_flags_t` .

*verify*: An OR'ed list of `dane_verify_status_t` .

This function will verify the given certificate chain against the CA constraints and/or the certificate available via DANE. If no information via DANE can be obtained the flag `DANE_VERIFY_NO_DANE_INFO` is set. If a DNSSEC signature is not available for the DANE record then the verify flag `DANE_VERIFY_NO_DNSSEC_DATA` is set.

Due to the many possible options of DANE, there is no single threat model countered. When notifying the user about DANE verification results it may be better to mention: DANE verification did not reject the certificate, rather than mentioning a successful DANE verification.

Note that this function is designed to be run in addition to PKIX - certificate chain - verification. To be run independently the `DANE_VFLAG_ONLY_CHECK_EE_USAGE` flag should be specified; then the function will check whether the key of the peer matches the key advertised in the DANE entry.

If the `q` parameter is provided it will be used for caching entries.

**Returns:** On success, `DANE_E_SUCCESS` (0) is returned, otherwise a negative error value.

## `dane_verify_cert_raw`

```
int dane_verify_cert_raw (dane_state_t s, const gnutls_datum_t *      [Function]
    chain, unsigned chain_size, gnutls_certificate_type_t chain_type,
    dane_query_t r, unsigned int sflags, unsigned int vflags, unsigned int *
    verify)
```

*s*: A DANE state structure (may be NULL)

*chain*: A certificate chain

*chain\_size*: The size of the chain

*chain\_type*: The type of the certificate chain

*r*: DANE data to check against

*sflags*: Flags for the initialization of *s* (if NULL)

*vflags*: Verification flags; an OR'ed list of `dane_verify_flags_t` .

*verify*: An OR'ed list of `dane_verify_status_t` .

This function will verify the given certificate chain against the CA constraints and/or the certificate available via DANE. If no information via DANE can be obtained the flag `DANE_VERIFY_NO_DANE_INFO` is set. If a DNSSEC signature is not available for the DANE record then the verify flag `DANE_VERIFY_NO_DNSSEC_DATA` is set.

Due to the many possible options of DANE, there is no single threat model countered. When notifying the user about DANE verification results it may be better to mention: DANE verification did not reject the certificate, rather than mentioning a successful DANE verification.

Note that this function is designed to be run in addition to PKIX - certificate chain - verification. To be run independently the `DANE_VFLAG_ONLY_CHECK_EE_USAGE` flag should be specified; then the function will check whether the key of the peer matches the key advertised in the DANE entry.



If the `q` parameter is provided it will be used for caching entries.

**Returns:** On success, `DANE_E_SUCCESS` (0) is returned, otherwise a negative error value.

## `dane_verify_session_crt`

```
int dane_verify_session_crt (dane_state_t s, gnutls_session_t session, const char * hostname, const char * proto, unsigned int port,
                             unsigned int sflags, unsigned int vflags, unsigned int * verify) [Function]
```

*s*: A DANE state structure (may be NULL)

*session*: A gnutls session

*hostname*: The hostname associated with the chain

*proto*: The protocol of the service connecting (e.g. tcp)

*port*: The port of the service connecting (e.g. 443)

*sflags*: Flags for the the initialization of *s* (if NULL)

*vflags*: Verification flags; an OR'ed list of `dane_verify_flags_t`.

*verify*: An OR'ed list of `dane_verify_status_t`.

This function will verify session's certificate chain against the CA constraints and/or the certificate available via DANE. See `dane_verify_crt()` for more information.

This will not verify the chain for validity; unless the DANE verification is restricted to end certificates, this must be performed separately using `gnutls_certificate_verify_peers3()`.

**Returns:** On success, `DANE_E_SUCCESS` (0) is returned, otherwise a negative error value.

## E.11 Cryptographic API

The following functions are to be used for low-level cryptographic operations. Their prototypes lie in `gnutls/crypto.h`.

### `gnutls_cipher_add_auth`

```
int gnutls_cipher_add_auth (gnutls_cipher_hd_t handle, const void * text, size_t text_size) [Function]
```

*handle*: is a `gnutls_cipher_hd_t` structure.

*text*: the data to be authenticated

*text\_size*: The length of the data

This function operates on authenticated encryption with associated data (AEAD) ciphers and authenticate the input data. This function can only be called once and before any encryption operations.

**Returns:** Zero or a negative error code on error.

**Since:** 3.0

**gnutls\_cipher\_decrypt**

**int gnutls\_cipher\_decrypt** (*gnutls\_cipher\_hd\_t handle*, void \* *ciphertext*, size\_t *ciphertextlen*) [Function]

*handle*: is a **gnutls\_cipher\_hd\_t** structure.

*ciphertext*: the data to encrypt

*ciphertextlen*: The length of data to encrypt

This function will decrypt the given data using the algorithm specified by the context.

Note that in AEAD ciphers, this will not check the tag. You will need to compare the tag sent with the value returned from **gnutls\_cipher\_tag()** .

**Returns:** Zero or a negative error code on error.

**Since:** 2.10.0

**gnutls\_cipher\_decrypt2**

**int gnutls\_cipher\_decrypt2** (*gnutls\_cipher\_hd\_t handle*, const void \* *ciphertext*, size\_t *ciphertextlen*, void \* *text*, size\_t *textlen*) [Function]

*handle*: is a **gnutls\_cipher\_hd\_t** structure.

*ciphertext*: the data to encrypt

*ciphertextlen*: The length of data to encrypt

*text*: the decrypted data

*textlen*: The available length for decrypted data

This function will decrypt the given data using the algorithm specified by the context.

Note that in AEAD ciphers, this will not check the tag. You will need to compare the tag sent with the value returned from **gnutls\_cipher\_tag()** .

**Returns:** Zero or a negative error code on error.

**Since:** 2.12.0

**gnutls\_cipher\_deinit**

**void gnutls\_cipher\_deinit** (*gnutls\_cipher\_hd\_t handle*) [Function]

*handle*: is a **gnutls\_cipher\_hd\_t** structure.

This function will deinitialize all resources occupied by the given encryption context.

**Since:** 2.10.0

**gnutls\_cipher\_encrypt**

**int gnutls\_cipher\_encrypt** (*gnutls\_cipher\_hd\_t handle*, void \* *text*, size\_t *textlen*) [Function]

*handle*: is a **gnutls\_cipher\_hd\_t** structure.

*text*: the data to encrypt

*textlen*: The length of data to encrypt

This function will encrypt the given data using the algorithm specified by the context.

**Returns:** Zero or a negative error code on error.

**Since:** 2.10.0

**gnutls\_cipher\_encrypt2**

**int gnutls\_cipher\_encrypt2** (*gnutls\_cipher\_hd\_t* **handle**, *const void* [Function]  
*\*text*, *size\_t* **textlen**, *void \****ciphertext**, *size\_t* **ciphertextlen**)

*handle*: is a *gnutls\_cipher\_hd\_t* structure.

*text*: the data to encrypt

*textlen*: The length of data to encrypt

*ciphertext*: the encrypted data

*ciphertextlen*: The available length for encrypted data

This function will encrypt the given data using the algorithm specified by the context.

**Returns:** Zero or a negative error code on error.

**Since:** 2.12.0

**gnutls\_cipher\_get\_block\_size**

**int gnutls\_cipher\_get\_block\_size** (*gnutls\_cipher\_algorithm\_t* [Function]  
*algorithm*)

*algorithm*: is an encryption algorithm

**Returns:** the block size of the encryption algorithm.

**Since:** 2.10.0

**gnutls\_cipher\_get\_iv\_size**

**int gnutls\_cipher\_get\_iv\_size** (*gnutls\_cipher\_algorithm\_t* [Function]  
*algorithm*)

*algorithm*: is an encryption algorithm

Get block size for encryption algorithm.

**Returns:** block size for encryption algorithm.

**Since:** 3.2.0

**gnutls\_cipher\_get\_tag\_size**

**int gnutls\_cipher\_get\_tag\_size** (*gnutls\_cipher\_algorithm\_t* [Function]  
*algorithm*)

*algorithm*: is an encryption algorithm

**Returns:** the tag size of the authenticated encryption algorithm.

**Since:** 3.2.2

**gnutls\_cipher\_init**

**int gnutls\_cipher\_init** (*gnutls\_cipher\_hd\_t \****handle**, [Function]  
*gnutls\_cipher\_algorithm\_t* **cipher**, *const gnutls\_datum\_t \****key**, *const*  
*gnutls\_datum\_t \****iv**)

*handle*: is a *gnutls\_cipher\_hd\_t* structure.

*cipher*: the encryption algorithm to use

*key*: The key to be used for encryption

*iv*: The IV to use (if not applicable set NULL)

This function will initialize an context that can be used for encryption/decryption of data. This will effectively use the current crypto backend in use by gnutls or the cryptographic accelerator in use.

**Returns:** Zero or a negative error code on error.

**Since:** 2.10.0

## gnutls\_cipher\_set\_iv

```
void gnutls_cipher_set_iv (gnutls_cipher_hd_t handle, void * iv,      [Function]
                          size_t ivlen)
```

*handle*: is a `gnutls_cipher_hd_t` structure.

*iv*: the IV to set

*ivlen*: The length of the IV

This function will set the IV to be used for the next encryption block.

**Since:** 3.0

## gnutls\_cipher\_tag

```
int gnutls_cipher_tag (gnutls_cipher_hd_t handle, void * tag, size_t   [Function]
                      tag_size)
```

*handle*: is a `gnutls_cipher_hd_t` structure.

*tag*: will hold the tag

*tag-size*: The length of the tag to return

This function operates on authenticated encryption with associated data (AEAD) ciphers and will return the output tag.

**Returns:** Zero or a negative error code on error.

**Since:** 3.0

## gnutls\_hash

```
int gnutls_hash (gnutls_hash_hd_t handle, const void * text, size_t    [Function]
                 textlen)
```

*handle*: is a `gnutls_cipher_hd_t` structure.

*text*: the data to hash

*textlen*: The length of data to hash

This function will hash the given data using the algorithm specified by the context.

**Returns:** Zero or a negative error code on error.

**Since:** 2.10.0

## gnutls\_hash\_deinit

**void** gnutls\_hash\_deinit (*gnutls\_hash\_hd\_t* *handle*, *void \***digest*) [Function]

*handle*: is a *gnutls\_hash\_hd\_t* structure.

*digest*: is the output value of the hash

This function will deinitialize all resources occupied by the given hash context.

**Since:** 2.10.0

## gnutls\_hash\_fast

**int** gnutls\_hash\_fast (*gnutls\_digest\_algorithm\_t* *algorithm*, *const void \***text*, *size\_t* *textlen*, *void \***digest*) [Function]

*algorithm*: the hash algorithm to use

*text*: the data to hash

*textlen*: The length of data to hash

*digest*: is the output value of the hash

This convenience function will hash the given data and return output on a single call.

**Returns:** Zero or a negative error code on error.

**Since:** 2.10.0

## gnutls\_hash\_get\_len

**int** gnutls\_hash\_get\_len (*gnutls\_digest\_algorithm\_t* *algorithm*) [Function]

*algorithm*: the hash algorithm to use

This function will return the length of the output data of the given hash algorithm.

**Returns:** The length or zero on error.

**Since:** 2.10.0

## gnutls\_hash\_init

**int** gnutls\_hash\_init (*gnutls\_hash\_hd\_t \***dig*, *gnutls\_digest\_algorithm\_t* *algorithm*) [Function]

*dig*: is a *gnutls\_hash\_hd\_t* structure.

*algorithm*: the hash algorithm to use

This function will initialize an context that can be used to produce a Message Digest of data. This will effectively use the current crypto backend in use by gnutls or the cryptographic accelerator in use.

**Returns:** Zero or a negative error code on error.

**Since:** 2.10.0

## gnutls\_hash\_output

**void** gnutls\_hash\_output (*gnutls\_hash\_hd\_t* *handle*, *void \***digest*) [Function]

*handle*: is a *gnutls\_hash\_hd\_t* structure.

*digest*: is the output value of the hash

This function will output the current hash value and reset the state of the hash.

**Since:** 2.10.0

## gnutls\_hmac

`int gnutls_hmac (gnutls_hmac_hd_t handle, const void * text, size_t textlen)` [Function]

*handle*: is a `gnutls_cipher_hd_t` structure.

*text*: the data to hash

*textlen*: The length of data to hash

This function will hash the given data using the algorithm specified by the context.

**Returns:** Zero or a negative error code on error.

**Since:** 2.10.0

## gnutls\_hmac\_deinit

`void gnutls_hmac_deinit (gnutls_hmac_hd_t handle, void * digest)` [Function]

*handle*: is a `gnutls_hmac_hd_t` structure.

*digest*: is the output value of the MAC

This function will deinitialize all resources occupied by the given hmac context.

**Since:** 2.10.0

## gnutls\_hmac\_fast

`int gnutls_hmac_fast (gnutls_mac_algorithm_t algorithm, const void * key, size_t keylen, const void * text, size_t textlen, void * digest)` [Function]

*algorithm*: the hash algorithm to use

*key*: the key to use

*keylen*: The length of the key

*text*: the data to hash

*textlen*: The length of data to hash

*digest*: is the output value of the hash

This convenience function will hash the given data and return output on a single call.

