# GnuTLS

Transport Layer Security Library for the GNU system
for version 3.0.9, 25 November 2011

**Nikos Mavrogiannopoulos**
**Simon Josefsson** (bug-gnutls@gnu.org)

This manual is last updated 25 November 2011 for version 3.0.9 of GnuTLS.

# Table of Contents

# 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/` and `http://www.gnu.org/software/gnutls/`.

# 2 Introduction to GnuTLS

## 2.1 Introduction

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.
- 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 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[1] library. The "Cryptographic back-end" is provided by the nettle[2] library.

## 2.2 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 Libnettle, and you will need to install it before installing GnuTLS. Libnettle is available from `http://www.lysator.liu.se/~nisse/nettle/`. 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.

The compression library, libz, as well as p11-kit are a optional dependencies. You can get libz from `http://www.zlib.net/` and p11-kit from `http://p11-glue.freedesktop.org/`.

[1] `http://www.gnu.org/software/libtasn1/`

[2] `http://www.lysator.liu.se/~nisse/nettle/`

The X.509 part of GnuTLS needs ASN.1 functionality, from 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/gnutls/download.html`.

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.

```
--disable-srp-authentication
--disable-psk-authentication
--disable-anon-authentication
--disable-extra-pki
--disable-openpgp-authentication
--disable-openssl-compatibility
--without-p11-kit
```

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

## 2.3 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 Chapter 4 [Certificate authentication], page 17, and shared-key as well anonymous authentication in Chapter 5 [Shared-key and anonymous authentication], page 24. We elaborate on certificate authentication by demonstrating advanced usage of the API in Chapter 6 [More on certificate authentication], page 27. The core of the TLS library is presented in Chapter 7 [How to use GnuTLS in applications], page 48 and example applications are listed in Chapter 8 [GnuTLS application examples], page 67. In Chapter 9 [Other included programs], page 123 the usage of few included programs that may assist debugging is presented. The last chapter is Chapter 10 [Internal architecture of GnuTLS], page 129 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[1], 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.

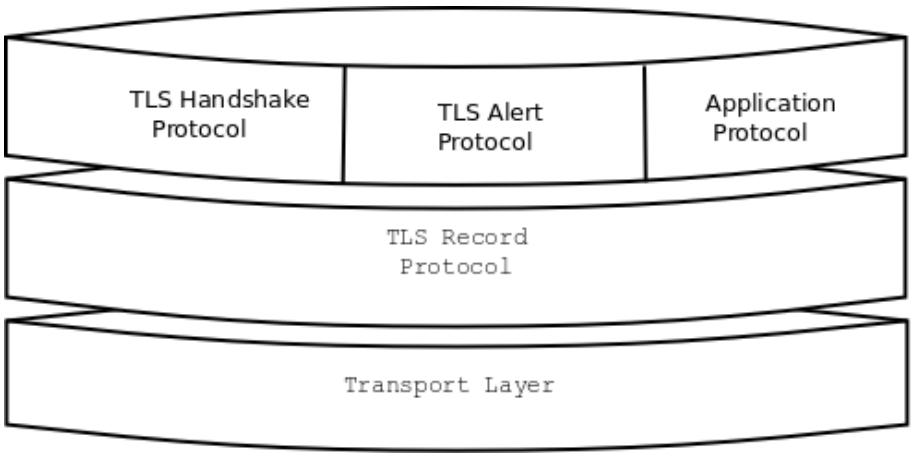


Figure 3.1: The TLS protocol layers.

## 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

[1] 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 7.5 [Setting up the transport layer], page 55).

## 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. For this reason programs using DTLS should call the receive functions for every packet received by the peer, even if no data were expected.

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.

- [gnutls_record_send], page 191
- [gnutls_record_recv], page 190
- [gnutls_record_recv_seq], page 190

### 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.

| Algorithm | Description |
|---|---|
| 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. |
| ARCFOUR_40 | This is the ARCFOUR cipher fed with a 40 bit key, which is considered weak. |
| 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.

| Algorithm | Description |
|---|---|
| MAC_MD5 | This is a cryptographic hash algorithm designed by Ron Rivest. Outputs 128 bits of data. |
| MAC_SHA1 | A cryptographic hash algorithm designed by NSA. Outputs 160 bits of data. |
| MAC_SHA256 | A cryptographic hash algorithm designed by NSA. 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 in ⟨undefined⟩ [gnutls_compression_method_t], page ⟨undefined⟩.

### 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. 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 random 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 implementation that take advantage of this text, and pad records by a random length.

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.

GnuTLS implements a work around for this problem. However, it has to be enabled specifically. It can be enabled by using [gnutls_record_disable_padding], page 189, or [gnutls_priority_set], page 184 with the `%COMPAT` priority string (see Section 7.8 [Priority Strings], page 58).

If you implement an application that have 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 184. 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 [tab:alerts], page 8.

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. The available functions to control the alert protocol are shown below.

- [gnutls_alert_get], page 148
- [gnutls_alert_send], page 148
- [gnutls_error_to_alert], page 171
- [gnutls_alert_get_name], page 148

Available alert messages:

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

## 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 174. 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[2] 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 341.

### 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.

- 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.

[2] MAC stands for Message Authentication Code. It can be described as a keyed hash algorithm. See RFC2104.

| 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. |
| RSA_EXPORT | The RSA algorithm is used to encrypt a key and send it to the peer. In the EXPORT algorithm, the server signs temporary RSA parameters of 512 bits — which are considered weak — and sends them to the client. |
| 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 to use this key exchange algorithm. |

Table 3.3: Supported key exchange algorithms.

Each authentication method is associated with a key exchange method, shown in Table 3.3, 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 164). A mapping of the key exchange methods with the credential types is shown in Table 3.4.

| 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 and certificate | 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, KX_DHE_PSK, KX_ECDHE_PSK | CRD_PSK | CRD_PSK |

Table 3.4: Key exchange algorithms and the corresponding credential types.

### 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 154 function. We elaborate in Section 7.4.1 [Certificate credentials], page 52.

### 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 12).

Session resumption is an integral part of GnuTLS, and Section 7.9.1 [Session resumption], page 61 and [ex:resume-client], page 93 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:

- Maximum fragment length negotiation
- Server name indication
- Session tickets

- Safe Renegotiation

and they will be 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.

- [gnutls_record_get_max_size], page 190
- [gnutls_record_set_max_size], page 191

### 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 193 and [gnutls_server_name_get], page 193 can be used to enable this extension, or to retrieve the name sent by a client.

- [gnutls_server_name_set], page 193
- [gnutls_server_name_get], page 193

### 3.6.3 Session tickets

To resume a TLS session the server normally store session parameters. This complicates deployment, and could be avoiding 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 Ticket extension implements this idea, and it is documented in RFC 5077 [*TLSTKT*].

### 3.6.4 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 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 7.8 [Priority Strings], page 58). 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 192 function is used to check if the extension has been negotiated on a session, and can be used both by clients and servers.

## 3.7 How to use TLS in application protocols

This chapter is intended to provide some hints on how to use the 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. That way two separate ports were assigned, one for the non secure sessions, and one for the secured ones. This has the benefit that if a user requests a secure session then the client will try to connect to the secure port and fail otherwise. The only possible attack with this method is a denial of service one. The most famous example of this method is the famous "HTTP over TLS" or HTTPS protocol [*RFC2818*].

Despite its wide use, this method is not as good as it seems. 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[3]. 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 obsoleted.

### 3.7.2 Upward negotiation

Other application protocols[4] 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 approach does not require any extra port to be reserved. There is even an extension to HTTP protocol to support that 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 see an example of a conversation where someone is acting in between:

[3] See also the Server Name Indication extension on [serverind], page 12.
[4] See LDAP, IMAP etc.

(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 dummy enough to send the confidential data in the clear.

How to avoid the above attack? As you may have already noticed this one 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 use 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 Certificate authentication

## 4.1 Introduction

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.

## 4.2 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.

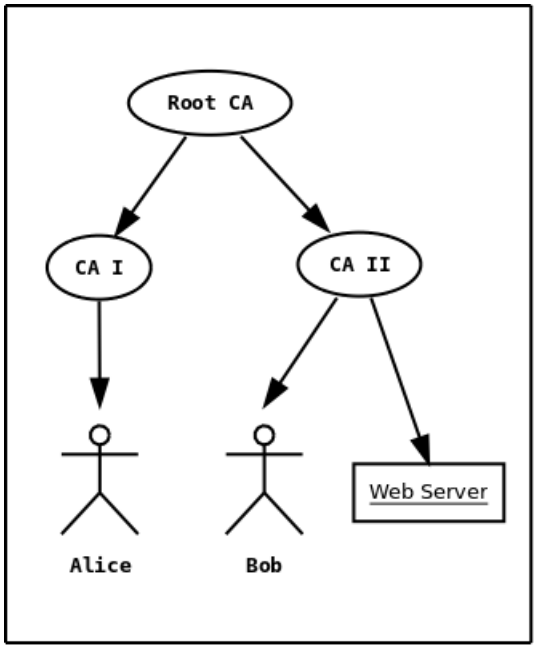


Figure 4.1: An example of the X.509 hierarchical trust model.

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.

### 4.2.1 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 4.1.

| 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 4.1: 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.

Certificate *extensions* are there to include information about the certificate's subject that did not fit in the typical certificate fields. 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 4.2.

| Extension | OID | Description |
|---|---|---|
| Subject key id | 2.5.29.14 | An identifier of the key of the subject. |
| Authority key id | 2.5.29.35 | An identifier of the authority's key used to sign the certificate. |
| Subject alternative name | 2.5.29.17 | Alternative names to subject's distinguished name. |
| Key usage | 2.5.29.15 | Constraints the key's usage of the certificate. |
| Extended key usage | 2.5.29.37 | Constraints the purpose of the certificate. |
| Basic constraints | 2.5.29.19 | Indicates whether this is a CA certificate or not, and specify the maximum path lengths of certificate chains. |
| CRL distribution points | 2.5.29.31 | This extension is set by the CA, in order to inform about the issued CRLs. |
| 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. Proxy Certificates are specified in [*RFC3820*]. |

Table 4.2: X.509 certificate extensions.

In GnuTLS the X.509 certificate structures are handled using the `gnutls_x509_crt_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 at [ex:x509-info], page 119.

### 4.2.2 Verifying X.509 certificate paths

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

- [gnutls_x509_trust_list_add_cas], page 271
- [gnutls_x509_trust_list_add_named_crt], page 272
- [gnutls_x509_trust_list_add_crls], page 272
- [gnutls_x509_trust_list_verify_crt], page 273
- [gnutls_x509_trust_list_verify_named_crt], page 274

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 ⟨undefined⟩ [gnutls_certificate_status_t], page ⟨undefined⟩.

An example of certificate verification is shown in [ex:verify2], page 86. 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_crt], page 272 and [gnutls_x509_trust_list_verify_named_crt], page 274.

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

When operating in the context of a TLS session, the trusted certificate authority list has been set via the [gnutls_certificate_set_x509_trust_file], page 159 and [gnutls_certificate_set_x509_crl_file], page 157, thus it is not required to setup a trusted list as above. Convenience functions such as [gnutls_certificate_verify_peers2], page 161 are equivalent and will verify the peer's certificate chain in a TLS session.

There is also the possibility to pass some input to the verification functions in the form of flags. For [gnutls_x509_trust_list_verify_crt], page 273 the flags are passed straightforward, but [gnutls_certificate_verify_peers2], page 161 depends on the flags set by calling [gnutls_certificate_set_verify_flags], page 156. All the available flags are part of the enumeration `gnutls_certificate_verify_flags` shown in ⟨undefined⟩ [gnutls_certificate_verify_flags], page ⟨undefined⟩.

Although the verification of a certificate path indicates that the certificate is signed by trusted authority, does not reveal anything about the peer's identity. It is required to verify if the certificate's owner is the one you expect. For more information consult [*RFC2818*] and section [ex:verify], page 69 for an example.

## 4.3 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 4.2, David trusts Alice to be an introducer and Alice signed Bob's key thus Dave trusts Bob's key to be the real one.

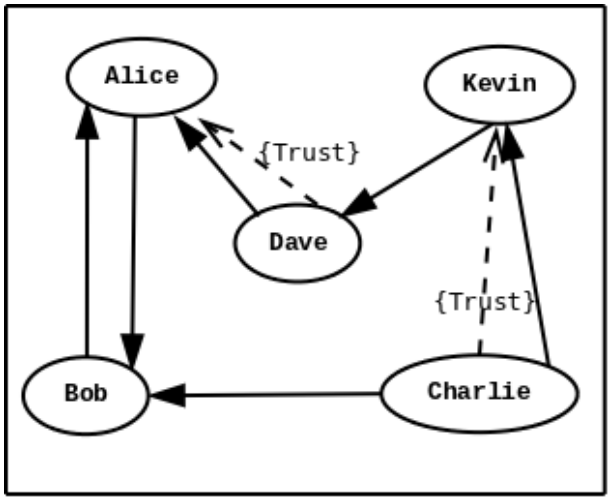


Figure 4.2: An example of the OpenPGP trust model.

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.

### 4.3.1 OpenPGP certificate structure

In GnuTLS the OpenPGP key structures [*RFC2440*] are handled using the `gnutls_openpgp_crt_t` type and the corresponding private keys with the `gnutls_openpgp_privkey_t` type. All the prototypes for the key handling functions can be found at '`gnutls/openpgp.h`'.

### 4.3.2 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[1] is recommended.

In GnuTLS there is a verification function for OpenPGP certificates, the [gnutls_openpgp_crt_verify_ring], page 285. 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 ⟨undefined⟩ [gnutls_certificate_verify_flags], page ⟨undefined⟩).

[1] http://www.gnupg.org/related_software/gpgme/

- [gnutls_openpgp_crt_verify_ring], page 285
- [gnutls_openpgp_crt_verify_self], page 285

### 4.3.3 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_peers2], page 161 to verify the signatures in the certificate sent by the peer.

- [gnutls_certificate_set_openpgp_keyring_file], page 276

## 4.4 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 € to Greenpeace and find out that he donated 1.000.000 € 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.

### 4.4.1 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 4.2.2 [Verifying X.509 certificate paths], page 19), 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_peers2], page 161 to verify the certificate chain, you can call [gnutls_certificate_set_verify_flags], page 156 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 tell the verifier algorithm to enable `RSA-MD5` when verifying the certificates.

If you are using [gnutls_x509_crt_verify], page 261 or [gnutls_x509_crt_list_verify], page 253, 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 154 to extract the raw server's certificate chain, [gnutls_x509_crt_list_import], page 252 to parse each of the certificates, and then [gnutls_x509_crt_get_signature_algorithm], page 249 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.

# 5 Shared-key and anonymous authentication

## 5.1 SRP authentication

### 5.1.1 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. The Stanford SRP libraries, include a PAM module that synchronizes the system's users passwords with the SRP password files. That way SRP authentication could be used for all users of a system.

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.

Helper functions are included in GnuTLS, used to generate and maintain SRP verifiers and password files. A program to manipulate the required parameters for SRP authentication is also included. See [srptool], page 24, for more information.

- [gnutls_srp_verifier], page 202
- [gnutls_srp_base64_encode], page 199
- [gnutls_srp_base64_decode], page 198

### 5.1.2 Invoking srptool

The '`srptool`' is a very simple program that emulates the programs in the *Stanford SRP libraries*[1]. It requires two files, one called `tpasswd` which holds usernames and verifiers, and `tpasswd.conf` which holds generators and primes.

To create tpasswd.conf which holds the generator and prime values for the SRP protocol, run:

[1] See `http://srp.stanford.edu/`.

```
$ 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 in a way that is compatible with libsrp.

```
$ 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
```

## 5.2 PSK authentication

### 5.2.1 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.

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

- [gnutls_key_generate], page 177
- [gnutls_hex_encode], page 176
- [gnutls_hex_decode], page 176

### 5.2.2 Invoking psktool

This is a program to manage PSK username and keys. It will generate random keys for the indicated username, using a simple password file format.

```
PSKtool help
Usage : psktool [options]
     -u, --username username
                                specify username.
     -p, --passwd FILE          specify a password file.
     -s, --keysize SIZE         specify the key size in bytes.
     -v, --version              prints the program's version number
     -h, --help                 shows this help text
```

The generation of a PSK password file is illustrated in the example below. The password is provided in the prompt.

```
$ ./psktool -u psk_identity -p psks.txt
Generating a random key for user 'psk_identity'
Key stored to psks.txt
```

```
$ cat psks.txt
psk_identity:88f3824b3e5659f52d00e959bacab954b6540344
$
```

## 5.3 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. Moreover it is useful when complete anonymity is required. Unless in one of the above cases, do not use anonymous authentication.

The available key exchange algorithms for anonymous authentication are shown below.

`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.

# 6 More on certificate authentication

## 6.1 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.

- [gnutls_x509_crq_set_version], page 233
- [gnutls_x509_crq_set_dn_by_oid], page 231
- [gnutls_x509_crq_set_key_usage], page 232
- [gnutls_x509_crq_set_key_purpose_oid], page 231
- [gnutls_x509_crq_set_basic_constraints], page 230

The [gnutls_x509_crq_set_key], page 231 and [gnutls_x509_crq_sign2], page 233 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 6.6 [Abstract key types], page 46.

- [gnutls_x509_crq_set_key], page 231
- [gnutls_x509_crq_sign2], page 233

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_NORMAL);
  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;

}
```

## 6.2 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 6.1. 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.

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. 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 6.1: Certificate revocation list fields.

- [gnutls_x509_crl_set_version], page 220
- [gnutls_x509_crl_set_crt_serial], page 219
- [gnutls_x509_crl_set_crt], page 219
- [gnutls_x509_crl_set_next_update], page 219
- [gnutls_x509_crl_set_this_update], page 220

The [gnutls_x509_crl_sign2], page 220 and [gnutls_x509_crl_privkey_sign], page 323 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.

- [gnutls_x509_crl_sign2], page 220
- [gnutls_x509_crl_privkey_sign], page 323

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

- [gnutls_x509_crl_set_number], page 219
- [gnutls_x509_crl_set_authority_key_id], page 218

## 6.3 Managing encrypted keys

Transferring or storing private keys in plain might not be a good idea. Any access on the keys becomes a fatal compromise. Storing the keys in hardware tokens (see Section 6.5 [Hardware tokens], page 40) could solve the storage problem but it is not always practical or efficient enough. This section describes alternative ways that involve encryption of the private keys to store and transfer.

There are two alternatives to use for key encryption, PKCS #8 and #12 methods of private key encryption. The PKCS #8 method only allows encryption of the private key, whilst the

PKCS #12 method allows in addition the bundling of other data into the structure. That could be bundling together the certificate as well as the trusted CA certificate.

### 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.

- [gnutls_x509_privkey_import_pkcs8], page 268
- [gnutls_x509_privkey_export_pkcs8], page 265

### 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.

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.

The following functions are available to read a PKCS #12 structure.

- [gnutls_pkcs12_get_bag], page 298
- [gnutls_pkcs12_verify_mac], page 299
- [gnutls_pkcs12_bag_decrypt], page 294
- [gnutls_pkcs12_bag_get_count], page 294
- [gnutls_pkcs12_bag_get_data], page 294
- [gnutls_pkcs12_bag_get_key_id], page 295
- [gnutls_pkcs12_bag_get_friendly_name], page 295

The functions below are used to generate a PKCS #12 structure. An example of their usage is also shown.

- [gnutls_pkcs12_set_bag], page 298
- [gnutls_pkcs12_bag_encrypt], page 294
- [gnutls_pkcs12_generate_mac], page 297
- [gnutls_pkcs12_bag_set_data], page 296
- [gnutls_pkcs12_bag_set_crl], page 296
- [gnutls_pkcs12_bag_set_crt], page 296
- [gnutls_pkcs12_bag_set_key_id], page 297
- [gnutls_pkcs12_bag_set_friendly_name], page 296

```
/* 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;
  FILE *fd;

  /* A good idea might be to use gnutls_x509_privkey_get_key_id()
   * to obtain a unique ID.
   */
  gnutls_datum_t key_id = { (char *) "\x00\x00\x07", 3 };

  gnutls_global_init ();

  /* Firstly we create two helper bags, which hold the certificate,
   * and the (encrypted) key.
   */

  gnutls_pkcs12_bag_init (&bag);
  gnutls_pkcs12_bag_init (&key_bag);

  ret = gnutls_pkcs12_bag_set_data (bag, GNUTLS_BAG_CERTIFICATE, cert);
  if (ret < 0)
    {
      fprintf (stderr, "ret: %s\n", gnutls_strerror (ret));
      return 1;
    }
```

```
/* ret now holds the bag's index.
 */
bag_index = ret;

/* Associate a friendly name with the given certificate. Used
 * by browsers.
 */
gnutls_pkcs12_bag_set_friendly_name (bag, bag_index, "My name");

/* Associate the certificate with the key using a unique key
 * ID.
 */
gnutls_pkcs12_bag_set_key_id (bag, bag_index, &key_id);

/* use weak encryption for the certificate.
 */
gnutls_pkcs12_bag_encrypt (bag, password, GNUTLS_PKCS_USE_PKCS12_RC2_40);

/* Now the key.
 */

ret = gnutls_pkcs12_bag_set_data (key_bag,
                                  GNUTLS_BAG_PKCS8_ENCRYPTED_KEY,
                                  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;
}
```

## 6.4 The certtool application

This is a program to generate X.509 certificates, certificate requests, CRLs and private keys.

```
Certtool help
Usage: certtool [options]
     -s, --generate-self-signed
                              Generate a self-signed certificate.
     -c, --generate-certificate
                              Generate a signed certificate.
     --generate-proxy         Generate a proxy certificate.
     --generate-crl           Generate a CRL.
```

```
-u, --update-certificate
                        Update a signed certificate.
-p, --generate-privkey  Generate a private key.
-q, --generate-request  Generate a PKCS #10 certificate
                        request.
-e, --verify-chain      Verify a PEM encoded certificate chain.
                        The last certificate in the chain must
                        be a self signed one.
--verify                Verify a PEM encoded certificate chain.
                        CA certificates must be loaded with
                        --load-ca-certificate.
--verify-crl            Verify a CRL.
--generate-dh-params    Generate PKCS #3 encoded Diffie-Hellman
                        parameters.
--get-dh-params         Get the included PKCS #3 encoded
                        Diffie-Hellman parameters.
--load-privkey FILE     Private key file to use.
--load-pubkey FILE      Public key file to use.
--load-request FILE     Certificate request file to use.
--load-certificate FILE
                        Certificate file to use.
--load-ca-privkey FILE  Certificate authority's private key
                        file to use.
--load-ca-certificate FILE
                        Certificate authority's certificate
                        file to use.
--password PASSWORD     Password to use.
-i, --certificate-info  Print information on a certificate.
--certificate-pubkey    Print certificate public key.
--pgp-certificate-info  Print information on a OpenPGP
                        certificate.
--pgp-ring-info         Print information on a keyring
                        structure.
-l, --crl-info          Print information on a CRL.
--crq-info              Print information on a Certificate
                        Request.
--no-crq-extensions     Do not use extensions in certificate
                        requests.
--p12-info              Print information on a PKCS #12
                        structure.
--p7-info               Print information on a PKCS #7
                        structure.
--smime-to-p7           Convert S/MIME to PKCS #7 structure.
-k, --key-info          Print information on a private key.
--pgp-key-info          Print information on a OpenPGP private
                        key.
--pubkey-info           Print information on a public key.
```

```
--fix-key                Regenerate the parameters in a private
                         key.
--v1                     Generate an X.509 version 1 certificate
                         (no extensions).
--to-p12                 Generate a PKCS #12 structure.
--to-p8                  Generate a PKCS #8 key structure.
-8, --pkcs8              Use PKCS #8 format for private keys.
--dsa                    Use DSA keys.
--ecc                    Use ECC (ECDSA) keys.
--hash STR               Hash algorithm to use for signing
                         (MD5,SHA1,RMD160,SHA256,SHA384,SHA512).
--export-ciphers         Use weak encryption algorithms.
--inder                  Use DER format for input certificates
                         and private keys.
--inraw                  Use RAW/DER format for input
                         certificates and private keys.
--outder                 Use DER format for output certificates
                         and private keys.
--outraw                 Use RAW/DER format for output
                         certificates and private keys.
--bits BITS              specify the number of bits for key
                         generation.
--sec-param PARAM        specify the security level
                         [low|normal|high|ultra].
--disable-quick-random   Use /dev/random for key generationg,
                         thus increasing the quality of
                         randomness used.
--outfile FILE           Output file.
--infile FILE            Input file.
--template FILE          Template file to use for non
                         interactive operation.
--pkcs-cipher CIPHER     Cipher to use for pkcs operations
                         (3des,3des-pkcs12,aes-128,aes-192,aes-25
                         6,rc2-40,arcfour).
-d, --debug LEVEL        specify the debug level. Default is 1.
-h, --help               shows this help text
-v, --version            shows the program's version
```

The program can be used interactively or non interactively by specifying the `--template` command line option. See below for an example of a template file.

### Diffie-Hellman parameter generation

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

```
$ certtool --generate-dh-params --outfile dh.pem
```

### Self-signed certificate generation

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.

### Private key generation

To create a private key (RSA by default), run:

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

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

### Certificate generation

To generate a certificate using the private key, 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
```

Alternatively you may create a certificate request, which is needed when the certificate will be signed by a third party authority.

```
$ 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 (see Section 6.5.6 [The p11tool application], page 44 on how to obtain the URL).

```
$ certtool --generate-request --load-privkey pkcs11:(PRIVKEY URL) \
  --load-pubkey pkcs11:(PUBKEY URL) --outfile request.pem
```

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
```

### 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
```

### 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'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 cert.pem --load-privkey key.pem  \
   --template cert.cfg \
   --load-ca-certificate ca-cert.pem --load-ca-privkey ca-key.pem
```

An example certtool template file:

```
# 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"

# 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." "2.5.4.65" "jackal"

# This is deprecated and should not be used in new
# certificates.
# pkcs9_email = "none@none.org"

# The serial number of the certificate
serial = 007

# In how many days, counting from today, this certificate will expire.
expiration_days = 700

# X.509 v3 extensions

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

# 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"

# 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

# 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
```

## 6.5 Security modules

### 6.5.1 Introduction

In this section we present the smart-card and hardware security module 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 allows decoupling cryptographic keys from the applications that use them providing an additional security layer. Since this can also be achieved in software components such as in Gnome keyring, we will use the term security module to describe such an isolation interface.

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[1], the trusted platform module (TPM)[2] 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.

[1] http://www.opensc-project.org

[2] http://trousers.sourceforge.net/

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 6.1. That way applications could load their trusted certificate list, as well as user certificates from a common PKCS #11 module. Such a provider exists in the Gnome system, being the Gnome Keyring.

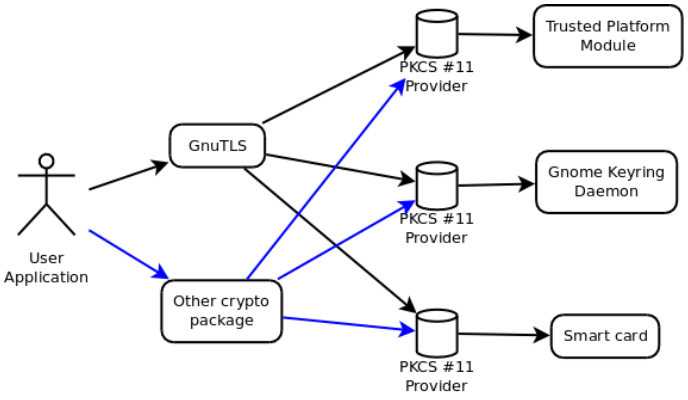


Figure 6.1: PKCS #11 module usage.

### 6.5.2 Initialization

To allow all the GnuTLS applications to access PKCS #11 tokens you can use a configuration per module, stored in `/etc/pkcs11/modules/`. These are the configuration files of p11-kit[3]. For example a file that will load the OpenSC module, could be named `/etc/pkcs11/modules/opensc` and contain the following:

```
module: /usr/lib/opensc-pkcs11.so
```

If you use this file, then there is no need for other initialization in GnuTLS, except for the PIN and token functions. Those allow retrieving a PIN when accessing a protected object, such as a private key, as well as probe the user to insert the token. All the initialization functions are below.

- [gnutls_pkcs11_init], page 301
- [gnutls_pkcs11_set_token_function], page 305
- [gnutls_pkcs11_set_pin_function], page 305
- [gnutls_pkcs11_add_provider], page 299

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.

### 6.5.3 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:

[3] http://p11-glue.freedesktop.org/

```
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.

- [gnutls_pkcs11_obj_import_url], page 302
- [gnutls_pkcs11_obj_export_url], page 302
- [gnutls_pkcs11_obj_get_info], page 302
- [gnutls_x509_crt_import_pkcs11], page 308
- [gnutls_x509_crt_import_pkcs11_url], page 308
- [gnutls_x509_crt_list_import_pkcs11], page 308

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.

