GNU TLS

Transport Layer Security Library for the GNU system
for version 1.4.4, 21 March 2006

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

This manual is last updated 21 March 2006 for version 1.4.4 of GNU TLS.

Copyright (C) 2001, 2002, 2003, 2004, 2005 Free Software Foundation, Inc.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Table of Contents

1 Preface 1

2 The Library 2

2.1 General Idea 2
2.2 Error handling 4
2.3 Memory handling 4
2.4 Callback functions 4

3 Introduction to TLS 5

3.1 TLS layers 5
3.2 The transport layer 6
3.3 The TLS record protocol 6
3.3.1 Encryption algorithms used in the record layer 7
3.3.2 Compression algorithms used in the record layer 7
3.3.3 Weaknesses and countermeasures 8
3.4 The TLS Alert Protocol 8
3.5 The TLS Handshake Protocol 8
3.5.1 TLS cipher suites 9
3.5.2 Client authentication 9
3.5.3 Resuming Sessions 10
3.5.4 Resuming internals 10
3.6 TLS Extensions 10
3.6.1 Maximum fragment length negotiation 11
3.6.2 Server name indication 11
3.7 On SSL 2 and older protocols 11

4 Authentication methods 12

4.1 Certificate authentication 12
4.1.1 Authentication using X.509 certificates 12
4.1.2 Authentication using OpenPGP keys 12
4.1.3 Using certificate authentication 12
4.2 Anonymous authentication 14
4.3 Authentication using SRP 14
4.4 Authentication using PSK 15
4.5 Authentication and credentials 16
4.6 Parameters stored in credentials 16

5 More on certificate authentication 18

5.1 The X.509 trust model 18
5.1.1 X.509 certificates 18
5.1.2 Verifying X.509 certificate paths 19
5.1.3 PKCS #10 certificate requests 21
5.1.4 PKCS #12 structures 21
5.2 The OpenPGP trust model 21
5.2.1 OpenPGP keys 22
5.2.2 Verifying an OpenPGP key 22
5.3 Digital signatures 23
5.3.1 Supported algorithms 24
5.3.2 Trading security for interoperability 24

6 How to use TLS in application protocols 26

6.1 Separate ports 26
6.2 Upward negotiation 26

7 How to use GnuTLS in applications 28

7.1 Preparation 28
7.1.1 Headers 28
7.1.2 Version check 28
7.1.3 Building the source 28
7.2 Multi-threaded applications 29
7.3 Client examples 30
7.3.1 Simple client example with anonymous authentication 30
7.3.2 Simple client example with X.509 certificate support 32
7.3.3 Obtaining session information 34
7.3.4 Verifying peer's certificate 37
7.3.5 Using a callback to select the certificate to use 43
7.3.6 Client with Resume capability example 49
7.3.7 Simple client example with SRP authentication 52
7.4 Server examples 55
7.4.1 Echo Server with X.509 authentication 55
7.4.2 Echo Server with X.509 authentication II 59
7.4.3 Echo Server with OpenPGP authentication 67
7.4.4 Echo Server with SRP authentication 71
7.4.5 Echo Server with anonymous authentication 75
7.5 Miscellaneous examples 79
7.5.1 Checking for an alert 79
7.5.2 X.509 certificate parsing example 79
7.5.3 Certificate request generation 82
7.5.4 PKCS #12 structure generation 83
7.6 Compatibility with the OpenSSL library 86

8 Included programs 87
8.1 Invoking srptool 87
8.2 Invoking gnutls-cli 87
8.3 Invoking gnutls-cli-debug 88
8.4 Invoking gnutls-serv 89
8.5 Invoking certtool 90

9 Function reference 94
9.1 Core functions 94
9.2 X.509 certificate functions 141
9.3 GnuTLS-extra functions 183
9.4 OpenPGP functions 183
9.5 TLS Inner Application (TLS/IA) functions 192
9.6 Error codes and descriptions 199

10 Certificate to XML convertion functions .. 205
10.1 An X.509 certificate 205
10.2 An OpenPGP key 208

11 All the supported ciphersuites in GnuTLS 210

12 Internal architecture of GnuTLS 212
12.1 The TLS protocol 212
12.2 TLS Handshake protocol 213
12.3 TLS authentication methods 214
12.4 TLS Extension handling 215
12.5 Certificate handling 215

Appendix A Copying This Manual 217
A.1 GNU Free Documentation License 217
A.1.1 ADDENDUM: How to use this License for your documents 223

Concept Index 224

Function and Data Index 225

Bibliography 229

1 Preface

This document tries to demonstrate and explain 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 he risks to offer just a false sense of security. Security and the network security terms are very general terms even for computer software thus cannot be easily restricted to a single cryptographic library. For that reason, do not consider a 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 PKI knowlegde is assumed in most of it. A good introduction to networking can be found in [STEVENS] (See [Bibliography], page 229.) and for Public Key Infrastructure in [GUTPKI] (See [Bibliography], page 229.) .

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 The Library

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 TLS 1.1 and SSL 3.0 protocols (See Chapter 3 [Introduction to TLS], page 5, for a more detailed description of the protocols), accompanied with the required framework for authentication and public key infrastructure. The library is available under the GNU Lesser GPL license[1]. Important features of the GnuTLS library include:

- Support for TLS 1.0, TLS 1.1, and SSL 3.0 protocols.
- Support for both X.509 and OpenPGP certificates.
- Support for handling and verification of certificates.
- Support for SRP for TLS authentication.
- Support for PSK for TLS authentication.
- Support for TLS Extension mechanism.
- Support for TLS Compression Methods.

Additionally GnuTLS provides a limited emulation API for the widely used OpenSSL[2] library, to ease integration with existing applications.

GnuTLS consists of three independent parts, namely the "TLS protocol part", the "Certificate part", and the "Crypto backend" 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 which is partially implemented in the GnuTLS library. The Libtasn1[3], a library which offers ASN.1 parsing capabilities, is used for the X.509 certificate parsing functions, and Opencdk[4] is used for the OpenPGP key support in GnuTLS. The "Crypto backend" is provided by the Libgcrypt[5] library.

In order to ease integration in embedded systems, parts of the GnuTLS library can be disabled at compile time. That way a small library, with the required features, can be generated.

---

[1] A copy of the license is included in the distribution

[2] `http://www.openssl.org/`

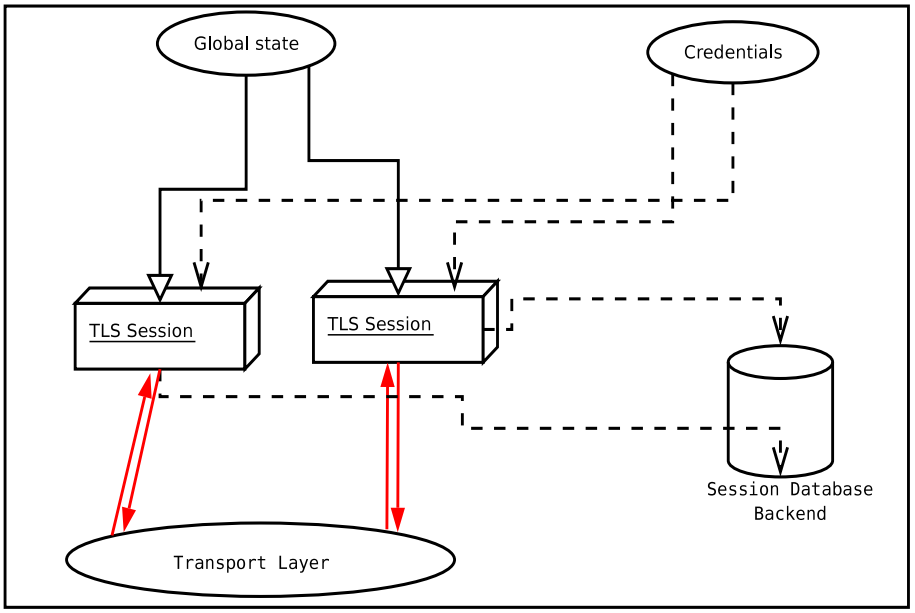
[3] `ftp://ftp.gnupg.org/gcrypt/alpha/gnutls/libtasn1/`

[4] `ftp://ftp.gnupg.org/gcrypt/alpha/gnutls/opencdk/`

[5] `ftp://ftp.gnupg.org/gcrypt/alpha/libgcrypt/`

2.1 General Idea

A brief description of how GnuTLS works internally is shown at the figure below. This section may be easier to understand after having seen the examples (see [examples], page 28).



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 some structures needed for the ASN.1 parser. This structure is never modified by any GnuTLS function, except for the deinitialization function which frees all memory allocated in the global structure and is called after the program has permanently finished using GnuTLS.

The credentials structure is used by some authentication methods, such as certificate authentication (see [Certificate Authentication], page 18). A credentials structure may contain certificates, private keys, temporary parameters for diffie hellman or RSA key exchange, and other stuff that may be shared between several TLS sessions.

This structure should be initialized using the appropriate initialization functions. For example an application which uses certificate authentication would probably initialize the credentials, using the appropriate functions, and put its trusted certificates in this structure. The next step is to associate the credentials structure with each TLS session.

A GnuTLS session contains all the required stuff for a session to handle one secure connection. This session calls directly to the transport layer functions, in order to communicate with the peer. Every session has a unique session ID shared with the peer.

Since TLS sessions can be resumed, servers would probably need a database backend to hold the session's parameters. Every GnuTLS session after a successful handshake calls the appropriate backend function (See [resume], page 10, for information on initialization) to store the newly negotiated session. The session database is examined by the server just after having received the client hello[6], and if the session ID sent by the client, matches a

[6] The first message in a TLS handshake

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.

2.2 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. An example of a fatal error code 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 127. 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 115.

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 [Error Codes], page 199, for all the error codes.

2.3 Memory handling

GnuTLS internally handles heap allocated objects differently, depending on the sensitivity of the data they contain. However for performance reasons, the default memory functions do not overwrite sensitive data from memory, nor protect such objects from being written to the swap. In order to change the default behavior the [gnutls_global_set_mem_functions], page 117 function is available which can be used to set other memory handlers than the defaults.

The Libgcrypt library on which GnuTLS depends, has such secure memory allocation functions available. These should be used in cases where even the system's swap memory is not considered secure. See the documentation of Libgcrypt for more information.

2.4 Callback functions

There are several cases where GnuTLS may need some 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 141
- [gnutls_transport_set_pull_function], page 140

Other callback functions such as the one set by [gnutls_srp_set_server_credentials_function], page 138, may require more complicated input, including data to be allocated. These callbacks should allocate and free memory using the functions shown below.

- [gnutls_malloc], page 120
- [gnutls_free], page 115

3 Introduction to TLS

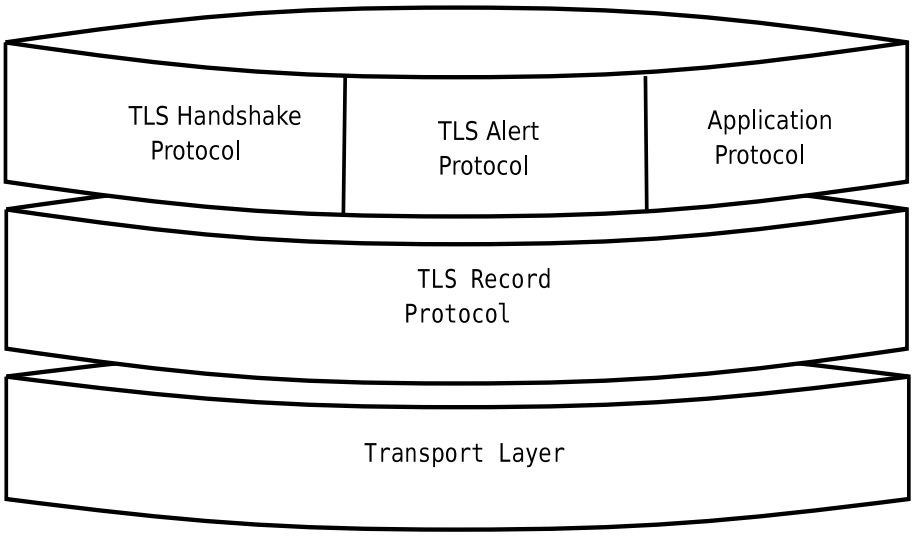
TLS stands for "Transport Layer Security" and is the successor of SSL, the Secure Sockets Layer protocol [SSL3] (See [Bibliography], page 229.) designed by Netscape. TLS is an Internet protocol, defined by IETF[1], described in RFC 2246 and also in [RESCOLA] (See [Bibliography], page 229.) . The protocol provides confidentiality, and authentication layers over any reliable transport layer. The description, below, refers to TLS 1.0 but also applies to TLS 1.1 [RFC4346] (See [Bibliography], page 229.) and SSL 3.0, since the differences of these protocols are minor. Older protocols such as SSL 2.0 are not discussed nor implemented in GnuTLS since they are not considered secure today.

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 8, for more information about the handshake protocol. The protocol layering in TLS is shown in the figure below.



[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.

3.2 The transport layer

TLS is not limited to one transport layer, it can be used above any transport layer, as long as it is a reliable one. A set of functions is provided and their purpose is to load to GnuTLS the required callbacks to access the transport layer.

- [gnutls_transport_set_push_function], page 141
- [gnutls_transport_set_pull_function], page 140
- [gnutls_transport_set_ptr], page 140
- [gnutls_transport_set_lowat], page 140

These functions accept a callback function as a parameter. The callback functions should return the number of bytes written, or -1 on error and should set `errno` appropriately.

GnuTLS currently only interprets the EINTR and EAGAIN errno values and returns the corresponding GnuTLS error codes `GNUTLS_E_INTERRUPTED` and `GNUTLS_E_AGAIN`. These values are usually returned by interrupted system calls, or when non blocking IO is used. All GnuTLS functions can be resumed (called again), if any of these error codes is returned. The error codes above refer to the system call, not the GnuTLS function, since signals do not interrupt GnuTLS' functions.

For non blocking sockets or other custom made pull/push functions the [gnutls_transport_set_lowat], page 140 must be called, with a zero low water mark value.

By default, if the transport functions are not set, GnuTLS will use the Berkeley Sockets functions. In this case GnuTLS will use some hacks in order for `select` to work, thus making it easy to add TLS support to existing TCP/IP servers.