**Returns:** Zero or a negative error code on error.

**Since:** 2.10.0

## gnutls\_hmac\_get\_len

`int gnutls_hmac_get_len (gnutls_mac_algorithm_t algorithm)` [Function]

*algorithm*: the hmac algorithm to use

This function will return the length of the output data of the given hmac algorithm.

**Returns:** The length or zero on error.

**Since:** 2.10.0

## gnutls\_hmac\_init

`int gnutls_hmac_init (gnutls_hmac_hd_t * dig, [Function]  
                   gnutls_mac_algorithm_t algorithm, const void * key, size_t keylen)`

*dig*: is a `gnutls_hmac_hd_t` structure.

*algorithm*: the HMAC algorithm to use

*key*: The key to be used for encryption

*keylen*: The length of the key

This function will initialize an context that can be used to produce a Message Authentication Code (MAC) of data. This will effectively use the current crypto backend in use by gnutls or the cryptographic accelerator in use.

Note that despite the name of this function, it can be used for other MAC algorithms than HMAC.

**Returns:** Zero or a negative error code on error.

**Since:** 2.10.0

## gnutls\_hmac\_output

`void gnutls_hmac_output (gnutls_hmac_hd_t handle, void * digest) [Function]`

*handle*: is a `gnutls_hmac_hd_t` structure.

*digest*: is the output value of the MAC

This function will output the current MAC value and reset the state of the MAC.

**Since:** 2.10.0

## gnutls\_hmac\_set\_nonce

`void gnutls_hmac_set_nonce (gnutls_hmac_hd_t handle, const void * [Function]  
                   nonce, size_t nonce_len)`

*handle*: is a `gnutls_cipher_hd_t` structure.

*nonce*: the data to set as nonce

*nonce\_len*: The length of data

This function will set the nonce in the MAC algorithm.

**Since:** 3.2.0

## gnutls\_mac\_get\_nonce\_size

`size_t gnutls_mac_get_nonce_size (gnutls_mac_algorithm_t [Function]  
                   algorithm)`

*algorithm*: is an encryption algorithm

Returns the size of the nonce used by the MAC in TLS.

**Returns:** length (in bytes) of the given MAC nonce size, or 0.

**Since:** 3.2.0

## gnutls\_rnd

**int gnutls\_rnd** (*gnutls\_rnd\_level\_t level*, void \* *data*, *size\_t len*) [Function]

*level*: a security level

*data*: place to store random bytes

*len*: The requested size

This function will generate random data and store it to output buffer.

This function is thread-safe and also fork-safe.

**Returns:** Zero on success, or a negative error code on error.

**Since:** 2.12.0

## gnutls\_rnd\_refresh

**void gnutls\_rnd\_refresh** () [Function]

This function refreshes the random generator state. That is the current precise time, CPU usage, and other values are input into its state.

On a slower rate input from /dev/urandom is mixed too.

**Since:** 3.1.7

## E.12 Compatibility API

The following functions are carried over from old GnuTLS released. They might be removed at a later version. Their prototypes lie in `gnutls/compat.h`.

### gnutls\_certificate\_client\_set\_retrieve\_function

**void gnutls\_certificate\_client\_set\_retrieve\_function** [Function]  
(*gnutls\_certificate\_credentials\_t cred*, *gnutls\_certificate\_client\_retrieve\_function* \* *func*)

*cred*: is a `gnutls_certificate_credentials_t` structure.

*func*: is the callback function

This function sets a callback to be called in order to retrieve the certificate to be used in the handshake. You are advised to use `gnutls_certificate_set_retrieve_function2()` because it is much more efficient in the processing it requires from gnutls.

The callback's function prototype is: `int (*callback)(gnutls_session_t, const gnutls_datum_t* req_ca_dn, int nreqs, const gnutls_pk_algorithm_t* pk_algos, int pk_algos_length, gnutls_retr_st* st);`

`req_ca_cert` is only used in X.509 certificates. Contains a list with the CA names that the server considers trusted. Normally we should send a certificate that is signed by one of these CAs. These names are DER encoded. To get a more meaningful value use the function `gnutls_x509_rdn_get()`.

`pk_algos` contains a list with server's acceptable signature algorithms. The certificate returned should support the server's given algorithms.

`st` should contain the certificates and private keys.



If the callback function is provided then gnutls will call it, in the handshake, if a certificate is requested by the server (and after the certificate request message has been received).

The callback function should set the certificate list to be sent, and return 0 on success. If no certificate was selected then the number of certificates should be set to zero. The value (-1) indicates error and the handshake will be terminated.

### **gnutls\_certificate\_server\_set\_retrieve\_function**

```
void gnutls_certificate_server_set_retrieve_function      [Function]
      (gnutls_certificate_credentials_t cred, gnutls_certificate_server_retrieve_function
       * func)
```

*cred*: is a `gnutls_certificate_credentials_t` structure.

*func*: is the callback function

This function sets a callback to be called in order to retrieve the certificate to be used in the handshake. You are advised to use `gnutls_certificate_set_retrieve_function2()` because it is much more efficient in the processing it requires from gnutls.

The callback's function prototype is: `int (*callback)(gnutls_session_t, gnutls_retr_st* st);`

*st* should contain the certificates and private keys.

If the callback function is provided then gnutls will call it, in the handshake, after the certificate request message has been received.

The callback function should set the certificate list to be sent, and return 0 on success. The value (-1) indicates error and the handshake will be terminated.

### **gnutls\_certificate\_set\_rsa\_export\_params**

```
void gnutls_certificate_set_rsa_export_params            [Function]
      (gnutls_certificate_credentials_t res, gnutls_rsa_params_t rsa_params)
```

*res*: is a `gnutls_certificate_credentials_t` structure

*rsa\_params*: is a structure that holds temporary RSA parameters.

This function will set the temporary RSA parameters for a certificate server to use. These parameters will be used in RSA-EXPORT cipher suites.

### **gnutls\_certificate\_type\_set\_priority**

```
int gnutls_certificate_type_set_priority (gnutls_session_t      [Function]
      session, const int * list)
```

*session*: is a `gnutls_session_t` structure.

*list*: is a 0 terminated list of `gnutls_certificate_type_t` elements.

Sets the priority on the certificate types supported by gnutls. Priority is higher for elements specified before others. After specifying the types you want, you must append a 0. Note that the certificate type priority is set on the client. The server does not use the cert type priority except for disabling types that were not specified.

**Returns:** GNUTLS\_E\_SUCCESS on success, or an error code.

## gnutls\_cipher\_set\_priority

**int gnutls\_cipher\_set\_priority** (*gnutls\_session\_t session*, *const int \* list*) [Function]

*session*: is a `gnutls_session_t` structure.

*list*: is a 0 terminated list of `gnutls_cipher_algorithm_t` elements.

Sets the priority on the ciphers supported by gnutls. Priority is higher for elements specified before others. After specifying the ciphers you want, you must append a 0. Note that the priority is set on the client. The server does not use the algorithm's priority except for disabling algorithms that were not specified.

**Returns:** GNUTLS\_E\_SUCCESS (0) on success, or a negative error code.

## gnutls\_compression\_set\_priority

**int gnutls\_compression\_set\_priority** (*gnutls\_session\_t session*, *const int \* list*) [Function]

*session*: is a `gnutls_session_t` structure.

*list*: is a 0 terminated list of `gnutls_compression_method_t` elements.

Sets the priority on the compression algorithms supported by gnutls. Priority is higher for elements specified before others. After specifying the algorithms you want, you must append a 0. Note that the priority is set on the client. The server does not use the algorithm's priority except for disabling algorithms that were not specified.

TLS 1.0 does not define any compression algorithms except NULL. Other compression algorithms are to be considered as gnutls extensions.

**Returns:** GNUTLS\_E\_SUCCESS on success, or an error code.

## gnutls\_global\_set\_mem\_functions

**void gnutls\_global\_set\_mem\_functions** (*gnutls\_alloc\_function alloc\_func*, *gnutls\_alloc\_function secure\_alloc\_func*, *gnutls\_is\_secure\_function is\_secure\_func*, *gnutls\_realloc\_function realloc\_func*, *gnutls\_free\_function free\_func*) [Function]

*alloc\_func*: it's the default memory allocation function. Like `malloc()` .

*secure\_alloc\_func*: This is the memory allocation function that will be used for sensitive data.

*is\_secure\_func*: a function that returns 0 if the memory given is not secure. May be NULL.

*realloc\_func*: A realloc function

*free\_func*: The function that frees allocated data. Must accept a NULL pointer.

**Deprecated:** since 3.3.0 it is no longer possible to replace the internally used memory allocation functions

This is the function where you set the memory allocation functions gnutls is going to use. By default the libc's allocation functions (`malloc()` , `free()` ), are used by gnutls, to allocate both sensitive and not sensitive data. This function is provided to set the memory allocation functions to something other than the defaults

This function must be called before `gnutls_global_init()` is called. This function is not thread safe.

### **gnutls\_kx\_set\_priority**

**int gnutls\_kx\_set\_priority** (*gnutls\_session\_t session*, *const int \* list*) [Function]

*session*: is a `gnutls_session_t` structure.

*list*: is a 0 terminated list of `gnutls_kx_algorithm_t` elements.

Sets the priority on the key exchange algorithms supported by gnutls. Priority is higher for elements specified before others. After specifying the algorithms you want, you must append a 0. Note that the priority is set on the client. The server does not use the algorithm's priority except for disabling algorithms that were not specified.

**Returns:** GNUTLS\_E\_SUCCESS on success, or an error code.

### **gnutls\_mac\_set\_priority**

**int gnutls\_mac\_set\_priority** (*gnutls\_session\_t session*, *const int \* list*) [Function]

*session*: is a `gnutls_session_t` structure.

*list*: is a 0 terminated list of `gnutls_mac_algorithm_t` elements.

Sets the priority on the mac algorithms supported by gnutls. Priority is higher for elements specified before others. After specifying the algorithms you want, you must append a 0. Note that the priority is set on the client. The server does not use the algorithm's priority except for disabling algorithms that were not specified.

**Returns:** GNUTLS\_E\_SUCCESS on success, or an error code.

### **gnutls\_openpgp\_privkey\_sign\_hash**

**int gnutls\_openpgp\_privkey\_sign\_hash** (*gnutls\_openpgp\_privkey\_t key*, *const gnutls\_datum\_t \* hash*, *gnutls\_datum\_t \* signature*) [Function]

*key*: Holds the key

*hash*: holds the data to be signed

*signature*: will contain newly allocated signature

This function will sign the given hash using the private key. You should use `gnutls_openpgp_privkey_set_preferred_key_id()` before calling this function to set the subkey to use.

**Returns:** On success, GNUTLS\_E\_SUCCESS (0) is returned, otherwise a negative error value.

**Deprecated:** Use `gnutls_privkey_sign_hash()` instead.

### **gnutls\_privkey\_sign\_raw\_data**

**int gnutls\_privkey\_sign\_raw\_data** (*gnutls\_privkey\_t key*, *unsigned flags*, *const gnutls\_datum\_t \* data*, *gnutls\_datum\_t \* signature*) [Function]

*key*: Holds the key

*flags*: should be zero

*data*: holds the data to be signed

*signature*: will contain the signature allocate with `gnutls_malloc()`

This function will sign the given data using a signature algorithm supported by the private key. Note that this is a low-level function and does not apply any preprocessing or hash on the signed data. For example on an RSA key the input `data` should be of the DigestInfo PKCS 1 1.5 format. Use it only if you know what are you doing.

Note this function is equivalent to using the `GNUTLS_PRIVKEY_SIGN_FLAG_TLS1_RSA` flag with `gnutls_privkey_sign_hash()` .

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Since:** 3.1.10

## `gnutls_protocol_set_priority`

```
int gnutls_protocol_set_priority (gnutls_session_t session, const [Function]
                                int *list)
```

*session*: is a `gnutls_session_t` structure.

*list*: is a 0 terminated list of `gnutls_protocol_t` elements.

Sets the priority on the protocol versions supported by gnutls. This function actually enables or disables protocols. Newer protocol versions always have highest priority.

**Returns:** `GNUTLS_E_SUCCESS` on success, or an error code.

## `gnutls_rsa_export_get_modulus_bits`

```
int gnutls_rsa_export_get_modulus_bits (gnutls_session_t [Function]
                                         session)
```

*session*: is a gnutls session

Get the export RSA parameter's modulus size.

**Returns:** The bits used in the last RSA-EXPORT key exchange with the peer, or a negative error code in case of error.

## `gnutls_rsa_export_get_pubkey`

```
int gnutls_rsa_export_get_pubkey (gnutls_session_t session, [Function]
                                  gnutls_datum_t *exponent, gnutls_datum_t *modulus)
```

*session*: is a gnutls session

*exponent*: will hold the exponent.

*modulus*: will hold the modulus.

This function will return the peer's public key exponent and modulus used in the last RSA-EXPORT authentication. The output parameters must be freed with `gnutls_free()` .

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise an error code is returned.