- [gnutls_pkcs11_token_init], page 307
- [gnutls_pkcs11_token_get_url], page 306
- [gnutls_pkcs11_token_get_info], page 306
- [gnutls_pkcs11_token_get_flags], page 306
- [gnutls_pkcs11_token_set_pin], page 307

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();
```

```
#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 i, ret;

  obj_list_size = 0;
  ret = gnutls_pkcs11_obj_list_import_url (NULL, &obj_list_size, URL,
                                           GNUTLS_PKCS11_OBJ_ATTR_CRT_WITH_PRIVKEY,
                                           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_CRT_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_CRT_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;
}
```

### 6.5.4 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).

- [gnutls_pkcs11_copy_x509_privkey], page 300
- [gnutls_pkcs11_copy_x509_crt], page 300
- [gnutls_pkcs11_delete_url], page 300

### 6.5.5 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 89. 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.

- [gnutls_certificate_set_x509_trust_file], page 159
- [gnutls_certificate_set_x509_key_file], page 158

### 6.5.6 The p11tool application

p11tool is a program that is used to access tokens and security modules that support the PKCS #11 API. It requires individual PKCS #11 modules to be loaded either with the `--provider` option, or by setting up the GnuTLS configuration file for PKCS #11 as in Section 6.5 [Hardware tokens], page 40.

```
p11tool help
Usage: p11tool [options]
Usage: p11tool --list-tokens
Usage: p11tool --list-all
Usage: p11tool --export 'pkcs11:...'

     --export URL              Export an object specified by a pkcs11
                               URL
     --list-tokens             List all available tokens
     --list-mechanisms URL     List all available mechanisms in token.
     --list-all                List all objects specified by a PKCS#11
                               URL
     --list-all-certs          List all certificates specified by a
                               PKCS#11 URL
     --list-certs              List certificates that have a private
                               key specified by a PKCS#11 URL
     --list-privkeys           List private keys specified by a
                               PKCS#11 URL
     --list-trusted            List certificates marked as trusted,
                               specified by a PKCS#11 URL
```

```
--initialize URL          Initializes a PKCS11 token.
--write URL               Writes loaded certificates, private or
                          secret keys to a PKCS11 token.
--delete URL              Deletes objects matching the URL.
--label label             Sets a label for the write operation.
--trusted                 Marks the certificate to be written as
                          trusted.
--private                 Marks the object to be written as
                          private (requires PIN).
--no-private              Marks the object to be written as not
                          private.
--login                   Force login to token
--detailed-url            Export detailed URLs.
--no-detailed-url         Export less detailed URLs.
--secret-key HEX_KEY      Provide a hex encoded secret key.
--load-privkey FILE       Private key file to use.
--load-pubkey FILE        Private key file to use.
--load-certificate FILE
                          Certificate file to use.
-8, --pkcs8               Use PKCS #8 format for private keys.
--inder                   Use DER format for input certificates
                          and private keys.
--inraw                   Use RAW/DER format for input
                          certificates and private keys.
--provider Library        Specify the pkcs11 provider library
--outfile FILE            Output file.
-d, --debug LEVEL         specify the debug level. Default is 1.
-h, --help                shows this help text
```

After being provided the available PKCS #11 modules, it can list all tokens available in your system, the objects on the tokens, and perform operations on them.

Some examples on how to use p11tool are illustrated in the following paragraphs.

### List all tokens

```
$ p11tool --list-tokens
```

### List all objects

The following command will list all objects in a token. The `--login` is required to show objects marked as private.

```
$ p11tool --login --list-all
```

### Exporting an object

To retrieve an object stored in the card use the following command. Note however that objects marked as sensitive (typically PKCS #11 private keys) are not allowed to be extracted from the token.

```
$ p11tool --login --export [OBJECT URL]
```

### Copy an object to a token

To copy an object, such as a certificate or private key to a token use the following command.

```
$ p11tool --login --write [TOKEN URL] \
  --load-certificate cert.pem --label "my_cert"
```

## 6.6 Abstract key types

Since there are many forms of a public or private keys supported by GnuTLS such as X.509, OpenPGP, or PKCS #11 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 must be used.

```
#inlude <gnutls/abstract.h>
#inlude <gnutls/pkcs11.h>

void sign_cert( gnutls_x509_crt_t to_be_signed)
{
gnutls_pkcs11_privkey_t ca_key;
gnutls_x509_crt_t ca_cert;
gnutls_privkey_t abs_key;

  /* load the PKCS #11 key and certificates */
  gnutls_pkcs11_privkey_init(&ca_key);
  gnutls_pkcs11_privkey_import_url(ca_key, key_url);

  gnutls_x509_crt_init(&ca_cert);
  gnutls_x509_crt_import_pkcs11_url(&ca_cert, cert_url);

  /* initialize the abstract key */
  gnutls_privkey_init(&abs_key);
  gnutls_privkey_import_pkcs11(abs_key, ca_key);

  /* sign the certificate to be signed */
  gnutls_x509_crt_privkey_sign(to_be_signed, ca_cert, ca_key,
                               GNUTLS_DIG_SHA256, 0);
}
```

### 6.6.1 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_crt_t`, or through an ASN.1 encoding of the X.509 `SubjectPublicKeyInfo` sequence.

- [gnutls_pubkey_import_x509], page 322
- [gnutls_pubkey_import_openpgp], page 320
- [gnutls_pubkey_import_pkcs11], page 320
- [gnutls_pubkey_import_pkcs11_url], page 321

- [gnutls_pubkey_import_privkey], page 321

Additional functions are available that will return information over a public key.

- [gnutls_pubkey_get_pk_algorithm], page 317
- [gnutls_pubkey_get_preferred_hash_algorithm], page 318
- [gnutls_pubkey_get_key_id], page 315

### 6.6.2 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 PKCS #11 keys that are not extractable.

- [gnutls_privkey_import_x509], page 313
- [gnutls_privkey_import_openpgp], page 313
- [gnutls_privkey_import_pkcs11], page 313
- [gnutls_privkey_import_ext], page 312
- [gnutls_privkey_get_pk_algorithm], page 312
- [gnutls_privkey_get_type], page 312

### 6.6.3 Operations

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

- [gnutls_pubkey_verify_data2], page 323
- [gnutls_pubkey_verify_hash], page 323
- [gnutls_privkey_sign_data], page 314
- [gnutls_privkey_sign_hash], page 314

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.

- [gnutls_x509_crq_set_pubkey], page 324
- [gnutls_x509_crt_set_pubkey], page 325
- [gnutls_x509_crt_privkey_sign], page 324
- [gnutls_x509_crl_privkey_sign], page 323
- [gnutls_x509_crq_privkey_sign], page 324

# 7 How to use GnuTLS in applications

## 7.1 Introduction

### 7.1.1 General idea

A brief description of how GnuTLS works internally is shown at Figure 7.1. This section may become more clear after having read the rest of this section. 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, and structures needed for the ASN.1 parser. This structure is never modified by any GnuTLS function, except for the deinitialization function which frees all allocated memory and is called after the program has permanently finished using GnuTLS.

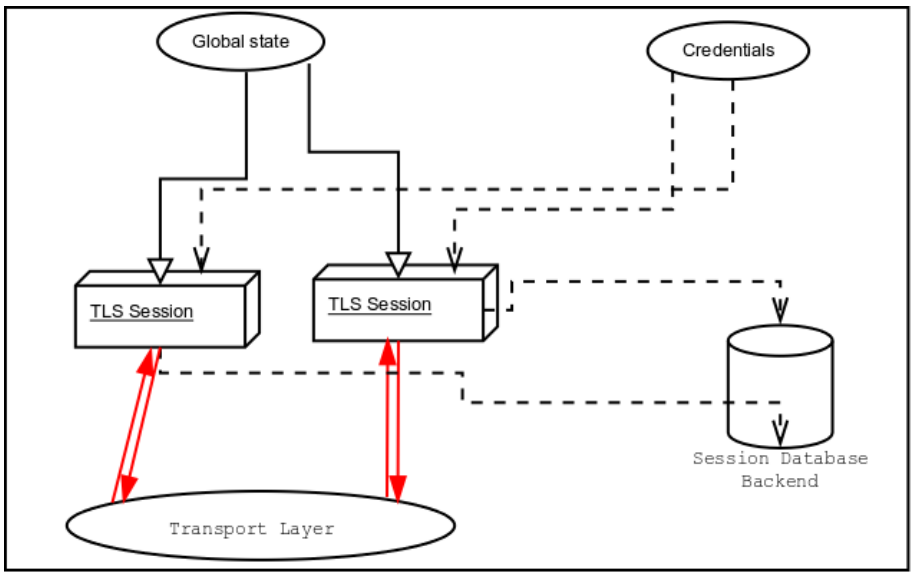


Figure 7.1: High level design of GnuTLS.

The credentials structures are used by the authentication methods, such as certificate authentication. They store certificates, privates keys, and other information that is needed to prove the identity to the peer, and/or verify the indentity 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 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 11) to store the newly negotiated session. The session

database is examined by the server just after having received the client hello[1], 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.

### 7.1.2 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 190. 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 171.

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. See Appendix B [Error codes], page 140, for a description of the available error codes.

### 7.1.3 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 173 and [gnutls_global_set_log_function], page 172 are to be used. Those will print verbose information on the GnuTLS functions internal flow.

- [gnutls_global_set_log_level], page 173
- [gnutls_global_set_log_function], page 172

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 172. The provided function will receive an message and the corresponding TLS session. The session information might be used to derive IP addresses or other information about the peer involved.

- [gnutls_global_set_audit_log_function], page 172

### 7.1.4 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 (see Section 3.5.2 [Authentication], page 9), is also thread-safe.

The random generator of the cryptographic back-end, is not thread safe and requires mutex locks which are setup by GnuTLS. Applications can either call [gnutls_global_init],

[1] The first message in a TLS handshake

page 172 which will initialize the default operating system provided locks (i.e. `pthreads` on GNU/Linux and `CriticalSection` on Windows), or specify manually the locking system using the function [gnutls_global_set_mutex], page 173 before calling [gnutls_global_init], page 172. Setting manually mutexes is recommended only to applications that have full control of the underlying libraries. If this is not the case, the use of the operating system defaults is recommended. An example of non-native thread usage is shown below.

```
#include <gnutls.h>

/* Other thread packages
 */

int main()
{
   gnutls_global_set_mutex (mutex_init, mutex_deinit,
                            mutex_lock, mutex_unlock);
   gnutls_global_init();
}
```

- [gnutls_global_set_mutex], page 173

### 7.1.5 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.

- [gnutls_transport_set_push_function], page 205
- [gnutls_transport_set_pull_function], page 204

Other callback functions may require more complicated input and data to be allocated. Such an example is [gnutls_srp_set_server_credentials_function], page 201. All callbacks should allocate and free memory using `gnutls_malloc` and `gnutls_free`.

## 7.2 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.

### 7.2.1 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.

### 7.2.2 Initialization

GnuTLS must be initialized before it can be used. The library is initialized by calling [gnutls_global_init], page 172. The resources allocated by the initialization process can be released if the application no longer has a need to call GnuTLS functions, this is done by calling [gnutls_global_deinit], page 172.

In order to take advantage of the internationalization features in GnuTLS, such as translated error messages, the application must set the current locale using `setlocale` before initializing GnuTLS.

### 7.2.3 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 161.

### 7.2.4 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 '`-ltasn1`' 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`
```

## 7.3 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 177 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).

- [gnutls_init], page 177

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 184. We

elaborate on them in Section 7.8 [Priority Strings], page 58. 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 164.

- [gnutls_credentials_set], page 164

## 7.4 Associating the credentials

### 7.4.1 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. On 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.

- [gnutls_certificate_allocate_credentials], page 152
- [gnutls_certificate_free_credentials], page 153

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.

- [gnutls_certificate_set_x509_key_mem], page 158
- [gnutls_certificate_set_x509_key], page 158
- [gnutls_certificate_set_x509_key_file], page 158
- [gnutls_certificate_set_openpgp_key_mem], page 275
- [gnutls_certificate_set_openpgp_key], page 274
- [gnutls_certificate_set_openpgp_key_file], page 275
- [gnutls_certificate_set_key], page 309

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 197. Both functions are shown below.

- [gnutls_certificate_set_retrieve_function], page 155
- [gnutls_sign_algorithm_get_requested], page 197

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.

- [gnutls_certificate_set_x509_trust_file], page 159
- [gnutls_certificate_set_openpgp_keyring_file], page 276

Note however that the peer's certificate is not automatically verified, you should call [gnutls_certificate_verify_peers2], page 161, after a successful handshake or during if [gnutls_certificate_set_verify_function], page 156 has been used, to verify the certificate's signature. An alternative way, which reports a more detailed verification output, is to use [gnutls_certificate_get_peers], page 154 to obtain the raw certificate of the peer and verify it using the functions discussed in Section 4.2 [X.509 certificates], page 17.

- [gnutls_certificate_verify_peers2], page 161

In a handshake, the negotiated cipher suite also depends on the certificate's parameters, so some key exchange methods might not be available with some 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 3.3 are available in certificate authentication.

- [gnutls_certificate_set_verify_function], page 156

Note that the DHE key exchange methods are generally slower[2] than the 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 7.9.2 [Parameter generation], page 62).

### 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 154. 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 154. This however will have the side-effect of not restricting the client to certificates signed by server's acceptable signers.

- [gnutls_certificate_server_set_request], page 154
- [gnutls_certificate_send_x509_rdn_sequence], page 154

## 7.4.2 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 200 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.

- [gnutls_srp_allocate_server_credentials], page 198
- [gnutls_srp_allocate_client_credentials], page 198

[2] It depends on the group used. Primes with lesser bits are always faster, but also easier to break. See Section 7.11 [Selecting cryptographic key sizes], page 65 for the acceptable security levels.

- [gnutls_srp_free_server_credentials], page 200
- [gnutls_srp_free_client_credentials], page 199
- [gnutls_srp_set_client_credentials], page 200
- [gnutls_srp_set_client_credentials_function], page 200

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 201 should be called, to set an appropriate callback.

- [gnutls_srp_set_server_credentials_file], page 201
- [gnutls_srp_set_server_credentials_function], page 201

### 7.4.3 PSK

The initialization functions in PSK credentials differ between client and server.

- [gnutls_psk_allocate_server_credentials], page 186
- [gnutls_psk_allocate_client_credentials], page 185
- [gnutls_psk_free_server_credentials], page 186
- [gnutls_psk_free_client_credentials], page 186

Clients supporting PSK should supply the username and key before a TLS session is established. Alternatively [gnutls_psk_set_client_credentials_function], page 187 can be used to specify a callback function. This has the advantage that the callback will be called only if PSK has been negotiated.

- [gnutls_psk_set_client_credentials], page 187
- [gnutls_psk_set_client_credentials_function], page 187

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 188. 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 188.

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 188. The client can retrieve the hint, for example in the callback function, using [gnutls_psk_client_get_hint], page 186.

- [gnutls_psk_set_server_credentials_file], page 188
- [gnutls_psk_set_server_credentials_function], page 188
- [gnutls_psk_set_server_credentials_hint], page 188
- [gnutls_psk_client_get_hint], page 186

### 7.4.4 Anonymous

The initialization functions for the credentials are shown below.

- [gnutls_anon_allocate_server_credentials], page 149
- [gnutls_anon_allocate_client_credentials], page 149
- [gnutls_anon_free_server_credentials], page 150
- [gnutls_anon_free_client_credentials], page 149

Note that the key exchange methods for anonymous authentication require Diffie-Hellman parameters to be generated by the server and associated with an anonymous credentials structure. Check Section 7.9.2 [Parameter generation], page 62 for more information.

## 7.5 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_ptr2], page 204 would be sufficient to specify the socket descriptor.

- [gnutls_transport_set_ptr2], page 204
- [gnutls_transport_set_ptr], page 204

If however another transport layer than TCP is selected, then the following functions have to be specified.

- [gnutls_transport_set_push_function], page 205
- [gnutls_transport_set_vec_push_function], page 205
- [gnutls_transport_set_pull_function], page 204

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 203 with the intended errno value instead of setting `errno` directly.

- [gnutls_transport_set_errno], page 203

GnuTLS currently only interprets the EINTR and EAGAIN errno values and returns the corresponding GnuTLS error codes:

- `GNUTLS_E_INTERRUPTED`
- `GNUTLS_E_AGAIN`

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.

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. This requires the [gnutls_transport_set_pull_timeout_function], page 205 function to be used.

- [gnutls_transport_set_pull_timeout_function], page 205

### 7.5.1 Asynchronous operation

GnuTLS can be used with asynchronous socket or event-driven programming. During a TLS protocol session GnuTLS does not block for anything except calculations. The only blocking operations are due to the transport layer (sockets) functions. Those, however, in an asynchronous scenario are typically set to non-blocking mode, which forces them to return `EAGAIN` error code instead of blocking. In that case GnuTLS functions will return the `GNUTLS_E_AGAIN` error code and can be resumed the same way as a system call would. The only exception is [gnutls_record_send], page 191, 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 the session is read completely. That can be achieved by checking there are no data waiting to be read (using [gnutls_record_check_pending], page 189), either before the `select` system call, or after a call to [gnutls_record_recv], page 190. GnuTLS does not keep a write buffer, thus when writing `select` need only to be consulted.

In the DTLS, however, GnuTLS might block due to timers required by the protocol. To prevent those timers from blocking a DTLS handshake, the [gnutls_init], page 177 should be called with the `GNUTLS_NONBLOCK` flag (see Section 7.3 [Session initialization], page 51).

### 7.5.2 DTLS sessions

Because datagram TLS can operate over connections where the peer of a server cannot be reliably verified, functionality is available to prevent denial of service attacks. GnuTLS requires a server to generate a secret key that is used to sign a cookie[3]. That cookie is sent to the client using [gnutls_dtls_cookie_send], page 206, 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 206. If successful the session should be initialized and associated with the cookie using [gnutls_dtls_prestate_set], page 207, before proceeding to the handshake.

- [gnutls_key_generate], page 177
- [gnutls_dtls_cookie_send], page 206
- [gnutls_dtls_cookie_verify], page 206
- [gnutls_dtls_prestate_set], page 207

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 174.

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*].

[3] A key of 128 bits or 16 bytes should be sufficient for this purpose.

- [gnutls_dtls_set_mtu], page 207
- [gnutls_dtls_get_mtu], page 207
- [gnutls_dtls_get_data_mtu], page 206

## 7.6 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.

- [gnutls_handshake], page 174

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 156 has been used. In both cases the [gnutls_certificate_verify_peers2], page 161 function can be used to verify the peer's certificate (see Chapter 4 [Certificate authentication], page 17 for more information).

- [gnutls_certificate_verify_peers2], page 161

## 7.7 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 171 to check whether the error codes returned by these functions are fatal for the protocol or can be ignored.

- [gnutls_record_send], page 191
- [gnutls_record_recv], page 190
- [gnutls_error_is_fatal], page 171

In DTLS it is adviceable to use the extended receive function shown below, because it allows the extraction of the sequence number. This is required in DTLS because messages may arrive out of order.

- [gnutls_record_recv_seq], page 190

The [gnutls_record_check_pending], page 189 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 189 reports no data to be read, `select` should be called to check for data in the network buffers.

- [gnutls_record_check_pending], page 189
- [gnutls_record_get_direction], page 189

Once a TLS or DTLS session is no longer needed, it is recommended to use [gnutls_bye], page 151 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 166 function.

- [gnutls_bye], page 151
- [gnutls_deinit], page 166

## 7.8 Priority strings

In order to specify cipher suite preferences on a TLS session there are priority functions that accept a string specifying the enabled for the handshake algorithms. That string may contain a high level keyword such as in Table 7.1 or combination of a high level keyword, additional algorithm keywords and special keywords.

- [gnutls_priority_set_direct], page 184
- [gnutls_priority_set], page 184

| Keyword | Description |
|---|---|
| PERFORMANCE | All the "secure" ciphersuites are enabled, limited to 128 bit ciphers and sorted by terms of speed performance. |
| NORMAL | Means all "secure" ciphersuites. 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. |
| SECURE256 | Currently alias for SECURE192. |
| SUITEB128 | Means all the NSA Suite B cryptography (RFC5430) ciphersuites with an 128 bit security level. |
| SUITEB192 | Means all the NSA Suite B cryptography (RFC5430) ciphersuites with an 192 bit security level. |
| 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. It should be followed by the algorithms to be enabled. |

Table 7.1: Supported priority string keywords.

Unless the first 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 (except for the RSA-EXPORT which is only enabled in EXPORT level). The NONE keyword, if used, must followed by the algorithms to be enabled, and

is used to provide the exact list of requested algorithms[4]. The order with which every algorithm is specified is significant. Similar algorithms specified before others will take precedence. The individual algorithms are shown in Table 7.2 and special keywords are in Table 7.3. The prefixes for individual algorithms are:

'!' 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, RSA-EXPORT. The Catch all name is KX-ALL which will add all the algorithms from NORMAL priority. |
| MAC | MD5, SHA1, SHA256, 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. Catch all is VERS-TLS-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-SECP224R1, CURVE-SECP256R1, CURVE-SECP384R1, CURVE-SECP521R1. Catch all is CURVE-ALL. |

Table 7.2: The supported algorithm keywords in priority strings.

[4] To avoid collisions in order to specify a compression algorithm in this 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.

| 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. |
| %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. |
| %DISABLE_SAFE_RENEGOTIATION | will disable safe renegotiation completely. Do not use unless you know what you are doing. Testing purposes only. |
| %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. |
| %SSL3_RECORD_VERSION | will use SSL3.0 record version in client hello. This is the default. |

## 7.9 Advanced and other topics

### 7.9.1 Session resumption

#### Client side

To reduce time and roundtrips spent in a handshake the client can utilize session resumption. This requires the client to retrieve and store the session parameters. On new sessions to the same server the parameters must be re-associated with sessions using [gnutls_session_set_data], page 195.

- [gnutls_session_get_data], page 194
- [gnutls_session_get_id], page 195
- [gnutls_session_set_data], page 195

Keep in mind that sessions might be expired after some time, and it may be normal for a server not to resume a session even it was requested. That 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.

It is highly recommended clients to enable the session ticket extension using [gnutls_session_ticket_enable_client], page 196 in order to allow resumption with servers that do not store any state.

- [gnutls_session_ticket_enable_client], page 196

#### Server side

In order to support resumption a server might do it either by storing the session security parameters in a local database or by using session tickets (see Section 3.6.3 [Session tickets], page 12) to delegate storage to the client. Because session tickets might not be supported by all clients, servers might 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 164 for expiration.

- [gnutls_db_set_retrieve_function], page 166
- [gnutls_db_set_store_function], page 166
- [gnutls_db_set_ptr], page 165
- [gnutls_db_set_remove_function], page 165
- [gnutls_db_check_entry], page 164

A server utilizing tickets should use [gnutls_session_ticket_key_generate], page 196 to generate a ticket encryption key and call [gnutls_session_ticket_enable_server], page 196 to enable the extension.

- [gnutls_session_ticket_enable_server], page 196
- [gnutls_session_ticket_key_generate], page 196

### 7.9.2 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 be generated on the fly using [gnutls_dh_params_generate2], page 169 or imported from pregenerated data using [gnutls_dh_params_import_pkcs3], page 169. The parameters can be used in a TLS session by calling [gnutls_certificate_set_dh_params], page 155 or [gnutls_anon_set_server_dh_params], page 150 for anonymous sessions.

- [gnutls_dh_params_generate2], page 169
- [gnutls_dh_params_import_pkcs3], page 169
- [gnutls_certificate_set_dh_params], page 155
- [gnutls_anon_set_server_dh_params], page 150

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 192 (see Section 7.11 [Selecting cryptographic key sizes], page 65) to determine the bit size of the generated parameters.

The ciphersuites that involve the RSA-EXPORT key exchange require additional parameters. Those ciphersuites are rarely used today because they are by design insecure, thus if you have no requirement for them, the rest of this section can be skipped. The RSA-EXPORT key exchange requires 512-bit RSA keys to be generated. It is recommended those parameters to be refreshed (regenerated) in short intervals. The following functions can be used for these parameters.

- [gnutls_rsa_params_generate2], page 335
- [gnutls_certificate_set_rsa_export_params], page 331
- [gnutls_rsa_params_import_pkcs1], page 335
- [gnutls_rsa_params_export_pkcs1], page 334

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.

- [gnutls_certificate_set_params_function], page 155

### 7.9.3 Keying material exporters

The TLS PRF can be used by other protocols to derive data. The API to use is [gnutls_prf], page 182. The function needs to be provided with the label in the parameter `label`, and the extra data to mix in the `extra` parameter. Depending on whether you want to mix in the client or server random data first, you can set the `server_random_first` parameter.

For example, after establishing a TLS session using [gnutls_handshake], page 174, you can invoke the TLS PRF with this call:

```
#define MYLABEL "EXPORTER-FOO"
#define MYCONTEXT "some context data"
char out[32];
rc = gnutls_prf (session, strlen (MYLABEL), MYLABEL, 0,
                 strlen (MYCONTEXT), MYCONTEXT, 32, out);
```

If you don't want to mix in the client/server random, there is a more low-level TLS PRF interface called [gnutls_prf_raw], page 182.

### 7.9.4 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 194 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");
    }
}
```

### 7.9.5 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 7.8 [Priority Strings], page 58). An example priority string that is known to provide wide compatibility even with broken peers is shown below:

```
NORMAL:-VERS-TLS-ALL:+VERS-TLS1.0:+VERS-SSL3.0:%COMPAT
```

This priority string will only enable SSL 3.0 and TLS 1.0 as protocols and will disable, via the `%COMPAT` keyword, several TLS protocol options that are known to cause compatibility

problems. Note however that there are known attacks against those protocol versions and this mode trades security for compatibility.

### 7.9.6 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.

## 7.10 Using the 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 135), providing symmetric crypto, hash and HMAC algorithms, as well access to the random number generation.

### 7.10.1 Symmetric cryptography

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.

- [gnutls_cipher_init], page 327
- [gnutls_cipher_encrypt2], page 327
- [gnutls_cipher_decrypt2], page 326
- [gnutls_cipher_set_iv], page 327
- [gnutls_cipher_deinit], page 326

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.

- [gnutls_cipher_add_auth], page 325
- [gnutls_cipher_tag], page 328

### 7.10.2 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.

- [gnutls_hmac_init], page 330
- [gnutls_hmac], page 329
- [gnutls_hmac_output], page 331
- [gnutls_hmac_deinit], page 330
- [gnutls_hmac_get_len], page 330

- [gnutls_hmac_fast], page 330

The available functions to access hash functions are shown below. The supported hash functions are the same as the HMAC algorithms.

- [gnutls_hash_init], page 329
- [gnutls_hash], page 328
- [gnutls_hash_output], page 329
- [gnutls_hash_deinit], page 328
- [gnutls_hash_get_len], page 329
- [gnutls_hash_fast], page 328

### 7.10.3 Random number generation

Access to the random number generator is provided using the [gnutls_rnd], page 331 function. It allows obtaining random data of various levels.

- [gnutls_rnd], page 331

## 7.11 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 7.4 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 | 816 | 128 | `WEAK` | Very short term protection against small organizations |
| 80 | 1248 | 160 | `LOW` | Very short term protection against agencies |
| 112 | 2432 | 224 | `NORMAL` | Medium-term protection |
| 128 | 3248 | 256 | `HIGH` | Long term protection |
| 256 | 15424 | 512 | `ULTRA` | Foreseeable future |

Table 7.4: 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:

- [gnutls_sec_param_to_pk_bits], page 192
- [gnutls_pk_bits_to_sec_param], page 181

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.

# 8 GnuTLS application examples

## 8.1 Client examples

This section contains examples of TLS and SSL clients, using GnuTLS. Note that these examples contain little or no error checking. Some of the examples require functions implemented by another example.

### 8.1.1 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 is integrity and privacy protected.