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 following functions are available:

[gnutls_record_send], page 128:
: To send a record packet (with application data).

[gnutls_record_recv], page 127:
: To receive a record packet (with application data).

[gnutls_record_get_direction], page 127:
: To get the direction of the last interrupted function call.

As you may have already noticed, the functions which access the Record protocol, are quite limited, given the importance of this protocol in TLS. This is because the Record protocol's parameters are all set by the Handshake protocol.

The Record protocol initially starts with NULL parameters, which means no encryption, and no MAC is used. Encryption and authentication begin just after the handshake protocol has finished.

3.3.1 Encryption algorithms used in the record layer

Confidentiality in the record layer is achieved by using symmetric block encryption algorithms like `3DES`, `AES`[2], or stream algorithms like `ARCFOUR_128`[3]. Ciphers are encryption algorithms that use a single, secret, key to encrypt and decrypt data. Block algorithms in TLS 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.

Supported cipher algorithms:

`3DES_CBC` `3DES_CBC` 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 is a fast stream cipher.

`ARCFOUR_40`
This is the ARCFOUR cipher that is 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. This is not officially supported in TLS.

Supported MAC algorithms:

`MAC_MD5` MD5 is a cryptographic hash algorithm designed by Ron Rivest. Outputs 128 bits of data.

`MAC_SHA` SHA is a cryptographic hash algorithm designed by NSA. Outputs 160 bits of data.

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. All the algorithms except for DEFLATE which is referenced in [RFC3749] (See [Bibliography], page 229.) , should be considered as GnuTLS' extensions[4], and should be advertised only when the peer is known to have a compliant client, to avoid interoperability problems.

The included algorithms perform really good when text, or other compressible data are to be transfered, 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. As a drawback, compression increases latency.

The record layer compression in GnuTLS is implemented based on the proposal [RFC3749] (See [Bibliography], page 229.) . The supported compression algorithms are:

`DEFLATE` Zlib compression, using the deflate algorithm.

[2] AES, or Advanced Encryption Standard, is actually the RIJNDAEL algorithm. This is the algorithm that replaced DES.

[3] `ARCFOUR_128` is a compatible algorithm with RSA's RC4 algorithm, which is considered to be a trade secret.

[4] You should use [gnutls_handshake_set_private_extensions], page 118 to enable private extensions.

`LZO` LZO is a very fast compression algorithm. This algorithm is only available if the GnuTLS-extra library has been initialized and the private extensions are enabled.

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] (See [Bibliography], page 229.) which is implemented in GnuTLS. For a detailed discussion see the archives of the TLS Working Group mailing list and the paper [CBCATT] (See [Bibliography], page 229.) .

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 (see `GNUTLS_A_CLOSE_NOTIFY`), and others refer to the application protocol solely (see `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 renegotiations using the current session ID.

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.

[gnutls_alert_send], page 94:
To send an alert signal.

[gnutls_error_to_alert], page 115:
To map a gnutls error number to an alert signal.

[gnutls_alert_get], page 94:
Returns the last received alert.

[gnutls_alert_get_name], page 94:
Returns the name, in a character array, of the given alert.

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. Available functions to control the handshake protocol include:

[gnutls_cipher_set_priority], page 108:
To set the priority of bulk cipher algorithms.

[gnutls_mac_set_priority], page 120:
To set the priority of MAC algorithms.

[gnutls_kx_set_priority], page 119:
To set the priority of key exchange algorithms.

[gnutls_compression_set_priority], page 108:
To set the priority of compression methods.

[gnutls_certificate_type_set_priority], page 106:
To set the priority of certificate types (e.g., OpenPGP, X.509).

[gnutls_protocol_set_priority], page 124:
To set the priority of protocol versions (e.g., SSL 3.0, TLS 1.0).

[gnutls_set_default_priority], page 135:
To set some defaults in the current session. That way you don't have to call each priority function, independently, but you have to live with the defaults.

[gnutls_credentials_set], page 109:
To set the appropriate credentials structures.

[gnutls_certificate_server_set_request], page 100:
To set whether client certificate is required or not.

[gnutls_handshake], page 118:
To initiate the handshake.

3.5.1 TLS cipher suites

The Handshake Protocol of TLS negotiates cipher suites of the form `TLS_DHE_RSA_WITH_3DES_CBC_SHA`. The usual cipher suites contain these parameters:

- The key exchange algorithm. `DHE_RSA` in the example.
- The Symmetric encryption algorithm and mode `3DES_CBC` in this example.
- The MAC[5] 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.

The priority functions, dicussed above, allow the application layer to enable and set priorities on the individual ciphers. It may imply that all combinations of ciphersuites are allowed, but this is not true. For several reasons, not discussed here, some combinations were not defined in the TLS protocol. The supported ciphersuites are shown in [ciphersuites], page 210.

3.5.2 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 100 function. If a certificate is to be requested from the client during the handshake, the server will send a certificate request message

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

that contains a list of acceptable certificate signers. In GnuTLS the certificate signers list is constructed using the trusted Certificate Authorities by the server. That is the ones set using

- [gnutls_certificate_set_x509_trust_file], page 105
- [gnutls_certificate_set_x509_trust_mem], page 105

Sending of the names of the CAs can be controlled using [gnutls_certificate_send_x509_rdn_sequence], page 100. The client, then, may send a certificate, signed by one of the server's acceptable signers.

3.5.3 Resuming Sessions

The [gnutls_handshake], page 118 function, is expensive since a lot of calculations are performed. In order to support many fast connections to the same server a client may use session resuming. **Session resuming** 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 using the previously established keys. GnuTLS supports this feature, and the example (see [ex:resume-client], page 49) illustrates a typical use of it.

Keep in mind that sessions are expired after some time, for security reasons, thus it may be normal for a server not to resume a session even if you requested that. Also note that you must enable, using the priority functions, at least the algorithms used in the last session.

3.5.4 Resuming internals

The resuming capability, mostly in the server side, is one of the problems of a thread-safe TLS implementations. The problem is that all threads must share information in order to be able to resume sessions. The gnutls approach is, in case of a client, to leave all the burden of resuming to the client. I.e., copy and keep the necessary parameters. See the functions:

- [gnutls_session_get_data], page 133
- [gnutls_session_get_id], page 133
- [gnutls_session_set_data], page 134

The server side is different. A server has to specify some callback functions which store, retrieve and delete session data. These can be registered with:

- [gnutls_db_set_remove_function], page 110
- [gnutls_db_set_store_function], page 111
- [gnutls_db_set_retrieve_function], page 111
- [gnutls_db_set_ptr], page 110

It might also be useful to be able to check for expired sessions in order to remove them, and save space. The function [gnutls_db_check_entry], page 109 is provided for that reason.

3.6 TLS Extensions

A number of extensions to the TLS protocol have been proposed mainly in [TLSEXT] (See [Bibliography], page 229.) . The extensions supported in GnuTLS are:

- Maximum fragment length negotiation

- Server name indication

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. See the [gnutls_record_set_max_size], page 128 and the [gnutls_record_get_max_size], page 127 functions.

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 132 and [gnutls_server_name_get], page 131 can be used to enable this extension, or to retrieve the name sent by a client.

3.7 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.

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

4 Authentication methods

The TLS protocol provides confidentiality and encryption, but also offers authentication, which is a prerequisite for a secure connection. The available authentication methods in GnuTLS are:

- Certificate authentication
- Anonymous authentication
- SRP authentication
- PSK authentication

4.1 Certificate authentication

4.1.1 Authentication using X.509 certificates

X.509 certificates contain the public parameters, of a public key algorithm, and an authority's signature, which proves the authenticity of the parameters. See Section 5.1 [The X.509 trust model], page 18, for more information on X.509 protocols.

4.1.2 Authentication using OpenPGP keys

OpenPGP keys also contain public parameters of a public key algorithm, and signatures from several other parties. Depending on whether a signer is trusted the key is considered trusted or not. GnuTLS's OpenPGP authentication implementation is based on the [TLSPGP] (See [Bibliography], page 229.) proposal.

See Section 5.2 [The OpenPGP trust model], page 21, for more information about the OpenPGP trust model. For a more detailed introduction to OpenPGP and GnuPG see [GPGH] (See [Bibliography], page 229.) .

4.1.3 Using certificate authentication

In GnuTLS both the OpenPGP and X.509 certificates are part of the certificate authentication and thus are handled using a common API.

When using certificates the server is required to have at least one certificate and private key pair. A client may or may not have such a pair. The certificate and key pair should be loaded, before any TLS session is initialized, in a certificate credentials structure. This should be done by using [gnutls_certificate_set_x509_key_file], page 103 or [gnutls_certificate_set_openpgp_key_file], page 183 depending on the certificate type. In the X.509 case, the functions will also accept and use a certificate list that leads to a trusted authority. The certificate list 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.

As an alternative, a callback may be used so the server or the client specify the certificate and the key at the handshake time. That callback can be set using the functions:

- [gnutls_certificate_server_set_retrieve_function], page 101
- [gnutls_certificate_client_set_retrieve_function], page 98

Certificate verification is possible by loading the trusted authorities into the credentials structure by using [gnutls_certificate_set_x509_trust_file], page 105 or

[gnutls_certificate_set_openpgp_keyring_file], page 184 for openpgp keys. Note however that the peer's certificate is not automatically verified, you should call [gnutls_certificate_verify_peers2], page 106, after a successful handshake, to verify the signatures of the certificate. An alternative way, which reports a more detailed verification output, is to use [gnutls_certificate_get_peers], page 100 to obtain the raw certificate of the peer and verify it using the functions discussed in Section 5.1 [The X.509 trust model], page 18.

In a handshake, the negotiated cipher suite depends on the certificate's parameters, so not all key exchange methods will 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, but only the `DHE_RSA` ones. It is recommended not to use RSA keys for both signing and encryption. If possible use the same key for the `DHE_RSA` and `RSA_EXPORT` ciphersuites, which use signing, and a different key for the plain RSA ciphersuites, which use encryption. All the key exchange methods shown below are available in certificate authentication.

Note that the DHE key exchange methods are generally slower[1] than plain RSA and require Diffie Hellman parameters to be generated and associated with a credentials structure, by the server. The `RSA-EXPORT` method also requires 512 bit RSA parameters, that should also be generated and associated with the credentials structure. See the functions:

- [gnutls_dh_params_generate2], page 113
- [gnutls_certificate_set_dh_params], page 101
- [gnutls_rsa_params_generate2], page 130
- [gnutls_certificate_set_rsa_export_params], page 101

Sometimes in order to avoid bottlenecks in programs it is usefull to store and read parameters from formats that can be generated by external programs such as `certtool`. This is possible with GnuTLS by using the following functions:

- [gnutls_dh_params_import_pkcs3], page 114
- [gnutls_rsa_params_import_pkcs1], page 131
- [gnutls_dh_params_export_pkcs3], page 113
- [gnutls_rsa_params_export_pkcs1], page 130

Key exchange algorithms for OpenPGP and X.509 certificates:

`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

[1] It really depends on the group used. Primes with lesser bits are always faster, but also easier to break. Values less than 768 should not be used today

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.

`DHE_DSS:` The DSS 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. DSS stands for Digital Signature Standard.

4.2 Anonymous authentication

The anonymous key exchange performs encryption but there is no indication of the identity of the peer. This kind of authentication is vulnerable to a man in the middle attack, but this protocol can be used even if there is no prior communication and trusted parties with the peer, or when full anonymity is required. Unless really required, do not use anonymous authentication. Available key exchange methods are shown below.

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.

Supported anonymous key exchange algorithms:

`ANON_DH:` This algorithm exchanges Diffie Hellman parameters.

4.3 Authentication using SRP

Authentication via the Secure Remote Password protocol, SRP[2], is supported. The SRP key exchange is an extension to the TLS protocol, and it is a password based authentication (unlike X.509 or OpenPGP that use certificates). The two 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 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. Check [TOMSRP] (See [Bibliography], page 229.) for a detailed description of the SRP protocol and the Stanford SRP libraries, which includes a PAM module that synchronizes the system's users passwords with the SRP password files. That way SRP authentication could be used for all the system's users.

The implementation in GnuTLS is based on paper [TLSSRP] (See [Bibliography], page 229.) . The supported SRP key exchange methods are:

`SRP:` Authentication using the SRP protocol.

`SRP_DSS:` Client authentication using the SRP protocol. Server is authenticated using a certificate with DSA parameters.

[2] SRP is described in [RFC2945] (See [Bibliography], page 229.)

`SRP_RSA:` Client authentication using the SRP protocol. Server is authenticated using a certificate with RSA parameters.

If clients supporting SRP know the username and password before the connection, should initialize the client credentials and call the function [gnutls_srp_set_client_credentials], page 138. Alternatively they could specify a callback function by using the function [gnutls_srp_set_client_credentials_function], page 137. This has the advantage that allows probing the server for SRP support. In that case the callback function will be called twice per handshake. The first time is before the ciphersuite is negotiated, and if the callback returns a negative error code, the callback will be called again if SRP has been negotiated. This uses a special TLS-SRP handshake idiom in order to avoid, in interactive applications, to ask the user for SRP password and username if the server does not negotiate an SRP ciphersuite.

In server side the default behaviour of GnuTLS is to read the usernames and SRP verifiers from password files. These password files are the ones used by the *Stanford srp libraries* and can be specified using the [gnutls_srp_set_server_credentials_file], page 138. If a different password file format is to be used, then the function [gnutls_srp_set_server_credentials_function], page 138, should be called, in order to set an appropriate callback.

Some helper functions such as

- [gnutls_srp_verifier], page 139
- [gnutls_srp_base64_encode], page 136
- [gnutls_srp_base64_decode], page 136

are included in GnuTLS, and can be 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 87, for more information.

4.4 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 paper [TLSPSK] (See [Bibliography], page 229.) . 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.

Clients supporting PSK should supply the username and key before the connection to the client credentials by calling the function [gnutls_psk_set_client_credentials], page 125. Alternatively they could specify a callback function by using the function [gnutls_psk_set_client_credentials_function], page 125. This has the advantage that the callback will be called only if PSK has been negotiated.

In server side the default behaviour 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 126. If a different password file format is to be used, then the function [gnutls_psk_set_server_credentials_function], page 126, should be used instead.