**gnutls\_rsa\_params\_cpy**

**int gnutls\_rsa\_params\_cpy** (*gnutls\_rsa\_params\_t dst*, [Function]  
*gnutls\_rsa\_params\_t src*)

*dst*: Is the destination structure, which should be initialized.

*src*: Is the source structure

This function will copy the RSA parameters structure from source to destination.

**Returns:** GNUTLS\_E\_SUCCESS on success, or an negative error code.

**gnutls\_rsa\_params\_deinit**

**void gnutls\_rsa\_params\_deinit** (*gnutls\_rsa\_params\_t rsa\_params*) [Function]

*rsa\_params*: Is a structure that holds the parameters

This function will deinitialize the RSA parameters structure.

**gnutls\_rsa\_params\_export\_pkcs1**

**int gnutls\_rsa\_params\_export\_pkcs1** (*gnutls\_rsa\_params\_t params*, [Function]  
*gnutls\_x509\_crt\_fmt\_t format*, unsigned char \* *params\_data*, size\_t \*  
*params\_data\_size*)

*params*: Holds the RSA parameters

*format*: the format of output params. One of PEM or DER.

*params\_data*: will contain a PKCS1 RSAPrivateKey structure PEM or DER encoded

*params\_data\_size*: holds the size of *params\_data* (and will be replaced by the actual size of parameters)

This function will export the given RSA parameters to a PKCS1 RSAPrivateKey structure. If the buffer provided is not long enough to hold the output, then GNUTLS\_E\_SHORT\_MEMORY\_BUFFER will be returned.

If the structure is PEM encoded, it will have a header of "BEGIN RSA PRIVATE KEY".

**Returns:** GNUTLS\_E\_SUCCESS on success, or an negative error code.

**gnutls\_rsa\_params\_export\_raw**

**int gnutls\_rsa\_params\_export\_raw** (*gnutls\_rsa\_params\_t rsa*, [Function]  
*gnutls\_datum\_t \* m*, *gnutls\_datum\_t \* e*, *gnutls\_datum\_t \* d*, *gnutls\_datum\_t \* p*,  
*gnutls\_datum\_t \* q*, *gnutls\_datum\_t \* u*, unsigned int \* *bits*)

*rsa*: a structure that holds the rsa parameters

*m*: will hold the modulus

*e*: will hold the public exponent

*d*: will hold the private exponent

*p*: will hold the first prime (p)

*q*: will hold the second prime (q)

*u*: will hold the coefficient

*bits*: if non null will hold the prime's number of bits

This function will export the RSA parameters found in the given structure. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

**Returns:** GNUTLS\_E\_SUCCESS on success, or an negative error code.

## gnutls\_rsa\_params\_generate2

`int gnutls_rsa_params_generate2 (gnutls_rsa_params_t params, [Function]  
                                   unsigned int bits)`

*params*: The structure where the parameters will be stored

*bits*: is the prime's number of bits

This function will generate new temporary RSA parameters for use in RSA-EXPORT ciphersuites. This function is normally slow.

Note that if the parameters are to be used in export cipher suites the bits value should be 512 or less. Also note that the generation of new RSA parameters is only useful to servers. Clients use the parameters sent by the server, thus it's no use calling this in client side.

**Returns:** GNUTLS\_E\_SUCCESS on success, or an negative error code.

## gnutls\_rsa\_params\_import\_pkcs1

`int gnutls_rsa_params_import_pkcs1 (gnutls_rsa_params_t params, [Function]  
                                   const gnutls_datum_t * pkcs1_params, gnutls_x509_crt_fmt_t format)`

*params*: A structure where the parameters will be copied to

*pkcs1\_params*: should contain a PKCS1 RSAPrivateKey structure PEM or DER encoded

*format*: the format of params. PEM or DER.

This function will extract the RSAPrivateKey found in a PKCS1 formatted structure. If the structure is PEM encoded, it should have a header of "BEGIN RSA PRIVATE KEY".

**Returns:** GNUTLS\_E\_SUCCESS on success, or an negative error code.

## gnutls\_rsa\_params\_import\_raw

`int gnutls_rsa_params_import_raw (gnutls_rsa_params_t [Function]  
                                   rsa_params, const gnutls_datum_t * m, const gnutls_datum_t * e, const  
                                   gnutls_datum_t * d, const gnutls_datum_t * p, const gnutls_datum_t * q, const  
                                   gnutls_datum_t * u)`

*rsa\_params*: Is a structure will hold the parameters

*m*: holds the modulus

*e*: holds the public exponent

*d*: holds the private exponent

*p*: holds the first prime (p)

*q*: holds the second prime (q)

*u*: holds the coefficient

This function will replace the parameters in the given structure. The new parameters should be stored in the appropriate `gnutls_datum`.

**Returns:** GNUTLS\_E\_SUCCESS on success, or an negative error code.

### **gnutls\_rsa\_params\_init**

**int gnutls\_rsa\_params\_init** (*gnutls\_rsa\_params\_t* \* *rsa\_params*) [Function]  
*rsa\_params*: Is a structure that will hold the parameters

This function will initialize the temporary RSA parameters structure.

**Returns:** GNUTLS\_E\_SUCCESS on success, or an negative error code.

### **gnutls\_set\_default\_export\_priority**

**int gnutls\_set\_default\_export\_priority** (*gnutls\_session\_t* *session*) [Function]

*session*: is a `gnutls_session_t` structure.

Sets some default priority on the ciphers, key exchange methods, macs and compression methods. This function also includes weak algorithms.

This is the same as calling:

```
gnutls_priority_set_direct (session, "EXPORT", NULL);
```

This function is kept around for backwards compatibility, but because of its wide use it is still fully supported. If you wish to allow users to provide a string that specify which ciphers to use (which is recommended), you should use `gnutls_priority_set_direct()` or `gnutls_priority_set()` instead.

**Returns:** GNUTLS\_E\_SUCCESS on success, or an error code.

### **gnutls\_sign\_callback\_get**

**gnutls\_sign\_func gnutls\_sign\_callback\_get** (*gnutls\_session\_t* *session*, void \*\* *userdata*) [Function]

*session*: is a gnutls session

*userdata*: if non-NULL , will be set to abstract callback pointer.

Retrieve the callback function, and its userdata pointer.

**Returns:** The function pointer set by `gnutls_sign_callback_set()` , or if not set, NULL .

**Deprecated:** Use the PKCS 11 interfaces instead.

### **gnutls\_sign\_callback\_set**

**void gnutls\_sign\_callback\_set** (*gnutls\_session\_t* *session*, *gnutls\_sign\_func* *sign\_func*, void \* *userdata*) [Function]

*session*: is a gnutls session

*sign\_func*: function pointer to application's sign callback.

*userdata*: void pointer that will be passed to sign callback.

Set the callback function. The function must have this prototype:

```
typedef int (*gnutls_sign_func) (gnutls_session_t session, void *userdata,
gnutls_certificate_type_t cert_type, const gnutls_datum_t * cert, const
gnutls_datum_t * hash, gnutls_datum_t * signature);
```

The `userdata` parameter is passed to the `sign_func` verbatim, and can be used to store application-specific data needed in the callback function. See also `gnutls_sign_callback_get()` .

**Deprecated:** Use the PKCS 11 or `gnutls_privkey_t` interfaces like `gnutls_privkey_import_ext()` instead.

## gnutls\_x509\_crl\_sign

```
int gnutls_x509_crl_sign (gnutls_x509_crl_t crl, gnutls_x509_cert_t issuer, gnutls_privkey_t issuer_key) [Function]
```

`crl`: should contain a `gnutls_x509_crl_t` structure

`issuer`: is the certificate of the certificate issuer

`issuer_key`: holds the issuer's private key

This function is the same as `gnutls_x509_crl_sign2()` with no flags, and SHA1 as the hash algorithm.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Deprecated:** Use `gnutls_x509_crl_privkey_sign()` .

## gnutls\_x509\_crq\_sign

```
int gnutls_x509_crq_sign (gnutls_x509_crq_t crq, gnutls_privkey_t key) [Function]
```

`crq`: should contain a `gnutls_x509_crq_t` structure

`key`: holds a private key

This function is the same as `gnutls_x509_crq_sign2()` with no flags, and SHA1 as the hash algorithm.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Deprecated:** Use `gnutls_x509_crq_privkey_sign()` instead.

## gnutls\_x509\_cert\_get\_preferred\_hash\_algorithm

```
int gnutls_x509_cert_get_preferred_hash_algorithm (gnutls_x509_cert_t crt, gnutls_digest_algorithm_t * hash, unsigned int * mand) [Function]
```

`crt`: Holds the certificate

`hash`: The result of the call with the hash algorithm used for signature

`mand`: If non-zero it means that the algorithm MUST use this hash. May be NULL.

This function will read the certificate and return the appropriate digest algorithm to use for signing with this certificate. Some certificates (i.e. DSA might not be able to sign without the preferred algorithm).

**Deprecated:** Please use `gnutls_pubkey_get_preferred_hash_algorithm()` .



**Returns:** the 0 if the hash algorithm is found. A negative error code is returned on error.

**Since:** 2.12.0

## gnutls\_x509\_cert\_get\_verify\_algorithm

```
int gnutls_x509_cert_get_verify_algorithm (gnutls_x509_cert_t [Function]
                                           crt, const gnutls_datum_t * signature, gnutls_digest_algorithm_t * hash)
```

*crt*: Holds the certificate

*signature*: contains the signature

*hash*: The result of the call with the hash algorithm used for signature

This function will read the certificate and the signed data to determine the hash algorithm used to generate the signature.

**Deprecated:** Use `gnutls_pubkey_get_verify_algorithm()` instead.

**Returns:** the 0 if the hash algorithm is found. A negative error code is returned on error.

**Since:** 2.8.0

## gnutls\_x509\_cert\_verify\_data

```
int gnutls_x509_cert_verify_data (gnutls_x509_cert_t crt, unsigned [Function]
                                   int flags, const gnutls_datum_t * data, const gnutls_datum_t * signature)
```

*crt*: Holds the certificate

*flags*: should be 0 for now

*data*: holds the data to be signed

*signature*: contains the signature

This function will verify the given signed data, using the parameters from the certificate.

Deprecated. This function cannot be easily used securely. Use `gnutls_pubkey_verify_data2()` instead.

**Returns:** In case of a verification failure `GNUTLS_E_PK_SIG_VERIFY_FAILED` is returned, and zero or positive code on success.

## gnutls\_x509\_cert\_verify\_hash

```
int gnutls_x509_cert_verify_hash (gnutls_x509_cert_t crt, unsigned [Function]
                                   int flags, const gnutls_datum_t * hash, const gnutls_datum_t * signature)
```

*crt*: Holds the certificate

*flags*: should be 0 for now

*hash*: holds the hash digest to be verified

*signature*: contains the signature

This function will verify the given signed digest, using the parameters from the certificate.

Deprecated. This function cannot be easily used securely. Use `gnutls_pubkey_verify_hash2()` instead.

**Returns:** In case of a verification failure `GNUTLS_E_PK_SIG_VERIFY_FAILED` is returned, and zero or positive code on success.

## **gnutls\_x509\_privkey\_sign\_data**

```
int gnutls_x509_privkey_sign_data (gnutls_x509_privkey_t key,          [Function]
                                   gnutls_digest_algorithm_t digest, unsigned int flags, const gnutls_datum_t *
                                   data, void * signature, size_t * signature_size)
```

*key*: Holds the key

*digest*: should be MD5 or SHA1

*flags*: should be 0 for now

*data*: holds the data to be signed

*signature*: will contain the signature

*signature\_size*: holds the size of signature (and will be replaced by the new size)

This function will sign the given data using a signature algorithm supported by the private key. Signature algorithms are always used together with a hash functions. Different hash functions may be used for the RSA algorithm, but only SHA-1 for the DSA keys.

If the buffer provided is not long enough to hold the output, then `* signature_size` is updated and `GNUTLS_E_SHORT_MEMORY_BUFFER` will be returned.

Use `gnutls_x509_crt_get_preferred_hash_algorithm()` to determine the hash algorithm.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

**Deprecated:** Use `gnutls_privkey_sign_data()` .

## **gnutls\_x509\_privkey\_sign\_hash**

```
int gnutls_x509_privkey_sign_hash (gnutls_x509_privkey_t key,          [Function]
                                   const gnutls_datum_t * hash, gnutls_datum_t * signature)
```

*key*: Holds the key

*hash*: holds the data to be signed

*signature*: will contain newly allocated signature

This function will sign the given hash using the private key. Do not use this function directly unless you know what it is. Typical signing requires the data to be hashed and stored in special formats (e.g. BER Digest-Info for RSA).

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, otherwise a negative error value.

Deprecated in: 2.12.0

## Appendix F Copying Information

### GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.  
<http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

#### 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

#### 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

The “publisher” means any person or entity that distributes copies of the Document to the public.

A section “Entitled XYZ” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “Acknowledgements”, “Dedications”, “Endorsements”, or “History”.) To “Preserve the Title” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

## 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

## 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

## 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at

your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

## 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

## 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

## 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

## 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

## 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.



## 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy’s public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

## 11. RELICENSING

“Massive Multiauthor Collaboration Site” (or “MMC Site”) means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A “Massive Multiauthor Collaboration” (or “MMC”) contained in the site means any set of copyrightable works thus published on the MMC site.