```
/* 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_ptr (session, (gnutls_transport_ptr_t) sd);

/* Perform the TLS handshake
 */
ret = gnutls_handshake (session);

if (ret < 0)
  {
    fprintf (stderr, "*** Handshake failed\n");
    gnutls_perror (ret);
    goto end;
  }
else
  {
    printf ("- Handshake was completed\n");
  }

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_anon_free_client_credentials (anoncred);

  gnutls_global_deinit ();

  return 0;
}
```

### 8.1.2 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.
```

```
 */

#define MAX_BUF 1024
#define CAFILE "ca.pem"
#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;

  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);

  /* Initialize TLS session
   */
  gnutls_init (&session, GNUTLS_CLIENT);

  gnutls_session_set_ptr (session, (void *) "my_host_name");

  /* 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);

  /* connect to the peer
   */
  sd = tcp_connect ();

  gnutls_transport_set_ptr (session, (gnutls_transport_ptr_t) sd);

  /* Perform the TLS handshake
   */
  ret = gnutls_handshake (session);

  if (ret < 0)
    {
      fprintf (stderr, "*** Handshake failed\n");
      gnutls_perror (ret);
      goto end;
    }
  else
    {
      printf ("- Handshake was completed\n");
    }

  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_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;
  const gnutls_datum_t *cert_list;
  unsigned int cert_list_size;
  int ret;
  gnutls_x509_crt_t cert;
  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_peers2 (session, &status);
  if (ret < 0)
    {
      printf ("Error\n");
      return GNUTLS_E_CERTIFICATE_ERROR;
    }

  if (status & GNUTLS_CERT_INVALID)
    printf ("The certificate is not trusted.\n");

  if (status & GNUTLS_CERT_SIGNER_NOT_FOUND)
    printf ("The certificate hasn't got a known issuer.\n");

  if (status & GNUTLS_CERT_REVOKED)
    printf ("The certificate has been revoked.\n");
```

```
if (status & GNUTLS_CERT_EXPIRED)
  printf ("The certificate has expired\n");

if (status & GNUTLS_CERT_NOT_ACTIVATED)
  printf ("The certificate is not yet activated\n");

/* Up to here the process is the same for X.509 certificates and
 * OpenPGP keys. From now on X.509 certificates are assumed. This can
 * be easily extended to work with openpgp keys as well.
 */
if (gnutls_certificate_type_get (session) != GNUTLS_CRT_X509)
  return GNUTLS_E_CERTIFICATE_ERROR;

if (gnutls_x509_crt_init (&cert) < 0)
  {
    printf ("error in initialization\n");
    return GNUTLS_E_CERTIFICATE_ERROR;
  }

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;
  }

/* This is not a real world example, since we only check the first
 * certificate in the given chain.
 */
if (gnutls_x509_crt_import (cert, &cert_list[0], GNUTLS_X509_FMT_DER) < 0)
  {
    printf ("error parsing certificate\n");
    return GNUTLS_E_CERTIFICATE_ERROR;
  }


if (!gnutls_x509_crt_check_hostname (cert, hostname))
  {
    printf ("The certificate's owner does not match hostname '%s'\n",
            hostname);
    return GNUTLS_E_CERTIFICATE_ERROR;
  }

gnutls_x509_crt_deinit (cert);

/* notify gnutls to continue handshake normally */
return 0;
```

```
}
```

### 8.1.3 Simple datagram TLS client example

This is a client that uses UDP to connect to a server. This is the DTLS equivalent to the example in Section 8.1.2 [Simple client example with X.509 certificate support], page 69.

```
/* 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/dtls.h>

/* A very basic Datagram TLS client, over UDP with X.509 authentication.
 */

#define MAX_BUF 1024
#define CAFILE "ca.pem"
#define MSG "GET / HTTP/1.0\r\n\r\n"

extern int udp_connect (void);
extern void udp_close (int sd);
extern 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;

  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);

/* 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);

/* connect to the peer */
sd = udp_connect ();

gnutls_transport_set_ptr (session, (gnutls_transport_ptr_t) sd);

/* set the connection MTU */
gnutls_dtls_set_mtu (session, 1000);

/* Perform the TLS handshake */
ret = gnutls_handshake (session);

if (ret < 0)
  {
    fprintf (stderr, "*** Handshake failed\n");
    gnutls_perror (ret);
    goto end;
  }
else
  {
    printf ("- Handshake was completed\n");
  }

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);

  /* 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;
}
```

### 8.1.4 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 174.

```
/* 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

    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;
}
```

### 8.1.5 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 "ca.pem"

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 crt;
gnutls_privkey_t key;

/* Helper functions to load a certificate and key
 * files into memory.
 */
static gnutls_datum_t
load_file (const char *file)
{
  FILE *f;
  gnutls_datum_t loaded_file = { NULL, 0 };
  long filelen;
  void *ptr;

  if (!(f = fopen (file, "r"))
```

```
      || fseek (f, 0, SEEK_END) != 0
      || (filelen = ftell (f)) < 0
      || fseek (f, 0, SEEK_SET) != 0
      || !(ptr = malloc ((size_t) filelen))
      || fread (ptr, 1, (size_t) filelen, f) < (size_t) filelen)
    {
      return loaded_file;
    }

  loaded_file.data = ptr;
  loaded_file.size = (unsigned int) filelen;
  return loaded_file;
}

static void
unload_file (gnutls_datum_t data)
{
  free (data.data);
}

/* Load the certificate and the private key.
 */
static void
load_keys (void)
{
  int ret;
  gnutls_datum_t data;
  gnutls_x509_privkey_t x509_key;

  data = load_file (CERT_FILE);
  if (data.data == NULL)
    {
      fprintf (stderr, "*** Error loading certificate file.\n");
      exit (1);
    }

  ret = gnutls_pcert_import_x509_raw (&crt, &data, GNUTLS_X509_FMT_PEM, 0);
  if (ret < 0)
    {
      fprintf (stderr, "*** Error loading certificate file: %s\n",
               gnutls_strerror (ret));
      exit (1);
    }

  unload_file (data);

  data = load_file (KEY_FILE);
```

```
  if (data.data == NULL)
    {
      fprintf (stderr, "*** Error loading key file.\n");
      exit (1);
    }

  gnutls_x509_privkey_init (&x509_key);

  ret = gnutls_x509_privkey_import (x509_key, &data, GNUTLS_X509_FMT_PEM);
  if (ret < 0)
    {
      fprintf (stderr, "*** Error loading key file: %s\n",
               gnutls_strerror (ret));
      exit (1);
    }

  gnutls_privkey_init (&key);

  ret =
    gnutls_privkey_import_x509 (key, x509_key,
                                GNUTLS_PRIVKEY_IMPORT_AUTO_RELEASE);
  if (ret < 0)
    {
      fprintf (stderr, "*** Error importing key: %s\n",
               gnutls_strerror (ret));
      exit (1);
    }

  unload_file (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;
  /* Allow connections to servers that have OpenPGP keys as well.
   */

  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_ptr (session, (gnutls_transport_ptr_t) sd);

/* Perform the TLS handshake
 */
ret = gnutls_handshake (session);

if (ret < 0)
  {
    fprintf (stderr, "*** Handshake failed\n");
    gnutls_perror (ret);
    goto end;
  }
else
  {
    printf ("- Handshake was completed\n");
  }

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_CRT_X509)
    {
      *pcert_length = 1;
      *pcert = &crt;
      *pkey = key;
    }
  else
    {
      return -1;
    }

  return 0;

}
```

### 8.1.6 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_crt_t *ca_list;
int ca_list_size;

static int print_details_func (gnutls_x509_crt_t cert,
                               gnutls_x509_crt_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_crt_t *cert;

  unsigned int output;
```

```
/* Initialize the trusted certificate list. This should be done
 * once on initialization. gnutls_x509_crt_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_crt_init (&cert[i]);
    gnutls_x509_crt_import (cert[i], &cert_chain[i], GNUTLS_X509_FMT_DER);
  }

gnutls_x509_trust_list_verify_named_crt (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_crt (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_crt_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_crt_t cert,
                    gnutls_x509_crt_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_crt_get_issuer_dn (cert, issuer_name, &issuer_name_size);

  name_size = sizeof (name);
  gnutls_x509_crt_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_crt_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;
}
```

### 8.1.7 Using a PKCS #11 token with TLS

This example will demonstrate how to load keys and certificates from a PKCS #11 token, and use it with 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 "ca.pem"

/* 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_PKCS11_PIN_FINAL_TRY)
    printf ("*** This is the final try before locking!\n");
  if (flags & GNUTLS_PKCS11_PIN_COUNT_LOW)
    printf ("*** Only few tries left before locking!\n");
  if (flags & GNUTLS_PKCS11_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, 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.
 */

gnutls_global_init ();
/* PKCS11 private key operations might 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_ptr (session, (gnutls_transport_ptr_t) sd);

/* Perform the TLS handshake
 */
ret = gnutls_handshake (session);

if (ret < 0)
  {
    fprintf (stderr, "*** Handshake failed\n");
    gnutls_perror (ret);
```

```
      goto end;
    }
  else
    {
      printf ("- Handshake was completed\n");
    }

  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;
}
```

### 8.1.8 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 "ca.pem"
#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_ptr (session, (gnutls_transport_ptr_t) sd);

    /* Perform the TLS handshake
     */
    ret = gnutls_handshake (session);

    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)
    {
      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);

}                           /* for() */
```

```
  gnutls_certificate_free_credentials (xcred);

  gnutls_global_deinit ();

  return 0;
}
```

### 8.1.9 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 "ca.pem"
#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_srp_client_credentials_t srp_cred;
  gnutls_certificate_credentials_t cert_cred;

  gnutls_global_init ();
```

```
gnutls_srp_allocate_client_credentials (&srp_cred);
gnutls_certificate_allocate_credentials (&cert_cred);

gnutls_certificate_set_x509_trust_file (cert_cred, CAFILE,
                                        GNUTLS_X509_FMT_PEM);
gnutls_srp_set_client_credentials (srp_cred, USERNAME, PASSWORD);

/* connects to server
 */
sd = tcp_connect ();

/* Initialize TLS session
 */
gnutls_init (&session, GNUTLS_CLIENT);


/* Set the priorities.
 */
gnutls_priority_set_direct (session, "NORMAL:+SRP:+SRP-RSA:+SRP-DSS", NULL);

/* put the SRP credentials to the current session
 */
gnutls_credentials_set (session, GNUTLS_CRD_SRP, srp_cred);
gnutls_credentials_set (session, GNUTLS_CRD_CERTIFICATE, cert_cred);

gnutls_transport_set_ptr (session, (gnutls_transport_ptr_t) sd);

/* Perform the TLS handshake
 */
ret = gnutls_handshake (session);

if (ret < 0)
  {
    fprintf (stderr, "*** Handshake failed\n");
    gnutls_perror (ret);
    goto end;
  }
else
  {
    printf ("- Handshake was completed\n");
  }

gnutls_record_send (session, MSG, strlen (MSG));

ret = gnutls_record_recv (session, buffer, MAX_BUF);
if (gnutls_error_is_fatal (ret) == 1 || 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;
}
```

### 8.1.10 Simple client example using the C++ API

The following client is a simple example of a client 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) 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;
}
```

### 8.1.11 Helper function for TCP connections

This helper function abstracts 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>

#define SA struct sockaddr

/* 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, (SA *) & 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);
}
```

## 8.2 Server examples

This section contains examples of TLS and SSL servers, using GnuTLS.

### 8.2.1 Echo server with X.509 authentication

This example is a very simple echo server which supports X.509 authentication, using the RSA ciphersuites.

```
/* 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 "ca.pem"
#define CRLFILE "crl.pem"

/* This is a sample TLS 1.0 echo server, using X.509 authentication.
```

```
 */


#define SA struct sockaddr
#define SOCKET_ERR(err,s) if(err==-1) {perror(s);return(1);}
#define MAX_BUF 1024
#define PORT 5556               /* listen to 5556 port */
#define DH_BITS 1024

/* These are global */
gnutls_certificate_credentials_t x509_cred;
gnutls_priority_t priority_cache;

static gnutls_session_t
initialize_tls_session (void)
{
  gnutls_session_t session;

  gnutls_init (&session, GNUTLS_SERVER);

  gnutls_priority_set (session, priority_cache);

  gnutls_credentials_set (session, GNUTLS_CRD_CERTIFICATE, x509_cred);

  /* request client certificate if any.
   */
  gnutls_certificate_server_set_request (session, GNUTLS_CERT_REQUEST);

  /* Set maximum compatibility mode. This is only suggested on public webservers
   * that need to trade security for compatibility
   */
  gnutls_session_enable_compatibility_mode (session);

  return session;
}

static gnutls_dh_params_t dh_params;

static int
generate_dh_params (void)
{

  /* Generate Diffie-Hellman parameters - for use with DHE
   * kx algorithms. When short bit length is used, it might
   * be wise to regenerate parameters.
   *
   * Check the ex-serv-export.c example for using static
```

```
   * parameters.
   */
  gnutls_dh_params_init (&dh_params);
  gnutls_dh_params_generate2 (dh_params, DH_BITS);

  return 0;
}

int
main (void)
{
  int err, listen_sd;
  int sd, ret;
  struct sockaddr_in sa_serv;
  struct sockaddr_in sa_cli;
  int client_len;
  char topbuf[512];
  gnutls_session_t session;
  char buffer[MAX_BUF + 1];
  int optval = 1;

  /* 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);

  gnutls_certificate_set_x509_key_file (x509_cred, CERTFILE, KEYFILE,
                                        GNUTLS_X509_FMT_PEM);

  generate_dh_params ();

  gnutls_priority_init (&priority_cache, "NORMAL", NULL);


  gnutls_certificate_set_dh_params (x509_cred, 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, (SA *) & 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 (;;)
  {
    session = initialize_tls_session ();

    sd = accept (listen_sd, (SA *) & 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_ptr (session, (gnutls_transport_ptr_t) 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 (;;)
      {
        memset (buffer, 0, MAX_BUF + 1);
        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)
            {
              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, strlen (buffer));
            }
        }
      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;

}
```

### 8.2.2 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 SA struct sockaddr
#define SOCKET_ERR(err,s) if(err==-1) {perror(s);return(1);}
#define MAX_BUF 1024
#define PORT 5556               /* listen to 5556 port */
#define DH_BITS 1024

/* These are global */
gnutls_certificate_credentials_t cred;
gnutls_dh_params_t dh_params;

static int
generate_dh_params (void)
{

  /* 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, DH_BITS);

  return 0;
}

static gnutls_session_t
```

```
initialize_tls_session (void)
{
  gnutls_session_t session;

  gnutls_init (&session, GNUTLS_SERVER);

  gnutls_priority_set_direct (session, "NORMAL:+CTYPE-OPENPGP", NULL);

  /* request client certificate if any.
   */
  gnutls_certificate_server_set_request (session, GNUTLS_CERT_REQUEST);

  gnutls_dh_set_prime_bits (session, DH_BITS);

  return session;
}

int
main (void)
{
  int err, listen_sd;
  int sd, ret;
  struct sockaddr_in sa_serv;
  struct sockaddr_in sa_cli;
  int client_len;
  char topbuf[512];
  gnutls_session_t session;
  char buffer[MAX_BUF + 1];
  int optval = 1;
  char name[256];

  strcpy (name, "Echo Server");

  /* this must be called once in the program
   */
  gnutls_global_init ();

  gnutls_certificate_allocate_credentials (&cred);
  gnutls_certificate_set_openpgp_keyring_file (cred, RINGFILE,
                                               GNUTLS_OPENPGP_FMT_BASE64);

  gnutls_certificate_set_openpgp_key_file (cred, CERTFILE, KEYFILE,
                                           GNUTLS_OPENPGP_FMT_BASE64);

  generate_dh_params ();

  gnutls_certificate_set_dh_params (cred, 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, (SA *) & 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 (;;)
  {
    session = initialize_tls_session ();

    sd = accept (listen_sd, (SA *) & 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_ptr (session, (gnutls_transport_ptr_t) 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 (;;)
        {
          memset (buffer, 0, MAX_BUF + 1);
          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)
            {
              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, strlen (buffer));
            }
        }
      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;

}
```

### 8.2.3 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"
#define CERTFILE "cert.pem"
#define CAFILE "ca.pem"

/* This is a sample TLS-SRP echo server.
 */

#define SA struct sockaddr
#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_srp_server_credentials_t srp_cred;
gnutls_certificate_credentials_t cert_cred;

static gnutls_session_t
initialize_tls_session (void)
{
  gnutls_session_t session;

  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);

  /* request client certificate if any.
   */
  gnutls_certificate_server_set_request (session, GNUTLS_CERT_IGNORE);

  return session;
}

int
main (void)
{
  int err, listen_sd;
  int sd, ret;
  struct sockaddr_in sa_serv;
  struct sockaddr_in sa_cli;
  int client_len;
  char topbuf[512];
  gnutls_session_t session;
  char buffer[MAX_BUF + 1];
  int optval = 1;
  char name[256];

  strcpy (name, "Echo Server");

  gnutls_global_init ();

  /* SRP_PASSWD a password file (created with the included srptool utility)
   */
  gnutls_srp_allocate_server_credentials (&srp_cred);
  gnutls_srp_set_server_credentials_file (srp_cred, SRP_PASSWD,
                                          SRP_PASSWD_CONF);

  gnutls_certificate_allocate_credentials (&cert_cred);
  gnutls_certificate_set_x509_trust_file (cert_cred, CAFILE,
                                          GNUTLS_X509_FMT_PEM);
  gnutls_certificate_set_x509_key_file (cert_cred, CERTFILE, KEYFILE,
                                        GNUTLS_X509_FMT_PEM);

  /* 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, (SA *) & 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 (;;)
  {
    session = initialize_tls_session ();

    sd = accept (listen_sd, (SA *) & 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_ptr (session, (gnutls_transport_ptr_t) 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");
    printf ("- User %s was connected\n", gnutls_srp_server_get_username(session));

    /* print_info(session); */

    for (;;)
      {
        memset (buffer, 0, MAX_BUF + 1);
        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)
            {
              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, strlen (buffer));
            }
        }
      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;

}
```

### 8.2.4 Echo Server with anonymous authentication

This example server support 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 SA struct sockaddr
#define SOCKET_ERR(err,s) if(err==-1) {perror(s);return(1);}
#define MAX_BUF 1024
#define PORT 5556               /* listen to 5556 port */
#define DH_BITS 1024

/* These are global */
gnutls_anon_server_credentials_t anoncred;

static gnutls_session_t
initialize_tls_session (void)
{
  gnutls_session_t session;

  gnutls_init (&session, GNUTLS_SERVER);

  gnutls_priority_set_direct (session, "NORMAL:+ANON-ECDH:+ANON-DH", NULL);

  gnutls_credentials_set (session, GNUTLS_CRD_ANON, anoncred);

  gnutls_dh_set_prime_bits (session, DH_BITS);

  return session;
}

static gnutls_dh_params_t dh_params;

static int
generate_dh_params (void)
{

  /* 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, DH_BITS);

  return 0;
}

int
main (void)
{
  int err, listen_sd;
  int sd, ret;
  struct sockaddr_in sa_serv;
  struct sockaddr_in sa_cli;
  int client_len;
  char topbuf[512];
  gnutls_session_t session;
  char buffer[MAX_BUF + 1];
  int optval = 1;

  /* this must be called once in the program
   */
  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, (SA *) & 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 (;;)
  {
    session = initialize_tls_session ();

    sd = accept (listen_sd, (SA *) & 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_ptr (session, (gnutls_transport_ptr_t) 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 (;;)
      {
        memset (buffer, 0, MAX_BUF + 1);
        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)
          {
            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, strlen (buffer));
            }
        }
      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;

}
```

## 8.3 Miscellaneous examples

### 8.3.1 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));
    }
}
```

### 8.3.2 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_CRT_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_CRT_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_crt_get_expiration_time (cert);
activation_time = gnutls_x509_crt_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_crt_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_crt_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_crt_get_version (cert));

size = sizeof (dn);
gnutls_x509_crt_get_dn (cert, dn, &size);
printf ("\tDN: %s\n", dn);

size = sizeof (dn);
gnutls_x509_crt_get_issuer_dn (cert, dn, &size);
printf ("\tIssuer's DN: %s\n", dn);
```

```
      gnutls_x509_crt_deinit (cert);

    }
}
```

# 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 The gnutls-cli tool

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.

```
GnuTLS test client
Usage:  gnutls-cli [options] hostname

     -d, --debug integer      Enable debugging
     -r, --resume             Connect, establish a session. Connect
                              again and resume this session.
     -s, --starttls           Connect, establish a plain session and
                              start TLS when EOF or a SIGALRM is
                              received.
     --crlf                   Send CR LF instead of LF.
     --x509fmtder             Use DER format for certificates to read
                              from.
     -f, --fingerprint        Send the openpgp fingerprint, instead
                              of the key.
     --disable-extensions     Disable all the TLS extensions.
     --print-cert             Print the certificate in PEM format.
     --recordsize integer     The maximum record size to advertize.
     -V, --verbose            More verbose output.
     --ciphers cipher1 cipher2...
                              Ciphers to enable.
     --protocols protocol1 protocol2...
                              Protocols to enable.
     --comp comp1 comp2...    Compression methods to enable.
     --macs mac1 mac2...      MACs to enable.
     --kx kx1 kx2...          Key exchange methods to enable.
     --ctypes certType1 certType2...
                              Certificate types to enable.
     --priority PRIORITY STRING
                              Priorities string.
     --x509cafile FILE        Certificate file to use.
     --x509crlfile FILE       CRL file to use.
     --pgpkeyfile FILE        PGP Key file to use.
     --pgpkeyring FILE        PGP Key ring file to use.
     --pgpcertfile FILE       PGP Public Key (certificate) file to
                              use.
     --pgpsubkey HEX|auto     PGP subkey to use.
```

```
--x509keyfile FILE       X.509 key file to use.
--x509certfile FILE      X.509 Certificate file to use.
--srpusername NAME       SRP username to use.
--srppasswd PASSWD       SRP password to use.
--pskusername NAME       PSK username to use.
--pskkey KEY             PSK key (in hex) to use.
--opaque-prf-input DATA
                         Use Opaque PRF Input DATA.
-p, --port PORT          The port to connect to.
--insecure               Don't abort program if server
                         certificate can't be validated.
-l, --list               Print a list of the supported
                         algorithms and modes.
-h, --help               prints this help
-v, --version            prints the program's version number
```

### 9.1.1 Example client PSK connection

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 \
  --pskkey 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.

## 9.2 The gnutls-serv tool

Simple server program that listens to incoming TLS connections.

```
GnuTLS test server
Usage: gnutls-serv [options]

-d, --debug integer      Enable debugging
-g, --generate           Generate Diffie-Hellman Parameters.
-p, --port integer       The port to connect to.
-q, --quiet              Suppress some messages.
--nodb                   Does not use the resume database.
--http                   Act as an HTTP Server.
--echo                   Act as an Echo Server.
```

```
--dhparams FILE          DH params file to use.
--x509fmtder             Use DER format for certificates
--x509cafile FILE        Certificate file to use.
--x509crlfile FILE       CRL file to use.
--pgpkeyring FILE        PGP Key ring file to use.
--pgpkeyfile FILE        PGP Key file to use.
--pgpcertfile FILE       PGP Public Key (certificate) file to
                         use.
--pgpsubkey HEX|auto     PGP subkey to use.
--x509keyfile FILE       X.509 key file to use.
--x509certfile FILE      X.509 Certificate file to use.
--x509dsakeyfile FILE    Alternative X.509 key file to use.
--x509dsacertfile FILE   Alternative X.509 certificate file to
                         use.
-r, --require-cert       Require a valid certificate.
-a, --disable-client-cert
                         Disable request for a client
                         certificate.
--pskpasswd FILE         PSK password file to use.
--pskhint HINT           PSK identity hint to use.
--srppasswd FILE         SRP password file to use.
--srppasswdconf FILE     SRP password conf file to use.
--opaque-prf-input DATA
                         Use Opaque PRF Input DATA.
--ciphers cipher1 cipher2...
                         Ciphers to enable.
--protocols protocol1 protocol2...
                         Protocols to enable.
--comp comp1 comp2...    Compression methods to enable.
--macs mac1 mac2...      MACs to enable.
--kx kx1 kx2...          Key exchange methods to enable.
--ctypes certType1 certType2...
                         Certificate types to enable.
--priority PRIORITY STRING
                         Priorities string.
-l, --list               Print a list of the supported
                         algorithms  and modes.
-h, --help               prints this help
-v, --version            prints the program's version number
```

### 9.2.1 Setting up a test HTTPS server

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
```

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.tmpl --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.tmpl
$ echo 'tls_www_client' >> client.tmpl
$ echo 'encryption_key' >> client.tmpl
$ echo 'signing_key' >> client.tmpl
$ certtool --generate-certificate --load-privkey x509-client-key.pem \
  --load-ca-certificate x509-ca.pem --load-ca-privkey x509-ca-key.pem \
  --template client.tmpl --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.tmpl
$ 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.tmpl \
  --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.tmpl --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 \
              --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 (see Section 5.1.2 [Invoking srptool], page 24). To start the server with SRP support:

```
gnutls-serv --http \
              --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` (see Section 5.2.2 [Invoking psktool], page 25).

```
gnutls-serv --http \
              --pskpasswd psk-passwd.txt
```

Finally, we start the server with all the earlier parameters and you get this command:

```
gnutls-serv --http \
              --x509cafile x509-ca.pem \
              --x509keyfile x509-server-key.pem \
              --x509certfile x509-server.pem \
              --x509dsakeyfile 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 The gnutls-cli-debug tool

This program was created to assist in debugging GnuTLS, but it might be useful to extract a TLS server's capabilities. It's purpose is to connect onto a TLS server, perform some tests and print the server's capabilities. If called with the '-v' parameter more checks will be performed. An example output is:

```
crystal:/cvs/gnutls/src$ ./gnutls-cli-debug localhost -p 5556
Resolving 'localhost'...
Connecting to '127.0.0.1:5556'...
Checking for TLS 1.1 support... yes
Checking fallback from TLS 1.1 to... N/A
Checking for TLS 1.0 support... yes
Checking for SSL 3.0 support... yes
Checking for version rollback bug in RSA PMS... no
Checking for version rollback bug in Client Hello... no
Checking whether we need to disable TLS 1.0... N/A
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 cipher suites not in SSL 3.0 spec... yes
Checking for certificate information... N/A
Checking for trusted CAs... N/A
Checking whether the server understands TLS closure alerts... yes
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 AES cipher support (TLS extension)... yes
Checking for 3DES 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 ZLIB compression support (TLS extension)... yes
Checking for max record size (TLS extension)... yes
Checking for SRP authentication support (TLS extension)... yes
Checking for OpenPGP authentication support (TLS extension)... 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.

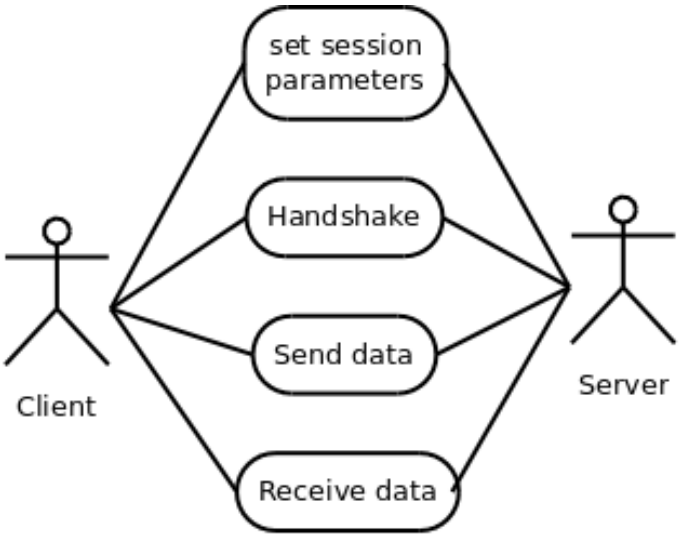


Figure 10.1: TLS protocol use case.

## 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.

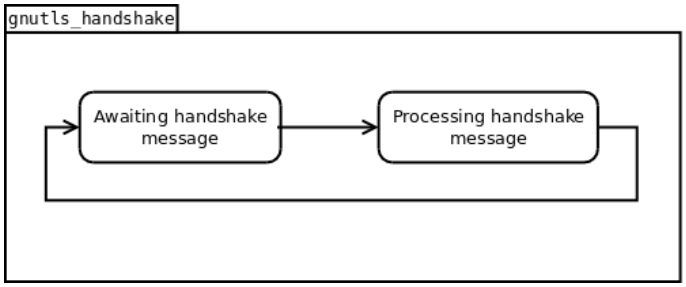


Figure 10.2: GnuTLS handshake state machine.

Also the way the input is processed varies per ciphersuite. Several implementations of the internal handlers are available and [gnutls_handshake], page 174 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.

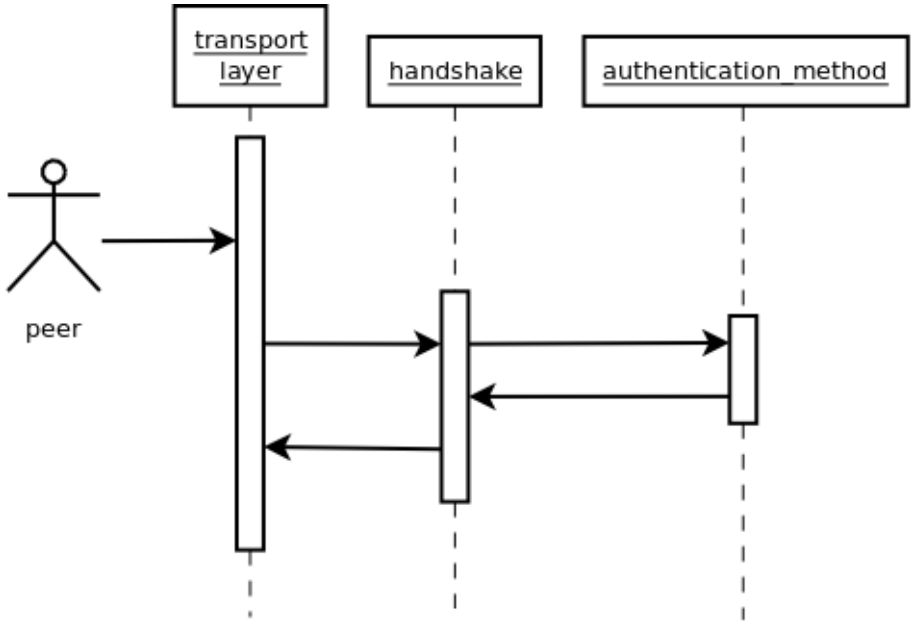


Figure 10.3: GnuTLS handshake process sequence.