Some helper functions such as:

- [gnutls_hex_encode], page 119
- [gnutls_hex_decode], page 118

are included in GnuTLS, and may be used to generate and maintain PSK keys.

4.5 Authentication and credentials

In GnuTLS every key exchange method is associated with a credentials type. So in order to enable to enable a specific method, the corresponding credentials type should be initialized and set using [gnutls_credentials_set], page 109. A mapping is shown below.

Key exchange algorithms and the corresponding credential types:

Key exchange	**Client credentials**	**Server credentials**
KX_RSA		
KX_DHE_RSA		
KX_DHE_DSS		
KX_RSA_EXPORT	CRD_CERTIFICATE	CRD_CERTIFICATE
KX_SRP_RSA	CRD_SRP	CRD_SRP
KX_SRP_DSS		CRD_CERTIFICATE
KX_SRP	CRD_SRP	CRD_SRP
KX_ANON_DH	CRD_ANON	CRD_ANON
KX_PSK	CRD_PSK	CRD_PSK

4.6 Parameters stored in credentials

Several parameters such as the ones used for Diffie-Hellman authentication are stored within the credentials structures, so all sessions can access them. Those parameters are stored in structures such as `gnutls_dh_params_t` and `gnutls_rsa_params_t`, and functions like [gnutls_certificate_set_dh_params], page 101 and [gnutls_certificate_set_rsa_export_params], page 101 can be used to associate those parameters with the given credentials structure.

Since those parameters need to be renewed from time to time and a global structure such as the credentials, may not be easy to modify since it is accessible by all sessions, an alternative interface is available using a callback function. This can be set using the [gnutls_certificate_set_params_function], page 101. An example is shown below.

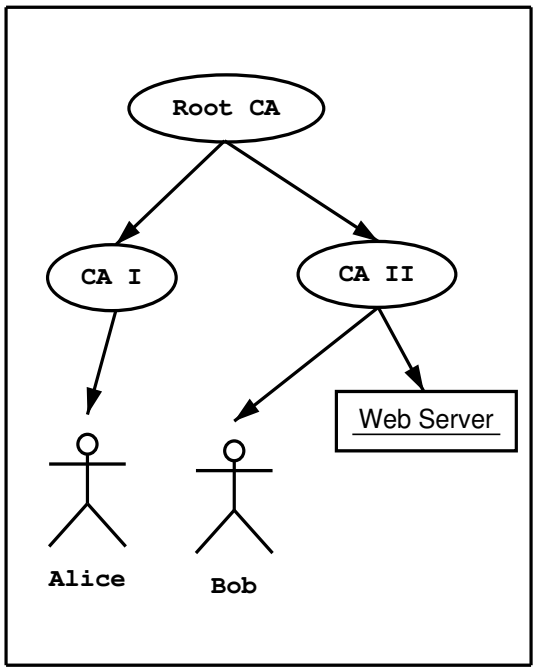
```
#include <gnutls.h>
```

```
gnutls_rsa_params_t rsa_params;
gnutls_dh_params_t dh_params;

/* This function will be called once a session requests DH
 * or RSA parameters. The parameters returned (if any) will
 * be used for the first handshake only.
 */
static int get_params( gnutls_session_t session,
        gnutls_params_type_t type,
        gnutls_params_st *st)
{
   if (type == GNUTLS_PARAMS_RSA_EXPORT)
      st->params.rsa_export = rsa_params;
   else if (type == GNUTLS_PARAMS_DH)
      st->params.dh = dh_params;
   else return -1;

   st->type = type;
   /* do not deinitialize those parameters.
    */
   st->deinit = 0;

   return 0;
}

int main()
{
   gnutls_certificate_credentials_t cert_cred;

   initialize_params();

   /* ...
    */

   gnutls_certificate_set_params_function( cert_cred, get_params);
}
```

5 More on certificate authentication

5.1 The X.509 trust model

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.



Two typical X.509 Certification paths

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. See the figure above for a typical example. The API for handling X.509 certificates is described at section [sec:x509api], page 141. Some examples are listed below.

5.1.1 X.509 certificates

An X.509 certificate usually contains information about the certificate holder, the signer, a unique serial number, expiration dates and some other fields [RFC3280] (See [Bibliography], page 229.) as shown in the table below.

`version:` The field that indicates the version of the certificate.

`serialNumber:`
This field holds a unique serial number per certificate.

`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.

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 IDs 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 the table below.

`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.

`CRL distribution points (2.5.29.31):`
: This extension is set by the CA, in order to inform about the issued CRLs.

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 section [ex:x509-info], page 79.

5.1.2 Verifying X.509 certificate paths

Verifying certificate paths is important in X.509 authentication. For this purpose the function [gnutls_x509_crt_verify], page 176 is provided. The output of this function is

the bitwise OR of the elements of the `gnutls_certificate_status_t` enumeration. A detailed description of these elements can be found in figure below. The function [gnutls_certificate_verify_peers2], page 106 is equivalent to the previous one, and will verify the peer's certificate in a TLS session.

`CERT_INVALID:`
: The certificate is not signed by one of the known authorities, or the signature is invalid.

`CERT_REVOKED:`
: The certificate has been revoked by its CA.

`CERT_SIGNER_NOT_FOUND:`
: The certificate's issuer is not known. This is the case when the issuer is not in the trusted certificates list.

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

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

There is also to possibility to pass some input to the verification functions in the form of flags. For [gnutls_x509_crt_verify], page 176 the flags are passed straightforward, but [gnutls_certificate_verify_peers2], page 106 depends on the flags set by calling [gnutls_certificate_set_verify_flags], page 102. All the available flags are part of the enumeration [gnutls_certificate_verify_flags], page 20 and are explained in the table below.

`GNUTLS_VERIFY_DISABLE_CA_SIGN:`
: If set a signer does not have to be a certificate authority. This flag should normaly be disabled, unless you know what this means.

`GNUTLS_VERIFY_ALLOW_X509_V1_CA_CRT:`
: Allow only trusted CA certificates that have version 1. This is safer than GNUTLS_VERIFY_ALLOW_ANY_X509_V1_CA_CRT, and should be used instead. That way only signers in your trusted list will be allowed to have certificates of version 1.

`GNUTLS_VERIFY_ALLOW_ANY_X509_V1_CA_CRT:`
: Allow CA certificates that have version 1 (both root and intermediate). This is dangerous since those haven't the basicConstraints extension. Must be used in combination with GNUTLS_VERIFY_ALLOW_X509_V1_CA_CRT.

`GNUTLS_VERIFY_DO_NOT_ALLOW_SAME:`
: If a certificate is not signed by anyone trusted but exists in the trusted CA list do not treat it as trusted.

`GNUTLS_VERIFY_ALLOW_SIGN_RSA_MD2:`
: Allow certificates to be signed using the old MD2 algorithm.

`GNUTLS_VERIFY_ALLOW_SIGN_RSA_MD5:`
Allow certificates to be signed using the broken MD5 algorithm.

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] (See [Bibliography], page 229.) and section [ex:verify], page 37 for an example.

5.1.3 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] (See [Bibliography], page 229.) . Other certificate request's format such as PKIX's [RFC2511] (See [Bibliography], page 229.) are not currently supported.

In GnuTLS the PKCS #10 structures are handled using the `gnutls_x509_crq_t` type. An example of a certificate request generation can be found at section [ex:crq], page 82.

5.1.4 PKCS #12 structures

A PKCS #12 structure [PKCS12] (See [Bibliography], page 229.) usually contains a user's private keys and certificates. It is commonly used in browsers to export and import the user's identities.

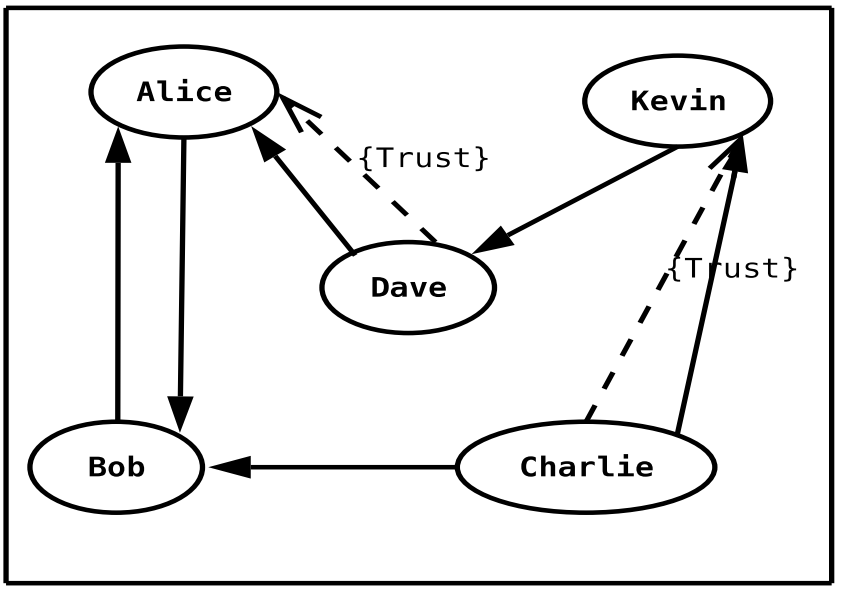
In GnuTLS the PKCS #12 structures are handled using the `gnutls_pkcs12_t` type. This is an abstract type that may hold several `gnutls_pkcs12_bag_t` types. The Bag types are the holders of the actual data, which may be certificates, private keys or encrypted data. An Bag of type encrypted should be decrypted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

An example of a PKCS #12 structure generation can be found at section [ex:pkcs12], page 83.

5.2 The OpenPGP trust model

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's else 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.



An example of the web of trust model

For example: If David trusts Alice to be an introducer, and Alice signed Bob's key, Dave also trusts Bob's key to be the real one.

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.

5.2.1 OpenPGP keys

In GnuTLS the OpenPGP key structures [RFC2440] (See [Bibliography], page 229.) are handled using the `gnutls_openpgp_key_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`'.

5.2.2 Verifying an OpenPGP key

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 (`http://www.gnupg.org/related_software/gpgme/`) is recommended.

There are two verification functions in GnuTLS, The [gnutls_openpgp_key_verify_ring], page 188 and the [gnutls_openpgp_key_verify_trustdb], page 189. The first one 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 is shown in the figure below. The latter function checks a GnuPG trust database for the given key. This function does not check the key signatures, only checks for disabled and revoked keys.

`CERT_INVALID:`
: A signature on the key is invalid. That means that the key was modified by somebody, or corrupted during transport.

`CERT_REVOKED:`
: The key has been revoked by its owner.

`CERT_SIGNER_NOT_FOUND:`
: The key was not signed by a known signer.

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

5.3 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] (See [Bibliography], page 229.) .

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.

5.3.1 Supported algorithms

The available digital signature algorithms in GnuTLS are listed below:

`RSA` RSA is public key cryptosystem designed by Ronald Rivest, Adi Shamir and Leonard Adleman. It can be used with any hash functions.

`DSA` DSA is the USA's Digital Signature Standard. It uses only the SHA-1 hash algorithm.

The supported cryptographic hash algorithms are:

`MD2` MD2 is a cryptographic hash algorithm designed by Ron Rivest. It is optimized for 8-bit processors. Outputs 128 bits of data. There are no known weaknesses of this algorithm but since this algorithm is rarely used and not really studied it should not be used today.

`MD5` MD5 is a cryptographic hash algorithm designed by Ron Rivest. Outputs 128 bits of data. It is considered to be broken.

`SHA-1` SHA is a cryptographic hash algorithm designed by NSA. Outputs 160 bits of data. It is also considered to be broken, though no practical attacks have been found.

`RMD160` RIPEMD is a cryptographic hash algorithm developed in the framework of the EU project RIPE. Outputs 160 bits of data.

5.3.2 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], page 20 validation error (see Section 5.1.2 [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 fail when attempting to verify 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.

First, 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 signed using `RSA-MD2` or `RSA-MD5`. The only attack today is that it is possible to generate certificates with colliding signatures (collision resistance); you cannot generate a certificate that has the same signature as an already existing signature (2nd preimage resistance).

If you are using [gnutls_certificate_verify_peers2], page 106 to verify the certificate chain, you can call [gnutls_certificate_set_verify_flags], page 102 with the `GNUTLS_VERIFY_ALLOW_SIGN_RSA_MD2` or `GNUTLS_VERIFY_ALLOW_SIGN_RSA_MD5` flag, as in:

```
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 176 or [gnutls_x509_crt_list_verify], page 170, 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 100 to extract the raw server's certificate chain, then use [gnutls_x509_crt_import], page 169 to parse each of the certificates, and then use [gnutls_x509_crt_get_signature_algorithm], page 168 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.

6 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.

6.1 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] (See [Bibliography], page 229.) .

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[1]. 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.

6.2 Upward negotiation

Other application protocols[2] use a different approach to enable the secure layer. They use something called 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 is a really neat idea and does not require an extra port.

This method is used by almost all modern protocols and there is even the [RFC2817] (See [Bibliography], page 229.) paper which proposes extensions to HTTP to support it.

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

[1] See also the Server Name Indication extension on [serverind], page 11.

[2] See LDAP, IMAP etc.

*** TLS STARTS

CLIENT: HERE ARE SOME CONFIDENTIAL DATA

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

(client connects to the server)

CLIENT: HELLO I'M MR. XXX

SERVER: NICE TO MEET YOU XXX

CLIENT: PLEASE START TLS

(here someone inserts this message)

SERVER: SORRY I DON'T HAVE THIS CAPABILITY

CLIENT: HERE ARE SOME CONFIDENTIAL DATA

As you can see above the client was fooled, and was dummy enough to send the confidential data in the clear.

How to avoid the above attack? As you may have already thought 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[3], or anything else!

[3] in SRP authentication

7 How to use GnuTLS in applications

7.1 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.

7.1.1 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.

The extra functionality of the GnuTLS-extra library is available by including the header file 'gnutls/extra.h' in your programs.

7.1.2 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 startup. See the function [gnutls_check_version], page 107.

7.1.3 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, GnuTLS ships with two small helper programs `libgnutls-config` and `libgnutls-extra-config` that knows about 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 `libgnutls-config`. The following example shows how it can be used at the command line:

```
gcc -c foo.c `libgnutls-config --cflags`
```

Adding the output of `libgnutls-config --cflags` to the compilers command line will ensure that the compiler can find the GnuTLS 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 `libgnutls-config` can be used. For convenience, this option also outputs all other options that are required to link the program with the GnuTLS libararies. The example shows how to link 'foo.o' with the GnuTLS libraries to a program *foo*.