“CC-BY-SA” means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

“Incorporate” means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is “eligible for relicensing” if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

## ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (C) *year* *your name*.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled ‘‘GNU Free Documentation License’’.

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the ‘‘with...Texts.’’ line with this:

with the Invariant Sections being *list their titles*, with the Front-Cover Texts being *list*, and with the Back-Cover Texts being *list*.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

## Bibliography

- [CBCATT] Bodo Moeller, "Security of CBC Ciphersuites in SSL/TLS: Problems and Countermeasures", 2002, available from <http://www.openssl.org/~bodo/tls-cbc.txt>.
- [GPGH] Mike Ashley, "The GNU Privacy Handbook", 2002, available from <http://www.gnupg.org/gph/en/manual.pdf>.
- [GUTPKI] Peter Gutmann, "Everything you never wanted to know about PKI but were forced to find out", Available from <http://www.cs.auckland.ac.nz/~pgut001/>.
- [KEYPIN] Chris Evans and Chris Palmer, "Public Key Pinning Extension for HTTP", Available from <http://tools.ietf.org/html/draft-ietf-websec-key-pinning-01>.
- [NISTSP80057] NIST Special Publication 800-57, "Recommendation for Key Management - Part 1: General (Revised)", March 2007, available from [http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57-Part1-revised2\\_Mar08-2007.pdf](http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57-Part1-revised2_Mar08-2007.pdf).
- [RFC2246] Tim Dierks and Christopher Allen, "The TLS Protocol Version 1.0", January 1999, Available from <http://www.ietf.org/rfc/rfc2246.txt>.
- [RFC4418] Ted Krovetz, "UMAC: Message Authentication Code using Universal Hashing", March 2006, Available from <http://www.ietf.org/rfc/rfc4418.txt>.
- [RFC4680] S. Santesson, "TLS Handshake Message for Supplemental Data", September 2006, Available from <http://www.ietf.org/rfc/rfc4680.txt>.
- [RFC4514] Kurt D. Zeilenga, "Lightweight Directory Access Protocol (LDAP): String Representation of Distinguished Names", June 2006, Available from <http://www.ietf.org/rfc/rfc4513.txt>.
- [RFC4346] Tim Dierks and Eric Rescorla, "The TLS Protocol Version 1.1", March 2006, Available from <http://www.ietf.org/rfc/rfc4346.txt>.
- [RFC4347] Eric Rescorla and Nagendra Modadugu, "Datagram Transport Layer Security", April 2006, Available from <http://www.ietf.org/rfc/rfc4347.txt>.
- [RFC5246] Tim Dierks and Eric Rescorla, "The TLS Protocol Version 1.2", August 2008, Available from <http://www.ietf.org/rfc/rfc5246.txt>.

- [RFC2440] Jon Callas, Lutz Donnerhacke, Hal Finney and Rodney Thayer, "OpenPGP Message Format", November 1998, Available from <http://www.ietf.org/rfc/rfc2440.txt>.
- [RFC4880] Jon Callas, Lutz Donnerhacke, Hal Finney, David Shaw and Rodney Thayer, "OpenPGP Message Format", November 2007, Available from <http://www.ietf.org/rfc/rfc4880.txt>.
- [RFC4211] J. Schaad, "Internet X.509 Public Key Infrastructure Certificate Request Message Format (CRMF)", September 2005, Available from <http://www.ietf.org/rfc/rfc4211.txt>.
- [RFC2817] Rohit Khare and Scott Lawrence, "Upgrading to TLS Within HTTP/1.1", May 2000, Available from <http://www.ietf.org/rfc/rfc2817.txt>
- [RFC2818] Eric Rescorla, "HTTP Over TLS", May 2000, Available from <http://www.ietf.org/rfc/rfc2818.txt>.
- [RFC2945] Tom Wu, "The SRP Authentication and Key Exchange System", September 2000, Available from <http://www.ietf.org/rfc/rfc2945.txt>.
- [RFC2986] Magnus Nystrom and Burt Kaliski, "PKCS 10 v1.7: Certification Request Syntax Specification", November 2000, Available from <http://www.ietf.org/rfc/rfc2986.txt>.
- [PKIX] D. Cooper, S. Santesson, S. Farrel, S. Boeyen, R. Housley, W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", May 2008, available from <http://www.ietf.org/rfc/rfc5280.txt>.
- [RFC3749] Scott Hollenbeck, "Transport Layer Security Protocol Compression Methods", May 2004, available from <http://www.ietf.org/rfc/rfc3749.txt>.
- [RFC3820] Steven Tuecke, Von Welch, Doug Engert, Laura Pearlman, and Mary Thompson, "Internet X.509 Public Key Infrastructure (PKI) Proxy Certificate Profile", June 2004, available from <http://www.ietf.org/rfc/rfc3820>.
- [RFC6520] R. Seggelmann, M. Tuexen, and M. Williams, "Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension", February 2012, available from <http://www.ietf.org/rfc/rfc6520>.

- [RFC5746] E. Rescorla, M. Ray, S. Dispensa, and N. Oskov, "Transport Layer Security (TLS) Renegotiation Indication Extension", February 2010, available from <http://www.ietf.org/rfc/rfc5746>.
- [RFC5280] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", May 2008, available from <http://www.ietf.org/rfc/rfc5280>.
- [TLSTKT] Joseph Salowey, Hao Zhou, Pasi Eronen, Hannes Tschofenig, "Transport Layer Security (TLS) Session Resumption without Server-Side State", January 2008, available from <http://www.ietf.org/rfc/rfc5077>.
- [PKCS12] RSA Laboratories, "PKCS 12 v1.0: Personal Information Exchange Syntax", June 1999, Available from <http://www.rsa.com>.
- [PKCS11] RSA Laboratories, "PKCS #11 Base Functionality v2.30: Cryptoki Draft 4", July 2009, Available from <http://www.rsa.com>.
- [RESCORLA] Eric Rescorla, "SSL and TLS: Designing and Building Secure Systems", 2001
- [SELKEY] Arjen Lenstra and Eric Verheul, "Selecting Cryptographic Key Sizes", 2003, available from <http://www.win.tue.nl/~klenstra/key.pdf>.
- [SSL3] Alan Freier, Philip Karlton and Paul Kocher, "The Secure Sockets Layer (SSL) Protocol Version 3.0", August 2011, Available from <http://www.ietf.org/rfc/rfc6101.txt>.
- [STEVENS] Richard Stevens, "UNIX Network Programming, Volume 1", Prentice Hall PTR, January 1998
- [TLSEXT] Simon Blake-Wilson, Magnus Nystrom, David Hopwood, Jan Mikkelsen and Tim Wright, "Transport Layer Security (TLS) Extensions", June 2003, Available from <http://www.ietf.org/rfc/rfc3546.txt>.
- [TLSPGP] Nikos Mavrogiannopoulos, "Using OpenPGP keys for TLS authentication", January 2011. Available from <http://www.ietf.org/rfc/rfc6091.txt>.
- [TLSSRP] David Taylor, Trevor Perrin, Tom Wu and Nikos Mavrogiannopoulos, "Using SRP for TLS Authentication", November 2007. Available from <http://www.ietf.org/rfc/rfc5054.txt>.
- [TLSPSK] Pasi Eronen and Hannes Tschofenig, "Pre-shared key Ciphersuites for TLS", December 2005, Available from <http://www.ietf.org/rfc/rfc4279.txt>.
- [TOMSRP] Tom Wu, "The Stanford SRP Authentication Project", Available at <http://srp.stanford.edu/>.

- [WEGER] Arjen Lenstra and Xiaoyun Wang and Benne de Weger, "Colliding X.509 Certificates", Cryptology ePrint Archive, Report 2005/067, Available at <http://eprint.iacr.org/>.
- [ECRYPT] European Network of Excellence in Cryptology II, "ECRYPT II Yearly Report on Algorithms and Keysizes (2009-2010)", Available at <http://www.ecrypt.eu.org/documents/D.SPA.13.pdf>.
- [RFC5056] N. Williams, "On the Use of Channel Bindings to Secure Channels", November 2007, available from <http://www.ietf.org/rfc/rfc5056>.
- [RFC5929] J. Altman, N. Williams, L. Zhu, "Channel Bindings for TLS", July 2010, available from <http://www.ietf.org/rfc/rfc5929>.
- [PKCS11URI] J. Pechanec, D. Moffat, "The PKCS#11 URI Scheme", September 2013, Work in progress, available from <http://tools.ietf.org/html/draft-pechanec-pkcs11uri-13>.
- [TPMURI] C. Latze, N. Mavrogiannopoulos, "The TPMKEY URI Scheme", January 2013, Work in progress, available from <http://tools.ietf.org/html/draft-mavrogiannopoulos-tpmuri-01>.
- [ANDERSON] R. J. Anderson, "Security Engineering: A Guide to Building Dependable Distributed Systems", John Wiley & Sons, Inc., 2001.
- [RFC4821] M. Mathis, J. Heffner, "Packetization Layer Path MTU Discovery", March 2007, available from <http://www.ietf.org/rfc/rfc4821.txt>.
- [RFC2560] M. Myers et al, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP", June 1999, Available from <http://www.ietf.org/rfc/rfc2560.txt>.
- [RIVESTCRL] R. L. Rivest, "Can We Eliminate Certificate Revocation Lists?", Proceedings of Financial Cryptography '98; Springer Lecture Notes in Computer Science No. 1465 (Rafael Hirschfeld, ed.), February 1998), pages 178–183, available from <http://people.csail.mit.edu/rivest/Rivest-CanWeEliminateCertificateRevocationLists.pdf>.

# Function and Data Index

## D

dane_cert_type_name .....	548
dane_cert_usage_name .....	548
dane_match_type_name .....	548
dane_query_data .....	548
dane_query_deinit .....	549
dane_query_entries .....	549
dane_query_status .....	549
dane_query_tlsa .....	549
dane_query_to_raw_tlsa .....	549
dane_raw_tlsa .....	550
dane_state_deinit .....	550
dane_state_init .....	550
dane_state_set_dlv_file .....	551
dane_strerror .....	551
dane_verification_status_print .....	551
dane_verify_cert .....	144, 551
dane_verify_cert_raw .....	552
dane_verify_session_cert .....	553

## G

gnutls_alert_get .....	131, 273
gnutls_alert_get_name .....	131, 273
gnutls_alert_get_strname .....	273
gnutls_alert_send .....	131, 273
gnutls_alert_send_appropriate .....	274
gnutls_alpn_get_selected_protocol .....	274
gnutls_alpn_set_protocols .....	274
gnutls_anon_allocate_client_credentials .....	275
gnutls_anon_allocate_server_credentials .....	275
gnutls_anon_free_client_credentials .....	275
gnutls_anon_free_server_credentials .....	275
gnutls_anon_set_params_function .....	275
gnutls_anon_set_server_dh_params .....	276
gnutls_anon_set_server_params_function ..	276
gnutls_auth_client_get_type .....	276
gnutls_auth_get_type .....	276
gnutls_auth_server_get_type .....	277
gnutls_bye .....	130, 277
gnutls_certificate_activation_time_peers .....	277
gnutls_certificate_allocate_credentials .....	278
gnutls_certificate_client_get_request_status .....	278
gnutls_certificate_client_set_retrieve_function .....	560
gnutls_certificate_expiration_time_peers .....	278
gnutls_certificate_free_ca_names .....	278
gnutls_certificate_free_cas .....	278

gnutls_certificate_free_credentials .....	279
gnutls_certificate_free_crls .....	279
gnutls_certificate_free_keys .....	279
gnutls_certificate_get_cert_raw .....	279
gnutls_certificate_get_issuer .....	280
gnutls_certificate_get_ours .....	280
gnutls_certificate_get_peers .....	280
gnutls_certificate_get_peers_subkey_id ..	280
gnutls_certificate_send_x509_rdn_sequence .....	117, 281
gnutls_certificate_server_set_request ...	117, 281
gnutls_certificate_server_set_retrieve_function .....	561
gnutls_certificate_set_dh_params .....	281
gnutls_certificate_set_key .....	115, 520
gnutls_certificate_set_ocsp_status_request_file .....	281
gnutls_certificate_set_ocsp_status_request_function .....	282
gnutls_certificate_set_openpgp_key .....	477
gnutls_certificate_set_openpgp_key_file .....	477
gnutls_certificate_set_openpgp_key_file2 .....	478
gnutls_certificate_set_openpgp_key_mem ..	478
gnutls_certificate_set_openpgp_key_mem2 .....	479
gnutls_certificate_set_openpgp_keyring_file .....	40, 479
gnutls_certificate_set_openpgp_keyring_mem .....	479
gnutls_certificate_set_params_function .....	146, 282
gnutls_certificate_set_pin_function .....	114, 283
gnutls_certificate_set_retrieve_function .....	283
gnutls_certificate_set_retrieve_function2 .....	521
gnutls_certificate_set_rsa_export_params .....	561
gnutls_certificate_set_trust_list .....	365
gnutls_certificate_set_verify_flags .....	284
gnutls_certificate_set_verify_function .....	118, 284
gnutls_certificate_set_verify_limits .....	284
gnutls_certificate_set_x509_crl .....	284
gnutls_certificate_set_x509_crl_file .....	285
gnutls_certificate_set_x509_crl_mem .....	285
gnutls_certificate_set_x509_key .....	285
gnutls_certificate_set_x509_key_file .....	286
gnutls_certificate_set_x509_key_file2 ...	286
gnutls_certificate_set_x509_key_mem .....	287
gnutls_certificate_set_x509_key_mem2 .....	288