## 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[1] 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`.

[1] such as the `gnutls_certificate_credentials_t` structures

### 10.4.1 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 `lib/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;

#if 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 doesn'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 B [Error codes], page 140. 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 LGPLv3+ 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.

## 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.

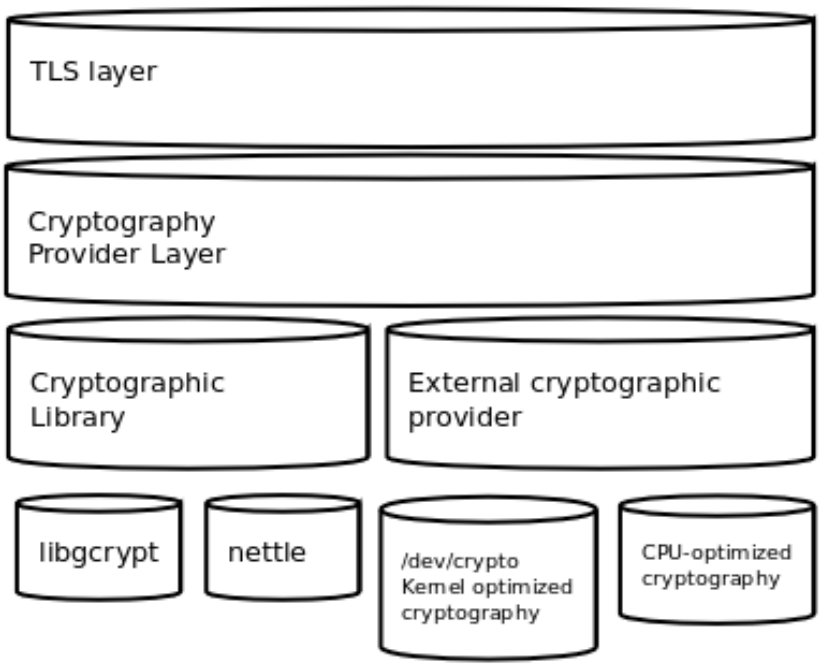


Figure 10.4: GnuTLS cryptographic back-end design.

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.

### 10.5.1 Cryptographic library layer

The Cryptographic library layer, currently supports only libnettle. Other cryptographic libraries might be supported in the future.

### 10.5.2 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[2] 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

[2] Check http://home.gna.org/cryptodev-linux/ for the Linux kernel implementation of `/dev/crypto`.

instruction sets. This is achieved using a mechanism that detects CPU capabilities and overrides parts of crypto backend 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.

## 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 backend, or even all of them. T following functions are provided for this reason.

- `gnutls_crypto_cipher_register`: To override the cryptographic algorithms backend.
- `gnutls_crypto_digest_register`: To override the digest algorithms backend.
- `gnutls_crypto_rnd_register`: To override the random number generator backend.
- `gnutls_crypto_bigint_register`: To override the big number number operations backend.
- `gnutls_crypto_pk_register`: To override the public key encryption backend. This is tied to the big number operations so either none or both of them should be overriden.

# Appendix A Support

## A.1 Getting Help

A mailing list where users may help each other exists, and you can reach it by sending e-mail to help-gnutls@gnu.org. Archives of the mailing list discussions, and an interface to manage subscriptions, is available through the World Wide Web at http://lists.gnu.org/mailman/listinfo/help-gnutls.

A mailing list for developers are also available, see http://www.gnu.org/software/gnutls/lists.html. Bug reports should be sent to bug-gnutls@gnu.org, see Section A.3 [Bug Reports], page 138.