```
gcc -o foo foo.o `libgnutls-config --libs`
```

Of course you can also combine both examples to a single command by specifying both options to 'libgnutls-config':

```
gcc -o foo foo.c `libgnutls-config --cflags --libs`
```

7.2 Multi-threaded applications

Although the GnuTLS library is thread safe by design, some parts of the crypto backend, such as the random generator, are not. Since *libgcrypt 1.1.92* there was an automatic detection of the thread library used by the application, so most applications wouldn't need to do any changes to ensure thread-safety. Due to the unportability of the automatic thread detection, this was removed from later releases of *libgcrypt*, so applications have now to register callback functions to ensure proper locking in sensitive parts of *libgcrypt*.

There are helper macros to help you properly initialize the libraries. Examples are shown below.

- POSIX threads

```
#include <gnutls.h>
#include <gcrypt.h>
#include <errno.h>
#include <pthread.h>
GCRY_THREAD_OPTION_PTHREAD_IMPL;

int main()
{
   /* The order matters.
    */
   gcry_control (GCRYCTL_SET_THREAD_CBS, &gcry_threads_pthread);
   gnutls_global_init();
}
```

- GNU PTH threads

```
#include <gnutls.h>
#include <gcrypt.h>
#include <errno.h>
#include <pth.h>
GCRY_THREAD_OPTION_PTH_IMPL;

int main()
{
   gcry_control (GCRYCTL_SET_THREAD_CBS, &gcry_threads_pth);
   gnutls_global_init();
}
```

- Other thread packages

```
/* The gcry_thread_cbs structure must have been
 * initialized.
 */
static struct gcry_thread_cbs gcry_threads_other = { ... };

int main()
{
   gcry_control (GCRYCTL_SET_THREAD_CBS, &gcry_threads_other);
}
```

7.3 Client examples

This section contains examples of TLS and SSL clients, using GnuTLS. Note that these examples contain little or no error checking.

7.3.1 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.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.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 SA struct sockaddr
#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. */
  const int kx_prio[] = { GNUTLS_KX_ANON_DH, 0 };

  gnutls_global_init ();

  gnutls_anon_allocate_client_credentials (&anoncred);

  /* Initialize TLS session
   */
```

```
gnutls_init (&session, GNUTLS_CLIENT);

/* Use default priorities */
gnutls_set_default_priority (session);
gnutls_kx_set_priority (session, kx_prio);

/* 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;
}
```

7.3.2 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, it does not support session resuming, not even certificate verification. The TCP functions defined in this example are used in most of the other examples below, without redefining them.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <gnutls/gnutls.h>

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

#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);
```

```
int
main (void)
{
  int ret, sd, ii;
  gnutls_session_t session;
  char buffer[MAX_BUF + 1];
  gnutls_certificate_credentials_t xcred;
  /* Allow connections to servers that have OpenPGP keys as well.
   */
  const int cert_type_priority[3] = { GNUTLS_CRT_X509,
    GNUTLS_CRT_OPENPGP, 0
  };

  gnutls_global_init ();

  /* X509 stuff */
  gnutls_certificate_allocate_credentials (&xcred);

  /* sets the trusted cas file
   */
  gnutls_certificate_set_x509_trust_file (xcred, CAFILE, GNUTLS_X509_FMT_PEM);

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

  /* Use default priorities */
  gnutls_set_default_priority (session);
  gnutls_certificate_type_set_priority (session, cert_type_priority);

  /* 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;
}
```

7.3.3 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 118.

```
#include <stdio.h>
#include <stdlib.h>
#include <gnutls/gnutls.h>
#include <gnutls/x509.h>

extern void print_x509_certificate_info (gnutls_session_t);

/* 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;

  /* 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_SRP:
      printf ("- SRP session with username %s\n",
              gnutls_srp_server_get_username (session));
      break;

    case GNUTLS_CRD_ANON:       /* anonymous authentication */

      printf ("- Anonymous DH using prime of %d bits\n",
              gnutls_dh_get_prime_bits (session));
      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)
      {
        printf ("\n- Ephemeral DH using prime of %d bits\n",
                gnutls_dh_get_prime_bits (session));
      }

    /* if the certificate list is available, then
     * print some information about it.
     */
    print_x509_certificate_info (session);

  }                           /* switch */

/* print the protocol's name (ie TLS 1.0)
 */
tmp = gnutls_protocol_get_name (gnutls_protocol_get_version (session));
printf ("- Protocol: %s\n", tmp);

/* print the certificate type of the peer.
 * ie X.509
 */
tmp =
  gnutls_certificate_type_get_name (gnutls_certificate_type_get (session));

printf ("- Certificate Type: %s\n", tmp);

/* print the compression algorithm (if any)
 */
tmp = gnutls_compression_get_name (gnutls_compression_get (session));
printf ("- Compression: %s\n", tmp);

/* print the name of the cipher used.
 * ie 3DES.
 */
tmp = gnutls_cipher_get_name (gnutls_cipher_get (session));
printf ("- Cipher: %s\n", tmp);

/* Print the MAC algorithms name.
 * ie SHA1
 */
tmp = gnutls_mac_get_name (gnutls_mac_get (session));
printf ("- MAC: %s\n", tmp);
```

```
  return 0;
}
```

7.3.4 Verifying peer's certificate

A TLS session is not secure just after the handshake procedure has finished. It must be considered secure, only after the peer's certificate and identity have been verified. That is, you have to verify the signature in peer's certificate, the hostname in the certificate, and expiration dates. Just after this step you should treat the connection as being a secure one.

```
#include <stdio.h>
#include <gnutls/gnutls.h>
#include <gnutls/x509.h>

/* This function will try to verify the peer's certificate, and
 * also check if the hostname matches, and the activation, expiration dates.
 */
void
verify_certificate (gnutls_session_t session, const char *hostname)
{
  unsigned int status;
  const gnutls_datum_t *cert_list;
  int cert_list_size, ret;
  gnutls_x509_crt_t cert;


  /* 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;
    }

  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");


  /* 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;

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

cert_list = gnutls_certificate_get_peers (session, &cert_list_size);
if (cert_list == NULL)
  {
    printf ("No certificate was found!\n");
    return;
  }

/* 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;
  }

/* Beware here we do not check for errors.
 */
if (gnutls_x509_crt_get_expiration_time (cert) < time (0))
  {
    printf ("The certificate has expired\n");
    return;
  }

if (gnutls_x509_crt_get_activation_time (cert) > time (0))
  {
    printf ("The certificate is not yet activated\n");
    return;
  }

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

```
    }

  gnutls_x509_crt_deinit (cert);

  return;
}
```

An other example is listed below which provides a more detailed verification output.

```
#include <stdio.h>
#include <stdlib.h>
#include <gnutls/gnutls.h>
#include <gnutls/x509.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 void verify_cert2 (gnutls_x509_crt_t crt,
                          gnutls_x509_crt_t issuer,
                          gnutls_x509_crl_t * crl_list, int crl_list_size);
static void verify_last_cert (gnutls_x509_crt_t crt,
                              gnutls_x509_crt_t * ca_list, int ca_list_size,
                              gnutls_x509_crl_t * crl_list,
                              int crl_list_size);


/* This function will try to verify the peer's certificate chain, and
 * also check if the hostname matches, and the activation, expiration dates.
 */
void
verify_certificate_chain (gnutls_session_t session,
                          const char *hostname,
                          const gnutls_datum_t * cert_chain,
                          int cert_chain_length)
{
  int i;
  gnutls_x509_crt_t *cert;

  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);
  }

/* If the last certificate in the chain is self signed ignore it.
 * That is because we want to check against our trusted certificate
 * list.
 */
if (gnutls_x509_crt_check_issuer (cert[cert_chain_length - 1],
                                  cert[cert_chain_length - 1]) > 0
    && cert_chain_length > 0)
  {
    cert_chain_length--;
  }

/* Now verify the certificates against their issuers
 * in the chain.
 */
for (i = 1; i < cert_chain_length; i++)
  {
    verify_cert2 (cert[i - 1], cert[i], crl_list, crl_list_size);
  }

/* Here we must verify the last certificate in the chain against
 * our trusted CA list.
 */
verify_last_cert (cert[cert_chain_length - 1],
                  ca_list, ca_list_size, crl_list, crl_list_size);

/* 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);
  }

for (i = 0; i < cert_chain_length; i++)
  gnutls_x509_crt_deinit (cert[i]);

return;
```

```
}


/* Verifies a certificate against an other certificate
 * which is supposed to be it's issuer. Also checks the
 * crl_list if the certificate is revoked.
 */
static void
verify_cert2 (gnutls_x509_crt_t crt, gnutls_x509_crt_t issuer,
              gnutls_x509_crl_t * crl_list, int crl_list_size)
{
  unsigned int output;
  int ret;
  time_t now = time (0);
  size_t name_size;
  char name[64];

  /* Print information about the certificates to
   * be checked.
   */
  name_size = sizeof (name);
  gnutls_x509_crt_get_dn (crt, name, &name_size);

  fprintf (stderr, "\nCertificate: %s\n", name);

  name_size = sizeof (name);
  gnutls_x509_crt_get_issuer_dn (crt, name, &name_size);

  fprintf (stderr, "Issued by: %s\n", name);

  /* Get the DN of the issuer cert.
   */
  name_size = sizeof (name);
  gnutls_x509_crt_get_dn (issuer, name, &name_size);

  fprintf (stderr, "Checking against: %s\n", name);

  /* Do the actual verification.
   */
  gnutls_x509_crt_verify (crt, &issuer, 1, 0, &output);

  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");

      fprintf (stderr, "\n");
    }
  else
    fprintf (stderr, "Trusted\n");


  /* Now check the expiration dates.
   */
  if (gnutls_x509_crt_get_activation_time (crt) > now)
    fprintf (stderr, "Not yet activated\n");

  if (gnutls_x509_crt_get_expiration_time (crt) < now)
    fprintf (stderr, "Expired\n");

  /* Check if the certificate is revoked.
   */
  ret = gnutls_x509_crt_check_revocation (crt, crl_list, crl_list_size);
  if (ret == 1)
    {                           /* revoked */
      fprintf (stderr, "Revoked\n");
    }
}


/* Verifies a certificate against our trusted CA list.
 * Also checks the crl_list if the certificate is revoked.
 */
static void
verify_last_cert (gnutls_x509_crt_t crt,
                  gnutls_x509_crt_t * ca_list, int ca_list_size,
                  gnutls_x509_crl_t * crl_list, int crl_list_size)
{
  unsigned int output;
  int ret;
  time_t now = time (0);
  size_t name_size;
  char name[64];

  /* Print information about the certificates to
   * be checked.
   */
  name_size = sizeof (name);
  gnutls_x509_crt_get_dn (crt, name, &name_size);
```

```
  fprintf (stderr, "\nCertificate: %s\n", name);

  name_size = sizeof (name);
  gnutls_x509_crt_get_issuer_dn (crt, name, &name_size);

  fprintf (stderr, "Issued by: %s\n", name);

  /* Do the actual verification.
   */
  gnutls_x509_crt_verify (crt, ca_list, ca_list_size,
                          GNUTLS_VERIFY_ALLOW_X509_V1_CA_CRT, &output);

  if (output & GNUTLS_CERT_INVALID)
    {
      fprintf (stderr, "Not trusted");

      if (output & GNUTLS_CERT_SIGNER_NOT_CA)
        fprintf (stderr, ": Issuer is not a CA\n");
      else
        fprintf (stderr, "\n");
    }
  else
    fprintf (stderr, "Trusted\n");


  /* Now check the expiration dates.
   */
  if (gnutls_x509_crt_get_activation_time (crt) > now)
    fprintf (stderr, "Not yet activated\n");

  if (gnutls_x509_crt_get_expiration_time (crt) < now)
    fprintf (stderr, "Expired\n");

  /* Check if the certificate is revoked.
   */
  ret = gnutls_x509_crt_check_revocation (crt, crl_list, crl_list_size);
  if (ret == 1)
    {                           /* revoked */
      fprintf (stderr, "Revoked\n");
    }
}
```

7.3.5 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.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <sys/stat.h>
#include <gnutls/gnutls.h>
#include <gnutls/x509.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_retr_st * st);

gnutls_x509_crt_t crt;
gnutls_x509_privkey_t key;

/* Helper functions to load a certificate and key
 * files into memory.
 */
static gnutls_datum
load_file (const char *file)
{
  FILE *f;
  gnutls_datum 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 data)
{
  free(data.data);
}

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

  data = load_file (CERT_FILE);
  if (data.data == NULL)
    {
      fprintf (stderr, "*** Error loading cert file.\n");
      exit (1);
    }
  gnutls_x509_crt_init (&crt);

  ret = gnutls_x509_crt_import (crt, &data, GNUTLS_X509_FMT_PEM);
  if (ret < 0)
    {
      fprintf (stderr, "*** Error loading key 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 (&key);

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

  unload_file (data);

}

int
main (void)
{
  int ret, sd, ii;
  gnutls_session_t session;
  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);

  /* sets the trusted cas file
   */
  gnutls_certificate_set_x509_trust_file (xcred, CAFILE, GNUTLS_X509_FMT_PEM);

  gnutls_certificate_client_set_retrieve_function (xcred, cert_callback);

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

```
/* Use default priorities */
gnutls_set_default_priority (session);

/* 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 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_retr_st * st)
{
  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)
    {
      st->type = type;
      st->ncerts = 1;

      st->cert.x509 = &crt;
      st->key.x509 = key;

      st->deinit_all = 0;
    }
  else
    {
      return -1;
    }

  return 0;

}
```

7.3.6 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.

```
#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 CRLFILE "crl.pem"
#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;
  size_t session_data_size;

  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_set_default_priority (session);

      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;
}
```