gnutls_certificate_set_x509_simple_pkcs12_		
file.....	288	
gnutls_certificate_set_x509_simple_pkcs12_		
mem.....	289	
gnutls_certificate_set_x509_system_trust		
.....	95, 289	
gnutls_certificate_set_x509_trust.....	290	
gnutls_certificate_set_x509_trust_dir...	290	
gnutls_certificate_set_x509_trust_file..	290	
gnutls_certificate_set_x509_trust_mem...	291	
gnutls_certificate_type_get.....	291	
gnutls_certificate_type_get_id.....	291	
gnutls_certificate_type_get_name.....	291	
gnutls_certificate_type_list.....	292	
gnutls_certificate_type_set_priority....	561	
gnutls_certificate_verification_status_		
print.....	292	
gnutls_certificate_verify_flags.....	34, 142	
gnutls_certificate_verify_peers.....	292	
gnutls_certificate_verify_peers2.....	293	
gnutls_certificate_verify_peers3....	118, 293	
gnutls_check_version.....	294	
gnutls_cipher_add_auth.....	553	
gnutls_cipher_decrypt.....	554	
gnutls_cipher_decrypt2.....	554	
gnutls_cipher_deinit.....	554	
gnutls_cipher_encrypt.....	554	
gnutls_cipher_encrypt2.....	555	
gnutls_cipher_get.....	294	
gnutls_cipher_get_block_size.....	555	
gnutls_cipher_get_id.....	294	
gnutls_cipher_get_iv_size.....	555	
gnutls_cipher_get_key_size.....	294	
gnutls_cipher_get_name.....	295	
gnutls_cipher_get_tag_size.....	555	
gnutls_cipher_init.....	555	
gnutls_cipher_list.....	295	
gnutls_cipher_set_iv.....	556	
gnutls_cipher_set_priority.....	562	
gnutls_cipher_suite_get_name.....	295	
gnutls_cipher_suite_info.....	295	
gnutls_cipher_tag.....	556	
gnutls_compression_get.....	296	
gnutls_compression_get_id.....	296	
gnutls_compression_get_name.....	296	
gnutls_compression_list.....	296	
gnutls_compression_set_priority.....	562	
gnutls_credentials_clear.....	296	
gnutls_credentials_get.....	297	
gnutls_credentials_set.....	112, 297	
gnutls_db_check_entry.....	297	
gnutls_db_check_entry_time.....	298	
gnutls_db_get_default_cache_expiration..	298	
gnutls_db_get_ptr.....	298	
gnutls_db_remove_session.....	298	
gnutls_db_set_cache_expiration.....	298	
gnutls_db_set_ptr.....	299	
gnutls_db_set_remove_function.....	299	
gnutls_db_set_retrieve_function.....	299	
gnutls_db_set_store_function.....	299	
gnutls_deinit.....	130, 300	
gnutls_dh_get_group.....	300	
gnutls_dh_get_peers_public_bits.....	300	
gnutls_dh_get_prime_bits.....	300	
gnutls_dh_get_pubkey.....	301	
gnutls_dh_get_secret_bits.....	301	
gnutls_dh_params_cpy.....	301	
gnutls_dh_params_deinit.....	301	
gnutls_dh_params_export_pkcs3.....	302	
gnutls_dh_params_export_raw.....	302	
gnutls_dh_params_export2_pkcs3.....	301	
gnutls_dh_params_generate2.....	303	
gnutls_dh_params_import_pkcs3.....	303	
gnutls_dh_params_import_raw.....	303	
gnutls_dh_params_init.....	304	
gnutls_dh_set_prime_bits.....	304	
gnutls_digest_get_id.....	304	
gnutls_digest_get_name.....	304	
gnutls_digest_list.....	305	
gnutls_dtls_cookie_send.....	362	
gnutls_dtls_cookie_verify.....	362	
gnutls_dtls_get_data_mtu.....	363	
gnutls_dtls_get_mtu.....	363	
gnutls_dtls_get_timeout.....	124, 363	
gnutls_dtls_prestate_set.....	363	
gnutls_dtls_set_data_mtu.....	364	
gnutls_dtls_set_mtu.....	364	
gnutls_dtls_set_timeouts.....	364	
gnutls_ecc_curve_get.....	305	
gnutls_ecc_curve_get_name.....	305	
gnutls_ecc_curve_get_size.....	305	
gnutls_ecc_curve_list.....	305	
gnutls_error_is_fatal.....	128, 306	
gnutls_error_to_alert.....	132, 306	
gnutls_est_record_overhead_size.....	306	
gnutls_fingerprint.....	307	
gnutls_fips140_mode_enabled.....	307	
gnutls_global_deinit.....	307	
gnutls_global_init.....	307	
gnutls_global_set_audit_log_function....	108, 308	
gnutls_global_set_log_function.....	308	
gnutls_global_set_log_level.....	308	
gnutls_global_set_mem_functions.....	562	
gnutls_global_set_mutex.....	109, 308	
gnutls_global_set_time_function.....	309	
gnutls_handshake.....	126, 309	
gnutls_handshake_description_get_name...	309	
gnutls_handshake_get_last_in.....	310	
gnutls_handshake_get_last_out.....	310	
gnutls_handshake_set_hook_function.....	310	
gnutls_handshake_set_max_packet_length..	311	
gnutls_handshake_set_post_client_hello_		
function.....	311	
gnutls_handshake_set_private_extensions		
.....	311	



gnutls_handshake_set_random.....	312	gnutls_ocsp_resp_get_nonce.....	473
gnutls_handshake_set_timeout.....	127, 312	gnutls_ocsp_resp_get_produced.....	473
gnutls_hash.....	556	gnutls_ocsp_resp_get_responder.....	473
gnutls_hash_deinit.....	557	gnutls_ocsp_resp_get_response.....	473
gnutls_hash_fast.....	557	gnutls_ocsp_resp_get_signature.....	474
gnutls_hash_get_len.....	557	gnutls_ocsp_resp_get_signature_algorithm.....	474
gnutls_hash_init.....	557	gnutls_ocsp_resp_get_single.....	51, 474
gnutls_hash_output.....	557	gnutls_ocsp_resp_get_status.....	475
gnutls_heartbeat_allowed.....	312	gnutls_ocsp_resp_get_version.....	475
gnutls_heartbeat_enable.....	312	gnutls_ocsp_resp_import.....	475
gnutls_heartbeat_get_timeout.....	313	gnutls_ocsp_resp_init.....	475
gnutls_heartbeat_ping.....	313	gnutls_ocsp_resp_print.....	476
gnutls_heartbeat_pong.....	313	gnutls_ocsp_resp_verify.....	476
gnutls_heartbeat_set_timeouts.....	314	gnutls_ocsp_resp_verify_direct.....	477
gnutls_hex_decode.....	314	gnutls_ocsp_status_request_enable_client.....	317
gnutls_hex_encode.....	315	gnutls_ocsp_status_request_get.....	318
gnutls_hex2bin.....	314	gnutls_ocsp_status_request_is_checked... ..	318
gnutls_hmac.....	558	gnutls_openpgp_cert_check_hostname.....	480
gnutls_hmac_deinit.....	558	gnutls_openpgp_cert_check_hostname2.....	480
gnutls_hmac_fast.....	558	gnutls_openpgp_cert_deinit.....	480
gnutls_hmac_get_len.....	558	gnutls_openpgp_cert_export.....	480
gnutls_hmac_init.....	559	gnutls_openpgp_cert_export2.....	481
gnutls_hmac_output.....	559	gnutls_openpgp_cert_get_auth_subkey.....	481
gnutls_hmac_set_nonce.....	559	gnutls_openpgp_cert_get_creation_time....	481
gnutls_init.....	112, 315	gnutls_openpgp_cert_get_expiration_time..	481
gnutls_key_generate.....	315	gnutls_openpgp_cert_get_fingerprint.....	481
gnutls_kx_get.....	316	gnutls_openpgp_cert_get_key_id.....	482
gnutls_kx_get_id.....	316	gnutls_openpgp_cert_get_key_usage.....	482
gnutls_kx_get_name.....	316	gnutls_openpgp_cert_get_name.....	482
gnutls_kx_list.....	316	gnutls_openpgp_cert_get_pk_algorithm.....	482
gnutls_kx_set_priority.....	563	gnutls_openpgp_cert_get_pk_dsa_raw.....	483
gnutls_load_file.....	316	gnutls_openpgp_cert_get_pk_rsa_raw.....	483
gnutls_mac_get.....	317	gnutls_openpgp_cert_get_preferred_key_id.....	483
gnutls_mac_get_id.....	317	gnutls_openpgp_cert_get_revoked_status... ..	484
gnutls_mac_get_key_size.....	317	gnutls_openpgp_cert_get_subkey_count.....	484
gnutls_mac_get_name.....	317	gnutls_openpgp_cert_get_subkey_creation_time.....	484
gnutls_mac_get_nonce_size.....	559	gnutls_openpgp_cert_get_subkey_expiration_time.....	484
gnutls_mac_list.....	317	gnutls_openpgp_cert_get_subkey_fingerprint.....	484
gnutls_mac_set_priority.....	563	gnutls_openpgp_cert_get_subkey_id.....	485
gnutls_ocsp_req_add_cert.....	467	gnutls_openpgp_cert_get_subkey_idx.....	485
gnutls_ocsp_req_add_cert_id.....	467	gnutls_openpgp_cert_get_subkey_pk_algorithm.....	485
gnutls_ocsp_req_deinit.....	468	gnutls_openpgp_cert_get_subkey_pk_dsa_raw.....	486
gnutls_ocsp_req_export.....	468	gnutls_openpgp_cert_get_subkey_pk_rsa_raw.....	486
gnutls_ocsp_req_get_cert_id.....	468	gnutls_openpgp_cert_get_subkey_revoked_status.....	486
gnutls_ocsp_req_get_extension.....	469	gnutls_openpgp_cert_get_subkey_usage.....	487
gnutls_ocsp_req_get_nonce.....	469	gnutls_openpgp_cert_get_version.....	487
gnutls_ocsp_req_get_version.....	469	gnutls_openpgp_cert_import.....	487
gnutls_ocsp_req_import.....	470	gnutls_openpgp_cert_init.....	487
gnutls_ocsp_req_init.....	470		
gnutls_ocsp_req_print.....	470		
gnutls_ocsp_req_randomize_nonce.....	470		
gnutls_ocsp_req_set_extension.....	471		
gnutls_ocsp_req_set_nonce.....	471		
gnutls_ocsp_resp_check_cert.....	471		
gnutls_ocsp_resp_deinit.....	471		
gnutls_ocsp_resp_export.....	472		
gnutls_ocsp_resp_get_certs.....	472		
gnutls_ocsp_resp_get_extension.....	472		