## A.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
```

If your company provides support related to GnuTLS and would like to be mentioned here, contact the authors.

## A.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:

'`bug-gnutls@gnu.org`'

## A.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 A.3 [Bug Reports], page 138. 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.

# Appendix B Error Codes and Descriptions

The error codes used throughout the library are described below. The return code `GNUTLS_E_SUCCESS` indicate successful operation, and is guaranteed to have the value 0, so you can use it in logical expressions.

`GNUTLS_E_AGAIN:`
: Resource temporarily unavailable, try again.

`GNUTLS_E_ASN1_DER_ERROR:`
: ASN1 parser: Error in DER parsing.

`GNUTLS_E_ASN1_DER_OVERFLOW:`
: ASN1 parser: Overflow in DER parsing.

`GNUTLS_E_ASN1_ELEMENT_NOT_FOUND:`
: ASN1 parser: Element was not found.

`GNUTLS_E_ASN1_GENERIC_ERROR:`
: ASN1 parser: Generic parsing error.

`GNUTLS_E_ASN1_IDENTIFIER_NOT_FOUND:`
: ASN1 parser: Identifier was not found

`GNUTLS_E_ASN1_SYNTAX_ERROR:`
: ASN1 parser: Syntax error.

`GNUTLS_E_ASN1_TAG_ERROR:`
: ASN1 parser: Error in TAG.

`GNUTLS_E_ASN1_TAG_IMPLICIT:`
: ASN1 parser: error in implicit tag

`GNUTLS_E_ASN1_TYPE_ANY_ERROR:`
: ASN1 parser: Error in type 'ANY'.

`GNUTLS_E_ASN1_VALUE_NOT_FOUND:`
: ASN1 parser: Value was not found.

`GNUTLS_E_ASN1_VALUE_NOT_VALID:`
: ASN1 parser: Value is not valid.

`GNUTLS_E_BAD_COOKIE:`
: The cookie was bad.

`GNUTLS_E_BASE64_DECODING_ERROR:`
: Base64 decoding error.

`GNUTLS_E_BASE64_ENCODING_ERROR:`
: Base64 encoding error.

`GNUTLS_E_BASE64_UNEXPECTED_HEADER_ERROR:`
: Base64 unexpected header error.

`GNUTLS_E_CERTIFICATE_ERROR:`
: Error in the certificate.

`GNUTLS_E_CERTIFICATE_KEY_MISMATCH:`
: The certificate and the given key do not match.

`GNUTLS_E_CERTIFICATE_LIST_UNSORTED:`
: The provided X.509 certificate list is not sorted (in subject to issuer order)

`GNUTLS_E_CHANNEL_BINDING_NOT_AVAILABLE:`
: Channel binding data not available

`GNUTLS_E_COMPRESSION_FAILED:`
: Compression of the TLS record packet has failed.

`GNUTLS_E_CONSTRAINT_ERROR:`
: Some constraint limits were reached.

`GNUTLS_E_CRYPTODEV_DEVICE_ERROR:`
: Error opening /dev/crypto

`GNUTLS_E_CRYPTODEV_IOCTL_ERROR:`
: Error interfacing with /dev/crypto

`GNUTLS_E_CRYPTO_ALREADY_REGISTERED:`
: There is already a crypto algorithm with lower priority.

`GNUTLS_E_CRYPTO_INIT_FAILED:`
: The initialization of crypto backend has failed.

`GNUTLS_E_DB_ERROR:`
: Error in Database backend.

`GNUTLS_E_DECOMPRESSION_FAILED:`
: Decompression of the TLS record packet has failed.

`GNUTLS_E_DECRYPTION_FAILED:`
: Decryption has failed.

`GNUTLS_E_DH_PRIME_UNACCEPTABLE:`
: The Diffie-Hellman prime sent by the server is not acceptable (not long enough).

`GNUTLS_E_ECC_NO_SUPPORTED_CURVES:`
: No supported ECC curves were found

`GNUTLS_E_ECC_UNSUPPORTED_CURVE:`
: The curve is unsupported

`GNUTLS_E_ENCRYPTION_FAILED:`
: Encryption has failed.

`GNUTLS_E_ERROR_IN_FINISHED_PACKET:`
: An error was encountered at the TLS Finished packet calculation.

`GNUTLS_E_EXPIRED:`
: The requested session has expired.

`GNUTLS_E_FATAL_ALERT_RECEIVED:`
: A TLS fatal alert has been received.

`GNUTLS_E_FILE_ERROR:`
: Error while reading file.

`GNUTLS_E_GOT_APPLICATION_DATA:`
: TLS Application data were received, while expecting handshake data.

`GNUTLS_E_HANDSHAKE_TOO_LARGE:`
: The handshake data size is too large.

`GNUTLS_E_HASH_FAILED:`
: Hashing has failed.

`GNUTLS_E_IA_VERIFY_FAILED:`
: Verifying TLS/IA phase checksum failed

`GNUTLS_E_ILLEGAL_PARAMETER:`
: An illegal parameter was found.

`GNUTLS_E_ILLEGAL_SRP_USERNAME:`
: The SRP username supplied is illegal.

`GNUTLS_E_INCOMPATIBLE_GCRYPT_LIBRARY:`
: The crypto library version is too old.

`GNUTLS_E_INCOMPATIBLE_LIBTASN1_LIBRARY:`
: The tasn1 library version is too old.

`GNUTLS_E_INCOMPAT_DSA_KEY_WITH_TLS_PROTOCOL:`
: The given DSA key is incompatible with the selected TLS protocol.

`GNUTLS_E_INSUFFICIENT_CREDENTIALS:`
: Insufficient credentials for that request.

`GNUTLS_E_INTERNAL_ERROR:`
: GnuTLS internal error.

`GNUTLS_E_INTERRUPTED:`
: Function was interrupted.

`GNUTLS_E_INVALID_PASSWORD:`
: The given password contains invalid characters.

`GNUTLS_E_INVALID_REQUEST:`
: The request is invalid.

`GNUTLS_E_INVALID_SESSION:`
: The specified session has been invalidated for some reason.

`GNUTLS_E_KEY_USAGE_VIOLATION:`
: Key usage violation in certificate has been detected.

`GNUTLS_E_LARGE_PACKET:`
: A large TLS record packet was received.

`GNUTLS_E_LOCKING_ERROR:`
: Thread locking error

`GNUTLS_E_MAC_VERIFY_FAILED:`
: The Message Authentication Code verification failed.

`GNUTLS_E_MEMORY_ERROR:`
: Internal error in memory allocation.

`GNUTLS_E_MPI_PRINT_FAILED:`
: Could not export a large integer.

`GNUTLS_E_MPI_SCAN_FAILED:`
: The scanning of a large integer has failed.

`GNUTLS_E_NO_CERTIFICATE_FOUND:`
: The peer did not send any certificate.

`GNUTLS_E_NO_CIPHER_SUITES:`
: No supported cipher suites have been found.

`GNUTLS_E_NO_COMPRESSION_ALGORITHMS:`
: No supported compression algorithms have been found.

`GNUTLS_E_NO_TEMPORARY_DH_PARAMS:`
: No temporary DH parameters were found.

`GNUTLS_E_NO_TEMPORARY_RSA_PARAMS:`
: No temporary RSA parameters were found.

`GNUTLS_E_OPENPGP_FINGERPRINT_UNSUPPORTED:`
: The OpenPGP fingerprint is not supported.

`GNUTLS_E_OPENPGP_GETKEY_FAILED:`
: Could not get OpenPGP key.

`GNUTLS_E_OPENPGP_KEYRING_ERROR:`
: Error loading the keyring.

`GNUTLS_E_OPENPGP_PREFERRED_KEY_ERROR:`
: The OpenPGP key has not a preferred key set.

`GNUTLS_E_OPENPGP_SUBKEY_ERROR:`
: Could not find OpenPGP subkey.

`GNUTLS_E_OPENPGP_UID_REVOKED:`
: The OpenPGP User ID is revoked.

`GNUTLS_E_PARSING_ERROR:`
: Error in parsing.

`GNUTLS_E_PKCS11_ATTRIBUTE_ERROR:`
: PKCS #11 error in attribute

`GNUTLS_E_PKCS11_DATA_ERROR:`
: PKCS #11 error in data

`GNUTLS_E_PKCS11_DEVICE_ERROR:`
: PKCS #11 error in device

`GNUTLS_E_PKCS11_ERROR:`
PKCS #11 error.

`GNUTLS_E_PKCS11_KEY_ERROR:`
PKCS #11 error in key

`GNUTLS_E_PKCS11_LOAD_ERROR:`
PKCS #11 initialization error.

`GNUTLS_E_PKCS11_PIN_ERROR:`
PKCS #11 error in PIN.

`GNUTLS_E_PKCS11_PIN_EXPIRED:`
PKCS #11 PIN expired

`GNUTLS_E_PKCS11_PIN_LOCKED:`
PKCS #11 PIN locked

`GNUTLS_E_PKCS11_REQUESTED_OBJECT_NOT_AVAILBLE:`
The requested PKCS #11 object is not available

`GNUTLS_E_PKCS11_SESSION_ERROR:`
PKCS #11 error in session

`GNUTLS_E_PKCS11_SIGNATURE_ERROR:`
PKCS #11 error in signature

`GNUTLS_E_PKCS11_SLOT_ERROR:`
PKCS #11 error in slot

`GNUTLS_E_PKCS11_TOKEN_ERROR:`
PKCS #11 error in token

`GNUTLS_E_PKCS11_UNSUPPORTED_FEATURE_ERROR:`
PKCS #11 unsupported feature

`GNUTLS_E_PKCS11_USER_ERROR:`
PKCS #11 user error

`GNUTLS_E_PKCS1_WRONG_PAD:`
Wrong padding in PKCS1 packet.

`GNUTLS_E_PK_DECRYPTION_FAILED:`
Public key decryption has failed.

`GNUTLS_E_PK_ENCRYPTION_FAILED:`
Public key encryption has failed.

`GNUTLS_E_PK_SIGN_FAILED:`
Public key signing has failed.

`GNUTLS_E_PK_SIG_VERIFY_FAILED:`
Public key signature verification has failed.

`GNUTLS_E_PREMATURE_TERMINATION:`
The TLS connection was non-properly terminated.

`GNUTLS_E_PULL_ERROR:`
: Error in the pull function.

`GNUTLS_E_PUSH_ERROR:`
: Error in the push function.

`GNUTLS_E_RANDOM_FAILED:`
: Failed to acquire random data.

`GNUTLS_E_RECEIVED_ILLEGAL_EXTENSION:`
: An illegal TLS extension was received.

`GNUTLS_E_RECEIVED_ILLEGAL_PARAMETER:`
: An illegal parameter has been received.

`GNUTLS_E_RECORD_LIMIT_REACHED:`
: The upper limit of record packet sequence numbers has been reached. Wow!

`GNUTLS_E_REHANDSHAKE:`
: Rehandshake was requested by the peer.

`GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE:`
: The requested data were not available.

`GNUTLS_E_SAFE_RENEGOTIATION_FAILED:`
: Safe renegotiation failed.

`GNUTLS_E_SHORT_MEMORY_BUFFER:`
: The given memory buffer is too short to hold parameters.

`GNUTLS_E_SRP_PWD_ERROR:`
: Error in password file.

`GNUTLS_E_SRP_PWD_PARSING_ERROR:`
: Parsing error in password file.

`GNUTLS_E_SUCCESS:`
: Success.

`GNUTLS_E_TIMEDOUT:`
: The operation timed out

`GNUTLS_E_TOO_MANY_EMPTY_PACKETS:`
: Too many empty record packets have been received.

`GNUTLS_E_TOO_MANY_HANDSHAKE_PACKETS:`
: Too many handshake packets have been received.

`GNUTLS_E_UNEXPECTED_HANDSHAKE_PACKET:`
: An unexpected TLS handshake packet was received.

`GNUTLS_E_UNEXPECTED_PACKET:`
: An unexpected TLS packet was received.

`GNUTLS_E_UNEXPECTED_PACKET_LENGTH:`
: A TLS packet with unexpected length was received.

`GNUTLS_E_UNKNOWN_ALGORITHM:`
: The specified algorithm or protocol is unknown.

`GNUTLS_E_UNKNOWN_CIPHER_SUITE:`
: Could not negotiate a supported cipher suite.

`GNUTLS_E_UNKNOWN_CIPHER_TYPE:`
: The cipher type is unsupported.

`GNUTLS_E_UNKNOWN_COMPRESSION_ALGORITHM:`
: Could not negotiate a supported compression method.

`GNUTLS_E_UNKNOWN_HASH_ALGORITHM:`
: The hash algorithm is unknown.

`GNUTLS_E_UNKNOWN_PKCS_BAG_TYPE:`
: The PKCS structure's bag type is unknown.

`GNUTLS_E_UNKNOWN_PKCS_CONTENT_TYPE:`
: The PKCS structure's content type is unknown.

`GNUTLS_E_UNKNOWN_PK_ALGORITHM:`
: An unknown public key algorithm was encountered.

`GNUTLS_E_UNKNOWN_SRP_USERNAME:`
: The SRP username supplied is unknown.

`GNUTLS_E_UNSAFE_RENEGOTIATION_DENIED:`
: Unsafe renegotiation denied.

`GNUTLS_E_UNSUPPORTED_CERTIFICATE_TYPE:`
: The certificate type is not supported.

`GNUTLS_E_UNSUPPORTED_SIGNATURE_ALGORITHM:`
: The signature algorithm is not supported.

`GNUTLS_E_UNSUPPORTED_VERSION_PACKET:`
: A record packet with illegal version was received.

`GNUTLS_E_UNWANTED_ALGORITHM:`
: An algorithm that is not enabled was negotiated.

`GNUTLS_E_USER_ERROR:`
: The operation was cancelled due to user error

`GNUTLS_E_WARNING_ALERT_RECEIVED:`
: A TLS warning alert has been received.

`GNUTLS_E_WARNING_IA_FPHF_RECEIVED:`
: Received a TLS/IA Final Phase Finished message

`GNUTLS_E_WARNING_IA_IPHF_RECEIVED:`
: Received a TLS/IA Intermediate Phase Finished message

`GNUTLS_E_X509_UNKNOWN_SAN:`
: Unknown Subject Alternative name in X.509 certificate.

`GNUTLS_E_X509_UNSUPPORTED_ATTRIBUTE:`
: The certificate has unsupported attributes.

`GNUTLS_E_X509_UNSUPPORTED_CRITICAL_EXTENSION:`
: Unsupported critical extension in X.509 certificate.

`GNUTLS_E_X509_UNSUPPORTED_OID:`
: The OID is not supported.

# Appendix C API reference

## C.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:** 3.0.0

### 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_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()`. howshould 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, or a negative error code in case of error.

### 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 advertize the CAs they support to clients.

### gnutls_certificate_free_cas

`void gnutls_certificate_free_cas (`*gnutls_certificate_credentials_t* `sc``)` [Function]

*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 (`*gnutls_certificate_credentials_t* `sc``)` [Function]

*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* `sc``)` [Function]

*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* `sc``)` [Function]

*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_issuer

`int gnutls_certificate_get_issuer (`*gnutls_certificate_credentials_t* `sc`, *gnutls_x509_crt_t* `cert`, *gnutls_x509_crt_t** `issuer`, *unsigned int* `flags`)` [Function]

*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.

This function will return the issuer of a given certificate.

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

**Since:** 3.0.0

### 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 certificates, 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

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 our certificates, or `NULL` in case of an error or if no certificate was used.

### gnutls_certificate_send_x509_rdn_sequence

`void gnutls_certificate_send_x509_rdn_sequence` [Function]
(*gnutls_session_t* `session`, *int* `status`)

*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 advertize it's trusted CAs to the peer. If status is zero then the default behaviour will take effect, which is to advertize 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* [Function]
`session`, *gnutls_certificate_request_t* `req`)

*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` [Function]
(*gnutls_certificate_credentials_t* ***res***, *gnutls_dh_params_t* ***dh_params***)

*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 if you deallocate the parameters before the certificate is deallocated, you must change the parameters stored in the certificate first.

## 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_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_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, 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:** 3.0.0

## 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 at verification of the certificates. Flags must be 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:** `GNUTLS_E_SUCCESS` (0) on success, or a negative error code.

**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 its own end entity certificate (e.g., also an intermediate CA cert) then put the certificate chain in `cert_list`.

**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.

This function can also accept PKCS `11` URLs at `keyfile`and `certfile`. In that case it will import the private key and certificate indicated by the URLs.

**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_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_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 PKCS 11 URLs. In that case it will import all certificates that are marked as trusted.

**Returns:** number of certificates processed, or a negative error code on error.

### 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_CRT_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. Note that to be able to use OpenPGP certificates, you must link to libgnutls-extra and call `gnutls_global_init_extra()`.

**Returns:** a (0)-terminated list of `gnutls_certificate_type_t` integers indicating the available certificate types.

### 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 try to verify the peer's certificate and return its status (trusted, invalid etc.). The value of `status`should be one or more of the gnutls_certificate_status_t enumerated elements bitwise or'd. 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.

This function uses `gnutls_x509_crt_list_verify()` with the CAs in the credentials as trusted CAs.

**Returns:** a negative error code on error and `GNUTLS_E_SUCCESS` (0) on success.

### 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* [Function]
`session`)

*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 MAC 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 algorithm)` [Function]

*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 (void)` [Function]

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 (gnutls_kx_algorithm_t kx_algorithm, gnutls_cipher_algorithm_t cipher_algorithm, gnutls_mac_algorithm_t mac_algorithm)` [Function]

*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 * cs_id*, *gnutls_kx_algorithm_t * kx*, *gnutls_cipher_algorithm_t * cipher*, *gnutls_mac_algorithm_t * mac*, *gnutls_protocol_t * min_version*) [Function]

*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` (*gnutls_session_t* **session**) [Function]

*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` (*const char * name*) [Function]

*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` (*gnutls_compression_method_t* **algorithm**) [Function]

*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_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)

Check if database entry has expired. This function is to be used when you want to clear unnesessary session which occupy space in your backend.

**Returns:** Returns `GNUTLS_E_EXPIRED`, if the database entry has expired or 0 otherwise.

### 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 writing this.

### 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 from the resumed sessions database. This function must remove 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()`.

**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, does 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()`.

**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_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* **`params`**, *unsigned int* **`bits`**) [Function]

*params*: 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`**, *const gnutls_datum_t ** **`pkcs3_params`**, *gnutls_x509_crt_fmt_t* **`format`**) [Function]

*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`**, *const gnutls_datum_t ** **`prime`**, *const gnutls_datum_t ** **`generator`**) [Function]

*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 an 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.

This function has no effect in server side.

### gnutls_ecc_curve_get

`gnutls_ecc_curve_t gnutls_ecc_curve_get (gnutls_session_t session)` [Function]

*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:** 3.0.0

### gnutls_ecc_curve_get_name

`const char * gnutls_ecc_curve_get_name (gnutls_ecc_curve_t curve)` [Function]

*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:** 3.0.0

### 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:** 3.0.0

## 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. 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 the record layer or the handshake layer.

**Returns:** 1 if the error code is fatal, for positive `error`values, 0 is returned. For unknown `error`values, -1 is returned.

## 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_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 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_global_deinit

`void gnutls_global_deinit ( void)` [Function]

This function deinitializes the global data, that were initialized using `gnutls_global_init()`.

Note! This function is not thread safe. See the discussion for `gnutls_global_init()` for more information.

## gnutls_global_init

`int gnutls_global_init ( void)` [Function]

This function initializes the global data to defaults. Every gnutls application has a global data which holds common parameters shared by gnutls session structures. You should call `gnutls_global_deinit()` when gnutls usage is no longer needed

Note that this function will also initialize the underlying crypto backend, if it has not been initialized before.

This function increment 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.

Note! This function is not thread safe. If two threads call this function simultaneously, they can cause a race between checking the global counter and incrementing it, causing both threads to execute the library initialization code. That would lead to a memory leak. To handle this, your application could invoke this function after aquiring a thread mutex. To ignore the potential memory leak is also an option.

**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` [Function]
`(gnutls_audit_log_func log_func)`

*log_func*: it is the audit log function

This is the function where you set the logging function gnutls is going to use. This is different from `gnutls_global_set_log_function()` because it will report the session of the event if any. Note that that session might 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, int level, const char*);

**Since:** 3.0.0

## 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 9.

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_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.

This is the function were 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_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. Instead only initialize gnutls and the default OS mutex locks will be used.

This function must be called before `gnutls_global_init()`.

**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 such as `GNUTLS_E_AGAIN` and `GNUTLS_E_INTERRUPTED` interrupt the handshake procedure, which should be later be resumed. Call this function again, until it returns 0; cf. `gnutls_record_get_direction()` and `gnutls_error_is_fatal()`.

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 might also mean that some data were pending.

**Returns:** `GNUTLS_E_SUCCESS` on success, otherwise a negative error code.

## gnutls_handshake_get_last_in

`gnutls_handshake_description_t gnutls_handshake_get_last_in (`*gnutls_session_t* `session``)` [Function]

*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_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` [Function]
(*gnutls_session_t* `session`, *gnutls_handshake_post_client_hello_func* `func`)

*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.

**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 advertized 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_hex2bin

`int gnutls_hex2bin (`*const char * hex_data*, *size_t hex_size*, *char * 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*, *char * 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*, *size_t * result_size*`)` [Function]

*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.

**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.

**Returns:** `GNUTLS_E_SUCCESS` on success, or an error code.

## gnutls_key_generate

`int gnutls_key_generate` (*gnutls_datum_t * key, unsigned int key_size*) [Function]

*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_bytes`size.

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

**Since:** 3.0.0

## 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_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

Get size of MAC key.

**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. For example, MD2 is not supported as a cipher suite, but is supported for other purposes (e.g., X.509 signature verification or similar).

This function is not thread safe.

**Returns:** Return a (0)-terminated list of `gnutls_mac_algorithm_t` integers indicating the available MACs.

### gnutls_openpgp_send_cert

`void gnutls_openpgp_send_cert (`*gnutls_session_t* ***session***`,` [Function]
*gnutls_openpgp_crt_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_pem_base64_decode

`int gnutls_pem_base64_decode (`*const char * **header***, *const* [Function]
*gnutls_datum_t * **b64_data***, *unsigned char * **result***, *size_t * **result_size***`)`

*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* [Function]
*gnutls_datum_t * **b64_data***, *gnutls_datum_t * **result***`)`

*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` [Function]
(*gnutls_pk_algorithm_t* **algorithm**)

*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_prf

`int gnutls_prf (`*gnutls_session_t* **`session`**, *size_t* **`label_size`**, *const char ** **`label`**, *int* **`server_random_first`**, *size_t* **`extra_size`**, *const char ** **`extra`**, *size_t* **`outsize`**, *char ** **`out`**`)` [Function]

*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-0 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-allocate buffer to hold the generated data.

Apply the TLS Pseudo-Random-Function (PRF) using the master secret on some data, seeded with the client and server random fields.

The `label`variable usually contain a string denoting the purpose for the generated data. The `server_random_first`indicate whether the client random field or the server random field should be first in the seed. Non-0 indicate 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 tie 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`**, *const char ** **`label`**, *size_t* **`seed_size`**, *const char ** **`seed`**, *size_t* **`outsize`**, *char ** **`out`**`)` [Function]

*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-allocate buffer to hold the generated data.

Apply the TLS Pseudo-Random-Function (PRF) using the master secret on some data.

The `label`variable usually contain a string denoting the purpose for the generated data. The `seed`usually contain 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`include 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 seed 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_deinit

`void gnutls_priority_deinit (`*gnutls_priority_t* ***priority_cache***`)` [Function]

*priority_cache*: is a `gnutls_prioritity_t` structure.

Deinitializes the priority cache.

## gnutls_priority_get_cipher_suite_index

`int gnutls_priority_get_cipher_suite_index (`*gnutls_priority_t* ***pcache***, *unsigned int* ***idx***, *unsigned int ** ***sidx***`)` [Function]

*pcache*: is a `gnutls_prioritity_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***, *const char ** ***priorities***, *const char *** ***err_pos***`)` [Function]

*priority_cache*: is a `gnutls_prioritity_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 occured

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.

"PERFORMANCE" means all the "secure" ciphersuites are enabled, limited to 128 bit ciphers and sorted by terms of speed performance.

"NORMAL" means all "secure" ciphersuites. 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.

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.

"SECURE:-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",

"NORMAL:COMPAT" is the most compatible mode.

**Returns:** On syntax error GNUTLS_E_INVALID_REQUEST is returned, GNUTLS_E_SUCCESS on success, or an error code.

## 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 occured

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()`.

**Returns:** On syntax error `GNUTLS_E_INVALID_REQUEST` is returned, `GNUTLS_E_SUCCESS` on success, or an error code.

### 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 version*`)` [Function]

*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 (`*gnutls_session_t session*`)` [Function]

*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 threat 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 (`*gnutls_psk_client_credentials_t * sc*`)` [Function]

*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* `session`) [Function]

*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_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 guarranteed not to block.

**Returns:** Returns the size of the data or zero.

### 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.

Normally only servers that require maximum compatibility with everything out there, need to call this function.

### 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.

**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_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_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:** 3.0.0

## 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 full information. You can replace the default push function by using `gnutls_transport_set_ptr2()` with a call to `send()` with a MSG_DONTWAIT flag if blocking is a problem. If the EINTR is returned by the internal push function (the default is `send()`) 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 same parameters; alternatively you could provide a `NULL` pointer for data, and 0 for size. cf. `gnutls_record_get_direction()`.

**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_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(=2^9), 1024(=2^10), 2048(=2^11) and 4096(=2^12). 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_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 will should 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 public key algorithm, 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_server_name_get

`int gnutls_server_name_get (`*gnutls_session_t* **`session`**, *void ** **`data`**, *size_t ** **`data_length`**, *unsigned int ** **`type`**, *unsigned int* **`indx`**`)` [Function]

*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`**, *gnutls_server_name_type_t* **`type`**, *const void ** **`name`**, *size_t* **`name_length`**`)` [Function]

*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`**, *gnutls_channel_binding_t* **`cbtype`**, *gnutls_datum_t* * **`cb`**`)` [Function]

*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 (`*gnutls_session_t* **`session`**`)` [Function]

*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. It is equivalent to calling: `gnutls_record_disable_padding()`

Normally only servers that require maximum compatibility with everything out there, need to call this function.

### 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, in order to support resuming. The client should call this, and keep the returned session, if he wants to resume that current version later by calling `gnutls_session_set_data()` This function must be called after a successful handshake.

Resuming sessions is really useful and speedups connections after a successful one.

**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, in order to support resuming. The client should call this, and keep the returned session, if he wants to resume that current version later by calling `gnutls_session_set_data()`. This function must be called after a successful handshake. The returned datum must be freed with `gnutls_free()`.

Resuming sessions is really useful and speedups connections after a successful one.

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

### 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*: is the session id's size, or it will be set by the function.

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. This is because resumed sessions have the same sessionID with the original session.

Session id is some data set 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_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_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_set_data

`int gnutls_session_set_data (`*gnutls_session_t* ***session***, *const 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'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_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 is 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 some default priority on the ciphers, key exchange methods, macs and compression methods.

**Thisisthesameascalling:** gnutls_priority_set_direct (session, "NORMAL", 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_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_id

`gnutls_sign_algorithm_t gnutls_sign_get_id (`*const char ** ***name***`)` [Function]

*name*: is a MAC algorithm name

The names are compared in a case insensitive way.

**Returns:** return a `gnutls_sign_algorithm_t` value corresponding to the specified cipher, 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_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 (gnutls_srp_client_credentials_t * sc)` [Function]

*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 (gnutls_srp_server_credentials_t * sc)` [Function]

*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, char * result, size_t * result_size)` [Function]

*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 * b64_data, gnutls_datum_t * result)` [Function]

*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`**, *unsigned int* **`bits`**`)` [Function]

*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* g, gnutls_datum_t* n);

`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.

In case the callback returned a negative number then gnutls will assume that the username does not exist.

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).

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_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_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_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* **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`is EAGAIN and EINTR, 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_transport_set_errno_function

`void gnutls_transport_set_errno_function` (*gnutls_session_t* ***session***, *gnutls_errno_func* ***errno_func***) [Function]

*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_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. The callback function is used in DTLS only.

`gnutls_pull_timeout_func`is of the form, ssize_t (*gnutls_pull_timeout_func)(gnutls_transport_ptr_t, unsigned int ms);

**Since:** 3.0.0

### gnutls_transport_set_push_function

`void gnutls_transport_set_push_function` (*gnutls_session_t* ***session***, *gnutls_push_func* ***push_func***) [Function]

*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* ***session***, *gnutls_vec_push_func* ***vec_func***) [Function]

*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 giovec_t * iov, int iovcnt);

**Since:** 2.12.0

## C.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* **`client_data`**, *size_t* **`client_data_size`**, *gnutls_dtls_prestate_st* **`prestate`**, *gnutls_transport_ptr_t* **`ptr`**, *gnutls_push_func* **`push_func`**`)` [Function]

*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

**Description:** 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.

**Returns:** the number of bytes sent, or a negative error code.

**Since:** 3.0.0

## gnutls_dtls_cookie_verify

`int gnutls_dtls_cookie_verify (`*gnutls_datum_t* **`key`**, *void* **`client_data`**, *size_t* **`client_data_size`**, *void* **`_msg`**, *size_t* **`msg_size`**, *gnutls_dtls_prestate_st* **`prestate`**`)` [Function]

*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.

**Description:** This function will verify an incoming message for a valid cookie. If a valid cookie is returned then it should be associated with the session using `gnutls_dtls_prestate_set()`;

**Returns:** `GNUTLS_E_SUCCESS` (0) on success, or a negative error code.

**Since:** 3.0.0

## gnutls_dtls_get_data_mtu

`unsigned int gnutls_dtls_get_data_mtu (`*gnutls_session_t* **`session`**`)` [Function]

*session*: is a `gnutls_session_t` structure.

**Description:** This function will return the actual maximum transfer unit for application data. I.e. DTLS headers are subtracted from the actual MTU.

**Returns:** the maximum allowed transfer unit.

**Since:** 3.0.0

### gnutls_dtls_get_mtu

`unsigned int gnutls_dtls_get_mtu (`*gnutls_session_t* **`session`**`)` [Function]

*session*: is a `gnutls_session_t` structure.

**Description:** 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:** 3.0.0

### gnutls_dtls_prestate_set

`void gnutls_dtls_prestate_set (`*gnutls_session_t* **`session`**`,` [Function]
*gnutls_dtls_prestate_st** **`prestate`**`)`

*session*: a new session

*prestate*: contains the client's prestate

**Description:** This function will associate the prestate acquired by the cookie authentication with the client, with the newly established session.

**Since:** 3.0.0

### gnutls_dtls_set_mtu

`void gnutls_dtls_set_mtu (`*gnutls_session_t* **`session`**`,` *unsigned int* [Function]
**`mtu`**`)`

*session*: is a `gnutls_session_t` structure.

*mtu*: The maximum transfer unit of the interface

**Description:** This function will set the maximum transfer unit of the interface that DTLS packets are expected to leave from.

**Since:** 3.0.0

### gnutls_dtls_set_timeouts

`void gnutls_dtls_set_timeouts (`*gnutls_session_t* **`session`**`,` *unsigned* [Function]
*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.

**Description:** 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.

If the retransmission timeout is zero then the handshake will operate in a non-blocking way, i.e., return `GNUTLS_E_AGAIN`.

**Since:** 3.0.0

### gnutls_record_get_discarded

`unsigned int gnutls_record_get_discarded (`*gnutls_session_t* `session``)` [Function]

*session*: is a `gnutls_session_t` structure.

**Description:** Returns the number of discarded packets in a DTLS connection.

**Returns:** The number of discarded packets.

**Since:** 3.0.0

## C.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_pkcs7_deinit

`void gnutls_pkcs7_deinit (`*gnutls_pkcs7_t* `pkcs7`) [Function]

*pkcs7*: The structure to be initialized

**Description:** 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

**Description:** 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_crt

`int gnutls_pkcs7_delete_crt (`*gnutls_pkcs7_t* `pkcs7`, *int* `indx`) [Function]

*pkcs7*: should contain a gnutls_pkcs7_t structure

*indx*: the index of the certificate to delete

**Description:** 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`, *gnutls_x509_crt_fmt_t* `format`, *void * * `output_data`, *size_t * `output_data_size`) [Function]

*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)

**Description:** 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_get_crl_count

`int gnutls_pkcs7_get_crl_count` (*gnutls_pkcs7_t* ***pkcs7***) [Function]

*pkcs7*: should contain a gnutls_pkcs7_t structure

**Description:** This function will return the number of certifcates 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

**Description:** 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_crt_count

`int gnutls_pkcs7_get_crt_count` (*gnutls_pkcs7_t* ***pkcs7***) [Function]

*pkcs7*: should contain a `gnutls_pkcs7_t` structure

**Description:** This function will return the number of certifcates in the PKCS7 or RFC2630 certificate set.

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

### gnutls_pkcs7_get_crt_raw

`int gnutls_pkcs7_get_crt_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

**Description:** 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 gnutls_datum_t* * **data**, *gnutls_x509_crt_fmt_t* **format**) [Function]

*pkcs7*: The structure to store the parsed PKCS7.

*data*: The DER or PEM encoded PKCS7.

*format*: One of DER or PEM

**Description:** 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

**Description:** 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* **crl**) [Function]

*pkcs7*: should contain a `gnutls_pkcs7_t` structure

*crl*: the DER encoded crl to be added

**Description:** 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 gnutls_datum_t ** ***crl***`)` [Function]

*pkcs7*: should contain a `gnutls_pkcs7_t` structure

*crl*: the DER encoded crl to be added

**Description:** 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_crt

`int gnutls_pkcs7_set_crt (`*gnutls_pkcs7_t* ***pkcs7***, *gnutls_x509_crt_t* ***crt***`)` [Function]

*pkcs7*: should contain a `gnutls_pkcs7_t` structure

*crt*: the certificate to be copied.

**Description:** This function will add a parsed certificate to the PKCS7 or RFC2630 certificate set. This is a wrapper function over `gnutls_pkcs7_set_crt_raw()` .

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

### gnutls_pkcs7_set_crt_raw

`int gnutls_pkcs7_set_crt_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

**Description:** 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_x509_crl_check_issuer

`int gnutls_x509_crl_check_issuer (`*gnutls_x509_crl_t* ***crl***, *gnutls_x509_crt_t* ***issuer***`)` [Function]

*crl*: is the CRL to be checked

*issuer*: is the certificate of a possible issuer

**Description:** This function will check if the given CRL was issued by the given issuer certificate. It will return true (1) if the given CRL was issued by the given issuer, and false (0) if not.

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

## gnutls_x509_crl_deinit

`void gnutls_x509_crl_deinit (`*gnutls_x509_crl_t* **`crl`**`)` [Function]

*crl*: The structure to be initialized

**Description:** This function will deinitialize a CRL structure.

## gnutls_x509_crl_export

`int gnutls_x509_crl_export (`*gnutls_x509_crl_t* **`crl`**, *gnutls_x509_crt_fmt_t* **`format`**, *void ** **`output_data`**, *size_t ** **`output_data_size`**`)` [Function]

*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)

**Description:** 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. and a negative error code on failure.

## gnutls_x509_crl_get_authority_key_id

`int gnutls_x509_crl_get_authority_key_id (`*gnutls_x509_crl_t* **`crl`**, *void ** **`ret`**, *size_t ** **`ret_size`**, *unsigned int ** **`critical`**`)` [Function]

*crl*: should contain a `gnutls_x509_crl_t` structure

*ret*: The place where the identifier will be copied

*ret_size*: Holds the size of the result field.

*critical*: will be non (0) if the extension is marked as critical (may be null)

**Description:** 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.

**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

**Description:** 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

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

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