7.3.7 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.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <gnutls/gnutls.h>
#include <gnutls/extra.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 SA struct sockaddr
#define MSG "GET / HTTP/1.0\r\n\r\n"

const int kx_priority[] = { GNUTLS_KX_SRP, GNUTLS_KX_SRP_DSS,
  GNUTLS_KX_SRP_RSA, 0
};

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 ();

  /* now enable the gnutls-extra library which contains the
   * SRP stuff.
   */
  gnutls_global_init_extra ();

  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_set_default_priority (session);
gnutls_kx_set_priority (session, kx_priority);


/* 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;
}
```

7.4 Server examples

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

7.4.1 Echo Server with X.509 authentication

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

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.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_session_t
initialize_tls_session (void)
{
  gnutls_session_t session;

  gnutls_init (&session, GNUTLS_SERVER);

  /* avoid calling all the priority functions, since the defaults
   * are adequate.
   */
  gnutls_set_default_priority (session);

  gnutls_credentials_set (session, GNUTLS_CRD_CERTIFICATE, x509_cred);

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

  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, i;
  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_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, &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); */

    i = 0;
    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_global_deinit ();

  return 0;

}
```

7.4.2 Echo Server with X.509 authentication II

The following example is a server which supports X.509 authentication. This server supports the export-grade cipher suites, the DHE ciphersuites and session resuming.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```
#include <arpa/inet.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.
 * Export-grade ciphersuites and session resuming are supported.
 */

#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 cert_cred;

static void wrap_db_init (void);
static void wrap_db_deinit (void);
static int wrap_db_store (void *dbf, gnutls_datum_t key, gnutls_datum_t data);
static gnutls_datum_t wrap_db_fetch (void *dbf, gnutls_datum_t key);
static int wrap_db_delete (void *dbf, gnutls_datum_t key);

#define TLS_SESSION_CACHE 50

gnutls_session_t
initialize_tls_session (void)
{
  gnutls_session_t session;

  gnutls_init (&session, GNUTLS_SERVER);

  /* Use the default priorities, plus, export cipher suites.
   */
  gnutls_set_default_export_priority (session);

  gnutls_credentials_set (session, GNUTLS_CRD_CERTIFICATE, cert_cred);

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

```
  gnutls_dh_set_prime_bits (session, DH_BITS);

  if (TLS_SESSION_CACHE != 0)
    {
      gnutls_db_set_retrieve_function (session, wrap_db_fetch);
      gnutls_db_set_remove_function (session, wrap_db_delete);
      gnutls_db_set_store_function (session, wrap_db_store);
      gnutls_db_set_ptr (session, NULL);
    }

  return session;
}

gnutls_dh_params_t dh_params;
/* Export-grade cipher suites require temporary RSA
 * keys.
 */
gnutls_rsa_params_t rsa_params;

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. Depends on the
   * security requirements.
   */
  gnutls_dh_params_init (&dh_params);
  gnutls_dh_params_generate2 (dh_params, DH_BITS);

  return 0;
}

static int
generate_rsa_params (void)
{
  gnutls_rsa_params_init (&rsa_params);

  /* Generate RSA parameters - for use with RSA-export
   * cipher suites. These should be discarded and regenerated
   * once a day, once every 500 transactions etc. Depends on the
   * security requirements.
   */

  gnutls_rsa_params_generate2 (rsa_params, 512);
```

```
  return 0;
}

int
main (void)
{
  int err, listen_sd, i;
  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 (&cert_cred);

  gnutls_certificate_set_x509_trust_file (cert_cred, CAFILE,
                                          GNUTLS_X509_FMT_PEM);

  gnutls_certificate_set_x509_crl_file (cert_cred, CRLFILE,
                                        GNUTLS_X509_FMT_PEM);

  gnutls_certificate_set_x509_key_file (cert_cred, CERTFILE, KEYFILE,
                                        GNUTLS_X509_FMT_PEM);

  generate_dh_params ();
  generate_rsa_params ();

  if (TLS_SESSION_CACHE != 0)
    {
      wrap_db_init ();
    }

  gnutls_certificate_set_dh_params (cert_cred, dh_params);
  gnutls_certificate_set_rsa_export_params (cert_cred, rsa_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, &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");

    /* print_info(session); */

    i = 0;
    for (;;)
      {
        memset (buffer, 0, MAX_BUF + 1);
        ret = gnutls_record_recv (session, buffer, MAX_BUF);
```

```
          if (ret == 0)
            {
              printf ("\n- Peer has closed the TLS 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 (cert_cred);

  gnutls_global_deinit ();

  return 0;

}


/* Functions and other stuff needed for session resuming.
 * This is done using a very simple list which holds session ids
 * and session data.
 */

#define MAX_SESSION_ID_SIZE 32
#define MAX_SESSION_DATA_SIZE 512
```

```
typedef struct
{
  char session_id[MAX_SESSION_ID_SIZE];
  int session_id_size;

  char session_data[MAX_SESSION_DATA_SIZE];
  int session_data_size;
} CACHE;

static CACHE *cache_db;
static int cache_db_ptr = 0;

static void
wrap_db_init (void)
{

  /* allocate cache_db */
  cache_db = calloc (1, TLS_SESSION_CACHE * sizeof (CACHE));
}

static void
wrap_db_deinit (void)
{
  return;
}

static int
wrap_db_store (void *dbf, gnutls_datum_t key, gnutls_datum_t data)
{

  if (cache_db == NULL)
    return -1;

  if (key.size > MAX_SESSION_ID_SIZE)
    return -1;
  if (data.size > MAX_SESSION_DATA_SIZE)
    return -1;

  memcpy (cache_db[cache_db_ptr].session_id, key.data, key.size);
  cache_db[cache_db_ptr].session_id_size = key.size;

  memcpy (cache_db[cache_db_ptr].session_data, data.data, data.size);
  cache_db[cache_db_ptr].session_data_size = data.size;

  cache_db_ptr++;
  cache_db_ptr %= TLS_SESSION_CACHE;
```

```
  return 0;
}

static gnutls_datum_t
wrap_db_fetch (void *dbf, gnutls_datum_t key)
{
  gnutls_datum_t res = { NULL, 0 };
  int i;

  if (cache_db == NULL)
    return res;

  for (i = 0; i < TLS_SESSION_CACHE; i++)
    {
      if (key.size == cache_db[i].session_id_size &&
          memcmp (key.data, cache_db[i].session_id, key.size) == 0)
        {


          res.size = cache_db[i].session_data_size;

          res.data = gnutls_malloc (res.size);
          if (res.data == NULL)
            return res;

          memcpy (res.data, cache_db[i].session_data, res.size);

          return res;
        }
    }
  return res;
}

static int
wrap_db_delete (void *dbf, gnutls_datum_t key)
{
  int i;

  if (cache_db == NULL)
    return -1;

  for (i = 0; i < TLS_SESSION_CACHE; i++)
    {
      if (key.size == cache_db[i].session_id_size &&
          memcmp (key.data, cache_db[i].session_id, key.size) == 0)
        {
```

```
          cache_db[i].session_id_size = 0;
          cache_db[i].session_data_size = 0;

          return 0;
        }
    }

  return -1;

}
```

7.4.3 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.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <unistd.h>
#include <gnutls/gnutls.h>
/* Must be linked against gnutls-extra.
 */
#include <gnutls/extra.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;
```

```
const int cert_type_priority[2] = { GNUTLS_CRT_OPENPGP, 0 };
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;
}

/* These are global */
gnutls_certificate_credentials_t x509_cred;

gnutls_session_t
initialize_tls_session (void)
{
  gnutls_session_t session;

  gnutls_init (&session, GNUTLS_SERVER);

  /* avoid calling all the priority functions, since the defaults
   * are adequate.
   */
  gnutls_set_default_priority (session);

  gnutls_credentials_set (session, GNUTLS_CRD_CERTIFICATE, x509_cred);

  /* 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, i;
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_certificate_set_openpgp_key_file (cred, CERTFILE, KEYFILE);

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, &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 ();
  gnutls_certificate_type_set_priority (session, cert_type_priority);

  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); */

  i = 0;
  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;

}
```

7.4.4 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.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <unistd.h>
#include <gnutls/gnutls.h>
#include <gnutls/extra.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;

gnutls_session_t
initialize_tls_session (void)
{
  gnutls_session_t session;
  const int kx_priority[] = { GNUTLS_KX_SRP, GNUTLS_KX_SRP_DSS,
    GNUTLS_KX_SRP_RSA, 0
  };

  gnutls_init (&session, GNUTLS_SERVER);

  gnutls_set_default_priority (session);
  gnutls_kx_set_priority (session, kx_priority);

  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, i;
  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");

/* these must be called once in the program
 */
gnutls_global_init ();
gnutls_global_init_extra ();  /* for SRP */

/* 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, &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");

/* print_info(session); */

i = 0;
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;

}
```

7.4.5 Echo Server with anonymous authentication

This example server support anonymous authentication, and could be used to serve the example client for anonymous authentication.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.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;

gnutls_session_t
initialize_tls_session (void)
{
  gnutls_session_t session;
```

```
  const int kx_prio[] = { GNUTLS_KX_ANON_DH, 0 };

  gnutls_init (&session, GNUTLS_SERVER);

  /* avoid calling all the priority functions, since the defaults
   * are adequate.
   */
  gnutls_set_default_priority (session);
  gnutls_kx_set_priority (session, kx_prio);

  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, i;
  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, &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); */

    i = 0;
    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;

}
```

7.5 Miscellaneous examples

7.5.1 Checking for an alert

This is a function that checks if an alert has been received in the current session.

```
#include <stdio.h>
#include <stdlib.h>
#include <gnutls/gnutls.h>

/* This function will check whether the given return code from
 * a gnutls function (recv/send), is an alert, and will print
 * that alert.
 */
void
check_alert (gnutls_session_t session, int ret)
{
  int last_alert;

  if (ret == GNUTLS_E_WARNING_ALERT_RECEIVED
      || ret == GNUTLS_E_FATAL_ALERT_RECEIVED)
    {
      last_alert = gnutls_alert_get (session);

      /* The check for renegotiation is only useful if we are
       * a server, and we had requested a rehandshake.
       */
      if (last_alert == GNUTLS_A_NO_RENEGOTIATION &&
          ret == GNUTLS_E_WARNING_ALERT_RECEIVED)
        printf ("* Received NO_RENEGOTIATION alert. "
                "Client Does not support renegotiation.\n");
      else
        printf ("* Received alert '%d': %s.\n", last_alert,
                gnutls_alert_get_name (last_alert));
    }
}
```

7.5.2 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.

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <gnutls/gnutls.h>
#include <gnutls/x509.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[128];
  size_t size;
  unsigned int algo, bits;
  time_t expiration_time, activation_time;
  const gnutls_datum_t *cert_list;
  int cert_list_size = 0;
  gnutls_x509_crt_t cert;

  /* 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)
  {

    /* 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");

    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);

    size = sizeof (serial);
    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);

    }
}
```

7.5.3 Certificate request generation

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.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <gnutls/gnutls.h>
#include <gnutls/x509.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];
  int buffer_size = sizeof (buffer);

  gnutls_global_init ();

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

  gnutls_x509_privkey_init (&key);

  /* Generate a 1024 bit RSA private key.
   */
  gnutls_x509_privkey_generate (key, GNUTLS_PK_RSA, 1024, 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_sign (crq, key);

  /* 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;

}
```

7.5.4 PKCS #12 structure generation

The following example is about generating a PKCS #12 structure.

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <gnutls/gnutls.h>
#include <gnutls/pkcs12.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];
  int 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 = { "\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;
}
```

7.6 Compatibility with the OpenSSL library

To ease GnuTLS' integration with existing applications, a compatibility layer with the widely used OpenSSL library is included in the `gnutls-openssl` library. This compatibility layer is not complete and it is not intended to completely reimplement the OpenSSL API with GnuTLS. It only provides source-level compatibility. There is currently no attempt to make it binary-compatible with OpenSSL.

The prototypes for the compatibility functions are in the '`gnutls/openssl.h`' header file.

Current limitations imposed by the compatibility layer include:

- Error handling is not thread safe.

8 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.

8.1 Invoking srptool

The '`srptool`' is a very simple program that emulates the programs in the *Stanford SRP libraries*. It is intended for use in places where you don't expect SRP authentication to be the used for system users. Traditionally *libsrp* used two files. One called 'tpasswd' which holds usernames and verifiers, and 'tpasswd.conf' which holds generators and primes.

How to use srptool:

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

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

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

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

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

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

8.2 Invoking gnutls-cli

Simple client program to set up a TLS connection to some other computer. It sets up a TLS connection and forwards data from the standard input to the secured socket and vice versa.

```
GNU TLS 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.
     --xml                    Print the certificate information in
```

```
                          XML format.
--print-cert              Print the certificate in PEM format.
-p, --port integer        The port to connect to.
--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.
--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.
--pgptrustdb FILE         PGP trustdb file to use.
--pgpcertfile FILE        PGP Public Key (certificate) file 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.
--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
--copyright               prints the program's license
```

8.3 Invoking gnutls-cli-debug

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 a 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 whether the server can accept a bogus TLS record version in the client hello... 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 LZO compression support (GnuTLS 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
```

8.4 Invoking gnutls-serv

Simple server program that listens to incoming TLS connections.

```
GNU TLS 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.
      --pgptrustdb FILE        PGP trustdb file to use.
      --pgpkeyfile FILE        PGP Key file to use.
      --pgpcertfile FILE       PGP Public Key (certificate) file 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.
--srppasswd FILE          SRP password file to use.
--srppasswdconf FILE      SRP password conf file to use.
--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.
-l, --list                Print a list of the supported
                          algorithms  and modes.
-h, --help                prints this help
-v, --version             prints the program's version number
--copyright               prints the program's license
```

8.5 Invoking certtool

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

How to use certtool interactively:

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

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

- To generate parameters for the RSA-EXPORT key exchange, use the command:

  ```
  $ certtool --generate-privkey --bits 512 --outfile rsa.pem
  ```

- 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.

- To create a private key, run:

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

- To create a certificate request, run:

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

- To view the certificate information, use:

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

- 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
```

Certtool's template file format:

- Firstly create a file named 'cert.cfg' that contains the information about the certificate. An example file is listed below.
- Then execute:

```
$ 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"

# 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 prefered 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
```

9 Function reference

9.1 Core functions

The prototypes for the following functions lie in 'gnutls/gnutls.h'.

gnutls_alert_get_name

`const char * gnutls_alert_get_name (gnutls_alert_description_t alert)` [Function]

alert: is an alert number `gnutls_session_t` structure.