gnutls_openpgp_crt_print .....	487	gnutls_pem_base64_decode .....	319
gnutls_openpgp_crt_set_preferred_key_id .....	488	gnutls_pem_base64_decode_alloc .....	319
gnutls_openpgp_crt_verify_ring .....	39, 488	gnutls_pem_base64_encode .....	320
gnutls_openpgp_crt_verify_self .....	39, 488	gnutls_pem_base64_encode_alloc .....	320
gnutls_openpgp_keyring_check_id .....	489	gnutls_perror .....	320
gnutls_openpgp_keyring_deinit .....	489	gnutls_pk_algorithm_get_name .....	320
gnutls_openpgp_keyring_get_crt .....	489	gnutls_pk_bits_to_sec_param .....	140, 321
gnutls_openpgp_keyring_get_crt_count .....	489	gnutls_pk_get_id .....	321
gnutls_openpgp_keyring_import .....	489	gnutls_pk_get_name .....	321
gnutls_openpgp_keyring_init .....	490	gnutls_pk_list .....	321
gnutls_openpgp_privkey_deinit .....	490	gnutls_pk_to_sign .....	321
gnutls_openpgp_privkey_export .....	490	gnutls_pkcs11_add_provider .....	504
gnutls_openpgp_privkey_export_dsa_raw ...	491	gnutls_pkcs11_copy_secret_key .....	504
gnutls_openpgp_privkey_export_rsa_raw ...	491	gnutls_pkcs11_copy_x509_crt .....	95, 504
gnutls_openpgp_privkey_export_subkey_dsa_raw .....	492	gnutls_pkcs11_copy_x509_privkey .....	94, 505
gnutls_openpgp_privkey_export_subkey_rsa_raw .....	492	gnutls_pkcs11_crt_is_known .....	505
gnutls_openpgp_privkey_export2 .....	490	gnutls_pkcs11_deinit .....	506
gnutls_openpgp_privkey_get_fingerprint ..	493	gnutls_pkcs11_delete_url .....	95, 506
gnutls_openpgp_privkey_get_key_id .....	493	gnutls_pkcs11_get_pin_function .....	506
gnutls_openpgp_privkey_get_pk_algorithm .....	493	gnutls_pkcs11_get_raw_issuer .....	506
gnutls_openpgp_privkey_get_preferred_key_id .....	493	gnutls_pkcs11_init .....	89, 507
gnutls_openpgp_privkey_get_revoked_status .....	494	gnutls_pkcs11_obj_deinit .....	507
gnutls_openpgp_privkey_get_subkey_count .....	494	gnutls_pkcs11_obj_export .....	507
gnutls_openpgp_privkey_get_subkey_creation_time .....	494	gnutls_pkcs11_obj_export_url .....	508
gnutls_openpgp_privkey_get_subkey_expiration_time .....	494	gnutls_pkcs11_obj_export2 .....	507
gnutls_openpgp_privkey_get_subkey_fingerprint .....	494	gnutls_pkcs11_obj_export3 .....	508
gnutls_openpgp_privkey_get_subkey_id .....	495	gnutls_pkcs11_obj_flags_get_str .....	508
gnutls_openpgp_privkey_get_subkey_idx ...	495	gnutls_pkcs11_obj_get_exts .....	509
gnutls_openpgp_privkey_get_subkey_pk_algorithm .....	495	gnutls_pkcs11_obj_get_flags .....	509
gnutls_openpgp_privkey_get_subkey_revoked_status .....	496	gnutls_pkcs11_obj_get_info .....	92, 509
gnutls_openpgp_privkey_import .....	496	gnutls_pkcs11_obj_get_type .....	510
gnutls_openpgp_privkey_init .....	496	gnutls_pkcs11_obj_import_url .....	510
gnutls_openpgp_privkey_sec_param .....	496	gnutls_pkcs11_obj_init .....	510
gnutls_openpgp_privkey_set_preferred_key_id .....	497	gnutls_pkcs11_obj_list_import_url .....	510
gnutls_openpgp_privkey_sign_hash .....	563	gnutls_pkcs11_obj_list_import_url2 .....	511
gnutls_openpgp_send_cert .....	318	gnutls_pkcs11_obj_set_pin_function .....	511
gnutls_openpgp_set_rcv_key_function .....	497	gnutls_pkcs11_privkey_deinit .....	511
gnutls_packet_deinit .....	319	gnutls_pkcs11_privkey_export_pubkey .....	512
gnutls_packet_get .....	319	gnutls_pkcs11_privkey_export_url .....	512
gnutls_pcert_deinit .....	522	gnutls_pkcs11_privkey_generate .....	512
gnutls_pcert_import_openpgp .....	522	gnutls_pkcs11_privkey_generate2 .....	513
gnutls_pcert_import_openpgp_raw .....	522	gnutls_pkcs11_privkey_get_info .....	513
gnutls_pcert_import_x509 .....	522	gnutls_pkcs11_privkey_get_pk_algorithm ..	513
gnutls_pcert_import_x509_raw .....	523	gnutls_pkcs11_privkey_import_url .....	514
gnutls_pcert_list_import_x509_raw .....	523	gnutls_pkcs11_privkey_init .....	514
		gnutls_pkcs11_privkey_set_pin_function ..	514
		gnutls_pkcs11_privkey_status .....	514
		gnutls_pkcs11_reinit .....	515
		gnutls_pkcs11_set_pin_function .....	515
		gnutls_pkcs11_set_token_function .....	515
		gnutls_pkcs11_token_get_flags .....	515
		gnutls_pkcs11_token_get_info .....	516
		gnutls_pkcs11_token_get_mechanism .....	516
		gnutls_pkcs11_token_get_random .....	516
		gnutls_pkcs11_token_get_url .....	516
		gnutls_pkcs11_token_init .....	517
		gnutls_pkcs11_token_set_pin .....	517
		gnutls_pkcs11_type_get_name .....	517

gnutls_pkcs12_bag_decrypt .....	497	gnutls_privkey_export_rsa_raw .....	525
gnutls_pkcs12_bag_deinit .....	497	gnutls_privkey_generate .....	525
gnutls_pkcs12_bag_encrypt .....	498	gnutls_privkey_get_pk_algorithm .....	526
gnutls_pkcs12_bag_get_count .....	498	gnutls_privkey_get_type .....	526
gnutls_pkcs12_bag_get_data .....	498	gnutls_privkey_import_dsa_raw .....	526
gnutls_pkcs12_bag_get_friendly_name .....	498	gnutls_privkey_import_ecc_raw .....	527
gnutls_pkcs12_bag_get_key_id .....	499	gnutls_privkey_import_ext .....	527
gnutls_pkcs12_bag_get_type .....	499	gnutls_privkey_import_ext2 .....	84, 527
gnutls_pkcs12_bag_init .....	499	gnutls_privkey_import_openpgp .....	528
gnutls_pkcs12_bag_set_crl .....	499	gnutls_privkey_import_openpgp_raw .....	528
gnutls_pkcs12_bag_set_cert .....	499	gnutls_privkey_import_pkcs11 .....	529
gnutls_pkcs12_bag_set_data .....	500	gnutls_privkey_import_pkcs11_url .....	529
gnutls_pkcs12_bag_set_friendly_name .....	500	gnutls_privkey_import_rsa_raw .....	529
gnutls_pkcs12_bag_set_key_id .....	500	gnutls_privkey_import_tpm_raw .....	530
gnutls_pkcs12_deinit .....	500	gnutls_privkey_import_tpm_url .....	102, 530
gnutls_pkcs12_export .....	501	gnutls_privkey_import_url .....	84, 531
gnutls_pkcs12_export2 .....	501	gnutls_privkey_import_x509 .....	531
gnutls_pkcs12_generate_mac .....	501	gnutls_privkey_import_x509_raw .....	54, 531
gnutls_pkcs12_get_bag .....	501	gnutls_privkey_init .....	532
gnutls_pkcs12_import .....	502	gnutls_privkey_set_pin_function .....	532
gnutls_pkcs12_init .....	502	gnutls_privkey_sign_data .....	86, 532
gnutls_pkcs12_set_bag .....	502	gnutls_privkey_sign_hash .....	86, 533
gnutls_pkcs12_simple_parse .....	56, 503	gnutls_privkey_sign_raw_data .....	563
gnutls_pkcs12_verify_mac .....	503	gnutls_privkey_status .....	533
gnutls_pkcs7_deinit .....	365	gnutls_privkey_verify_params .....	533
gnutls_pkcs7_delete_crl .....	365	gnutls_protocol_get_id .....	327
gnutls_pkcs7_delete_cert .....	366	gnutls_protocol_get_name .....	327
gnutls_pkcs7_export .....	366	gnutls_protocol_get_version .....	327
gnutls_pkcs7_export2 .....	366	gnutls_protocol_list .....	328
gnutls_pkcs7_get_crl_count .....	367	gnutls_protocol_set_priority .....	564
gnutls_pkcs7_get_crl_raw .....	367	gnutls_psk_allocate_client_credentials ..	328
gnutls_pkcs7_get_cert_count .....	367	gnutls_psk_allocate_server_credentials ..	328
gnutls_pkcs7_get_cert_raw .....	367	gnutls_psk_client_get_hint .....	328
gnutls_pkcs7_import .....	368	gnutls_psk_free_client_credentials .....	328
gnutls_pkcs7_init .....	368	gnutls_psk_free_server_credentials .....	329
gnutls_pkcs7_set_crl .....	368	gnutls_psk_server_get_username .....	329
gnutls_pkcs7_set_crl_raw .....	368	gnutls_psk_set_client_credentials .....	329
gnutls_pkcs7_set_cert .....	369	gnutls_psk_set_client_credentials_function	
gnutls_pkcs7_set_cert_raw .....	369	.....	120, 329
gnutls_prf .....	322	gnutls_psk_set_params_function .....	330
gnutls_prf_raw .....	322	gnutls_psk_set_server_credentials_file	
gnutls_priority_certificate_type_list ..	323	.....	121, 330
gnutls_priority_cipher_list .....	323	gnutls_psk_set_server_credentials_function	
gnutls_priority_compression_list .....	323	.....	330
gnutls_priority_deinit .....	324	gnutls_psk_set_server_credentials_hint ..	330
gnutls_priority_ecc_curve_list .....	324	gnutls_psk_set_server_dh_params .....	331
gnutls_priority_get_cipher_suite_index ..	324	gnutls_psk_set_server_params_function ..	331
gnutls_priority_init .....	324	gnutls_pubkey_deinit .....	534
gnutls_priority_kx_list .....	326	gnutls_pubkey_encrypt_data .....	86, 534
gnutls_priority_mac_list .....	326	gnutls_pubkey_export .....	534
gnutls_priority_protocol_list .....	326	gnutls_pubkey_export_dsa_raw .....	535
gnutls_priority_set .....	326	gnutls_pubkey_export_ecc_raw .....	535
gnutls_priority_set_direct .....	326	gnutls_pubkey_export_ecc_x962 .....	535
gnutls_priority_sign_list .....	327	gnutls_pubkey_export_rsa_raw .....	536
gnutls_privkey_decrypt_data .....	87, 524	gnutls_pubkey_export2 .....	82, 534
gnutls_privkey_deinit .....	524	gnutls_pubkey_get_key_id .....	536
gnutls_privkey_export_dsa_raw .....	524	gnutls_pubkey_get_key_usage .....	536
gnutls_privkey_export_ecc_raw .....	524	gnutls_pubkey_get_openpgp_key_id .....	537

gnutls_pubkey_get_pk_algorithm .....	537	gnutls_rsa_params_generate2 .....	566
gnutls_pubkey_get_preferred_hash_algorithm .....	537	gnutls_rsa_params_import_pkcs1 .....	566
gnutls_pubkey_get_verify_algorithm .....	538	gnutls_rsa_params_import_raw .....	566
gnutls_pubkey_import .....	538	gnutls_rsa_params_init .....	567
gnutls_pubkey_import_dsa_raw .....	538	gnutls_safe_renegotiation_status .....	338
gnutls_pubkey_import_ecc_raw .....	539	gnutls_sec_param_get_name .....	338
gnutls_pubkey_import_ecc_x962 .....	539	gnutls_sec_param_to_pk_bits .....	140, 338
gnutls_pubkey_import_opengpg .....	539	gnutls_sec_param_to_symmetric_bits .....	338
gnutls_pubkey_import_opengpg_raw .....	540	gnutls_server_name_get .....	339
gnutls_pubkey_import_pkcs1 .....	540	gnutls_server_name_set .....	339
gnutls_pubkey_import_pkcs1_url .....	540	gnutls_session_channel_binding .....	339
gnutls_pubkey_import_privkey .....	541	gnutls_session_enable_compatibility_mode .....	340
gnutls_pubkey_import_rsa_raw .....	541	gnutls_session_force_valid .....	340
gnutls_pubkey_import_tpm_raw .....	541	gnutls_session_get_data .....	340
gnutls_pubkey_import_tpm_url .....	102, 542	gnutls_session_get_data2 .....	341
gnutls_pubkey_import_url .....	542	gnutls_session_get_desc .....	341
gnutls_pubkey_import_x509 .....	542	gnutls_session_get_id .....	341
gnutls_pubkey_import_x509_crq .....	543	gnutls_session_get_id2 .....	341
gnutls_pubkey_import_x509_raw .....	543	gnutls_session_get_ptr .....	342
gnutls_pubkey_init .....	543	gnutls_session_get_random .....	342
gnutls_pubkey_print .....	543	gnutls_session_is_resumed .....	141, 342
gnutls_pubkey_set_key_usage .....	544	gnutls_session_resumption_requested .....	142, 342
gnutls_pubkey_set_pin_function .....	544	gnutls_session_set_data .....	343
gnutls_pubkey_verify_data .....	544	gnutls_session_set_id .....	343
gnutls_pubkey_verify_data2 .....	85, 545	gnutls_session_set_premaster .....	343
gnutls_pubkey_verify_hash .....	545	gnutls_session_set_ptr .....	344
gnutls_pubkey_verify_hash2 .....	85, 545	gnutls_session_ticket_enable_client .....	344
gnutls_pubkey_verify_params .....	546	gnutls_session_ticket_enable_server .....	141, 344
gnutls_random_art .....	331	gnutls_session_ticket_key_generate ..	142, 344
gnutls_range_split .....	332	gnutls_set_default_export_priority .....	567
gnutls_record_can_use_length_hiding .....	332	gnutls_set_default_priority .....	345
gnutls_record_check_corked .....	332	gnutls_sign_algorithm_get .....	345
gnutls_record_check_pending .....	129, 332	gnutls_sign_algorithm_get_client .....	345
gnutls_record_cork .....	130, 333	gnutls_sign_algorithm_get_requested .....	345
gnutls_record_disable_padding .....	333	gnutls_sign_callback_get .....	567
gnutls_record_get_direction .....	125, 333	gnutls_sign_callback_set .....	567
gnutls_record_get_discarded .....	365	gnutls_sign_get_hash_algorithm .....	346
gnutls_record_get_max_size .....	333	gnutls_sign_get_id .....	346
gnutls_record_overhead_size .....	333	gnutls_sign_get_name .....	346
gnutls_record_rcv .....	128, 334	gnutls_sign_get_pk_algorithm .....	346
gnutls_record_rcv_packet .....	334	gnutls_sign_is_secure .....	346
gnutls_record_rcv_seq .....	129, 334	gnutls_sign_list .....	347
gnutls_record_send .....	127, 335	gnutls_srp_allocate_client_credentials ..	347
gnutls_record_send_range .....	335	gnutls_srp_allocate_server_credentials ..	347
gnutls_record_set_max_empty_records .....	336	gnutls_srp_base64_decode .....	347
gnutls_record_set_max_size .....	336	gnutls_srp_base64_decode_alloc .....	347
gnutls_record_set_timeout .....	337	gnutls_srp_base64_encode .....	348
gnutls_record_uncork .....	130, 337	gnutls_srp_base64_encode_alloc .....	348
gnutls_rehandshake .....	337	gnutls_srp_free_client_credentials .....	348
gnutls_rnd .....	230, 560	gnutls_srp_free_server_credentials .....	349
gnutls_rnd_refresh .....	560	gnutls_srp_server_get_username .....	349
gnutls_rsa_export_get_modulus_bits .....	564	gnutls_srp_set_client_credentials .....	349
gnutls_rsa_export_get_pubkey .....	564	gnutls_srp_set_client_credentials_function .....	119, 349
gnutls_rsa_params_cpy .....	565	gnutls_srp_set_prime_bits .....	350
gnutls_rsa_params_deinit .....	565		
gnutls_rsa_params_export_pkcs1 .....	565		
gnutls_rsa_params_export_raw .....	565		