### 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'

**Description:** 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`**, *int* **`indx`**, *void ** **`data`**, *size_t ** **`sizeof_data`**`)` [Function]

*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`

**Description:** 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 your have reached the last extension available `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

**Since:** 2.8.0

### gnutls_x509_crl_get_extension_info

`int gnutls_x509_crl_get_extension_info (`*gnutls_x509_crl_t* `crl`, *int* `indx`, *void * * `oid`, *size_t * * `sizeof_oid`, *int * * `critical`)` [Function]

*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.

**Description:** 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 your 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* `indx`, *void * * `oid`, *size_t * * `sizeof_oid`)` [Function]

*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`

**Description:** 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 your 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`, *char * * `buf`, *size_t * * `sizeof_buf`)` [Function]

*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`

**Description:** 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

RFC2253. 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_dn_by_oid

`int gnutls_x509_crl_get_issuer_dn_by_oid (`*gnutls_x509_crl_t* **`crl`**, *const char ** **`oid`**, *int* **`indx`**, *unsigned int* **`raw_flag`**, *void ** **`buf`**, *size_t ** **`sizeof_buf`**`)` [Function]

*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 (0) 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`

**Description:** 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 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()`.

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

**Description:** 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`**, *size_t ** **`ret_size`**, *unsigned int ** **`critical`**`)` [Function]

*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 (0) if the extension is marked as critical (may be null)

**Description:** 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`, *gnutls_datum_t ** `dn``)` [Function]

*crl*: should contain a gnutls_x509_crl_t structure

*dn*: will hold the starting point of the DN

**Description:** 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 ** `sig`, *size_t ** `sizeof_sig`)` [Function]

*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`

**Description:** This function will extract the signature field of a CRL.

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

### gnutls_x509_crl_get_signature_algorithm

`int gnutls_x509_crl_get_signature_algorithm (`*gnutls_x509_crl_t* `crl`)` [Function]

*crl*: should contain a `gnutls_x509_crl_t` structure

**Description:** 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

**Description:** 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

**Description:** 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 gnutls_datum_t ** **`data`**, *gnutls_x509_crt_fmt_t* **`format`**`)` [Function]

*crl*: The structure to store the parsed CRL.

*data*: The DER or PEM encoded CRL.

*format*: One of DER or PEM

**Description:** 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

**Description:** 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_list_import

`int gnutls_x509_crl_list_import (`*gnutls_x509_crl_t ** **`crls`**, *unsigned int ** **`crl_max`**, *const gnutls_datum_t ** **`data`**, *gnutls_x509_crt_fmt_t* **`format`**, *unsigned int* **`flags`**`)` [Function]

*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.

**Description:** 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:** 3.0.0

### gnutls_x509_crl_list_import2

`int gnutls_x509_crl_list_import2 (`*gnutls_x509_crl_t ** `crls`*, *unsigned int * `size`*, *const gnutls_datum_t * `data`*, *gnutls_x509_crt_fmt_t `format`*, *unsigned int `flags`*`)` [Function]

*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.

**Description:** 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:** 3.0.0

### gnutls_x509_crl_print

`int gnutls_x509_crl_print (`*gnutls_x509_crl_t `crl`*, *gnutls_certificate_print_formats_t `format`*, *gnutls_datum_t * `out`*`)` [Function]

*crl*: The structure to be printed

*format*: Indicate the format to use

*out*: Newly allocated datum with (0) terminated string.

**Description:** This function will pretty print a X.509 certificate revocation list, suitable for display to a human.

The output `out`needs to be deallocate 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.

**Description:** 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

**Description:** 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

**Description:** 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

**Description:** 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.

**Description:** 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

**Description:** 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.

**Description:** 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_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

**Description:** 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_crt_t * **CA_list***, *int **CA_list_length***, *unsigned int **flags***, *unsigned int * **verify***) [Function]

*crl*: is the crl to be verified

*CA_list*: is a certificate list that is considered to be trusted one

*CA_list_length*: 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.

**Description:** This function will try to verify the given crl and return its status. See `gnutls_x509_crt_list_verify()` for a detailed description of return values.

**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

**Description:** This function will deinitialize a PKCS10 certificate request structure.

### gnutls_x509_crq_export

`int gnutls_x509_crq_export` (*gnutls_x509_crq_t* **`crq`**, *gnutls_x509_crt_fmt_t **format***, *void * **output_data***, *size_t * **output_data_size***) [Function]

*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)

**Description:** 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_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 * sizeof_buf*`)` [Function]

*crq*: should contain a `gnutls_x509_crq_t` structure

*oid*: holds an Object Identified in (0)-terminated string

*indx*: In case multiple same OIDs exist in the attribute list, this specifies which to send, use (0) to get the first one

*buf*: a pointer to a structure to hold the attribute data (may be `NULL`)

*sizeof_buf*: initially holds the size of `buf`

**Description:** This function will return the attribute in the certificate request specified by the given Object ID. The attribute will be DER encoded.

**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 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`

**Description:** 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 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`.

**Description:** 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 your 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 (0) 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.

**Description:** 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 ** `sizeof_pass``)` [Function]

*crq*: should contain a `gnutls_x509_crq_t` structure

*pass*: will hold a (0)-terminated password string

*sizeof_pass*: Initially holds the size of `pass`.

**Description:** 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 ** `sizeof_buf``)` [Function]

*crq*: should contain a `gnutls_x509_crq_t` structure

*buf*: a pointer to a structure to hold the name (may be `NULL`)

*sizeof_buf*: initially holds the size of `buf`

**Description:** 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 * `sizeof_buf`will be updated with the required size. On success 0 is returned.

## gnutls_x509_crq_get_dn_by_oid

`int gnutls_x509_crq_get_dn_by_oid` (*gnutls_x509_crq_t* `crq`, *const char * oid*, *int* `indx`, *unsigned int* `raw_flag`, *void * buf*, *size_t * sizeof_buf*) [Function]

*crq*: should contain a gnutls_x509_crq_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 (0) returns the raw DER data of the DN part.

*buf*: a pointer to a structure to hold the name (may be `NULL`)

*sizeof_buf*: initially holds the size of `buf`

**Description:** 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 * `sizeof_buf`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`, *void * oid*, *size_t * sizeof_oid*) [Function]

*crq*: should contain a gnutls_x509_crq_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`

**Description:** 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* **`crq`**, *const char * `oid`, int `indx`, void * `buf`, size_t * `sizeof_buf`, unsigned int * `critical`*`)` [Function]

*crq*: should contain a `gnutls_x509_crq_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)

*sizeof_buf*: initially holds the size of `buf`

*critical*: will be non (0) if the extension is marked as critical

**Description:** 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_data

`int gnutls_x509_crq_get_extension_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 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`

**Description:** 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 your have reached the last extension available `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

**Since:** 2.8.0

### gnutls_x509_crq_get_extension_info

`int gnutls_x509_crq_get_extension_info (`*gnutls_x509_crq_t* **`crq`**, *int `indx`, void * `oid`, size_t * `sizeof_oid`, unsigned int * `critical`*`)` [Function]

*crq*: should contain a `gnutls_x509_crq_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.

**Description:** 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 int* **`flags`**, *unsigned char * * **`output_data`**, *size_t * * **`output_data_size`**`)` [Function]

*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)

**Description:** This function will return a unique ID the 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`**, *int* **`indx`**, *void * * **`oid`**, *size_t * * **`sizeof_oid`**, *unsigned int * * **`critical`**`)` [Function]

*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`.

**Description:** 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_oidwill 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`, *gnutls_datum_t * m*, *gnutls_datum_t * e*`)` [Function]

*crq*: Holds the certificate

*m*: will hold the modulus

*e*: will hold the public exponent

**Description:** 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`, *unsigned int * key_usage*, *unsigned int * critical*`)` [Function]

*crq*: should contain a gnutls_x509_crq_t structure

*key_usage*: where the key usage bits will be stored

*critical*: will be non (0) if the extension is marked as critical

**Description:** 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`, *unsigned int * bits*`)` [Function]

*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

**Description:** 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_subject_alt_name

int gnutls_x509_crq_get_subject_alt_name (*gnutls_x509_crq_t* **crq**, *unsigned int* **seq**, *void * **ret***, *size_t * **ret_size***, *unsigned int * **ret_type***, *unsigned int * **critical**) [Function]

*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 (0) if the extension is marked as critical (may be null)

**Description:** 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_typeeven 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_sizeis not large enough to hold the value. In that case ret_sizewill 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 (*gnutls_x509_crq_t* **crq**, *unsigned int* **seq**, *void * **ret***, *size_t * **ret_size**) [Function]

*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.

**Description:** 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

**Description:** 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

**Description:** 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

**Description:** 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 (0) terminated string.

**Description:** This function will pretty print a certificate request, suitable for display to a human.

The output `out`needs to be deallocate 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 sizeof_buf*`)` [Function]

*crq*: should contain a `gnutls_x509_crq_t` structure

*oid*: holds an Object Identified in (0)-terminated string

*buf*: a pointer to a structure that holds the attribute data

*sizeof_buf*: holds the size of `buf`

**Description:** This function will set the attribute in the certificate request specified by the given Object ID. The attribute must be be DER encoded.

**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.

**Description:** 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

**Description:** 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_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`

**Description:** 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`, *gnutls_x509_privkey_t* `key`)` [Function]

*crq*: should contain a `gnutls_x509_crq_t` structure

*key*: holds a private key

**Description:** 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`, *const void ** `oid`, *unsigned int* `critical`)` [Function]

*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

**Description:** 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`, *const gnutls_datum_t * m*, *const gnutls_datum_t * e*) [Function]

*crq*: should contain a `gnutls_x509_crq_t` structure

*m*: holds the modulus

*e*: holds the public exponent

**Description:** 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`, *unsigned int* `usage`) [Function]

*crq*: a certificate request of type `gnutls_x509_crq_t`

*usage*: an ORed sequence of the GNUTLS_KEY_* elements.

**Description:** 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_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.

**Description:** 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

**Description:** 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`, *gnutls_x509_privkey_t* `key`, *gnutls_digest_algorithm_t* `dig`, *unsigned int* `flags`) [Function]

*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

**Description:** 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* `flags`) [Function]

*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.

**Description:** This function will verify self signature in the certificate request and return its status.

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

Since 2.12.0

### gnutls_x509_crt_check_hostname

`int gnutls_x509_crt_check_hostname` (*gnutls_x509_crt_t* `cert`, *const char * * `hostname`) [Function]

*cert*: should contain an gnutls_x509_crt_t structure

*hostname*: A null terminated string that contains a DNS name

**Description:** This function will check if the given certificate's subject matches the given hostname. This is a basic implementation of the matching described in RFC2818 (HTTPS), which takes into account wildcards, and the DNSName/IPAddress subject alternative name PKIX extension.

**Returns:** non (0) for a successful match, and (0) on failure.

### gnutls_x509_crt_check_issuer

`int gnutls_x509_crt_check_issuer` (*gnutls_x509_crt_t* `cert`, *gnutls_x509_crt_t* `issuer`) [Function]

*cert*: is the certificate to be checked

*issuer*: is the certificate of a possible issuer

**Description:** This function will check if the given certificate was issued by the given issuer.

**Returns:** It will return true (1) if the given certificate is issued by the given issuer, and false (0) if not. A negative error code is returned in case of an error.

### gnutls_x509_crt_check_revocation

`int gnutls_x509_crt_check_revocation` (*gnutls_x509_crt_t* `cert`, *const gnutls_x509_crl_t ** `crl_list`, *int* `crl_list_length`) [Function]

*cert*: should contain a `gnutls_x509_crt_t` structure

*crl_list*: should contain a list of gnutls_x509_crl_t structures

*crl_list_length*: the length of the crl_list

**Description:** 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_crt_cpy_crl_dist_points

`int gnutls_x509_crt_cpy_crl_dist_points` (*gnutls_x509_crt_t* `dst`, *gnutls_x509_crt_t* `src`) [Function]

*dst*: a certificate of type `gnutls_x509_crt_t`

*src*: the certificate where the dist points will be copied from

**Description:** 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_crt_deinit

`void gnutls_x509_crt_deinit` (*gnutls_x509_crt_t* `cert`) [Function]

*cert*: The structure to be deinitialized

**Description:** This function will deinitialize a certificate structure.

### gnutls_x509_crt_export

`int gnutls_x509_crt_export (`*gnutls_x509_crt_t* **`cert`**, *gnutls_x509_crt_fmt_t* **`format`**, *void ** **`output_data`**, *size_t ** **`output_data_size`**`)` [Function]

*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)

**Description:** 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_crt_get_activation_time

`time_t gnutls_x509_crt_get_activation_time (`*gnutls_x509_crt_t* **`cert`**`)` [Function]

*cert*: should contain a `gnutls_x509_crt_t` structure

**Description:** This function will return the time this Certificate was or will be activated.

**Returns:** activation time, or (time_t)-1 on error.

### gnutls_x509_crt_get_authority_info_access

`int gnutls_x509_crt_get_authority_info_access` [Function]
`(`*gnutls_x509_crt_t* **`crt`**, *unsigned int* **`seq`**, *int* **`what`**, *gnutls_datum_t ** **`data`**, *int ** **`critical`**`)`

*crt*: 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-0 if the extension is marked as critical (may be `NULL`)

**Description:** 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:

<informalexample><programlisting>

**AuthorityInfoAccessSyntax :** := SEQUENCE SIZE (1..MAX) OF AccessDescription

**AccessDescription :** := SEQUENCE { accessMethod OBJECT IDENTIFIER, accessLocation GeneralName } </programlisting></informalexample>

The seqinput 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 seqvalue is out of bounds, GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE is returned.

The type of data returned in datais specified via whatwhich should be gnutls_info_access_what_t values.

If whatis GNUTLS_IA_ACCESSMETHOD_OID then datawill hold the accessMethod OID (e.g., "1.3.6.1.5.5.7.48.1").

If whatis GNUTLS_IA_ACCESSLOCATION_GENERALNAME_TYPE, datawill hold the accessLocation GeneralName type (e.g., "uniformResourceIdentifier").

If whatis GNUTLS_IA_URI, datawill hold the accessLocation URI data. Requesting this whatvalue leads to an error if the accessLocation is not of the "uniformResourceIdentifier" type.

If whatis GNUTLS_IA_OCSP_URI, datawill hold the OCSP URI. Requesting this whatvalue 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.

If whatis GNUTLS_IA_CAISSUERS_URI, datawill hold the caIssuers URI. Requesting this whatvalue 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.

More whatvalues may be allocated in the future as needed.

If datais NULL, the function does the same without storing the output data, that is, it will set criticaland do error checking as usual.

The value of the critical flag is returned in * critical. Supply a NULL criticalif 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 criticalto 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:** 3.0.0

## gnutls_x509_crt_get_authority_key_id

int gnutls_x509_crt_get_authority_key_id (*gnutls_x509_crt_t cert*, *void * ret*, *size_t * ret_size*, *unsigned int * critical*) [Function]

*cert*: should contain a gnutls_x509_crt_t structure

*ret*: The place where the identifier will be copied

*ret_size*: Holds the size of the result field.

*critical*: will be non (0) if the extension is marked as critical (may be null)

**Description:** 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.

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

### gnutls_x509_crt_get_basic_constraints

`int gnutls_x509_crt_get_basic_constraints` (*gnutls_x509_crt_t* `cert`, *unsigned int * * `critical`, *unsigned int * * `ca`, *int * * `pathlen`) [Function]

*cert*: should contain a `gnutls_x509_crt_t` structure

*critical*: will be non (0) 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.

**Description:** 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_crt_get_ca_status

`int gnutls_x509_crt_get_ca_status` (*gnutls_x509_crt_t* `cert`, *unsigned int * * `critical`) [Function]

*cert*: should contain a `gnutls_x509_crt_t` structure

*critical*: will be non (0) if the extension is marked as critical

**Description:** 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_crt_get_basic_constraints()` if you want to read the pathLenConstraint field too.

**Returns:** A negative error code may be returned in case of parsing error. If the certificate does not contain the basicConstraints extension `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

### gnutls_x509_crt_get_crl_dist_points

`int gnutls_x509_crt_get_crl_dist_points` (*gnutls_x509_crt_t* `cert`, *unsigned int* `seq`, *void * * `ret`, *size_t * * `ret_size`, *unsigned int * * `reason_flags`, *unsigned int * * `critical`) [Function]

*cert*: should contain a `gnutls_x509_crt_t` structure

*seq*: specifies the sequence number of the distribution point (0 for the first one, 1 for the second etc.)

*ret*: is the place where the distribution point will be copied to

*ret_size*: holds the size of ret.

*reason_flags*: Revocation reasons flags.

*critical*: will be non (0) if the extension is marked as critical (may be null)

**Description:** This function retrieves the CRL distribution points (2.5.29.31), contained in the given certificate in the X509v3 Certificate Extensions.

`reason_flags`should be an ORed sequence of `GNUTLS_CRL_REASON_UNUSED`, `GNUTLS_CRL_REASON_KEY_COMPROMISE`, `GNUTLS_CRL_REASON_CA_COMPROMISE`, `GNUTLS_CRL_REASON_AFFILIATION_CHANGED`, `GNUTLS_CRL_REASON_SUPERSEEDED`, `GNUTLS_CRL_REASON_CESSATION_OF_OPERATION`, `GNUTLS_CRL_REASON_CERTIFICATE_HOLD`, `GNUTLS_CRL_REASON_PRIVILEGE_WITHDRAWN`, `GNUTLS_CRL_REASON_AA_COMPROMISE`, or (0) for all possible reasons.

**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_crt_get_dn

`int gnutls_x509_crt_get_dn` (*gnutls_x509_crt_t* **`cert`**, *char ** **`buf`**, *size_t ** **`buf_size`**) [Function]

*cert*: should contain a `gnutls_x509_crt_t` structure

*buf*: a pointer to a structure to hold the name (may be null)

*buf_size*: initially holds the size of `buf`

**Description:** 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 RFC2253. 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. 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. On success 0 is returned.

## gnutls_x509_crt_get_dn_by_oid

`int gnutls_x509_crt_get_dn_by_oid` (*gnutls_x509_crt_t* **`cert`**, *const char ** **`oid`**, *int* **`indx`**, *unsigned int* **`raw_flag`**, *void ** **`buf`**, *size_t ** **`buf_size`**) [Function]

*cert*: should contain a `gnutls_x509_crt_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 (0) 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`

**Description:** 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 RFC2253. 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 RFC2253 – 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. On success 0 is returned.

## gnutls_x509_crt_get_dn_oid

`int gnutls_x509_crt_get_dn_oid (`*gnutls_x509_crt_t* `cert`*, int* `indx`*, void * oid, size_t * oid_size*`)` [Function]

*cert*: should contain a `gnutls_x509_crt_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`

**Description:** 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. If the `raw_flag`is not specified the output is always null terminated, although the `oid_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 `oid_size`will be updated with the required size. On success 0 is returned.

## gnutls_x509_crt_get_expiration_time

`time_t gnutls_x509_crt_get_expiration_time (`*gnutls_x509_crt_t* `cert`*)* [Function]

*cert*: should contain a `gnutls_x509_crt_t` structure

**Description:** This function will return the time this Certificate was or will be expired.

**Returns:** expiration time, or (time_t)-1 on error.

## gnutls_x509_crt_get_extension_by_oid

`int gnutls_x509_crt_get_extension_by_oid (`*gnutls_x509_crt_t* `cert`*, const char * oid, int indx, void * buf, size_t * buf_size, unsigned int * critical*`)` [Function]

*cert*: should contain a `gnutls_x509_crt_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 (0) if the extension is marked as critical

**Description:** 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_crt_get_extension_data

`int gnutls_x509_crt_get_extension_data` (*gnutls_x509_crt_t* `cert`, *int* `indx`, *void * data*, *size_t * sizeof_data*) [Function]

*cert*: should contain a `gnutls_x509_crt_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`

**Description:** 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_crt_get_extension_info()` to extract the OID and critical flag. Use `gnutls_x509_crt_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_crt_get_extension_info

`int gnutls_x509_crt_get_extension_info` (*gnutls_x509_crt_t* `cert`, *int* `indx`, *void * oid*, *size_t * oid_size*, *unsigned int * critical*) [Function]

*cert*: should contain a `gnutls_x509_crt_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.

**Description:** 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_crt_get_extension_data()` 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.

**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_crt_get_extension_oid

`int gnutls_x509_crt_get_extension_oid` (*gnutls_x509_crt_t* **cert**, *int* **indx**, *void * * **oid**, *size_t * * **oid_size**) [Function]

*cert*: should contain a `gnutls_x509_crt_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`

**Description:** This function will return the requested extension OID in the certificate. 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 is returned. If you have reached the last extension available `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` will be returned.

## gnutls_x509_crt_get_fingerprint

`int gnutls_x509_crt_get_fingerprint` (*gnutls_x509_crt_t* **cert**, *gnutls_digest_algorithm_t* **algo**, *void * * **buf**, *size_t * * **buf_size**) [Function]

*cert*: should contain a `gnutls_x509_crt_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`

**Description:** This function will calculate and copy the certificate's fingerprint in the provided buffer.

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_crt_get_issuer

`int gnutls_x509_crt_get_issuer` (*gnutls_x509_crt_t* **cert**, *gnutls_x509_dn_t * * **dn**) [Function]

*cert*: should contain a `gnutls_x509_crt_t` structure

*dn*: output variable with pointer to opaque DN

**Description:** Return the Certificate's Issuer DN as an opaque data type. You may use `gnutls_x509_dn_get_rdn_ava()` to decode the DN.

Note that `dn`should be treated as constant. Because points into the `cert`object, you may not deallocate `cert`and continue to access `dn`.

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

### gnutls_x509_crt_get_issuer_alt_name

`int gnutls_x509_crt_get_issuer_alt_name (`*gnutls_x509_crt_t* **`cert`**, *unsigned int* **`seq`**, *void ** **`ret`**, *size_t ** **`ret_size`**, *unsigned int ** **`critical`**`)` [Function]

*cert*: should contain a `gnutls_x509_crt_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.

*critical*: will be non (0) if the extension is marked as critical (may be null)

**Description:** 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_crt_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 `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 then `GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE` is returned.

**Since:** 2.10.0

### gnutls_x509_crt_get_issuer_alt_name2

`int gnutls_x509_crt_get_issuer_alt_name2 (`*gnutls_x509_crt_t* **`cert`**, *unsigned int* **`seq`**, *void ** **`ret`**, *size_t ** **`ret_size`**, *unsigned int ** **`ret_type`**, *unsigned int ** **`critical`**`)` [Function]

*cert*: should contain a `gnutls_x509_crt_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 type of the alternative name (one of gnutls_x509_subject_alt_name_t).

*critical*: will be non (0) if the extension is marked as critical (may be null)

**Description:** This function will return the alternative names, contained in the given certificate. It is the same as `gnutls_x509_crt_get_issuer_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 issuer 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 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_crt_get_issuer_alt_othername_oid

`int gnutls_x509_crt_get_issuer_alt_othername_oid` [Function]
(*gnutls_x509_crt_t* `cert`, *unsigned int* `seq`, *void ** `ret`, *size_t ** `ret_size`)

*cert*: should contain a `gnutls_x509_crt_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.

**Description:** 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. If the `raw_flag`is not specified the output is always null terminated, although the `oid_size`will not include the null character.

This function is only useful if `gnutls_x509_crt_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_crt_get_issuer_dn

`int gnutls_x509_crt_get_issuer_dn` (*gnutls_x509_crt_t* `cert`, *char ** `buf`, *size_t ** `buf_size`) [Function]

*cert*: should contain a `gnutls_x509_crt_t` structure

*buf*: a pointer to a structure to hold the name (may be null)

*buf_size*: initially holds the size of `buf`

**Description:** 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 RFC2253. 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. 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. On success 0 is returned.

## gnutls_x509_crt_get_issuer_dn_by_oid

`int gnutls_x509_crt_get_issuer_dn_by_oid` (*gnutls_x509_crt_t* `cert`, *const char * oid*, *int* `indx`, *unsigned int* `raw_flag`, *void * buf*, *size_t * buf_size*) [Function]

*cert*: should contain a `gnutls_x509_crt_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 (0) 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`

**Description:** 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 RFC2253. 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 RFC2253 – 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. On success 0 is returned.

## gnutls_x509_crt_get_issuer_dn_oid

`int gnutls_x509_crt_get_issuer_dn_oid` (*gnutls_x509_crt_t* `cert`, *int* `indx`, *void * oid*, *size_t * oid_size*) [Function]

*cert*: should contain a `gnutls_x509_crt_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`

**Description:** 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. If the `raw_flag`is not specified the output is always null terminated, although the `oid_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 `oid_size`will be updated with the required size. On success 0 is returned.

### gnutls_x509_crt_get_issuer_unique_id

`int gnutls_x509_crt_get_issuer_unique_id` (*gnutls_x509_crt_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.

**Description:** 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_crt_get_key_id

`int gnutls_x509_crt_get_key_id` (*gnutls_x509_crt_t* `crt`, *unsigned int* `flags`, *unsigned char * output_data*, *size_t * output_data_size*) [Function]

*crt*: 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)

**Description:** This function will return a unique ID the 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_crt_get_key_purpose_oid

`int gnutls_x509_crt_get_key_purpose_oid` (*gnutls_x509_crt_t* `cert`, *int indx*, *void * oid*, *size_t * oid_size*, *unsigned int * critical*) [Function]

*cert*: should contain a `gnutls_x509_crt_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

**Description:** 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. If the `raw_flag`is not specified the output is always null terminated, although the `oid_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 *oid_size will be updated with the required size. On success 0 is returned.

## gnutls_x509_crt_get_key_usage

`int gnutls_x509_crt_get_key_usage` (*gnutls_x509_crt_t* `cert`, *unsigned int * key_usage, unsigned int * critical*) [Function]

*cert*: should contain a `gnutls_x509_crt_t` structure

*key_usage*: where the key usage bits will be stored

*critical*: will be non (0) if the extension is marked as critical

**Description:** 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_crt_get_pk_algorithm

`int gnutls_x509_crt_get_pk_algorithm` (*gnutls_x509_crt_t* `cert`, *unsigned int * bits*) [Function]

*cert*: should contain a `gnutls_x509_crt_t` structure

*bits*: if bits is non null it will hold the size of the parameters' in bits

**Description:** 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_crt_get_pk_dsa_raw

`int gnutls_x509_crt_get_pk_dsa_raw` (*gnutls_x509_crt_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

**Description:** 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_crt_get_pk_rsa_raw

`int gnutls_x509_crt_get_pk_rsa_raw` (*gnutls_x509_crt_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

**Description:** 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_crt_get_proxy

`int gnutls_x509_crt_get_proxy` (*gnutls_x509_crt_t* `cert`, *unsigned int * critical*, *int * pathlen*, *char ** policyLanguage*, *char ** policy*, *size_t * sizeof_policy*) [Function]

*cert*: should contain a `gnutls_x509_crt_t` structure

*critical*: will be non (0) 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

**Description:** 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_crt_get_raw_dn

`int gnutls_x509_crt_get_raw_dn` (*gnutls_x509_crt_t* `cert`, *gnutls_datum_t * start*) [Function]

*cert*: should contain a `gnutls_x509_crt_t` structure

*start*: will hold the starting point of the DN

**Description:** This function will return a pointer to the DER encoded DN structure and the length.

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

## gnutls_x509_crt_get_raw_issuer_dn

`int gnutls_x509_crt_get_raw_issuer_dn (`*gnutls_x509_crt_t* **`cert`**`,` *gnutls_datum_t ** **`start`**`)` [Function]

*cert*: should contain a `gnutls_x509_crt_t` structure

*start*: will hold the starting point of the DN

**Description:** This function will return a pointer to the DER encoded DN structure and the length.

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

## gnutls_x509_crt_get_serial

`int gnutls_x509_crt_get_serial (`*gnutls_x509_crt_t* **`cert`**`,` *void ** **`result`**`,` *size_t ** **`result_size`**`)` [Function]

*cert*: should contain a `gnutls_x509_crt_t` structure

*result*: The place where the serial number will be copied

*result_size*: Holds the size of the result field.

**Description:** 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 opaque.

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

## gnutls_x509_crt_get_signature

`int gnutls_x509_crt_get_signature (`*gnutls_x509_crt_t* **`cert`**`,` *char ** **`sig`**`,` *size_t ** **`sizeof_sig`**`)` [Function]

*cert*: should contain a `gnutls_x509_crt_t` structure

*sig*: a pointer where the signature part will be copied (may be null).

*sizeof_sig*: initially holds the size of `sig`

**Description:** 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_crt_get_signature_algorithm

`int gnutls_x509_crt_get_signature_algorithm (`*gnutls_x509_crt_t* `cert``)` [Function]

*cert*: should contain a `gnutls_x509_crt_t` structure

**Description:** 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_crt_get_subject

`int gnutls_x509_crt_get_subject (`*gnutls_x509_crt_t* `cert`, *gnutls_x509_dn_t* * `dn``)` [Function]

*cert*: should contain a `gnutls_x509_crt_t` structure

*dn*: output variable with pointer to opaque DN.

**Description:** Return the Certificate's Subject DN as an opaque data type. You may use `gnutls_x509_dn_get_rdn_ava()` to decode the DN.

Note that `dn`should be treated as constant. Because points into the `cert`object, you may not deallocate `cert`and continue to access `dn`.

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

### gnutls_x509_crt_get_subject_alt_name

`int gnutls_x509_crt_get_subject_alt_name (`*gnutls_x509_crt_t* `cert`, *unsigned int* `seq`, *void* * `ret`, *size_t* * `ret_size`, *unsigned int* * `critical``)` [Function]

*cert*: should contain a `gnutls_x509_crt_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.

*critical*: will be non (0) if the extension is marked as critical (may be null)

**Description:** 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_crt_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 `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 then GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE is returned.

### gnutls_x509_crt_get_subject_alt_name2

int gnutls_x509_crt_get_subject_alt_name2 (*gnutls_x509_crt_t* `cert`, *unsigned int* `seq`, *void * *`ret`, *size_t * *`ret_size`, *unsigned int * *`ret_type`, *unsigned int * *`critical`) [Function]

*cert*: should contain a gnutls_x509_crt_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 type of the alternative name (one of gnutls_x509_subject_alt_name_t).

*critical*: will be non (0) if the extension is marked as critical (may be null)

**Description:** This function will return the alternative names, contained in the given certificate. It is the same as gnutls_x509_crt_get_subject_alt_name() except for the fact that it will return the type of the alternative name in ret_typeeven 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_sizeis not large enough to hold the value. In that case ret_sizewill 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_crt_get_subject_alt_othername_oid

int gnutls_x509_crt_get_subject_alt_othername_oid (*gnutls_x509_crt_t* `cert`, *unsigned int* `seq`, *void * *`oid`, *size_t * *`oid_size`) [Function]

*cert*: should contain a gnutls_x509_crt_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.

**Description:** 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_crt_get_subject_alt_name() returned GNUTLS_SAN_OTHERNAME.

If oidis null then only the size will be filled. If the raw_flagis not specified the output is always null terminated, although the oid_sizewill not include the null character.

**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_sizeis not large enough to hold the value. In that case ret_sizewill 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_crt_get_subject_key_id

`int gnutls_x509_crt_get_subject_key_id (`*gnutls_x509_crt_t* **`cert`**, *void * ret*, *size_t * ret_size*, *unsigned int * critical*`)` [Function]

*cert*: should contain a gnutls_x509_crt_t structure

*ret*: The place where the identifier will be copied

*ret_size*: Holds the size of the result field.

*critical*: will be non (0) if the extension is marked as critical (may be null)

**Description:** 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, otherwise a negative error value.

## gnutls_x509_crt_get_subject_unique_id

`int gnutls_x509_crt_get_subject_unique_id (`*gnutls_x509_crt_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.

**Description:** 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_crt_get_version

`int gnutls_x509_crt_get_version (`*gnutls_x509_crt_t* **`cert`**`)` [Function]

*cert*: should contain a gnutls_x509_crt_t structure

**Description:** This function will return the version of the specified Certificate.

**Returns:** version of certificate, or a negative error code on error.

## gnutls_x509_crt_import

`int gnutls_x509_crt_import (`*gnutls_x509_crt_t* **`cert`**, *const gnutls_datum_t * data*, *gnutls_x509_crt_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

**Description:** This function will convert the given DER or PEM encoded Certificate to the native gnutls_x509_crt_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_crt_init

`int gnutls_x509_crt_init (`*gnutls_x509_crt_t * `cert`*`)` [Function]

*cert*: The structure to be initialized

**Description:** 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_crt_list_import

`int gnutls_x509_crt_list_import (`*gnutls_x509_crt_t * `certs`*, *unsigned int * `cert_max`*, *const gnutls_datum_t * `data`*, *gnutls_x509_crt_fmt_t `format`*, *unsigned int `flags`*`)` [Function]

*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.

**Description:** 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.

The flag `GNUTLS_X509_CRT_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_CRT_LIST_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_crt_list_import2