This function will return a string that describes the given alert number or NULL. See `gnutls_alert_get()`.

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 if GNUTLS_E_WARNING_ALERT_RECEIVED or GNUTLS_E_FATAL_ALERT_RECEIVED has been returned by a gnutls function. The peer may send alerts if he thinks some things were not right. Check gnutls.h for the available alert descriptions.

If no alert has been received the returned value is undefined.

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 zero on success.

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 0 on success.

gnutls_anon_allocate_client_credentials

`int gnutls_anon_allocate_client_credentials` [Function]
(*gnutls_anon_client_credentials_t* * `sc`)
sc: is a pointer to an `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 a negative value in case of an error.

gnutls_anon_allocate_server_credentials

`int gnutls_anon_allocate_server_credentials` [Function]
(*gnutls_anon_server_credentials_t* * `sc`)
sc: is a pointer to an `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 a negative value in case of an error.

gnutls_anon_free_client_credentials

`void gnutls_anon_free_client_credentials` [Function]
(*gnutls_anon_client_credentials_t* `sc`)
sc: is an `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 an `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 zero 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 zero 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.

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: KX_RSA, KX_DHE_RSA), the same function are to be used to access the authentication data.

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.

gnutls_bye

`int gnutls_bye` (*gnutls_session_t* **session**, *gnutls_close_request_t* **how**) [Function]

session: is a `gnutls_session_t` structure.

how: is an integer

Terminates the current TLS/SSL connection. The connection should have been initiated using `gnutls_handshake()`. `how` should be one of GNUTLS_SHUT_RDWR, GNUTLS_SHUT_WR.

In case of GNUTLS_SHUT_RDWR then the TLS connection gets terminated and further receives and sends will be disallowed. If the return value is zero you may continue using the connection. GNUTLS_SHUT_RDWR actually sends an alert containing a close request and waits for the peer to reply with the same message.

In case of GNUTLS_SHUT_WR then the TLS connection 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, thus causing a transmission error to the peer. This error cannot be distinguished from a malicious party prematurely terminating the session, thus this behavior is not recommended.

This function may also return GNUTLS_E_AGAIN or GNUTLS_E_INTERRUPTED; cf. `gnutls_record_get_direction()`.

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.

gnutls_certificate_allocate_credentials

`int gnutls_certificate_allocate_credentials` [Function]
(*gnutls_certificate_credentials_t* * **res**)

res: is a pointer to an `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 0 on success.

gnutls_certificate_client_get_request_status

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

session: is a gnutls session

This function will return 0 if the peer (server) did not request client authentication or 1 otherwise. Returns a negative value in case of an error.

gnutls_certificate_client_set_retrieve_function

`void gnutls_certificate_client_set_retrieve_function` [Function]
(*gnutls_certificate_credentials_t* **`cred`**, *gnutls_certificate_client_retrieve_function* * **`func`**)

cred: is a `gnutls_certificate_credentials_t` structure.

func: is the callback function

This function sets a callback to be called in order to retrieve the certificate to be used in the handshake. The callback's function prototype is: int (*callback)(gnutls_session_t, const gnutls_datum_t* req_ca_dn, int nreqs, gnutls_pk_algorithm_t* pk_algos, int pk_algos_length, gnutls_retr_st* st);

`st` should contain the certificates and private keys.

`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.

If the callback function is provided then gnutls will call it, in the handshake, after the certificate request message has been received.

The callback function should set the certificate list to be sent, and return 0 on success. If no certificate was selected then the number of certificates should be set to zero. The value (-1) indicates error and the handshake will be terminated.

gnutls_certificate_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.

gnutls_certificate_free_ca_names

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

sc: is an `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.

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 an `gnutls_certificate_credentials_t` structure.

This function will delete all the CAs associated with the given credentials. Servers that do not use `gnutls_certificate_verify_peers2()` may call this to save some memory.

gnutls_certificate_free_credentials

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

sc: is an `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 an `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 an `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_ours

`const gnutls_datum_t * gnutls_certificate_get_ours` [Function]
(*gnutls_session_t* **`session`**)

session: is a gnutls session

This function will return the certificate as sent to the peer, in the last handshake. These certificates are in raw format. In X.509 this is a certificate list. In OpenPGP this is a single certificate. Returns 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

This function will return 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 NULL in case of an error, or if no certificate was sent.

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 an `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_server_set_retrieve_function

`void gnutls_certificate_server_set_retrieve_function` [Function]
(*gnutls_certificate_credentials_t* **`cred`**, *gnutls_certificate_server_retrieve_function* * **`func`**)

cred: is a `gnutls_certificate_credentials_t` structure.

func: is the callback function

This function sets a callback to be called in order to retrieve the certificate to be used in the handshake. The callback's function prototype is: int (*callback)(gnutls_session_t, gnutls_retr_st* st);

`st` should contain the certificates and private keys.

If the callback function is provided then gnutls will call it, in the handshake, after the certificate request message has been received.

The callback function should set the certificate list to be sent, and return 0 on success. The value (-1) indicates error and the handshake will be terminated.

gnutls_certificate_set_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 zero on success.

gnutls_certificate_set_rsa_export_params

`void gnutls_certificate_set_rsa_export_params` [Function]
(*gnutls_certificate_credentials_t* **`res`**, *gnutls_rsa_params_t* **`rsa_params`**)

res: is a gnutls_certificate_credentials_t structure

rsa_params: is a structure that holds temporary RSA parameters.

This function will set the temporary RSA parameters for a certificate server to use. These parameters will be used in RSA-EXPORT cipher suites.

gnutls_certificate_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_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.

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 an `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 the number of CRLs processed or a negative value 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 an `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 the number of CRLs processed or a negative value on error.

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 an `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 0 on success.

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 an `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).

Currently only PKCS-1 encoded RSA and DSA private keys are accepted by this function.

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 an `gnutls_certificate_credentials_t` structure.

cert: contains a certificate list (path) for the specified private key

key: is 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).

Currently are supported: RSA PKCS-1 encoded private keys, DSA private keys.

DSA private keys are encoded the OpenSSL way, which is an ASN.1 DER sequence of 6 INTEGERs - version, p, q, g, pub, priv.

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.

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 an `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).

gnutls_certificate_set_x509_simple_pkcs12_file

`int gnutls_certificate_set_x509_simple_pkcs12_file` [Function]
(*gnutls_certificate_credentials_t* **`res`**, *const char ** **`pkcs12file`**, *gnutls_x509_crt_fmt_t* **`type`**, *const char ** **`password`**)

res: is an `gnutls_certificate_credentials_t` structure.

pkcs12file: filename of file containing PKCS12 blob.

type: is PEM or DER of the `pkcs12file`.

password: optional password used to decrypt PKCS12 file, bags and keys.

This function sets a certificate/private key pair and/or a CRL in the gnutls_certificate_credentials_t structure. This function may be called more than once (in case multiple keys/certificates exist for the server).

MAC: ed PKCS12 files are supported. Encrypted PKCS12 bags are supported. Encrypted PKCS8 private keys are supported. However, only password based security, and the same password for all operations, are supported.

The private keys may be RSA PKCS1 or DSA private keys encoded in the OpenSSL way.

PKCS12 file may contain many keys and/or certificates, and there is no way to identify which key/certificate pair you want. You should make sure the PKCS12 file only contain one key/certificate pair and/or one CRL.

It is believed that the limitations of this function is acceptable for most usage, and that any more flexibility would introduce complexity that would make it harder to use this functionality at all.

Return value: Returns 0 on success, or an error code.

gnutls_certificate_set_x509_trust_file

`int gnutls_certificate_set_x509_trust_file` [Function]
(*gnutls_certificate_credentials_t* `res`, *const char ** `cafile`, *gnutls_x509_crt_fmt_t* `type`)

res: is an `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()`.

Returns the number of certificates processed or a negative value 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 an `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 value on error.

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 an `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 0 on success.

gnutls_certificate_type_get_name

`const char * gnutls_certificate_type_get_name` [Function]
(*gnutls_certificate_type_t* ***type***)

type: is a certificate type

Returns a string (or NULL) that contains the name of the specified certificate type.

gnutls_certificate_type_get

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

session: is a `gnutls_session_t` structure.

Returns the currently used certificate type. The certificate type is by default X.509, unless it is negotiated as a TLS extension.

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.

Sets the priority on the certificate types supported by gnutls. Priority is higher for types 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 0 on success.

gnutls_certificate_verify_peers2

`int gnutls_certificate_verify_peers2` (*gnutls_session_t* ***session***, *unsigned int * **status***) [Function]

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.

Returns a negative error code on error and zero on success.

This is the same as `gnutls_x509_verify_certificate()` and uses the loaded CAs in the credentials as trusted CAs.

Note that some commonly used X.509 Certificate Authorities are still using Version 1 certificates. If you want to accept them, you need to call `gnutls_certificate_set_verify_flags()` with, e.g., `GNUTLS_VERIFY_ALLOW_X509_V1_CA_CRT` parameter.

gnutls_certificate_verify_peers

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

session: is a gnutls session

This function will try to verify the peer's certificate and return its status (trusted, invalid etc.). However you must also check the peer's name in order to check if the verified certificate belongs to the actual peer.

The return value should be one or more of the gnutls_certificate_status_t enumerated elements bitwise or'd, or a negative value on error.

This is the same as `gnutls_x509_verify_certificate()`.

Deprecated: Use `gnutls_certificate_verify_peers2()` instead.

gnutls_check_version

`const char * gnutls_check_version (`*const char * req_version*`)` [Function]

req_version: the version to check

Check that the version of the library is at minimum the requested one and return the version string; return NULL if the condition is not satisfied. If a NULL is passed to this function, no check is done, but the version string is simply returned.

gnutls_cipher_get_key_size

`size_t gnutls_cipher_get_key_size (`*gnutls_cipher_algorithm_t* ***algorithm***`)` [Function]

algorithm: is an encryption algorithm

Returns the length (in bytes) of the given cipher's key size. Returns 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

Returns a pointer to a string that contains the name of the specified cipher or NULL.

gnutls_cipher_get

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

session: is a `gnutls_session_t` structure.

Returns the currently used cipher.

gnutls_cipher_set_priority

`int gnutls_cipher_set_priority (`*gnutls_session_t* **`session`**, *const int ** **`list`**`)` [Function]

session: is a `gnutls_session_t` structure.

list: is a 0 terminated list of gnutls_cipher_algorithm_t elements.

Sets the priority on the ciphers supported by gnutls. Priority is higher for ciphers 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 0 on success.

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

Returns a string that contains the name of a TLS cipher suite, specified by the given algorithms, or NULL.

Note that the full cipher suite name must be prepended by TLS or SSL depending of the protocol in use.

gnutls_compression_get_name

`const char * gnutls_compression_get_name (`*gnutls_compression_method_t* **`algorithm`**`)` [Function]

algorithm: is a Compression algorithm

Returns a pointer to a string that contains the name of the specified compression algorithm or NULL.

gnutls_compression_get

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

session: is a `gnutls_session_t` structure.

Returns the currently used compression method.

gnutls_compression_set_priority

`int gnutls_compression_set_priority (`*gnutls_session_t* **`session`**, *const int ** **`list`**`)` [Function]

session: is a `gnutls_session_t` structure.

list: is a 0 terminated list of gnutls_compression_method_t elements.

Sets the priority on the compression algorithms supported by gnutls. Priority is higher for algorithms 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 0 on success.

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`**, *gnutls_credentials_type_t* **`type`**, *void ** **`cred`**`)` [Function]

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 (void* 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.

gnutls_db_check_entry

`int gnutls_db_check_entry (`*gnutls_session_t* **`session`**, *gnutls_datum_t* **`session_entry`**`)` [Function]

session: is a `gnutls_session_t` structure.

session_entry: is the session data (not key)

This function returns GNUTLS_E_EXPIRED, if the database entry has expired or 0 otherwise. This function is to be used when you want to clear unnesessary session which occupy space in your backend.

gnutls_db_get_ptr

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

session: is a `gnutls_session_t` structure.

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***`,` [Function]
int ***seconds***`)`

session: is a `gnutls_session_t` structure.

seconds: is the number of seconds.

Sets 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***`,` [Function]
gnutls_db_remove_func ***rem_func***`)`

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`**`,` [Function]
gnutls_db_retr_func **`retr_func`**`)`

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`**`,` [Function]
gnutls_db_store_func **`store_func`**`)`

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 ** [Function]
`raw_gen``,` *gnutls_datum_t ** **`raw_prime`**`)`

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 authentication 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 a negative value in case of an error.

gnutls_dh_get_peers_public_bits

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

session: is a gnutls session

This function will return the bits used in the last Diffie Hellman authentication with the peer. Should be used for both anonymous and ephemeral diffie Hellman. Returns a negative value in case of an 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 authentication with the peer. Should be used for both anonymous and ephemeral diffie Hellman. Returns a negative value in case of an error.

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 authentication. This function should be used for both anonymous and ephemeral diffie Hellman. The output parameters must be freed with `gnutls_free()`.

Returns a negative value in case of an error.

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 authentication with the peer. Should be used for both anonymous and ephemeral diffie Hellman. Returns a negative value in case of an error.

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.

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`**, *gnutls_x509_crt_fmt_t* **`format`**, *unsigned char ** **`params_data`**, *size_t ** **`params_data_size`**`)` [Function]

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".

In case of failure a negative value will be returned, and 0 on success.

gnutls_dh_params_export_raw

`int gnutls_dh_params_export_raw (`*gnutls_dh_params_t* **`params`**, *gnutls_datum_t ** **`prime`**, *gnutls_datum_t ** **`generator`**, *unsigned int ** **`bits`**`)` [Function]

params: Holds the DH parameters

prime: will hold the new prime

generator: will hold the new generator

bits: if non null will hold is the prime'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.

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.

Note that the bits value should be one of 768, 1024, 2048, 3072 or 4096. 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.

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".

In case of failure a negative value will be returned, and 0 on success.

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.

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.

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.

gnutls_error_is_fatal

`int gnutls_error_is_fatal (int error)` [Function]

error: is an error returned by a gnutls function. Error should be a negative value.