<code>gnutls_srp_set_server_credentials_file</code>	119, 350
<code>gnutls_srp_set_server_credentials_function</code>	120, 350
<code>gnutls_srp_set_server_fake_salt_seed</code>	351
<code>gnutls_srp_verifier</code>	75, 351
<code>gnutls_srtcp_get_keys</code>	14, 352
<code>gnutls_srtcp_get_mki</code>	352
<code>gnutls_srtcp_get_profile_id</code>	353
<code>gnutls_srtcp_get_profile_name</code>	353
<code>gnutls_srtcp_get_selected_profile</code>	353
<code>gnutls_srtcp_set_mki</code>	353
<code>gnutls_srtcp_set_profile</code>	354
<code>gnutls_srtcp_set_profile_direct</code>	354
<code>gnutls_store_commitment</code>	143, 354
<code>gnutls_store_pubkey</code>	143, 355
<code>gnutls_strerror</code>	355
<code>gnutls_strerror_name</code>	355
<code>gnutls_subject_alt_names_deinit</code>	369
<code>gnutls_subject_alt_names_get</code>	369
<code>gnutls_subject_alt_names_init</code>	370
<code>gnutls_subject_alt_names_set</code>	370
<code>gnutls_supplemental_get_name</code>	356
<code>gnutls_tdb_deinit</code>	356
<code>gnutls_tdb_init</code>	356
<code>gnutls_tdb_set_store_commitment_func</code>	356
<code>gnutls_tdb_set_store_func</code>	357
<code>gnutls_tdb_set_verify_func</code>	357
<code>gnutls_tpm_get_registered</code>	519
<code>gnutls_tpm_key_list_deinit</code>	519
<code>gnutls_tpm_key_list_get_url</code>	519
<code>gnutls_tpm_privkey_delete</code>	101, 103, 519
<code>gnutls_tpm_privkey_generate</code>	101, 520
<code>gnutls_transport_get_int</code>	357
<code>gnutls_transport_get_int2</code>	357
<code>gnutls_transport_get_ptr</code>	358
<code>gnutls_transport_get_ptr2</code>	358
<code>gnutls_transport_set_errno</code>	123, 358
<code>gnutls_transport_set_errno_function</code>	358
<code>gnutls_transport_set_int</code>	359
<code>gnutls_transport_set_int2</code>	359
<code>gnutls_transport_set_ptr</code>	359
<code>gnutls_transport_set_ptr2</code>	359
<code>gnutls_transport_set_pull_function</code>	123, 359
<code>gnutls_transport_set_pull_timeout_function</code>	123, 124, 360
<code>gnutls_transport_set_push_function</code>	122, 360
<code>gnutls_transport_set_vec_push_function</code>	122, 360
<code>gnutls_url_is_supported</code>	83, 361
<code>gnutls_verify_stored_pubkey</code>	142, 361
<code>gnutls_x509_aia_deinit</code>	370
<code>gnutls_x509_aia_get</code>	370
<code>gnutls_x509_aia_init</code>	371
<code>gnutls_x509_aia_set</code>	371
<code>gnutls_x509_aki_deinit</code>	371
<code>gnutls_x509_aki_get_cert_issuer</code>	371
<code>gnutls_x509_aki_get_id</code>	372
<code>gnutls_x509_aki_init</code>	372
<code>gnutls_x509_aki_set_cert_issuer</code>	372
<code>gnutls_x509_aki_set_id</code>	373
<code>gnutls_x509_crl_check_issuer</code>	373
<code>gnutls_x509_crl_deinit</code>	373
<code>gnutls_x509_crl_dist_points_deinit</code>	373
<code>gnutls_x509_crl_dist_points_get</code>	373
<code>gnutls_x509_crl_dist_points_init</code>	374
<code>gnutls_x509_crl_dist_points_set</code>	374
<code>gnutls_x509_crl_export</code>	374
<code>gnutls_x509_crl_export2</code>	375
<code>gnutls_x509_crl_get_authority_key_gn_serial</code>	375
<code>gnutls_x509_crl_get_authority_key_id</code>	375
<code>gnutls_x509_crl_get_cert_count</code>	376
<code>gnutls_x509_crl_get_cert_serial</code>	46, 376
<code>gnutls_x509_crl_get_dn_oid</code>	376
<code>gnutls_x509_crl_get_extension_data</code>	377
<code>gnutls_x509_crl_get_extension_data2</code>	377
<code>gnutls_x509_crl_get_extension_info</code>	377
<code>gnutls_x509_crl_get_extension_oid</code>	378
<code>gnutls_x509_crl_get_issuer_dn</code>	378
<code>gnutls_x509_crl_get_issuer_dn_by_oid</code>	379
<code>gnutls_x509_crl_get_issuer_dn2</code>	378
<code>gnutls_x509_crl_get_next_update</code>	379
<code>gnutls_x509_crl_get_number</code>	379
<code>gnutls_x509_crl_get_raw_issuer_dn</code>	380
<code>gnutls_x509_crl_get_signature</code>	380
<code>gnutls_x509_crl_get_signature_algorithm</code>	380
<code>gnutls_x509_crl_get_this_update</code>	380
<code>gnutls_x509_crl_get_version</code>	381
<code>gnutls_x509_crl_import</code>	381
<code>gnutls_x509_crl_init</code>	381
<code>gnutls_x509_crl_iter_cert_serial</code>	381
<code>gnutls_x509_crl_iter_deinit</code>	382
<code>gnutls_x509_crl_list_import</code>	382
<code>gnutls_x509_crl_list_import2</code>	382
<code>gnutls_x509_crl_print</code>	383
<code>gnutls_x509_crl_privkey_sign</code>	48, 546
<code>gnutls_x509_crl_set_authority_key_id</code>	383
<code>gnutls_x509_crl_set_cert</code>	383
<code>gnutls_x509_crl_set_cert_serial</code>	384
<code>gnutls_x509_crl_set_next_update</code>	384
<code>gnutls_x509_crl_set_number</code>	384
<code>gnutls_x509_crl_set_this_update</code>	384
<code>gnutls_x509_crl_set_version</code>	385
<code>gnutls_x509_crl_sign</code>	568
<code>gnutls_x509_crl_sign2</code>	47, 385
<code>gnutls_x509_crl_verify</code>	385
<code>gnutls_x509_crq_deinit</code>	386
<code>gnutls_x509_crq_export</code>	386
<code>gnutls_x509_crq_export2</code>	386
<code>gnutls_x509_crq_get_attribute_by_oid</code>	387
<code>gnutls_x509_crq_get_attribute_data</code>	387
<code>gnutls_x509_crq_get_attribute_info</code>	387
<code>gnutls_x509_crq_get_basic_constraints</code>	388
<code>gnutls_x509_crq_get_challenge_password</code>	388



gnutls_x509_crq_get_dn.....	388	gnutls_x509_crt_get_dn.....	406
gnutls_x509_crq_get_dn_by_oid.....	389	gnutls_x509_crt_get_dn_by_oid.....	407
gnutls_x509_crq_get_dn_oid.....	390	gnutls_x509_crt_get_dn_oid.....	407
gnutls_x509_crq_get_dn2.....	389	gnutls_x509_crt_get_dn2.....	22, 407
gnutls_x509_crq_get_extension_by_oid....	390	gnutls_x509_crt_get_expiration_time.....	408
gnutls_x509_crq_get_extension_by_oid2...	390	gnutls_x509_crt_get_extension_by_oid....	408
gnutls_x509_crq_get_extension_data.....	391	gnutls_x509_crt_get_extension_by_oid2...	408
gnutls_x509_crq_get_extension_data2.....	391	gnutls_x509_crt_get_extension_data.....	409
gnutls_x509_crq_get_extension_info.....	391	gnutls_x509_crt_get_extension_data2.....	409
gnutls_x509_crq_get_key_id.....	392	gnutls_x509_crt_get_extension_info.....	410
gnutls_x509_crq_get_key_purpose_oid.....	392	gnutls_x509_crt_get_extension_oid.....	410
gnutls_x509_crq_get_key_rsa_raw.....	393	gnutls_x509_crt_get_fingerprint.....	410
gnutls_x509_crq_get_key_usage.....	393	gnutls_x509_crt_get_issuer.....	411
gnutls_x509_crq_get_pk_algorithm.....	393	gnutls_x509_crt_get_issuer_alt_name.....	411
gnutls_x509_crq_get_private_key_usage_		gnutls_x509_crt_get_issuer_alt_name2....	412
period.....	394	gnutls_x509_crt_get_issuer_alt_othername_	
gnutls_x509_crq_get_subject_alt_name....	394	oid.....	412
gnutls_x509_crq_get_subject_alt_othername_		gnutls_x509_crt_get_issuer_dn.....	413
oid.....	395	gnutls_x509_crt_get_issuer_dn_by_oid....	413
gnutls_x509_crq_get_version.....	395	gnutls_x509_crt_get_issuer_dn_oid.....	414
gnutls_x509_crq_import.....	395	gnutls_x509_crt_get_issuer_dn2.....	413
gnutls_x509_crq_init.....	396	gnutls_x509_crt_get_issuer_unique_id....	414
gnutls_x509_crq_print.....	396	gnutls_x509_crt_get_key_id.....	27, 415
gnutls_x509_crq_privkey_sign.....	546	gnutls_x509_crt_get_key_purpose_oid.....	415
gnutls_x509_crq_set_attribute_by_oid....	396	gnutls_x509_crt_get_key_usage.....	415
gnutls_x509_crq_set_basic_constraints...	396	gnutls_x509_crt_get_name_constraints....	416
gnutls_x509_crq_set_challenge_password..	397	gnutls_x509_crt_get_pk_algorithm.....	416
gnutls_x509_crq_set_dn.....	397	gnutls_x509_crt_get_pk_dsa_raw.....	417
gnutls_x509_crq_set_dn_by_oid.....	397	gnutls_x509_crt_get_pk_rsa_raw.....	417
gnutls_x509_crq_set_key.....	43, 398	gnutls_x509_crt_get_policy.....	417
gnutls_x509_crq_set_key_purpose_oid.....	398	gnutls_x509_crt_get_preferred_hash_	
gnutls_x509_crq_set_key_rsa_raw.....	398	algorithm.....	568
gnutls_x509_crq_set_key_usage.....	398	gnutls_x509_crt_get_private_key_usage_	
gnutls_x509_crq_set_private_key_usage_		period.....	418
period.....	399	gnutls_x509_crt_get_proxy.....	418
gnutls_x509_crq_set_pubkey.....	87, 547	gnutls_x509_crt_get_raw_dn.....	418
gnutls_x509_crq_set_subject_alt_name....	399	gnutls_x509_crt_get_raw_issuer_dn.....	419
gnutls_x509_crq_set_version.....	399	gnutls_x509_crt_get_serial.....	419
gnutls_x509_crq_sign.....	568	gnutls_x509_crt_get_signature.....	419
gnutls_x509_crq_sign2.....	43, 400	gnutls_x509_crt_get_signature_algorithm	
gnutls_x509_crq_verify.....	400	.....	419
gnutls_x509_crt_check_hostname.....	400	gnutls_x509_crt_get_subject.....	420
gnutls_x509_crt_check_hostname2.....	401	gnutls_x509_crt_get_subject_alt_name....	420
gnutls_x509_crt_check_issuer.....	401	gnutls_x509_crt_get_subject_alt_name2...	420
gnutls_x509_crt_check_revocation.....	401	gnutls_x509_crt_get_subject_alt_othername_	
gnutls_x509_crt_cpy_crl_dist_points....	402	oid.....	421
gnutls_x509_crt_deinit.....	402	gnutls_x509_crt_get_subject_key_id.....	421
gnutls_x509_crt_export.....	402	gnutls_x509_crt_get_subject_unique_id...	422
gnutls_x509_crt_export2.....	402	gnutls_x509_crt_get_verify_algorithm....	569
gnutls_x509_crt_get_activation_time....	403	gnutls_x509_crt_get_version.....	422
gnutls_x509_crt_get_authority_info_access		gnutls_x509_crt_import.....	422
.....	403	gnutls_x509_crt_import_pkcs11.....	518
gnutls_x509_crt_get_authority_key_gn_serial		gnutls_x509_crt_import_pkcs11_url.....	518
.....	404	gnutls_x509_crt_init.....	423
gnutls_x509_crt_get_authority_key_id....	405	gnutls_x509_crt_list_import.....	423
gnutls_x509_crt_get_basic_constraints...	405	gnutls_x509_crt_list_import_pkcs11.....	518
gnutls_x509_crt_get_ca_status.....	405	gnutls_x509_crt_list_import2.....	423
gnutls_x509_crt_get_crl_dist_points....	406	gnutls_x509_crt_list_verify.....	424