`int gnutls_x509_crt_list_import2 (`*gnutls_x509_crt_t ** `certs`*, *unsigned int * `size`*, *const gnutls_datum_t * `data`*, *gnutls_x509_crt_fmt_t `format`*, *unsigned int `flags`*`)` [Function]

*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.

**Description:** 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:** 3.0.0

## gnutls_x509_crt_list_verify

`int gnutls_x509_crt_list_verify` (*const gnutls_x509_crt_t * `cert_list`*, *int `cert_list_length`*, *const gnutls_x509_crt_t * `CA_list`*, *int `CA_list_length`*, *const gnutls_x509_crl_t * `CRL_list`*, *int `CRL_list_length`*, *unsigned int `flags`*, *unsigned int * `verify`*) [Function]

*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.

**Description:** 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_crt_verify()` per list element.

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

## gnutls_x509_crt_print

`int gnutls_x509_crt_print` (*gnutls_x509_crt_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.

**Description:** This function will pretty print a X.509 certificate, suitable for display to a human.

If the format is `GNUTLS_CRT_PRINT_FULL` then all fields of the certificate will be output, on multiple lines. The `GNUTLS_CRT_PRINT_ONELINE` format will generate one line with some selected fields, which is useful for logging purposes.

The output `out`needs to be deallocate using `gnutls_free()`.

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

## gnutls_x509_crt_set_activation_time

`int gnutls_x509_crt_set_activation_time (`*gnutls_x509_crt_t* `cert`, *time_t* `act_time`[Function]`)`

*cert*: a certificate of type `gnutls_x509_crt_t`

*act_time*: The actual time

**Description:** 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_crt_set_authority_key_id

`int gnutls_x509_crt_set_authority_key_id (`*gnutls_x509_crt_t* `cert`, *const void * * `id`, *size_t* `id_size`[Function]`)`

*cert*: a certificate of type `gnutls_x509_crt_t`

*id*: The key ID

*id_size*: Holds the size of the serial field.

**Description:** 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_crt_set_basic_constraints

`int gnutls_x509_crt_set_basic_constraints (`*gnutls_x509_crt_t* `crt`, *unsigned int* `ca`, *int* `pathLenConstraint`[Function]`)`

*crt*: a certificate of type `gnutls_x509_crt_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.

**Description:** This function will set the basicConstraints certificate extension.

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

### gnutls_x509_crt_set_ca_status

`int gnutls_x509_crt_set_ca_status (`*gnutls_x509_crt_t* **`crt`**, *unsigned int* **`ca`**`)` [Function]

*crt*: a certificate of type `gnutls_x509_crt_t`

*ca*: true(1) or false(0). Depending on the Certificate authority status.

**Description:** This function will set the basicConstraints certificate extension. Use `gnutls_x509_crt_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_crt_set_crl_dist_points

`int gnutls_x509_crt_set_crl_dist_points (`*gnutls_x509_crt_t* **`crt`**, *gnutls_x509_subject_alt_name_t* **`type`**, *const void ** **`data_string`**, *unsigned int* **`reason_flags`**`)` [Function]

*crt*: a certificate of type `gnutls_x509_crt_t`

*type*: is one of the gnutls_x509_subject_alt_name_t enumerations

*data_string*: The data to be set

*reason_flags*: revocation reasons

**Description:** 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_crt_set_crl_dist_points2

`int gnutls_x509_crt_set_crl_dist_points2 (`*gnutls_x509_crt_t* **`crt`**, *gnutls_x509_subject_alt_name_t* **`type`**, *const void ** **`data`**, *unsigned int* **`data_size`**, *unsigned int* **`reason_flags`**`)` [Function]

*crt*: a certificate of type `gnutls_x509_crt_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

**Description:** 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_crt_set_crq

`int gnutls_x509_crt_set_crq (`*gnutls_x509_crt_t* **`crt`**, *gnutls_x509_crq_t* **`crq`**`)` [Function]

*crt*: a certificate of type `gnutls_x509_crt_t`

*crq*: holds a certificate request

**Description:** 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.

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

### gnutls_x509_crt_set_crq_extensions

`int gnutls_x509_crt_set_crq_extensions (`*gnutls_x509_crt_t* **`crt`**, *gnutls_x509_crq_t* **`crq`**`)` [Function]

*crt*: a certificate of type `gnutls_x509_crt_t`

*crq*: holds a certificate request

**Description:** 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_crt_set_dn_by_oid

`int gnutls_x509_crt_set_dn_by_oid (`*gnutls_x509_crt_t* **`crt`**, *const char ** **`oid`**, *unsigned int* **`raw_flag`**, *const void ** **`name`**, *unsigned int* **`sizeof_name`**`)` [Function]

*crt*: a certificate of type `gnutls_x509_crt_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`

**Description:** 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_crt_set_expiration_time

`int gnutls_x509_crt_set_expiration_time (`*gnutls_x509_crt_t* **`cert`**, *time_t* **`exp_time`**`)` [Function]

*cert*: a certificate of type `gnutls_x509_crt_t`

*exp_time*: The actual time

**Description:** This function will set the time this Certificate will expire.

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

### gnutls_x509_crt_set_extension_by_oid

`int gnutls_x509_crt_set_extension_by_oid (`*gnutls_x509_crt_t* **`crt`**, *const char ** **`oid`**, *const void ** **`buf`**, *size_t* **`sizeof_buf`**, *unsigned int* **`critical`**`)` [Function]

*crt*: a certificate of type `gnutls_x509_crt_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 (0) if the extension is to be marked as critical

**Description:** 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_crt_set_issuer_dn_by_oid

`int gnutls_x509_crt_set_issuer_dn_by_oid (`*gnutls_x509_crt_t* **`crt`**, *const char ** **`oid`**, *unsigned int* **`raw_flag`**, *const void ** **`name`**, *unsigned int* **`sizeof_name`**`)` [Function]

*crt*: a certificate of type `gnutls_x509_crt_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`

**Description:** 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_crt_set_key

`int gnutls_x509_crt_set_key (`*gnutls_x509_crt_t* **`crt`**, *gnutls_x509_privkey_t* **`key`**`)` [Function]

*crt*: a certificate of type `gnutls_x509_crt_t`

*key*: holds a private key

**Description:** 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_crt_set_key_purpose_oid

`int gnutls_x509_crt_set_key_purpose_oid (`*gnutls_x509_crt_t* `cert`*, const void * oid, unsigned int* `critical``)` [Function]

*cert*: a certificate of type `gnutls_x509_crt_t`

*oid*: a pointer to a null terminated string that holds the OID

*critical*: Whether this extension will be critical or not

**Description:** 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_crt_set_key_usage

`int gnutls_x509_crt_set_key_usage (`*gnutls_x509_crt_t* `crt`*, unsigned int* `usage``)` [Function]

*crt*: a certificate of type `gnutls_x509_crt_t`

*usage*: an ORed sequence of the GNUTLS_KEY_* elements.

**Description:** This function will set the keyUsage certificate extension.

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

### gnutls_x509_crt_set_proxy

`int gnutls_x509_crt_set_proxy (`*gnutls_x509_crt_t* `crt`*, int* `pathLenConstraint`*, const char * policyLanguage, const char * policy, size_t* `sizeof_policy``)` [Function]

*crt*: a certificate of type `gnutls_x509_crt_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*: opaque byte array with policy language, can be `NULL`

*sizeof_policy*: size of `policy`.

**Description:** This function will set the proxyCertInfo extension.

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

### gnutls_x509_crt_set_proxy_dn

`int gnutls_x509_crt_set_proxy_dn (`*gnutls_x509_crt_t* `crt`*, gnutls_x509_crt_t* `eecrt`*, unsigned int* `raw_flag`*, const void * name, unsigned int* `sizeof_name``)` [Function]

*crt*: a gnutls_x509_crt_t structure with the new proxy cert

*eecrt*: 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`

**Description:** This function will set the subject in `crt`to the end entity's `eecrt`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_crt_set_dn_by_oid()` or similar.

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

## gnutls_x509_crt_set_serial

`int gnutls_x509_crt_set_serial` (*gnutls_x509_crt_t* `cert`, *const void * serial*, *size_t* `serial_size`) [Function]

*cert*: a certificate of type `gnutls_x509_crt_t`

*serial*: The serial number

*serial_size*: Holds the size of the serial field.

**Description:** This function will set the X.509 certificate's serial number. 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 opaque.

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

## gnutls_x509_crt_set_subject_alternative_name

`int gnutls_x509_crt_set_subject_alternative_name` (*gnutls_x509_crt_t* `crt`, *gnutls_x509_subject_alt_name_t* `type`, *const char * data_string*) [Function]

*crt*: a certificate of type `gnutls_x509_crt_t`

*type*: is one of the gnutls_x509_subject_alt_name_t enumerations

*data_string*: The data to be set, a (0) terminated string

**Description:** 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 incosistent with `gnutls_x509_crt_get_subject_alt_name()`.

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

## gnutls_x509_crt_set_subject_alt_name

`int gnutls_x509_crt_set_subject_alt_name` (*gnutls_x509_crt_t* `crt`, *gnutls_x509_subject_alt_name_t* `type`, *const void * data*, *unsigned int* `data_size`, *unsigned int* `flags`) [Function]

*crt*: a certificate of type `gnutls_x509_crt_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.

**Description:** 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_crt_set_subject_key_id

`int gnutls_x509_crt_set_subject_key_id (`*gnutls_x509_crt_t* **`cert`**, *const void * * **`id`**, *size_t* **`id_size`**`)` [Function]

*cert*: a certificate of type `gnutls_x509_crt_t`

*id*: The key ID

*id_size*: Holds the size of the serial field.

**Description:** 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_crt_set_version

`int gnutls_x509_crt_set_version (`*gnutls_x509_crt_t* **`crt`**, *unsigned int* **`version`**`)` [Function]

*crt*: a certificate of type `gnutls_x509_crt_t`

*version*: holds the version number. For X.509v1 certificates must be 1.

**Description:** 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_crt_set_subject_alt_name()` or `gnutls_x509_crt_set_key_usage()`.

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

### gnutls_x509_crt_sign

`int gnutls_x509_crt_sign (`*gnutls_x509_crt_t* **`crt`**, *gnutls_x509_crt_t* **`issuer`**, *gnutls_x509_privkey_t* **`issuer_key`**`)` [Function]

*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

**Description:** This function is the same a `gnutls_x509_crt_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_crt_sign2

`int gnutls_x509_crt_sign2 (`*gnutls_x509_crt_t* **`crt`**, *gnutls_x509_crt_t* **`issuer`**, *gnutls_x509_privkey_t* **`issuer_key`**, *gnutls_digest_algorithm_t* **`dig`**, *unsigned int* **`flags`**`)` [Function]

*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

**Description:** 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_verify

`int gnutls_x509_crt_verify (`*gnutls_x509_crt_t* **`cert`**, *const gnutls_x509_crt_t ** **`CA_list`**, *int* **`CA_list_length`**, *unsigned int* **`flags`**, *unsigned int ** **`verify`**`)` [Function]

*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.

**Description:** This function will try to verify the given certificate and return its status.

**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 opaque object pointer.

**Description:** 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**, *gnutls_x509_crt_fmt_t* **format**, *void ** **output_data**, *size_t ** **output_data_size**) [Function]

*dn*: Holds the opaque 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)

**Description:** 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_get_rdn_ava

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

*dn*: input variable with opaque DN pointer

*irdn*: index of RDN

*iava*: index of AVA.

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

**Description:** Get pointers to data within the DN.

Note that `ava`will contain pointers into the `dn`structure, so you should not modify any data or deallocate it. Note also that the DN in turn points into the original certificate structure, and thus you may not deallocate the certificate and continue to access `dn`.

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

## gnutls_x509_dn_import

`int gnutls_x509_dn_import` (*gnutls_x509_dn_t* **dn**, *const gnutls_datum_t ** **data**) [Function]

*dn*: the structure that will hold the imported DN

*data*: should contain a DER encoded RDN sequence

**Description:** 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

**Description:** 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

**Description:** This function will inform about known DN OIDs. This is useful since functions like `gnutls_x509_crt_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_privkey_cpy

`int gnutls_x509_privkey_cpy` (*gnutls_x509_privkey_t dst*, *gnutls_x509_privkey_t src*) [Function]

*dst*: The destination key, which should be initialized.

*src*: The source key

**Description:** 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

**Description:** This function will deinitialize a private key structure.

### gnutls_x509_privkey_export

`int gnutls_x509_privkey_export` (*gnutls_x509_privkey_t key*, *gnutls_x509_crt_fmt_t format*, *void * output_data*, *size_t * output_data_size*) [Function]

*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)

**Description:** 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_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

**Description:** 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

**Description:** 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:** 3.0.0

## 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)

**Description:** 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* `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`)` [Function]

*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

**Description:** 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* `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*) [Function]

*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 mod (p-1)

*e2*: will hold e2 = d mod (q-1)

**Description:** 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

**Description:** 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`, *gnutls_pk_algorithm_t* `algo`, *unsigned int* `bits`, *unsigned int* `flags`) [Function]

*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.

**Description:** This function will generate a random private key. Note that this function must be called on an empty private key.

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`, *unsigned int* `flags`, *unsigned char ** `output_data`, *size_t ** `output_data_size``)` [Function]

*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)

**Description:** This function will return a unique ID the 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

**Description:** 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

**Description:** 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 of "RSA PRIVATE KEY", or "DSA PRIVATE KEY".

**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

**Description:** 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

**Description:** 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:** 3.0.0

### 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.

**Description:** 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.

**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* `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*`)` [Function]

*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

**Description:** 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* `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*`)` [Function]

*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

*e1*: holds e1 = d mod (p-1)

*e2*: holds e2 = d mod (q-1)

**Description:** 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

**Description:** 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

**Description:** 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

**Description:** 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 * sizeof_buf*`)` [Function]

*idn*: should contain a DER encoded RDN sequence

*buf*: a pointer to a structure to hold the peer's name

*sizeof_buf*: holds the size of `buf`

**Description:** 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 RFC2253.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, or `GNUTLS_E_SHORT_MEMORY_BUFFER` is returned and * `sizeof_buf`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 * `sizeof_buf`*`)`* [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 (0) then the raw DER data are returned.

*buf*: a pointer to a structure to hold the peer's name

*sizeof_buf*: holds the size of `buf`

**Description:** This function will return the name of the given Object identifier, of the RDN sequence. The name will be encoded using the rules from RFC2253.

**Returns:** On success, `GNUTLS_E_SUCCESS` (0) is returned, or `GNUTLS_E_SHORT_MEMORY_BUFFER` is returned and * `sizeof_buf`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`, void * `buf`, size_t * `sizeof_buf`*`)`* [Function]

*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

*sizeof_buf*: holds the size of `buf`

**Description:** 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 * `sizeof_buf`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 `list`, const gnutls_x509_crt_t * `clist`, int `clist_size`, unsigned int `flags`*`)`* [Function]

*list*: The structure of the list

*clist*: A list of CAs

*clist_size*: The length of the CA list

*flags*: should be 0.

**Description:** 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.

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

*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

**Description:** 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.

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

**Since:** 3.0.0

### gnutls_x509_trust_list_add_named_crt

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

*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.

**Description:** This function will add the given certificate to the trusted list and associate it with a name. The certificate will not be 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

### gnutls_x509_trust_list_deinit

`void gnutls_x509_trust_list_deinit` (*gnutls_x509_trust_list_t* `list`, *unsigned int* `all`) [Function]

*list*: The structure to be deinitialized

*all*: if non-(0) it will deinitialize all the certificates and CRLs contained in the structure.

**Description:** This function will deinitialize a trust list.

**Since:** 3.0.0

### gnutls_x509_trust_list_get_issuer

`int gnutls_x509_trust_list_get_issuer (`*gnutls_x509_trust_list_t* `list`, *gnutls_x509_crt_t* `cert`, *gnutls_x509_crt_t ** `issuer`, *unsigned int* `flags``)` [Function]

*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 (0).

**Description:** This function will attempt to find the issuer of the given certificate.

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

**Since:** 3.0.0

### gnutls_x509_trust_list_init

`int gnutls_x509_trust_list_init (`*gnutls_x509_trust_list_t ** `list`, *unsigned int* `size``)` [Function]

*list*: The structure to be initialized

*size*: The size of the internal hash table. Use (0) for default size.

**Description:** 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_verify_crt

`int gnutls_x509_trust_list_verify_crt (`*gnutls_x509_trust_list_t* `list`, *gnutls_x509_crt_t ** `cert_list`, *unsigned int* `cert_list_size`, *unsigned int* `flags`, *unsigned int ** `verify`, *gnutls_verify_output_function* `func``)` [Function]

*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.

*verify*: will hold the certificate verification output.

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

**Description:** This function will try to verify the given certificate and return its status.

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

**Since:** 3.0.0

### gnutls_x509_trust_list_verify_named_crt

`int gnutls_x509_trust_list_verify_named_crt` [Function]
(*gnutls_x509_trust_list_t* `list`, *gnutls_x509_crt_t* `cert`, *const void ** `name`, *size_t* `name_size`, *unsigned int* `flags`, *unsigned int ** `verify`, *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.

*verify*: will hold the certificate verification output.

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

**Description:** This function will try to find a matching named certificate. If a match is found the certificate is considered valid. In addition to that this function will also check CRLs.

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

**Since:** 3.0.0

## C.4 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_crt_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

**Description:** 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_crt_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_crt_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

**Description:** This funtion 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_crt_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

**Description:** This funtion 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_crt_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_crt_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

**Description:** This funtion 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_crt_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

**Description:** This funtion 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_crt_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_crt_fmt_t* **format**)

*c*: A certificate credentials structure

*file*: filename of the keyring.

*format*: format of keyring.

**Description:** 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 opaque ** **data**, *size_t* **dlen**, *gnutls_openpgp_crt_fmt_t* **format**)

*c*: A certificate credentials structure

*data*: buffer with keyring data.

*dlen*: length of data buffer.

*format*: the format of the keyring

**Description:** 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_crt_check_hostname

`int gnutls_openpgp_crt_check_hostname (`*gnutls_openpgp_crt_t* **`key`**, *const char ** **`hostname`**`)` [Function]

*key*: should contain a gnutls_openpgp_crt_t structure

*hostname*: A null terminated string that contains a DNS name

**Description:** 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:** GNUTLS_E_SUCCESS on success, or an error code.

### gnutls_openpgp_crt_deinit

`void gnutls_openpgp_crt_deinit (`*gnutls_openpgp_crt_t* **`key`**`)` [Function]

*key*: The structure to be initialized

**Description:** This function will deinitialize a key structure.

### gnutls_openpgp_crt_export

`int gnutls_openpgp_crt_export (`*gnutls_openpgp_crt_t* **`key`**, *gnutls_openpgp_crt_fmt_t* **`format`**, *void ** **`output_data`**, *size_t ** **`output_data_size`**`)` [Function]

*key*: Holds the key.

*format*: One of gnutls_openpgp_crt_fmt_t elements.

*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)

**Description:** 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_crt_get_auth_subkey

`int gnutls_openpgp_crt_get_auth_subkey (`*gnutls_openpgp_crt_t* **`crt`**, *gnutls_openpgp_keyid_t* **`keyid`**, *unsigned int* **`flag`**`)` [Function]

*crt*: the structure that contains the OpenPGP public key.

*keyid*: the struct to save the keyid.

*flag*: Non (0) indicates that a valid subkey is always returned.

**Description:** Returns the 64-bit keyID of the first valid OpenPGP subkey marked for authentication. If flag is non (0) and no authentication subkey exists, then a valid subkey will be returned even if it is not marked for authentication. Returns the 64-bit keyID of the first valid OpenPGP subkey marked for authentication. If flag is non

(0) 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_crt_get_creation_time

`time_t gnutls_openpgp_crt_get_creation_time` [Function]
(*gnutls_openpgp_crt_t* `key`)

*key*: the structure that contains the OpenPGP public key.

**Description:** Get key creation time.

**Returns:** the timestamp when the OpenPGP key was created.

### gnutls_openpgp_crt_get_expiration_time

`time_t gnutls_openpgp_crt_get_expiration_time` [Function]
(*gnutls_openpgp_crt_t* `key`)

*key*: the structure that contains the OpenPGP public key.

**Description:** 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_crt_get_fingerprint

`int gnutls_openpgp_crt_get_fingerprint` (*gnutls_openpgp_crt_t* `key`, *void * fpr*, *size_t * fprlen*) [Function]

*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.

**Description:** 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_crt_get_key_id

`int gnutls_openpgp_crt_get_key_id` (*gnutls_openpgp_crt_t* `key`, *gnutls_openpgp_keyid_t* `keyid`) [Function]

*key*: the structure that contains the OpenPGP public key.

*keyid*: the buffer to save the keyid.

**Description:** Get key id string.

**Returns:** the 64-bit keyID of the OpenPGP key.

**Since:** 2.4.0

### gnutls_openpgp_crt_get_key_usage

`int gnutls_openpgp_crt_get_key_usage (`*gnutls_openpgp_crt_t* `key`, *unsigned int ** `key_usage``)` [Function]

*key*: should contain a gnutls_openpgp_crt_t structure

*key_usage*: where the key usage bits will be stored

**Description:** 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_crt_get_name

`int gnutls_openpgp_crt_get_name (`*gnutls_openpgp_crt_t* `key`, *int* `idx`, *char ** `buf`, *size_t ** `sizeof_buf``)` [Function]

*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`.

**Description:** 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_crt_get_pk_algorithm

`gnutls_pk_algorithm_t gnutls_openpgp_crt_get_pk_algorithm (`*gnutls_openpgp_crt_t* `key`, *unsigned int ** `bits``)` [Function]

*key*: is an OpenPGP key

*bits*: if bits is non null it will hold the size of the parameters' in bits

**Description:** 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_crt_get_pk_dsa_raw

`int gnutls_openpgp_crt_get_pk_dsa_raw (`*gnutls_openpgp_crt_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

**Description:** 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_crt_get_pk_rsa_raw

`int gnutls_openpgp_crt_get_pk_rsa_raw` (*gnutls_openpgp_crt_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

**Description:** 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_crt_get_preferred_key_id

`int gnutls_openpgp_crt_get_preferred_key_id` (*gnutls_openpgp_crt_t* `key`, *gnutls_openpgp_keyid_t* `keyid`) [Function]

*key*: the structure that contains the OpenPGP public key.

*keyid*: the struct to save the keyid.

**Description:** 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_crt_get_revoked_status

`int gnutls_openpgp_crt_get_revoked_status` (*gnutls_openpgp_crt_t* `key`) [Function]

*key*: the structure that contains the OpenPGP public key.

**Description:** 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_crt_get_subkey_count

`int gnutls_openpgp_crt_get_subkey_count (`*gnutls_openpgp_crt_t* `key``)` [Function]

*key*: is an OpenPGP key

**Description:** 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_crt_get_subkey_creation_time

`time_t gnutls_openpgp_crt_get_subkey_creation_time` [Function]
`(`*gnutls_openpgp_crt_t* `key`, *unsigned int* `idx``)`

*key*: the structure that contains the OpenPGP public key.

*idx*: the subkey index

**Description:** Get subkey creation time.

**Returns:** the timestamp when the OpenPGP sub-key was created.

**Since:** 2.4.0

### gnutls_openpgp_crt_get_subkey_expiration_time

`time_t gnutls_openpgp_crt_get_subkey_expiration_time` [Function]
`(`*gnutls_openpgp_crt_t* `key`, *unsigned int* `idx``)`

*key*: the structure that contains the OpenPGP public key.

*idx*: the subkey index

**Description:** 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_crt_get_subkey_fingerprint

`int gnutls_openpgp_crt_get_subkey_fingerprint` [Function]
`(`*gnutls_openpgp_crt_t* `key`, *unsigned int* `idx`, *void ** `fpr`, *size_t ** `fprlen``)`

*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.

**Description:** 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_crt_get_subkey_id

`int gnutls_openpgp_crt_get_subkey_id` (*gnutls_openpgp_crt_t* `key`, *unsigned int* `idx`, *gnutls_openpgp_keyid_t* `keyid`) [Function]

*key*: the structure that contains the OpenPGP public key.

*idx*: the subkey index

*keyid*: the buffer to save the keyid.

**Description:** Get the subkey's key-id.

**Returns:** the 64-bit keyID of the OpenPGP key.

### gnutls_openpgp_crt_get_subkey_idx

`int gnutls_openpgp_crt_get_subkey_idx` (*gnutls_openpgp_crt_t* `key`, *const gnutls_openpgp_keyid_t* `keyid`) [Function]

*key*: the structure that contains the OpenPGP public key.

*keyid*: the keyid.

**Description:** Get subkey's index.

**Returns:** the index of the subkey or a negative error value.

**Since:** 2.4.0

### gnutls_openpgp_crt_get_subkey_pk_algorithm

`gnutls_pk_algorithm_t` `gnutls_openpgp_crt_get_subkey_pk_algorithm` (*gnutls_openpgp_crt_t* `key`, *unsigned int* `idx`, *unsigned int * * `bits`) [Function]

*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

**Description:** 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_crt_get_subkey_pk_dsa_raw

`int gnutls_openpgp_crt_get_subkey_pk_dsa_raw` (*gnutls_openpgp_crt_t* `crt`, *unsigned int* `idx`, *gnutls_datum_t * * `p`, *gnutls_datum_t * * `q`, *gnutls_datum_t * * `g`, *gnutls_datum_t * * `y`) [Function]

*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

**Description:** 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_crt_get_subkey_pk_rsa_raw

`int gnutls_openpgp_crt_get_subkey_pk_rsa_raw` [Function]
(*gnutls_openpgp_crt_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

**Description:** 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_crt_get_subkey_revoked_status

`int gnutls_openpgp_crt_get_subkey_revoked_status` [Function]
(*gnutls_openpgp_crt_t* `key`, *unsigned int* `idx`)

*key*: the structure that contains the OpenPGP public key.

*idx*: is the subkey index

**Description:** 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_crt_get_subkey_usage

`int gnutls_openpgp_crt_get_subkey_usage` (*gnutls_openpgp_crt_t* `key`, *unsigned int* `idx`, *unsigned int* * `key_usage`) [Function]

*key*: should contain a gnutls_openpgp_crt_t structure

*idx*: the subkey index

*key_usage*: where the key usage bits will be stored

**Description:** 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_ENCIPHERMENT`.

A negative error code may be returned in case of parsing error.

**Returns:** key usage value.

**Since:** 2.4.0

## gnutls_openpgp_crt_get_version

`int gnutls_openpgp_crt_get_version (`*gnutls_openpgp_crt_t* `key``)` [Function]

*key*: the structure that contains the OpenPGP public key.

**Description:** Extract the version of the OpenPGP key.

**Returns:** the version number is returned, or a negative error code on errors.

## gnutls_openpgp_crt_import

`int gnutls_openpgp_crt_import (`*gnutls_openpgp_crt_t* `key`, *const gnutls_datum_t ** `data`, *gnutls_openpgp_crt_fmt_t* `format``)` [Function]

*key*: The structure to store the parsed key.

*data*: The RAW or BASE64 encoded key.

*format*: One of gnutls_openpgp_crt_fmt_t elements.

**Description:** This function will convert the given RAW or Base64 encoded key to the native `gnutls_openpgp_crt_t` format. The output will be stored in 'key'.

**Returns:** `GNUTLS_E_SUCCESS` on success, or an error code.

## gnutls_openpgp_crt_init

`int gnutls_openpgp_crt_init (`*gnutls_openpgp_crt_t ** `key``)` [Function]

*key*: The structure to be initialized

**Description:** This function will initialize an OpenPGP key structure.

**Returns:** `GNUTLS_E_SUCCESS` on success, or an error code.

## gnutls_openpgp_crt_print

`int gnutls_openpgp_crt_print (`*gnutls_openpgp_crt_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.

**Description:** 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_crt_set_preferred_key_id

`int gnutls_openpgp_crt_set_preferred_key_id` [Function]
(*gnutls_openpgp_crt_t* `key`, *const gnutls_openpgp_keyid_t* `keyid`)

*key*: the structure that contains the OpenPGP public key.

*keyid*: the selected keyid

**Description:** This allows setting a preferred key id for the given certificate. This key will be used by functions that involve key handling.

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

### gnutls_openpgp_crt_verify_ring

`int gnutls_openpgp_crt_verify_ring` (*gnutls_openpgp_crt_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.

**Description:** 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.

**Returns:** `GNUTLS_E_SUCCESS` on success, or an error code.

### gnutls_openpgp_crt_verify_self

`int gnutls_openpgp_crt_verify_self` (*gnutls_openpgp_crt_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.

**Description:** 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* [Function]
`ring`, *const gnutls_openpgp_keyid_t* `keyid`, *unsigned int* `flags`)

*ring*: holds the keyring to check against

*keyid*: will hold the keyid to check for.

*flags*: unused (should be 0)

**Description:** 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

**Description:** This function will deinitialize a keyring structure.

### gnutls_openpgp_keyring_get_crt

`int gnutls_openpgp_keyring_get_crt (`*gnutls_openpgp_keyring_t* ***ring***, *unsigned int* ***idx***, *gnutls_openpgp_crt_t ** ***cert***`)` [Function]

*ring*: Holds the keyring.

*idx*: the index of the certificate to export

*cert*: An uninitialized gnutls_openpgp_crt_t structure

**Description:** 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 deinited.

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

### gnutls_openpgp_keyring_get_crt_count

`int gnutls_openpgp_keyring_get_crt_count (`*gnutls_openpgp_keyring_t* ***ring***`)` [Function]

*ring*: is an OpenPGP key ring

**Description:** 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_crt_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.

**Description:** 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* * `keyring``)` [Function]

*keyring*: The structure to be initialized

**Description:** 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* `key``)` [Function]

*key*: The structure to be initialized

**Description:** This function will deinitialize a key structure.

### gnutls_openpgp_privkey_export

`int gnutls_openpgp_privkey_export (`*gnutls_openpgp_privkey_t* `key`, *gnutls_openpgp_crt_fmt_t* `format`, *const char* * `password`, *unsigned int* `flags`, *void* * `output_data`, *size_t* * `output_data_size``)` [Function]

*key*: Holds the key.

*format*: One of gnutls_openpgp_crt_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)

**Description:** 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_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

**Description:** 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

**Description:** 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

**Description:** 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

**Description:** 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.

**Description:** 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* `key`, *gnutls_openpgp_keyid_t* `keyid`) [Function]

*key*: the structure that contains the OpenPGP secret key.

*keyid*: the buffer to save the keyid.

**Description:** 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

**Description:** 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.

**Description:** 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.

**Description:** 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

**Description:** 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

**Description:** Get subkey creation time.

**Returns:** the timestamp when the OpenPGP key was created.

**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.

**Description:** 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.

**Description:** 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.

**Description:** 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` [Function]
`gnutls_openpgp_privkey_get_subkey_pk_algorithm` (*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

**Description:** 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

**Description:** 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)

**Description:** 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

**Description:** 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

**Description:** 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

**Description:** This allows setting a preferred key id for the given certificate. This key will be used by functions that involve key handling.

**Returns:** On success, 0 is returned, or an error code.

### gnutls_openpgp_set_recv_key_function

`void gnutls_openpgp_set_recv_key_function (`*gnutls_session_t session, gnutls_openpgp_recv_key_func func*`)` [Function]

*session*: a TLS session

*func*: the callback

**Description:** This funtion 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.

## C.5 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* `* pass`)` [Function]

*bag*: The bag

*pass*: The password used for encryption, must be ASCII.

**Description:** 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

**Description:** 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

**Description:** 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

**Description:** 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.

**Description:** 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)

**Description:** 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)

**Description:** 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

**Description:** 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

**Description:** 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.

**Description:** 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_crt

`int gnutls_pkcs12_bag_set_crt (`*gnutls_pkcs12_bag_t* `bag`, *gnutls_x509_crt_t* `crt``)` [Function]

*bag*: The bag

*crt*: the certificate to be copied.

**Description:** 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`, *gnutls_pkcs12_bag_type_t* `type`, *const gnutls_datum_t ** `data``)` [Function]

*bag*: The bag

*type*: The data's type

*data*: the data to be copied.

**Description:** 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`, *int* `indx`, *const char ** `name``)` [Function]

*bag*: The bag

*indx*: The bag's element to add the id

*name*: the name

**Description:** 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`**, *int* **`indx`**, *const gnutls_datum_t ** **`id`**`)` [Function]

*bag*: The bag

*indx*: The bag's element to add the id

*id*: the ID

**Description:** 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

**Description:** This function will deinitialize a PKCS12 structure.

### gnutls_pkcs12_export

`int gnutls_pkcs12_export (`*gnutls_pkcs12_t* **`pkcs12`**, *gnutls_x509_crt_fmt_t* **`format`**, *void ** **`output_data`**, *size_t ** **`output_data_size`**`)` [Function]

*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)

**Description:** 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_generate_mac

`int gnutls_pkcs12_generate_mac (`*gnutls_pkcs12_t* **`pkcs12`**, *const char ** **`pass`**`)` [Function]

*pkcs12*: should contain a gnutls_pkcs12_t structure

*pass*: The password for the MAC

**Description:** 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`**, *gnutls_pkcs12_bag_t* **`bag`**`)` [Function]

*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

**Description:** 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 gnutls_datum_t* * **`data`**, *gnutls_x509_crt_fmt_t* **`format`**, *unsigned int* **`flags`**`)` [Function]

*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

**Description:** 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

**Description:** 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`**, *gnutls_pkcs12_bag_t* **`bag`**`)` [Function]

*pkcs12*: should contain a gnutls_pkcs12_t structure

*bag*: An initialized bag

**Description:** 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_verify_mac

`int gnutls_pkcs12_verify_mac` (*gnutls_pkcs12_t* `pkcs12`, *const char ** `pass`) [Function]

*pkcs12*: should contain a gnutls_pkcs12_t structure

*pass*: The password for the MAC