If a function returns a negative value you may feed that value to this function to see if it is fatal. Returns 1 for a fatal error 0 otherwise. However you may want to check the error code manually, since some non-fatal errors to the protocol may be fatal for you (your program).

This is only useful if you are dealing with errors from the record layer or the handshake layer.

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

Returns 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 == GNUTLS_E_REHANDSHAKE, where a warning alert should be sent to the peer indicating that no renegotiation will be performed.

If the return value is GNUTLS_E_INVALID_REQUEST, then there was no mapping to an alert.

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 a negative value in case of an error.

gnutls_free

`void gnutls_free (void * ptr)` [Function]

This function will free data pointed by ptr.

The deallocation function used is the one set by `gnutls_global_set_mem_functions()`.

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 must call `gnutls_global_deinit()` when gnutls usage is no longer needed Returns zero on success.

Note that this function will also initialize libgcrypt, if it has not been initialized before. Thus if you want to manually initialize libgcrypt you must do it before calling this function. This is useful in cases you want to disable libgcrypt's internal lockings etc.

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.

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 (ie the gcrypt allocation functions).

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

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.

Returns the last handshake message received. 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.h for the available handshake descriptions.

gnutls_handshake_get_last_out

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

session: is a `gnutls_session_t` structure.

Returns the last handshake message sent. 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.h for the available handshake descriptions.

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 a handshake message. Handshake messages over this size are rejected. The default value is 16kb which is large enough. Set this to 0 if you do not want to set an upper limit.

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.

Unless this function is called with the option to allow (1), then no compression algorithms, like LZO. That is because these algorithms are not yet defined in any RFC or even internet draft.

Enabling the private ciphersuites when talking to other than gnutls servers and clients may cause interoperability problems.

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.

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***, gnutls_connection_end_t **con_end***)` [Function]

session: is a pointer to a `gnutls_session_t` structure.

con_end: is used to indicate if this session is to be used for server or client. Can be one of GNUTLS_CLIENT and GNUTLS_SERVER.

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 zero on success.

gnutls_kx_get_name

`const char * gnutls_kx_get_name (`*gnutls_kx_algorithm_t **algorithm***)` [Function]

algorithm: is a key exchange algorithm

Returns a pointer to a string that contains the name of the specified key exchange algorithm or NULL.

gnutls_kx_get

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

session: is a `gnutls_session_t` structure.

Returns the key exchange algorithm used in the last handshake.

gnutls_kx_set_priority

`int gnutls_kx_set_priority (`*gnutls_session_t **session***, *const int * **list***)` [Function]

session: is a `gnutls_session_t` structure.

list: is a 0 terminated list of gnutls_kx_algorithm_t elements.

Sets the priority on the key exchange algorithms supported by gnutls. Priority is higher for algorithms 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 0 on success.

gnutls_mac_get_name

`const char * gnutls_mac_get_name (`*gnutls_mac_algorithm_t* ***algorithm***`)` [Function]

algorithm: is a MAC algorithm

Returns a string that contains the name of the specified MAC algorithm or NULL.

gnutls_mac_get

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

session: is a `gnutls_session_t` structure.

Returns the currently used mac algorithm.

gnutls_mac_set_priority

`int gnutls_mac_set_priority (`*gnutls_session_t* ***session***, *const int ** ***list***`)` [Function]

session: is a `gnutls_session_t` structure.

list: is a 0 terminated list of gnutls_mac_algorithm_t elements.

Sets the priority on the mac algorithms supported by gnutls. Priority is higher for algorithms 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 0 on success.

gnutls_malloc

`void * gnutls_malloc (`*size_t* ***s***`)` [Function]

This function will allocate 's' bytes data, and return a pointer to memory. This function is supposed to be used by callbacks.

The allocation function used is the one set by `gnutls_global_set_mem_functions()`.

gnutls_openpgp_send_key

`void gnutls_openpgp_send_key (`*gnutls_session_t* ***session***, *gnutls_openpgp_key_status_t* ***status***`)` [Function]

session: is a pointer to a `gnutls_session_t` structure.

status: is one of OPENPGP_KEY, or OPENPGP_KEY_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_alloc

`int gnutls_pem_base64_decode_alloc (const char * header, const gnutls_datum_t * b64_data, gnutls_datum_t * result)` [Function]

header: The PEM header (eg. CERTIFICATE)

b64_data: contains the encoded data

result: the place where decoded data lie

This function will decode the given encoded data. The decoded data will be allocated, and stored into result. If the header given is non null this function will search for "——BEGIN header" and decode only this part. Otherwise it will decode the first PEM packet found.

You should use `gnutls_free()` to free the returned data.

gnutls_pem_base64_decode

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

header: A null terminated string with the PEM header (eg. CERTIFICATE)

b64_data: contain the encoded data

result: the place where decoded data will be copied

result_size: holds the size of the result

This function will decode the given encoded data. If the header given is non null this function will search for "——BEGIN header" and decode only this part. Otherwise it will decode the first PEM packet found.

Returns GNUTLS_E_SHORT_MEMORY_BUFFER 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.

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. If the provided buffer is not long enough GNUTLS_E_SHORT_MEMORY_BUFFER is returned.

The output string will be null terminated, although the size will not include the terminating null.

gnutls_perror

`void gnutls_perror (`*int* `error``)` [Function]

error: is an error returned by a gnutls function. Error is always a negative value.

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

Returns a string that contains the name of the specified public key algorithm or NULL.

gnutls_prf_raw

`int gnutls_prf_raw (`*gnutls_session_t* `session`, *size_t* `label_size`, [Function]
*const char * * `label`, *size_t* `seed_size`, *const char * * `seed`, *size_t* `outsize`, *char * * `out`)

session: is a `gnutls_session_t` structure.

label_size: length of the `label` variable.

label: label used in PRF computation, typically a short string.

seed_size: length of the `seed` variable.

seed: optional extra data to seed the PRF with.

outsize: size of pre-allocated output buffer to hold the output.

out: pre-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.

Return value: Return 0 on success, or an error code.

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.

Return value: Return 0 on success, or an error code.

gnutls_protocol_get_name

`const char * gnutls_protocol_get_name (`*gnutls_protocol_t* `version``)` [Function]

version: is a (gnutls) version number

Returns a string that contains the name of the specified TLS version 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.

Returns the version of the currently used protocol.

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.

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 0 on success.

gnutls_psk_allocate_client_credentials

`int gnutls_psk_allocate_client_credentials` (*gnutls_psk_client_credentials_t * sc*) [Function]

sc: is a pointer to an `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 0 on success.

gnutls_psk_allocate_server_credentials

`int gnutls_psk_allocate_server_credentials` (*gnutls_psk_server_credentials_t * sc*) [Function]

sc: is a pointer to an `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 0 on success.

gnutls_psk_free_client_credentials

`void gnutls_psk_free_client_credentials` (*gnutls_psk_client_credentials_t* `sc`) [Function]

sc: is an `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` (*gnutls_psk_server_credentials_t* `sc`) [Function]

sc: is an `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 function will return the username of the peer. This should only be called in case of PSK authentication and in case of a server. Returns NULL in case of an error.

gnutls_psk_set_client_credentials_function

`void gnutls_psk_set_client_credentials_function (gnutls_psk_client_credentials_t cred, gnutls_psk_client_credentials_function * func)` [Function]

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* key);

The `username` and `key` 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_client_credentials

`int gnutls_psk_set_client_credentials (gnutls_psk_client_credentials_t res, const char * username, const gnutls_datum * key, unsigned int flags)` [Function]

res: is an `gnutls_psk_client_credentials_t` structure.

username: is the user's zero-terminated userid

key: is the user's key

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 (not with the '0x' prefix).

Returns 0 on success.

gnutls_psk_set_params_function

`void gnutls_psk_set_params_function (gnutls_psk_server_credentials_t res, gnutls_params_function * func)` [Function]

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 zero 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 an `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 0 on success.

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_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 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 zero 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 any data to receive in the gnutls buffers. Returns the size of that data or 0. Notice that you may also use `select()` to check for data in a TCP connection, instead of this function. (gnutls leaves some data in the tcp buffer in order for select to work).

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.

gnutls_record_get_max_size

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

session: is a `gnutls_session_t` structure.

This function returns the maximum record packet size in this connection. The maximum record size is negotiated by the client after the first handshake message.

gnutls_record_recv

`ssize_t gnutls_record_recv` (*gnutls_session_t* **`session`**, *void ** **`data`**, *size_t* **`sizeofdata`**) [Function]

session: is a `gnutls_session_t` structure.

data: the buffer that the data will be read into

sizeofdata: the number of requested bytes

This function has the similar semantics with `recv()`. The only difference is that is 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 containing 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 `code{recv()}`) 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. `code{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. A negative error code is returned in case of an error. The number of bytes received might be less than `code{`count`}`.

gnutls_record_send

`ssize_t gnutls_record_send` (*gnutls_session_t* ***session***, *const void ** ***data***, *size_t* ***sizeofdata***) [Function]

session: is a `gnutls_session_t` structure.

data: contains the data to send

sizeofdata: is the length of the data

This function has the similar semantics with `send()`. The only difference is that is 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 `sizeofdata`. 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). Returns 0 on success. 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.

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.

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.

gnutls_rsa_export_get_modulus_bits

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

session: is a gnutls session

This function will return the bits used in the last RSA-EXPORT key exchange with the peer. Returns a negative value in case of an 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.

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 a negative value in case of an error.

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

This function will copy the RSA parameters structure from source to destination.

gnutls_rsa_params_deinit

`void gnutls_rsa_params_deinit (`*gnutls_rsa_params_t* ***rsa_params***`)` [Function]

rsa_params: Is a structure that holds the parameters

This function will deinitialize the RSA parameters structure.

gnutls_rsa_params_export_pkcs1

`int gnutls_rsa_params_export_pkcs1` (*gnutls_rsa_params_t* **params**, *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)

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".

In case of failure a negative value will be returned, and 0 on success.

gnutls_rsa_params_export_raw

`int gnutls_rsa_params_export_raw` (*gnutls_rsa_params_t* **params**, *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]

params: a structure that holds the rsa parameters

m: will hold the modulus

e: will hold the public exponent

d: will hold the private exponent

p: will hold the first prime (p)

q: will hold the second prime (q)

u: will hold the coefficient

bits: if non null will hold the prime's number of bits

This function will export the RSA parameters found in the given structure. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

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

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.

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.

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".

In case of failure a negative value will be returned, and 0 on success.

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

This function will replace the parameters in the given structure. The new parameters should be stored in the appropriate gnutls_datum.

gnutls_rsa_params_init

`int gnutls_rsa_params_init` (*gnutls_rsa_params_t * **`rsa_params`**) [Function]

rsa_params: Is a structure that will hold the parameters

This function will initialize the temporary RSA parameters structure.

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.

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 `ind` type. In case of GNUTLS_NAME_DNS, an ASCII or UTF-8 null terminated string, without the trailing dot, is expected. IPv4 or IPv6 addresses are not permitted.

gnutls_session_get_client_random

`const void * gnutls_session_get_client_random` (*gnutls_session_t* **`session`**) [Function]

session: is a `gnutls_session_t` structure.

Return a pointer to the 32-byte client random field used in the session. The pointer must not be modified or deallocated.

If a client random value has not yet been established, the output will be garbage; in particular, a `NULL` return value should not be expected.

Return value: pointer to client random.

gnutls_session_get_data2

`int gnutls_session_get_data2` (*gnutls_session_t* **`session`**, *gnutls_datum ** **`data`**) [Function]

session: is a `gnutls_session_t` structure.

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 succesful one.

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 succesful one.

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 zero on success.

gnutls_session_get_master_secret

`const void * gnutls_session_get_master_secret (`*gnutls_session_t* ***session***`)` [Function]

session: is a `gnutls_session_t` structure.

Return a pointer to the 48-byte master secret in the session. The pointer must not be modified or deallocated.

If a master secret value has not yet been established, the output will be garbage; in particular, a `NULL` return value should not be expected.

Consider using `gnutls_prf()` rather than extracting the master secret and use it to derive further data.

Return value: pointer to master secret.

gnutls_session_get_ptr

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

session: is a `gnutls_session_t` structure.

This function will return the user given pointer from the session structure. This is the pointer set with `gnutls_session_set_ptr()`.

gnutls_session_get_server_random

`const void * gnutls_session_get_server_random` [Function]
`(gnutls_session_t session)`

session: is a `gnutls_session_t` structure.

Return a pointer to the 32-byte server random field used in the session. The pointer must not be modified or deallocated.

If a server random value has not yet been established, the output will be garbage; in particular, a `NULL` return value should not be expected.

Return value: pointer to server random.

gnutls_session_is_resumed

`int gnutls_session_is_resumed (gnutls_session_t session)` [Function]

session: is a `gnutls_session_t` structure.

This function will return non zero if this session is a resumed one, or a zero if this is a new session.

gnutls_session_set_data

`int gnutls_session_set_data (gnutls_session_t session, const void *` [Function]
`session_data, size_t session_data_size)`

session: is a `gnutls_session_t` structure.

session_data: is a pointer to space to hold the session.

session_data_size: is the session's size

Sets all session parameters, in order to resume a previously established session. The session data given must be the one returned by `gnutls_session_get_data()`. This function should be called before `gnutls_handshake()`.

Keep in mind that session resuming is advisory. The server may choose not to resume the session, thus a full handshake will be performed.

Returns a negative value on error.

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 to the session structure. This is pointer can be accessed with `gnutls_session_get_ptr()`.

gnutls_set_default_export_priority

`int gnutls_set_default_export_priority (gnutls_session_t` [Function]
`session)`

session: is a `gnutls_session_t` structure.

Sets some default priority on the ciphers, key exchange methods, macs and compression methods. This is to avoid using the gnutls_*_priority() functions, if these defaults are ok. This function also includes weak algorithms. The order is TLS1, SSL3 for protocols, RSA, DHE_DSS, DHE_RSA, RSA_EXPORT for key exchange algorithms. SHA, MD5, RIPEMD160 for MAC algorithms, AES_256_CBC, AES_128_CBC, and 3DES_CBC, ARCFOUR_128, ARCFOUR_40 for ciphers.

Returns 0 on success.