gnutls_x509_crt_print.....	424	gnutls_x509_ext_export_name_constraints	
gnutls_x509_crt_privkey_sign.....	547	.....	439
gnutls_x509_crt_set_activation_time.....	425	gnutls_x509_ext_export_policies.....	439
gnutls_x509_crt_set_authority_info_access		gnutls_x509_ext_export_private_key_usage_	
.....	425	period.....	440
gnutls_x509_crt_set_authority_key_id.....	425	gnutls_x509_ext_export_proxy.....	440
gnutls_x509_crt_set_basic_constraints...	425	gnutls_x509_ext_export_subject_alt_names	
gnutls_x509_crt_set_ca_status.....	426	.....	440
gnutls_x509_crt_set_crl_dist_points.....	426	gnutls_x509_ext_export_subject_key_id...	441
gnutls_x509_crt_set_crl_dist_points2.....	426	gnutls_x509_ext_import_aia.....	441
gnutls_x509_crt_set_crq.....	427	gnutls_x509_ext_import_authority_key_id	
gnutls_x509_crt_set_crq_extensions.....	427	.....	441
gnutls_x509_crt_set_dn.....	427	gnutls_x509_ext_import_basic_constraints	
gnutls_x509_crt_set_dn_by_oid.....	427	.....	442
gnutls_x509_crt_set_expiration_time.....	428	gnutls_x509_ext_import_crl_dist_points..	442
gnutls_x509_crt_set_extension_by_oid.....	428	gnutls_x509_ext_import_key_purposes.....	442
gnutls_x509_crt_set_issuer_alt_name.....	428	gnutls_x509_ext_import_key_usage.....	442
gnutls_x509_crt_set_issuer_dn.....	429	gnutls_x509_ext_import_name_constraints	
gnutls_x509_crt_set_issuer_dn_by_oid.....	429	.....	443
gnutls_x509_crt_set_key.....	430	gnutls_x509_ext_import_policies.....	443
gnutls_x509_crt_set_key_purpose_oid.....	430	gnutls_x509_ext_import_private_key_usage_	
gnutls_x509_crt_set_key_usage.....	430	period.....	444
gnutls_x509_crt_set_name_constraints.....	430	gnutls_x509_ext_import_proxy.....	444
gnutls_x509_crt_set_pin_function.....	431	gnutls_x509_ext_import_subject_alt_names	
gnutls_x509_crt_set_policy.....	431	.....	444
gnutls_x509_crt_set_private_key_usage_		gnutls_x509_ext_import_subject_key_id...	445
period.....	431	gnutls_x509_ext_print.....	445
gnutls_x509_crt_set_proxy.....	432	gnutls_x509_key_purpose_deinit.....	445
gnutls_x509_crt_set_proxy_dn.....	432	gnutls_x509_key_purpose_get.....	445
gnutls_x509_crt_set_pubkey.....	87, 547	gnutls_x509_key_purpose_init.....	446
gnutls_x509_crt_set_serial.....	432	gnutls_x509_key_purpose_set.....	446
gnutls_x509_crt_set_subject_alt_name.....	433	gnutls_x509_name_constraints_add_excluded	
gnutls_x509_crt_set_subject_alternative_		.....	446
name.....	433	gnutls_x509_name_constraints_add_permitted	
gnutls_x509_crt_set_subject_key_id.....	433	.....	446
gnutls_x509_crt_set_version.....	434	gnutls_x509_name_constraints_check.....	447
gnutls_x509_crt_sign.....	434	gnutls_x509_name_constraints_check_crt..	447
gnutls_x509_crt_sign2.....	434	gnutls_x509_name_constraints_deinit.....	447
gnutls_x509_crt_verify.....	435	gnutls_x509_name_constraints_get_excluded	
gnutls_x509_crt_verify_data.....	569	.....	447
gnutls_x509_crt_verify_hash.....	569	gnutls_x509_name_constraints_get_permitted	
gnutls_x509_dn_deinit.....	435	.....	448
gnutls_x509_dn_export.....	435	gnutls_x509_name_constraints_init.....	448
gnutls_x509_dn_export2.....	436	gnutls_x509_othername_to_virtual.....	448
gnutls_x509_dn_get_rdn_ava.....	23, 436	gnutls_x509_policies_deinit.....	449
gnutls_x509_dn_import.....	436	gnutls_x509_policies_get.....	449
gnutls_x509_dn_init.....	437	gnutls_x509_policies_init.....	449
gnutls_x509_dn_oid_known.....	437	gnutls_x509_policies_set.....	449
gnutls_x509_dn_oid_name.....	437	gnutls_x509_policy_release.....	450
gnutls_x509_ext_deinit.....	437	gnutls_x509_privkey_cpy.....	450
gnutls_x509_ext_export_aia.....	437	gnutls_x509_privkey_deinit.....	450
gnutls_x509_ext_export_authority_key_id		gnutls_x509_privkey_export.....	450
.....	438	gnutls_x509_privkey_export_dsa_raw.....	452
gnutls_x509_ext_export_basic_constraints		gnutls_x509_privkey_export_ecc_raw.....	452
.....	438	gnutls_x509_privkey_export_pkcs8.....	452
gnutls_x509_ext_export_crl_dist_points..	438	gnutls_x509_privkey_export_rsa_raw.....	453
gnutls_x509_ext_export_key_purposes.....	439	gnutls_x509_privkey_export_rsa_raw2.....	453
gnutls_x509_ext_export_key_usage.....	439	gnutls_x509_privkey_export2.....	451

<code>gnutls_x509_privkey_export2_pkcs8</code> .....	451	<code>gnutls_x509_trust_list_add_crls</code> .....	29, 461
<code>gnutls_x509_privkey_fix</code> .....	454	<code>gnutls_x509_trust_list_add_named_cert</code> ....	28,
<code>gnutls_x509_privkey_generate</code> .....	454		461
<code>gnutls_x509_privkey_get_key_id</code> .....	454	<code>gnutls_x509_trust_list_add_system_trust</code>	
<code>gnutls_x509_privkey_get_pk_algorithm</code> ....	455	.....	32, 462
<code>gnutls_x509_privkey_get_pk_algorithm2</code> ...	455	<code>gnutls_x509_trust_list_add_trust_dir</code> ....	462
<code>gnutls_x509_privkey_import</code> .....	455	<code>gnutls_x509_trust_list_add_trust_file</code> ....	31,
<code>gnutls_x509_privkey_import_dsa_raw</code> .....	456		462
<code>gnutls_x509_privkey_import_ecc_raw</code> .....	456	<code>gnutls_x509_trust_list_add_trust_mem</code> ....	31,
<code>gnutls_x509_privkey_import_openssl</code> ... 57,	457		463
<code>gnutls_x509_privkey_import_pkcs8</code> .....	457	<code>gnutls_x509_trust_list_deinit</code> .....	463
<code>gnutls_x509_privkey_import_rsa_raw</code> .....	458	<code>gnutls_x509_trust_list_get_issuer</code> .....	463
<code>gnutls_x509_privkey_import_rsa_raw2</code> ....	458	<code>gnutls_x509_trust_list_init</code> .....	464
<code>gnutls_x509_privkey_import2</code> .....	54, 456	<code>gnutls_x509_trust_list_remove_cas</code> .....	464
<code>gnutls_x509_privkey_init</code> .....	459	<code>gnutls_x509_trust_list_remove_trust_file</code>	
<code>gnutls_x509_privkey_sec_param</code> .....	459	.....	464
<code>gnutls_x509_privkey_sign_data</code> .....	570	<code>gnutls_x509_trust_list_remove_trust_mem</code>	
<code>gnutls_x509_privkey_sign_hash</code> .....	570	.....	465
<code>gnutls_x509_privkey_verify_params</code> .....	459	<code>gnutls_x509_trust_list_verify_cert</code> ....	29, 465
<code>gnutls_x509_rdn_get</code> .....	459	<code>gnutls_x509_trust_list_verify_cert2</code> ...	30, 465
<code>gnutls_x509_rdn_get_by_oid</code> .....	459	<code>gnutls_x509_trust_list_verify_named_cert</code>	
<code>gnutls_x509_rdn_get_oid</code> .....	460	.....	30, 466
<code>gnutls_x509_trust_list_add_cas</code> .....	28, 460		



# Concept Index

## A

abstract types .....	81
alert protocol .....	8
ALPN .....	15
anonymous authentication .....	78
API reference .....	273
Application Layer Protocol Negotiation .....	15
authentication methods .....	18

## B

bad_record_mac .....	7
----------------------	---

## C

callback functions .....	110
certificate authentication .....	18, 42
certificate requests .....	42
certificate revocation lists .....	46
certificate status .....	48
Certificate status request .....	13
Certificate verification .....	40
certification .....	257
certtool .....	58
certtool help .....	58
channel bindings .....	147
ciphersuites .....	266
client certificate authentication .....	10
compression algorithms .....	6
contributing .....	257
CRL .....	46

## D

danetool .....	70
danetool help .....	71
DANE .....	40, 142
deriving keys .....	146
digital signatures .....	41
DNSSEC .....	40, 142
download .....	2

## E

Encrypted keys .....	53
error codes .....	259
example programs .....	149
examples .....	149
exporting keying material .....	146

## F

FDL, GNU Free Documentation License .....	571
fork .....	110

## G

generating parameters .....	146
gnutls-cli .....	231
gnutls-cli help .....	231
gnutls-cli-debug .....	239
gnutls-cli-debug help .....	240
gnutls-serv .....	236
gnutls-serv help .....	236

## H

hacking .....	257
handshake protocol .....	9
hardware security modules .....	88
hardware tokens .....	88
hash functions .....	229
heartbeat .....	11
HMAC functions .....	229

## I

installation .....	2
internal architecture .....	242

## K

key extraction .....	146
Key pinning .....	40, 142
key sizes .....	138
keying material exporters .....	146

## M

maximum fragment length .....	10
-------------------------------	----

## O

OCSP Functions .....	467
OCSP status request .....	13
ocsptool .....	68
ocsptool help .....	68
OCSP .....	48
Online Certificate Status Protocol .....	48
OpenPGP API .....	477
OpenPGP certificates .....	37
OpenPGP server .....	192
OpenSSL .....	148
OpenSSL encrypted keys .....	57

## P

p11tool .....	96
p11tool help .....	96

parameter generation ..... 146  
 PCT ..... 17  
 PKCS #10 ..... 42  
 PKCS #11 tokens ..... 88  
 PKCS #12 ..... 55  
 PKCS #8 ..... 55  
 Priority strings ..... 132  
 PSK authentication ..... 77  
 psktool ..... 77  
 psktool help ..... 77  
 public key algorithms ..... 229

## R

random numbers ..... 230  
 record padding ..... 7  
 record protocol ..... 5  
 renegotiation ..... 11  
 reporting bugs ..... 256  
 resuming sessions ..... 10, 140

## S

safe renegotiation ..... 11  
 Secure RTP ..... 13  
 server name indication ..... 10  
 session resumption ..... 10, 140  
 session tickets ..... 11  
 Smart card example ..... 173  
 smart cards ..... 88  
 SRP authentication ..... 74  
 srptool ..... 75  
 srptool help ..... 76  
 SRTP ..... 13  
 SSH-style authentication ..... 40, 142

SSL 2 ..... 17  
 symmetric algorithms ..... 229  
 symmetric cryptography ..... 229  
 symmetric encryption algorithms ..... 5

## T

thread safety ..... 109  
 tickets ..... 11  
 TLS extensions ..... 10, 11  
 TLS layers ..... 4  
 tpmtool ..... 103  
 tpmtool help ..... 103  
 TPM ..... 100  
 transport layer ..... 4  
 transport protocol ..... 4  
 Trust on first use ..... 40, 142  
 trusted platform module ..... 100

## U

upgrading ..... 253

## V

verifying certificate paths ..... 28, 34, 40  
 verifying certificate with pkcs11 ..... 35

## X

X.509 certificates ..... 19  
 X.509 distinguished name ..... 22  
 X.509 extensions ..... 23  
 X.509 Functions ..... 365