**Description:** This function will verify the MAC for the PKCS12 structure.

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

## C.6 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

**Description:** 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.

**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_*

**Description:** 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_crt

`int gnutls_pkcs11_copy_x509_crt (`*const char * `token_url`*, *gnutls_x509_crt_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_*

**Description:** 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`*, *gnutls_x509_privkey_t `key`*, *const char * `label`*, *unsigned int `key_usage`*, *unsigned int `flags`*`)` [Function]

*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

**Description:** 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_deinit

`void gnutls_pkcs11_deinit ( `*void*`)` [Function]

**Description:** This function will deinitialize the PKCS 11 subsystem in gnutls.

**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

**Description:** 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_init

`int gnutls_pkcs11_init (`*unsigned int* **`flags`**, *const char ** **`deprecated_config_file`**`)` [Function]

*flags*: `GNUTLS_PKCS11_FLAG_MANUAL` or `GNUTLS_PKCS11_FLAG_AUTO`

*deprecated_config_file*: either NULL or the location of a deprecated configuration file

**Description:** 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 by `gnutls_global_init()` using the `GNUTLS_PKCS11_FLAG_AUTO`. If other option is required then it must be called before it.

**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

**Description:** 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 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)

**Description:** This function will export the PKCS11 object data. It is normal for data to be inaccesible 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.

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 `GNUTLS_E_SUCCESS` (0) on success.

**Since:** 2.12.0

### gnutls_pkcs11_obj_export_url

`int gnutls_pkcs11_obj_export_url (`*gnutls_pkcs11_obj_t* `obj`, *gnutls_pkcs11_url_type_t* `detailed`, *char ** `url``)` [Function]

*obj*: Holds the PKCS 11 certificate

*detailed*: non zero if a detailed URL is required

*url*: will contain an allocated url

**Description:** 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_get_info

`int gnutls_pkcs11_obj_get_info (`*gnutls_pkcs11_obj_t* `crt`, *gnutls_pkcs11_obj_info_t* `itype`, *void * `output`*, *size_t * `output_size`*`)` [Function]

*crt*: 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

**Description:** 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 (`*gnutls_pkcs11_obj_t* `obj`*`)` [Function]

*obj*: Holds the PKCS 11 object

**Description:** This function will return the type of the certificate being stored in the structure.

**Returns:** The type of the certificate.

**Since:** 2.12.0

### gnutls_pkcs11_obj_import_url

`int gnutls_pkcs11_obj_import_url (`*gnutls_pkcs11_obj_t* `cert`, *const char * `url`*, *unsigned int* `flags`*`)` [Function]

*cert*: The structure to store the parsed certificate

*url*: a PKCS 11 url identifying the key

*flags*: One of GNUTLS_PKCS11_OBJ_* flags

**Description:** This function will "import" a PKCS 11 URL identifying a certificate key 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

**Description:** 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 * **p_list**, unsigned int * **n_list**, const char * **url**, gnutls_pkcs11_obj_attr_t **attrs**, unsigned int **flags**`)` [Function]

*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*: One of GNUTLS_PKCS11_OBJ_* flags

**Description:** 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_privkey_deinit

`void gnutls_pkcs11_privkey_deinit (`*gnutls_pkcs11_privkey_t **key**`)` [Function]

*key*: The structure to be initialized

**Description:** This function will deinitialize a private key structure.

## gnutls_pkcs11_privkey_export_url

`int gnutls_pkcs11_privkey_export_url (`*gnutls_pkcs11_privkey_t **key**, gnutls_pkcs11_url_type_t **detailed**, char ** **url**`)` [Function]

*key*: Holds the PKCS 11 key

*detailed*: non zero if a detailed URL is required

*url*: will contain an allocated url

**Description:** 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`, *gnutls_pk_algorithm_t* `pk`, *unsigned int* `bits`, *const char* `* label`, *unsigned int* `flags``)` [Function]

*url*: a token URL

*pk*: the public key algorithm

*bits*: the security bits

*label*: a label

*flags*: should be zero

**Description:** This function will generate a private key in the specified by the `urltoken`. The pivate 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:** 3.0.0

### gnutls_pkcs11_privkey_get_info

`int gnutls_pkcs11_privkey_get_info (`*gnutls_pkcs11_privkey_t* `pkey`, *gnutls_pkcs11_obj_info_t* `itype`, *void* `* output`, *size_t* `* output_size``)` [Function]

*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

**Description:** 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 (`*gnutls_pkcs11_privkey_t* `key`, *unsigned int* `* bits``)` [Function]

*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

**Description:** 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*: sequence of GNUTLS_PKCS_PRIVKEY_*

**Description:** 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

**Description:** This function will initialize an private key structure.

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

## gnutls_pkcs11_set_pin_function

`void gnutls_pkcs11_set_pin_function` [Function]
`(`*gnutls_pkcs11_pin_callback_t* `fn`*, void * * `userdata`*`)`

*fn*: The PIN callback, a `gnutls_pkcs11_pin_callback_t()` function.

*userdata*: data to be supplied to callback

**Description:** This function will set a callback function to be used when a PIN is required for PKCS 11 operations. See `gnutls_pkcs11_pin_callback_t()` on how the callback should behave.

**Since:** 2.12.0

## gnutls_pkcs11_set_token_function

`void gnutls_pkcs11_set_token_function` [Function]
`(`*gnutls_pkcs11_token_callback_t* `fn`*, void * * `userdata`*`)`

*fn*: The token callback

*userdata*: data to be supplied to callback

**Description:** 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_*)

**Description:** This function will return information about the PKCS 11 token flags. The flags from the `gnutls_pkcs11_token_info_t` enumeration.

**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`, gnutls_pkcs11_token_info_t `ttype`, void * `output`, size_t * `output_size`*`)`  [Function]

*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

**Description:** 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`, int `idx`, unsigned long * `mechanism`*`)`  [Function]

*url*: should contain a PKCS 11 URL

*idx*: The index of the mechanism

*mechanism*: The PKCS 11 mechanism ID

**Description:** 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_url

`int gnutls_pkcs11_token_get_url (`*unsigned int `seq`, gnutls_pkcs11_url_type_t `detailed`, char ** `url`*`)`  [Function]

*seq*: sequence number starting from 0

*detailed*: non zero if a detailed URL is required

*url*: will contain an allocated url

**Description:** 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

**Description:** 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_pkcs11_pin_flag_t`.

**Description:** 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`.

**Description:** 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_crt_import_pkcs11

`int gnutls_x509_crt_import_pkcs11 (`*gnutls_x509_crt_t* **`crt`**, *gnutls_pkcs11_obj_t* **`pkcs11_crt`**`)` [Function]

*crt*: A certificate of type `gnutls_x509_crt_t`

*pkcs11_crt*: A PKCS 11 object that contains a certificate

**Description:** This function will import a PKCS 11 certificate to a `gnutls_x509_crt_t` structure.

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

**Since:** 2.12.0

## gnutls_x509_crt_import_pkcs11_url

`int gnutls_x509_crt_import_pkcs11_url (`*gnutls_x509_crt_t* **`crt`**, *const char ** **`url`**, *unsigned int* **`flags`**`)` [Function]

*crt*: A certificate of type `gnutls_x509_crt_t`

*url*: A PKCS 11 url

*flags*: One of GNUTLS_PKCS11_OBJ_* flags

**Description:** 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_crt_list_import_pkcs11

`int gnutls_x509_crt_list_import_pkcs11 (`*gnutls_x509_crt_t ** **`certs`**, *unsigned int* **`cert_max`**, *gnutls_pkcs11_obj_t * const* **`objs`**, *unsigned int* **`flags`**`)` [Function]

*certs*: A list of certificates of type `gnutls_x509_crt_t`

*cert_max*: The maximum size of the list

*objs*: A list of PKCS 11 objects

*flags*: 0 for now

**Description:** This function will import a PKCS 11 certificate list to a list of `gnutls_x509_crt_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

## C.7 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* **res**, *const char*** **names**, *int* **names_size**, *gnutls_pcert_st ** **pcert_list**, *int* **pcert_list_size**, *gnutls_privkey_t* **key**) [Function]

*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_x509_privkey_t key

**Description:** 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`. The `pcert_list`and `key`will become part of the credentials structure and must not be deallocated. They will be automatically deallocated when `res`is deinitialized.

**Returns:** `GNUTLS_E_SUCCESS` (0) on success, or a negative error code.

**Since:** 3.0.0

### gnutls_pcert_deinit

`void gnutls_pcert_deinit` (*gnutls_pcert_st ** **pcert**) [Function]

*pcert*: The structure to be deinitialized

**Description:** This function will deinitialize a pcert structure.

**Since:** 3.0.0

### gnutls_pcert_import_openpgp

`int gnutls_pcert_import_openpgp` (*gnutls_pcert_st** **pcert**, *gnutls_openpgp_crt_t* **crt**, *unsigned int* **flags**) [Function]

*pcert*: The pcert structure

*crt*: The raw certificate to be imported

*flags*: zero for now

**Description:** 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:** 3.0.0

### gnutls_pcert_import_openpgp_raw

`int gnutls_pcert_import_openpgp_raw (`*gnutls_pcert_st * `pcert`*, *const gnutls_datum_t* `cert`*, *gnutls_openpgp_crt_fmt_t* `format`*, *gnutls_openpgp_keyid_t* `keyid`*, *unsigned int* `flags`*)* [Function]

*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

**Description:** 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:** 3.0.0

### gnutls_pcert_import_x509

`int gnutls_pcert_import_x509 (`*gnutls_pcert_st* `pcert`*, *gnutls_x509_crt_t* `crt`*, *unsigned int* `flags`*)* [Function]

*pcert*: The pcert structure

*crt*: The raw certificate to be imported

*flags*: zero for now

**Description:** 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:** 3.0.0

### gnutls_pcert_import_x509_raw

`int gnutls_pcert_import_x509_raw (`*gnutls_pcert_st * `pcert`*, const gnutls_datum_t* `cert`*, *gnutls_x509_crt_fmt_t* `format`*, *unsigned int* `flags`*)* [Function]

*pcert*: The pcert structure

*cert*: The raw certificate to be imported

*format*: The format of the certificate

*flags*: zero for now

**Description:** 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:** 3.0.0

## gnutls_pcert_list_import_x509_raw

`int gnutls_pcert_list_import_x509_raw (`*gnutls_pcert_st * `pcerts`, unsigned int * `pcert_max`, const gnutls_datum_t * `data`, gnutls_x509_crt_fmt_t `format`, unsigned int `flags`*`)`* [Function]

*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.

**Description:** 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:** 3.0.0

## gnutls_privkey_decrypt_data

`int gnutls_privkey_decrypt_data (`*gnutls_privkey_t `key`, unsigned int `flags`, const gnutls_datum_t * `ciphertext`, gnutls_datum_t * `plaintext`*`)`* [Function]

*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()`

**Description:** 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

**Description:** This function will deinitialize a private key structure.

**Since:** 2.12.0

## gnutls_privkey_get_pk_algorithm

`int gnutls_privkey_get_pk_algorithm` (*gnutls_privkey_t* `key`, *unsigned int* * `bits`) [Function]

*key*: should contain a `gnutls_privkey_t` structure

*bits*: If set will return the number of bits of the parameters (may be NULL)

**Description:** 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` (*gnutls_privkey_t* `key`) [Function]

*key*: should contain a `gnutls_privkey_t` structure

**Description:** 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_ext

`int gnutls_privkey_import_ext` (*gnutls_privkey_t* `pkey`, *gnutls_pk_algorithm_t* `pk`, *void** `userdata`, *gnutls_privkey_sign_func* `sign_func`, *gnutls_privkey_decrypt_func* `decrypt_func`, *unsigned int* `flags`) [Function]

*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

**Description:** This function will associate the given callbacks with the `gnutls_privkey_t` structure. At least one of the two callbacks must be non-null.

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

**Since:** 3.0.0

### gnutls_privkey_import_openpgp

`int gnutls_privkey_import_openpgp` (*gnutls_privkey_t* `pkey`, *gnutls_openpgp_privkey_t* `key`, *unsigned int* `flags`) [Function]

*pkey*: The private key

*key*: The private key to be imported

*flags*: Flags for the import

**Description:** 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_pkcs11

`int gnutls_privkey_import_pkcs11` (*gnutls_privkey_t* `pkey`, *gnutls_pkcs11_privkey_t* `key`, *unsigned int* `flags`) [Function]

*pkey*: The private key

*key*: The private key to be imported

*flags*: Flags for the import

**Description:** 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_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

**Description:** 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_init

`int gnutls_privkey_init (`*gnutls_privkey_t * **key***`)` [Function]

*key*: The structure to be initialized

**Description:** 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_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*: should be 0 for now

*data*: holds the data to be signed

*signature*: will contain the signature allocate with `gnutls_malloc()`

**Description:** 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.

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***, *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 for now

*hash_data*: holds the data to be signed

*signature*: will contain newly allocated signature

**Description:** 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.

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_pubkey_deinit

`void gnutls_pubkey_deinit (`*gnutls_pubkey_t* **`key`**`)` [Function]

*key*: The structure to be deinitialized

**Description:** This function will deinitialize a public key structure.

**Since:** 2.12.0

## 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)

**Description:** 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.

**Since:** 2.12.0

## gnutls_pubkey_get_key_id

`int gnutls_pubkey_get_key_id (`*gnutls_pubkey_t* **`key`**, *unsigned int* **`flags`**, *unsigned char ** **`output_data`**, *size_t ** **`output_data_size`**`)` [Function]

*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)

**Description:** This function will return a unique ID the 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 int ** `usage``)` [Function]

*key*: should contain a `gnutls_pubkey_t` structure

*usage*: If set will return the number of bits of the parameters (may be NULL)

**Description:** 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`, *unsigned int* `flags`, *unsigned char ** `output_data`, *size_t ** `output_data_size`, *unsigned int ** `subkey``)` [Function]

*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)

*subkey*: Will be non zero if the key ID corresponds to a subkey

**Description:** This function will return a unique ID the 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:** 3.0.0

### gnutls_pubkey_get_pk_algorithm

`int gnutls_pubkey_get_pk_algorithm (`*gnutls_pubkey_t* `key`, *unsigned int ** `bits``)` [Function]

*key*: should contain a `gnutls_pubkey_t` structure

*bits*: If set will return the number of bits of the parameters (may be NULL)

**Description:** 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_pk_dsa_raw

`int gnutls_pubkey_get_pk_dsa_raw (`*gnutls_pubkey_t* `key`, *gnutls_datum_t ** `p`, *gnutls_datum_t ** `q`, *gnutls_datum_t ** `g`, *gnutls_datum_t ** `y``)` [Function]

*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

**Description:** 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:** 2.12.0

### gnutls_pubkey_get_pk_ecc_raw

`int gnutls_pubkey_get_pk_ecc_raw (`*gnutls_pubkey_t* `key`, *gnutls_ecc_curve_t ** `curve`, *gnutls_datum_t ** `x`, *gnutls_datum_t ** `y`)` [Function]

*key*: Holds the public key

*curve*: will hold the curve

*x*: will hold x

*y*: will hold y

**Description:** 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:** 3.0.0

### gnutls_pubkey_get_pk_ecc_x962

`int gnutls_pubkey_get_pk_ecc_x962` (*gnutls_pubkey_t* **key**, *gnutls_datum_t** **parameters**, *gnutls_datum_t * **ecpoint**) [Function]

*key*: Holds the public key

*parameters*: DER encoding of an ANSI X9.62 parameters

*ecpoint*: DER encoding of ANSI X9.62 ECPoint

**Description:** 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:** 3.0.0

### gnutls_pubkey_get_pk_rsa_raw

`int gnutls_pubkey_get_pk_rsa_raw` (*gnutls_pubkey_t* **key**, *gnutls_datum_t * **m**, *gnutls_datum_t * **e**) [Function]

*key*: Holds the certificate

*m*: will hold the modulus

*e*: will hold the public exponent

**Description:** 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:** 2.12.0

### gnutls_pubkey_get_preferred_hash_algorithm

`int gnutls_pubkey_get_preferred_hash_algorithm` (*gnutls_pubkey_t* **key**, *gnutls_digest_algorithm_t * **hash**, *unsigned int * **mand**) [Function]

*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.

**Description:** This function will read the certifcate 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).

**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**, *const gnutls_datum_t * **signature**, *gnutls_digest_algorithm_t * **hash**) [Function]

*key*: Holds the certificate

*signature*: contains the signature

*hash*: The result of the call with the hash algorithm used for signature

**Description:** This function will read the certifcate 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 gnutls_datum_t * data*, *gnutls_x509_crt_fmt_t* `format`) [Function]

*key*: The structure to store the parsed public key.

*data*: The DER or PEM encoded certificate.

*format*: One of DER or PEM

**Description:** This function will convert the given DER or PEM encoded Public key to the native gnutls_pubkey_t format.The output will be stored in `key`. If the Certificate 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 gnutls_datum_t * p, const gnutls_datum_t * q, const gnutls_datum_t * g, const gnutls_datum_t * y*) [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

**Description:** 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`, *gnutls_ecc_curve_t* `curve`, *const gnutls_datum_t * x, const gnutls_datum_t * y*) [Function]

*key*: The structure to store the parsed key

*curve*: holds the curve

*x*: holds the x

*y*: holds the y

**Description:** 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:** 3.0.0

### gnutls_pubkey_import_ecc_x962

`int gnutls_pubkey_import_ecc_x962` (*gnutls_pubkey_t* `key`, *const gnutls_datum_t ** `parameters`, *const gnutls_datum_t ** `ecpoint`) [Function]

*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

**Description:** 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:** 3.0.0

### gnutls_pubkey_import_openpgp

`int gnutls_pubkey_import_openpgp` (*gnutls_pubkey_t* `key`, *gnutls_openpgp_crt_t* `crt`, *unsigned int* `flags`) [Function]

*key*: The public key

*crt*: The certificate to be imported

*flags*: should be zero

**Description:** 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_pkcs11

`int gnutls_pubkey_import_pkcs11` (*gnutls_pubkey_t* `key`, *gnutls_pkcs11_obj_t* `obj`, *unsigned int* `flags`) [Function]

*key*: The public key

*obj*: The parameters to be imported

*flags*: should be zero

**Description:** 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 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

**Description:** 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`**, *gnutls_privkey_t* **`pkey`**, *unsigned int* **`usage`**, *unsigned int* **`flags`**`)` [Function]

*key*: The public key

*pkey*: The private key

*usage*: GNUTLS_KEY_* key usage flags.

*flags*: should be zero

**Description:** 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 gnutls_datum_t ** **`m`**, *const gnutls_datum_t ** **`e`**`)` [Function]

*key*: Is a structure will hold the parameters

*m*: holds the modulus

*e*: holds the public exponent

**Description:** 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_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

**Description:** 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_init

`int gnutls_pubkey_init (`*gnutls_pubkey_t ** **`key`**`)` [Function]

*key*: The structure to be initialized

**Description:** 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_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.

**Description:** 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_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*: should be 0 for now

*data*: holds the signed data

*signature*: contains the signature

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

**Returns:** On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value (GNUTLS_E_PK_SIG_VERIFY_FAILED in verification failure).

**Since:** 2.12.0

### gnutls_pubkey_verify_data2

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

*pubkey*: Holds the public key

*algo*: The signature algorithm used

*flags*: should be 0 for now

*data*: holds the signed data

*signature*: contains the signature

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

**Returns:** On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value (GNUTLS_E_PK_SIG_VERIFY_FAILED in verification failure).

**Since:** 3.0.0

### gnutls_pubkey_verify_hash

`int gnutls_pubkey_verify_hash (`*gnutls_pubkey_t* **key**, *unsigned int* **flags**, *const gnutls_datum_t ** **hash**, *const gnutls_datum_t ** **signature**`)` [Function]

*key*: Holds the certificate

*flags*: should be 0 for now

*hash*: holds the hash digest to be verified

*signature*: contains the signature

**Description:** This function will verify the given signed digest, using the parameters from the certificate.

**Returns:** On success, GNUTLS_E_SUCCESS (0) is returned, otherwise a negative error value (GNUTLS_E_PK_SIG_VERIFY_FAILED in verification failure).

**Since:** 2.12.0

### gnutls_x509_crl_privkey_sign

`int gnutls_x509_crl_privkey_sign (`*gnutls_x509_crl_t* **crl**, *gnutls_x509_crt_t* **issuer**, *gnutls_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

**Description:** 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`**, *gnutls_privkey_t* **`key`**, *gnutls_digest_algorithm_t* **`dig`**, *unsigned int* **`flags`**`)` [Function]

*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

**Description:** 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`**, *gnutls_pubkey_t* **`key`**`)` [Function]

*crq*: should contain a `gnutls_x509_crq_t` structure

*key*: holds a public key

**Description:** 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`**, *gnutls_x509_crt_t* **`issuer`**, *gnutls_privkey_t* **`issuer_key`**, *gnutls_digest_algorithm_t* **`dig`**, *unsigned int* **`flags`**`)` [Function]

*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

**Description:** 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`**, *gnutls_pubkey_t* **`key`**`)` [Function]

*crt*: should contain a `gnutls_x509_crt_t` structure

*key*: holds a public key

**Description:** 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

## C.8 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

**Description:** 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:** 3.0.0

### 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

**Description:** This function will decrypt 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_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

**Description:** This function will decrypt 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_deinit

`void gnutls_cipher_deinit (`*gnutls_cipher_hd_t* **`handle`**`)` [Function]

*handle*: is a `gnutls_cipher_hd_t` structure.

**Description:** 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

**Description:** 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 * * `text`*, *size_t* **`textlen`**, *void * * `ciphertext`*, *size_t* **`ciphertextlen`**`)` [Function]

*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

**Description:** 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* **`algorithm`**`)` [Function]

*algorithm*: is an encryption algorithm

**Description:** Get block size for encryption algorithm.

**Returns:** block size for encryption algorithm.

**Since:** 2.10.0

### gnutls_cipher_init

`int gnutls_cipher_init (`*gnutls_cipher_hd_t * `**`handle`**, *gnutls_cipher_algorithm_t* **`cipher`**, *const gnutls_datum_t * `key`*, *const gnutls_datum_t * `iv`*`)` [Function]

*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)

**Description:** 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`*, *size_t* **`ivlen`**`)` [Function]

*handle*: is a `gnutls_cipher_hd_t` structure.

*iv*: the IV to set

*ivlen*: The length of the IV

**Description:** This function will set the IV to be used for the next encryption block.

**Since:** 3.0.0

### gnutls_cipher_tag

`int gnutls_cipher_tag (`*gnutls_cipher_hd_t* **`handle`**, *void ** **`tag`**, *size_t* **`tag_size`**`)` [Function]

*handle*: is a `gnutls_cipher_hd_t` structure.

*tag*: will hold the tag

*tag_size*: The length of the tag to return

**Description:** 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:** 3.0.0

### gnutls_hash

`int gnutls_hash (`*gnutls_hash_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

**Description:** 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

**Description:** 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

**Description:** 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

**Description:** 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

**Description:** 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

**Description:** This function will output the current hash value.

**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

**Description:** 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

**Description:** 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

**Description:** 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

**Description:** 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`**, *gnutls_digest_algorithm_t* **`algorithm`**, *const void ** **`key`**, *size_t* **`keylen`**`)` [Function]

*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

**Description:** 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.

**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

**Description:** This function will output the current MAC value.

**Since:** 2.10.0

### 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

**Description:** This function will generate random data and store it to output buffer.

**Returns:** Zero or a negative error code on error.

**Since:** 2.12.0

## C.9 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_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.

**Description:** 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* **`session`**, *const int ** **`list`**`)` [Function]

*session*: is a `gnutls_session_t` structure.

*list*: is a 0 terminated list of gnutls_certificate_type_t elements.

**Description:** 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.

**Description:** 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.

**Description:** 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_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.

**Description:** 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.

**Description:** 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

**Description:** 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_protocol_set_priority

`int gnutls_protocol_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_protocol_t elements.

**Description:** 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* **`session`**`)` [Function]

*session*: is a gnutls session

**Description:** 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`**, *gnutls_datum_t ** **`exponent`**, *gnutls_datum_t ** **`modulus`**`)` [Function]

*session*: is a gnutls session

*exponent*: will hold the exponent.

*modulus*: will hold the modulus.

**Description:** 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`**, *gnutls_rsa_params_t* **`src`**`)` [Function]

*dst*: Is the destination structure, which should be initialized.

*src*: Is the source structure

**Description:** 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

**Description:** This function will deinitialize the RSA parameters structure.

### gnutls_rsa_params_export_pkcs1

`int gnutls_rsa_params_export_pkcs1 (`*gnutls_rsa_params_t* **`params`**, *gnutls_x509_crt_fmt_t* **`format`**, *unsigned char ** **`params_data`**, *size_t ** **`params_data_size`**`)` [Function]

*params*: Holds the RSA parameters

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

*params_data*: will contain a PKCS1 RSAPublicKey structure PEM or DER encoded

*params_data_size*: holds the size of params_data (and will be replaced by the actual size of parameters)

**Description:** This function will export the given RSA parameters to a PKCS1 RSAPublicKey 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`**, *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`**`)` [Function]

*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

**Description:** 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`**, *unsigned int* **`bits`**) [Function]

*params*: The structure where the parameters will be stored

*bits*: is the prime's number of bits

**Description:** 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`**, *const gnutls_datum_t ** **`pkcs1_params`**, *gnutls_x509_crt_fmt_t* **`format`**) [Function]

*params*: A structure where the parameters will be copied to

*pkcs1_params*: should contain a PKCS1 RSAPublicKey structure PEM or DER encoded

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

**Description:** This function will extract the RSAPublicKey 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* `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*) [Function]

*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

**Description:** 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

**Description:** 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.

**Description:** 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.

**Description:** 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.

**Description:** 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` interfacess like `gnutls_privkey_import_ext()` instead.

## gnutls_x509_crl_sign

`int gnutls_x509_crl_sign (`*gnutls_x509_crl_t* ***crl***, *gnutls_x509_crt_t* ***issuer***, *gnutls_x509_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

**Description:** This function is the same a `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_x509_privkey_t* ***key***`)` [Function]

*crq*: should contain a `gnutls_x509_crq_t` structure

*key*: holds a private key

**Description:** This function is the same a `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_crt_get_preferred_hash_algorithm

`int gnutls_x509_crt_get_preferred_hash_algorithm` [Function]
(*gnutls_x509_crt_t* **`crt`**, *gnutls_digest_algorithm_t ** **`hash`**, *unsigned int ** **`mand`**)

*crt*: Holds the certificate

*hash*: The result of the call with the hash algorithm used for signature

*mand*: If non (0) it means that the algorithm MUST use this hash. May be NULL.

**Description:** This function will read the certifcate 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_crt_get_verify_algorithm

`int gnutls_x509_crt_get_verify_algorithm` (*gnutls_x509_crt_t* **`crt`**, *const gnutls_datum_t ** **`signature`**, *gnutls_digest_algorithm_t ** **`hash`**) [Function]

*crt*: Holds the certificate

*signature*: contains the signature

*hash*: The result of the call with the hash algorithm used for signature

**Description:** This function will read the certifcate 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_crt_verify_data

`int gnutls_x509_crt_verify_data` (*gnutls_x509_crt_t* **`crt`**, *unsigned int* **`flags`**, *const gnutls_datum_t ** **`data`**, *const gnutls_datum_t ** **`signature`**) [Function]

*crt*: Holds the certificate

*flags*: should be 0 for now

*data*: holds the data to be signed

*signature*: contains the signature

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

Deprecated. Please use `gnutls_pubkey_verify_data()`.

**Returns:** In case of a verification failure `GNUTLS_E_PK_SIG_VERIFY_FAILED` is returned, and a positive code on success.

### gnutls_x509_crt_verify_hash

`int gnutls_x509_crt_verify_hash (`*gnutls_x509_crt_t* **`crt`**, *unsigned int* **`flags`**, *const gnutls_datum_t ** **`hash`**, *const gnutls_datum_t ** **`signature`**`)` [Function]

*crt*: Holds the certificate

*flags*: should be 0 for now

*hash*: holds the hash digest to be verified

*signature*: contains the signature

**Description:** This function will verify the given signed digest, using the parameters from the certificate.

Deprecated. Please use `gnutls_pubkey_verify_data()`.

**Returns:** In case of a verification failure `GNUTLS_E_PK_SIG_VERIFY_FAILED` is returned, and a positive code on success.

### gnutls_x509_privkey_sign_data

`int gnutls_x509_privkey_sign_data (`*gnutls_x509_privkey_t* **`key`**, *gnutls_digest_algorithm_t* **`digest`**, *unsigned int* **`flags`**, *const gnutls_datum_t ** **`data`**, *void ** **`signature`**, *size_t ** **`signature_size`**`)` [Function]

*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)

**Description:** 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`**, *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

**Description:** 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 D Supported Ciphersuites in GnuTLS

Available cipher suites:

| | | |
|---|---|---|
| TLS_DH_ANON_ARCFOUR_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_SHA1 | 0x00 0x46 | TLS1.0 |
| TLS_DH_ANON_CAMELLIA_256_CBC_SHA1 | 0x00 0x89 | TLS1.0 |
| TLS_DH_ANON_AES_128_CBC_SHA256 | 0x00 0x6C | TLS1.2 |
| TLS_DH_ANON_AES_256_CBC_SHA256 | 0x00 0x6D | TLS1.2 |
| TLS_PSK_SHA_ARCFOUR_SHA1 | 0x00 0x8A | TLS1.0 |
| TLS_PSK_SHA_3DES_EDE_CBC_SHA1 | 0x00 0x8B | TLS1.0 |
| TLS_PSK_SHA_AES_128_CBC_SHA1 | 0x00 0x8C | TLS1.0 |
| TLS_PSK_SHA_AES_256_CBC_SHA1 | 0x00 0x8D | TLS1.0 |
| TLS_PSK_AES_128_CBC_SHA256 | 0x00 0xAE | TLS1.0 |
| TLS_PSK_AES_128_GCM_SHA256 | 0x00 0xA8 | TLS1.2 |
| TLS_PSK_NULL_SHA256 | 0x00 0xB0 | TLS1.0 |
| TLS_DHE_PSK_SHA_ARCFOUR_SHA1 | 0x00 0x8E | TLS1.0 |
| TLS_DHE_PSK_SHA_3DES_EDE_CBC_SHA1 | 0x00 0x8F | TLS1.0 |
| TLS_DHE_PSK_SHA_AES_128_CBC_SHA1 | 0x00 0x90 | TLS1.0 |
| TLS_DHE_PSK_SHA_AES_256_CBC_SHA1 | 0x00 0x91 | TLS1.0 |
| TLS_DHE_PSK_AES_128_CBC_SHA256 | 0x00 0xB2 | TLS1.0 |
| TLS_DHE_PSK_AES_128_GCM_SHA256 | 0x00 0xAA | TLS1.2 |
| TLS_DHE_PSK_NULL_SHA256 | 0x00 0xB4 | TLS1.0 |
| TLS_SRP_SHA_3DES_EDE_CBC_SHA1 | 0xC0 0x1A | TLS1.0 |
| TLS_SRP_SHA_AES_128_CBC_SHA1 | 0xC0 0x1D | TLS1.0 |
| TLS_SRP_SHA_AES_256_CBC_SHA1 | 0xC0 0x20 | TLS1.0 |
| TLS_SRP_SHA_DSS_3DES_EDE_CBC_SHA1 | 0xC0 0x1C | TLS1.0 |
| TLS_SRP_SHA_RSA_3DES_EDE_CBC_SHA1 | 0xC0 0x1B | TLS1.0 |
| TLS_SRP_SHA_DSS_AES_128_CBC_SHA1 | 0xC0 0x1F | TLS1.0 |
| TLS_SRP_SHA_RSA_AES_128_CBC_SHA1 | 0xC0 0x1E | TLS1.0 |
| TLS_SRP_SHA_DSS_AES_256_CBC_SHA1 | 0xC0 0x22 | TLS1.0 |
| TLS_SRP_SHA_RSA_AES_256_CBC_SHA1 | 0xC0 0x21 | TLS1.0 |
| TLS_DHE_DSS_ARCFOUR_SHA1 | 0x00 0x66 | TLS1.0 |
| 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_SHA1 | 0x00 0x44 | TLS1.0 |
| TLS_DHE_DSS_CAMELLIA_256_CBC_SHA1 | 0x00 0x87 | TLS1.0 |
| TLS_DHE_DSS_AES_128_CBC_SHA256 | 0x00 0x40 | TLS1.2 |
| TLS_DHE_DSS_AES_256_CBC_SHA256 | 0x00 0x6A | TLS1.2 |
| TLS_DHE_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_SHA1 | 0x00 0x45 | TLS1.0 |
| TLS_DHE_RSA_CAMELLIA_256_CBC_SHA1 | 0x00 0x88 | TLS1.0 |

| | | |
|---|---|---|
| TLS_DHE_RSA_AES_128_CBC_SHA256 | 0x00 0x67 | TLS1.2 |
| TLS_DHE_RSA_AES_256_CBC_SHA256 | 0x00 0x6B | TLS1.2 |
| 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_EXPORT_ARCFOUR_40_MD5 | 0x00 0x03 | SSL3.0 |
| TLS_RSA_ARCFOUR_SHA1 | 0x00 0x05 | SSL3.0 |
| TLS_RSA_ARCFOUR_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_SHA1 | 0x00 0x41 | TLS1.0 |
| TLS_RSA_CAMELLIA_256_CBC_SHA1 | 0x00 0x84 | TLS1.0 |
| 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_DHE_RSA_AES_128_GCM_SHA256 | 0x00 0x9E | TLS1.2 |
| TLS_DHE_DSS_AES_128_GCM_SHA256 | 0x00 0xA2 | TLS1.2 |
| TLS_DH_ANON_AES_128_GCM_SHA256 | 0x00 0xA6 | TLS1.2 |
| TLS_ECDH_ANON_NULL_SHA | 0xC0 0x15 | TLS1.0 |
| TLS_ECDH_ANON_3DES_EDE_CBC_SHA | 0xC0 0x17 | TLS1.0 |
| TLS_ECDH_ANON_AES_128_CBC_SHA | 0xC0 0x18 | TLS1.0 |
| TLS_ECDH_ANON_AES_256_CBC_SHA | 0xC0 0x19 | TLS1.0 |
| TLS_ECDHE_RSA_NULL_SHA | 0xC0 0x10 | TLS1.0 |
| TLS_ECDHE_RSA_3DES_EDE_CBC_SHA | 0xC0 0x12 | TLS1.0 |
| TLS_ECDHE_RSA_AES_128_CBC_SHA | 0xC0 0x13 | TLS1.0 |
| TLS_ECDHE_RSA_AES_256_CBC_SHA | 0xC0 0x14 | TLS1.0 |
| TLS_ECDHE_ECDSA_NULL_SHA | 0xC0 0x06 | TLS1.0 |
| TLS_ECDHE_ECDSA_3DES_EDE_CBC_SHA | 0xC0 0x08 | TLS1.0 |
| TLS_ECDHE_ECDSA_AES_128_CBC_SHA | 0xC0 0x09 | TLS1.0 |
| TLS_ECDHE_ECDSA_AES_256_CBC_SHA | 0xC0 0x0A | TLS1.0 |
| 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_AES_128_GCM_SHA256 | 0xC0 0x2B | TLS1.2 |
| TLS_ECDHE_RSA_AES_128_GCM_SHA256 | 0xC0 0x2F | TLS1.2 |
| TLS_ECDHE_PSK_3DES_EDE_CBC_SHA | 0xC0 0x34 | TLS1.0 |
| TLS_ECDHE_PSK_AES_128_CBC_SHA | 0xC0 0x35 | TLS1.0 |
| TLS_ECDHE_PSK_AES_256_CBC_SHA | 0xC0 0x36 | TLS1.0 |
| TLS_ECDHE_PSK_AES_128_CBC_SHA256 | 0xC0 0x37 | TLS1.0 |
| TLS_ECDHE_PSK_AES_256_CBC_SHA384 | 0xC0 0x38 | TLS1.0 |
| TLS_ECDHE_PSK_NULL_SHA256 | 0xC0 0x3A | TLS1.0 |
| TLS_ECDHE_PSK_NULL_SHA384 | 0xC0 0x3B | TLS1.0 |
| TLS_ECDHE_ECDSA_AES_256_GCM_SHA384 | 0xC0 0x2E | TLS1.2 |
| TLS_ECDHE_RSA_AES_256_GCM_SHA384 | 0xC0 0x30 | TLS1.2 |
| TLS_ECDHE_ECDSA_AES_256_CBC_SHA384 | 0xC0 0x24 | TLS1.2 |

Available certificate types:

- X.509
- OPENPGP

Available protocols:

- SSL3.0
- TLS1.0
- TLS1.1
- TLS1.2
- DTLS1.0

Available ciphers:

- AES-256-CBC
- AES-192-CBC
- AES-128-CBC
- AES-128-GCM
- AES-256-GCM
- 3DES-CBC
- DES-CBC
- ARCFOUR-128
- ARCFOUR-40
- RC2-40
- CAMELLIA-256-CBC
- CAMELLIA-128-CBC
- IDEA-PGP-CFB
- 3DES-PGP-CFB
- CAST5-PGP-CFB
- BLOWFISH-PGP-CFB
- SAFER-SK128-PGP-CFB
- AES-128-PGP-CFB
- AES-192-PGP-CFB
- AES-256-PGP-CFB
- TWOFISH-PGP-CFB
- NULL

Available MAC algorithms:

- SHA1
- MD5
- SHA256
- SHA384
- SHA512
- SHA224

- AEAD
- MD2
- RIPEMD160
- MAC-NULL

Available key exchange methods:

- ANON-DH
- ANON-ECDH
- RSA
- RSA-EXPORT
- DHE-RSA
- ECDHE-RSA
- ECDHE-ECDSA
- DHE-DSS
- SRP-DSS
- SRP-RSA
- SRP
- PSK
- DHE-PSK
- ECDHE-PSK

Available public key algorithms:

- RSA
- DSA
- ECC

Available public key signature algorithms:

- RSA-SHA1
- RSA-SHA224
- RSA-SHA256
- RSA-SHA384
- RSA-SHA512
- RSA-RMD160
- DSA-SHA1
- DSA-SHA224
- DSA-SHA256
- RSA-MD5
- RSA-MD2
- ECDSA-SHA1
- ECDSA-SHA224
- ECDSA-SHA256

- ECDSA-SHA384
- ECDSA-SHA512

Available compression methods:

- DEFLATE
- NULL

# Appendix E Copying Information

## E.1 GNU Free Documentation License

Version 1.3, 3 November 2008



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. Secondarily, 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/.

[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.

[RFC2246]
Tim Dierks and Christopher Allen, "The TLS Protocol Version 1.0", January 1999, Available from http://www.ietf.org/rfc/rfc2246.txt.

[RFC4346]
Tim Dierks and Eric Rescorla, "The TLS Protocol Version 1.1", Match 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/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.

[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.

[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", August 2011, Work in progress, available from http://tools.ietf.org/html/draft-pechanec-pkcs11uri-05.

[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.

# Function and Data Index

# Concept Index

## T



## V



## X