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. This is to avoid using the gnutls_*_priority() functions, if these defaults are ok. You may override any of the following priorities by calling the appropriate functions.

The order is TLS1, SSL3 for protocols. RSA, DHE_DSS, DHE_RSA for key exchange algorithms. SHA, MD5 and RIPEMD160 for MAC algorithms. AES_128_CBC, 3DES_CBC, and ARCFOUR_128 for ciphers.

Returns 0 on success.

gnutls_sign_algorithm_get_name

`const char * gnutls_sign_algorithm_get_name` [Function]
(*gnutls_sign_algorithm_t* `sign`)

Returns a string that contains the name of the specified sign algorithm or NULL.

gnutls_srp_allocate_client_credentials

`int gnutls_srp_allocate_client_credentials` [Function]
(*gnutls_srp_client_credentials_t* * `sc`)

sc: is a pointer to an `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 0 on success.

gnutls_srp_allocate_server_credentials

`int gnutls_srp_allocate_server_credentials` [Function]
(*gnutls_srp_server_credentials_t* * `sc`)

sc: is a pointer to an `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 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 found in libsrp.

You should use `gnutls_free()` to free the returned data.

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.

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.

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.

gnutls_srp_free_client_credentials

`void gnutls_srp_free_client_credentials` [Function]
(*gnutls_srp_client_credentials_t* `sc`)

sc: is an `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 an `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.

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, unsigned int times, char** username, char** password);

The `username` and `password` must be allocated using `gnutls_malloc()`. `times` will be 0 the first time called, and 1 the second. `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 or twice per handshake. The first time called, is before the ciphersuite is negotiated. At that time if the callback returns a negative error code, the callback will be called again if SRP has been negotiated. This uses a special TLS-SRP idiom in order to avoid asking the user for SRP password and username if the server does not support SRP.

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_client_credentials

`int gnutls_srp_set_client_credentials` [Function]
(*gnutls_srp_client_credentials_t* `res`, *const char ** `username`, *const char ** `password`)

res: is an `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 0 on success.

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 an `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 0 on success.

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 extra.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/extra.h or may be generated using the GCRYPT functions `gcry_prime_generate()` and `gcry_prime_group_generator()`. The verifier will be allocated with `malloc` and will be stored in `res` using binary format.

gnutls_strerror

`const char * gnutls_strerror` (*int **error***) [Function]

error: is an error returned by a gnutls function. Error is always a negative value.

This function is similar to `strerror()`. Differences: it accepts an error number returned by a gnutls function; In case of an unknown error a descriptive string is sent instead of NULL.

gnutls_transport_get_ptr2

`void gnutls_transport_get_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: 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_get_ptr

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

session: is a `gnutls_session_t` structure.

Used to get the first argument of the transport function (like PUSH and PULL). This must have been set using `gnutls_transport_set_ptr()`.

gnutls_transport_set_lowat

`void gnutls_transport_set_lowat (`*gnutls_session_t* **`session`**`,` *int* **`num`**`)` [Function]

session: is a `gnutls_session_t` structure.

num: is the low water value.

Used to set the lowat value in order for select to check if there are pending data to socket buffer. Used only if you have changed the default low water value (default is 1). Normally you will not need that function. This function is only useful if using berkeley style sockets. Otherwise it must be called and set lowat to zero.

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 (like PUSH and PULL). In berkeley style sockets this function will set the connection handle. With this function you can use two different pointers for receiving and sending.

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 (like PUSH and PULL). In berkeley style sockets this function will set the connection handle.

gnutls_transport_set_pull_function

`void gnutls_transport_set_pull_function (`*gnutls_session_t* **`session`**`,` *gnutls_pull_func* **`pull_func`**`)` [Function]

session: gnutls session

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.

PULL_FUNC is of the form, ssize_t (*gnutls_pull_func)(gnutls_transport_ptr_t, void*, size_t);

gnutls_transport_set_push_function

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

session: gnutls session

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.

PUSH_FUNC is of the form, ssize_t (*gnutls_push_func)(gnutls_transport_ptr_t, const void*, size_t);

9.2 X.509 certificate functions

The following functions are to be used for X.509 certificate handling. Their prototypes lie in '`gnutls/x509.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. This can only be ASCII.

This function will decrypt the given encrypted bag and return 0 on success.

gnutls_pkcs12_bag_deinit

`void gnutls_pkcs12_bag_deinit (`*gnutls_pkcs12_bag_t* **`bag`**`)` [Function]

bag: The structure to be initialized

This function will deinitialize a PKCS12 Bag structure.

gnutls_pkcs12_bag_encrypt

`int gnutls_pkcs12_bag_encrypt (`*gnutls_pkcs12_bag_t* **`bag`**, *const char * * **`pass`**, *unsigned int* **`flags`**`)` [Function]

bag: The bag

pass: The password used for encryption. This can only be ASCII.

flags: should be one of gnutls_pkcs_encrypt_flags_t elements bitwise or'd

This function will encrypt the given bag and return 0 on success.

gnutls_pkcs12_bag_get_count

`int gnutls_pkcs12_bag_get_count (`*gnutls_pkcs12_bag_t* **`bag`**`)` [Function]

bag: The bag

This function will return the number of the elements withing the bag.

gnutls_pkcs12_bag_get_data

`int gnutls_pkcs12_bag_get_data (`*gnutls_pkcs12_bag_t* **`bag`**, *int* **`indx`**, *gnutls_datum_t ** **`data`**`)` [Function]

bag: The bag

indx: The element of the bag to get the data from

data: where the bag's data will be. Should be treated as constant.

This function will return the bag's data. The data is a constant that is stored into the bag. Should not be accessed after the bag is deleted.

Returns 0 on success and a negative error code on error.

gnutls_pkcs12_bag_get_friendly_name

`int gnutls_pkcs12_bag_get_friendly_name (`*gnutls_pkcs12_bag_t* **`bag`**, *int* **`indx`**, *char *** **`name`**`)` [Function]

bag: The bag

indx: The bag's element to add the id

name: will hold a pointer to the name (to be treated as const)

This function will return the friendly name, of the specified bag element. The key ID is usually used to distinguish the local private key and the certificate pair.

Returns 0 on success, or a negative value on error.

gnutls_pkcs12_bag_get_key_id

`int gnutls_pkcs12_bag_get_key_id (`*gnutls_pkcs12_bag_t* **`bag`**, *int* **`indx`**, *gnutls_datum_t ** **`id`**`)` [Function]

bag: The bag

indx: The bag's element to add the id

id: where the ID will be copied (to be treated as const)

This function will return the key ID, of the specified bag element. The key ID is usually used to distinguish the local private key and the certificate pair.

Returns 0 on success, or a negative value on error.

gnutls_pkcs12_bag_get_type

`gnutls_pkcs12_bag_type_t gnutls_pkcs12_bag_get_type (`*gnutls_pkcs12_bag_t* **`bag`**, *int* **`indx`**`)` [Function]

bag: The bag

indx: The element of the bag to get the type

This function will return the bag's type. One of the gnutls_pkcs12_bag_type_t enumerations.

gnutls_pkcs12_bag_init

`int gnutls_pkcs12_bag_init (`*gnutls_pkcs12_bag_t* * `bag``)` [Function]

bag: The structure to be initialized

This function will initialize a PKCS12 bag structure. PKCS12 Bags usually contain private keys, lists of X.509 Certificates and X.509 Certificate revocation lists.

Returns 0 on success.

gnutls_pkcs12_bag_set_crl

`int gnutls_pkcs12_bag_set_crl (`*gnutls_pkcs12_bag_t* `bag`, *gnutls_x509_crl_t* `crl``)` [Function]

bag: The bag

crl: the CRL to be copied.

This function will insert the given CRL into the bag. This is just a wrapper over `gnutls_pkcs12_bag_set_data()`.

Returns the index of the added bag on success, or a negative value 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.

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.

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

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 0 on success, or a negative value 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

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 0 on success, or a negative value on error.

gnutls_pkcs12_deinit

`void gnutls_pkcs12_deinit (`*gnutls_pkcs12_t* `pkcs12``)` [Function]

pkcs12: The structure to be initialized

This function will deinitialize a PKCS12 structure.

gnutls_pkcs12_export

`int gnutls_pkcs12_export (`*gnutls_pkcs12_t* `pkcs12`*, 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)

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".

Return value: In case of failure a negative value 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

This function will generate a MAC for the PKCS12 structure. Returns 0 on success.

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

This function will return a Bag from the PKCS12 structure. Returns 0 on success.

After the last Bag has been read GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE will be returned.

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

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 0 on success.

gnutls_pkcs12_init

`int gnutls_pkcs12_init (`*gnutls_pkcs12_t ** `pkcs12`)` [Function]

pkcs12: The structure to be initialized

This function will initialize a PKCS12 structure. PKCS12 structures usually contain lists of X.509 Certificates and X.509 Certificate revocation lists.

Returns 0 on success.

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

This function will insert a Bag into the PKCS12 structure. Returns 0 on success.

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

This function will verify the MAC for the PKCS12 structure. Returns 0 on success.

gnutls_pkcs7_deinit

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

pkcs7: The structure to be initialized

This function will deinitialize a PKCS7 structure.

gnutls_pkcs7_delete_crl

`int gnutls_pkcs7_delete_crl (`*gnutls_pkcs7_t* **`pkcs7`**`, `*int* **`indx`**`)` [Function]

indx: the index of the crl to delete

This function will delete a crl from a PKCS7 or RFC2630 crl set. Index starts from 0. Returns 0 on success.

gnutls_pkcs7_delete_crt

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

indx: the index of the certificate to delete

This function will delete a certificate from a PKCS7 or RFC2630 certificate set. Index starts from 0. Returns 0 on success.

gnutls_pkcs7_export

`int gnutls_pkcs7_export (`*gnutls_pkcs7_t* **`pkcs7`**`,` [Function]
gnutls_x509_crt_fmt_t **`format`**`, `*void ** **`output_data`**`, `*size_t ** **`output_data_size`**`)`

pkcs7: Holds the pkcs7 structure

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

output_data: will contain a structure PEM or DER encoded

output_data_size: holds the size of output_data (and will be replaced by the actual size of parameters)

This function will export the pkcs7 structure to DER or PEM format.

If the buffer provided is not long enough to hold the output, then *output_data_size is updated and GNUTLS_E_SHORT_MEMORY_BUFFER will be returned.

If the structure is PEM encoded, it will have a header of "BEGIN PKCS7".

Return value: In case of failure a negative value will be returned, and 0 on success.

gnutls_pkcs7_get_crl_count

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

This function will return the number of certifcates in the PKCS7 or RFC2630 crl set.

Returns a negative value on failure.

gnutls_pkcs7_get_crl_raw

`int gnutls_pkcs7_get_crl_raw` (*gnutls_pkcs7_t* **pkcs7**, *int* **indx**, *void* * **crl**, *size_t* * **crl_size**) [Function]

indx: contains the index of the crl to extract

crl: the contents of the crl will be copied there (may be null)

crl_size: should hold the size of the crl

This function will return a crl of the PKCS7 or RFC2630 crl set. Returns 0 on success. 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]

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

Returns a negative value on failure.

gnutls_pkcs7_get_crt_raw

`int gnutls_pkcs7_get_crt_raw` (*gnutls_pkcs7_t* **pkcs7**, *int* **indx**, *void* * **certificate**, *size_t* * **certificate_size**) [Function]

indx: contains the index of the certificate to extract

certificate: the contents of the certificate will be copied there (may be null)

certificate_size: should hold the size of the certificate

This function will return a certificate of the PKCS7 or RFC2630 certificate set. Returns 0 on success. If the provided buffer is not long enough, then `certificate_size` is updated and GNUTLS_E_SHORT_MEMORY_BUFFER is returned.

After the last certificate has been read GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE will be 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

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 0 on success.

gnutls_pkcs7_init

`int gnutls_pkcs7_init (`*gnutls_pkcs7_t * pkcs7*`)` [Function]

pkcs7: The structure to be initialized

This function will initialize a PKCS7 structure. PKCS7 structures usually contain lists of X.509 Certificates and X.509 Certificate revocation lists.

Returns 0 on success.

gnutls_pkcs7_set_crl_raw

`int gnutls_pkcs7_set_crl_raw (`*gnutls_pkcs7_t pkcs7*, *const gnutls_datum_t * crl*`)` [Function]

crl: the DER encoded crl to be added

This function will add a crl to the PKCS7 or RFC2630 crl set. Returns 0 on success.

gnutls_pkcs7_set_crl

`int gnutls_pkcs7_set_crl (`*gnutls_pkcs7_t pkcs7*, *gnutls_x509_crl_t crl*`)` [Function]

crl: the DER encoded crl to be added

This function will add a parsed crl to the PKCS7 or RFC2630 crl set. Returns 0 on success.

gnutls_pkcs7_set_crt_raw

`int gnutls_pkcs7_set_crt_raw (`*gnutls_pkcs7_t pkcs7*, *const gnutls_datum_t * crt*`)` [Function]

crt: the DER encoded certificate to be added

This function will add a certificate to the PKCS7 or RFC2630 certificate set. Returns 0 on success.

gnutls_pkcs7_set_crt

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

crt: the certificate to be copied.

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

Returns 0 on success.

gnutls_x509_crl_check_issuer

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

issuer: is the certificate of a possible issuer

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.

A negative value is returned in case of an error.

gnutls_x509_crl_deinit

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

crl: The structure to be initialized

This function will deinitialize a CRL structure.

gnutls_x509_crl_export

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

crl: Holds the revocation list

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

output_data: will contain a private key PEM or DER encoded

output_data_size: holds the size of output_data (and will be replaced by the actual size of parameters)

This function will export the revocation list to DER or PEM format.

If the buffer provided is not long enough to hold the output, then GNUTLS_E_SHORT_MEMORY_BUFFER will be returned.

If the structure is PEM encoded, it will have a header of "BEGIN X509 CRL".

Returns 0 on success, and a negative value on failure.

gnutls_x509_crl_get_crt_count

`int gnutls_x509_crl_get_crt_count (`*gnutls_x509_crl_t* **`crl`**`)` [Function]

crl: should contain a gnutls_x509_crl_t structure

This function will return the number of revoked certificates in the given CRL.

Returns a negative value on failure.

gnutls_x509_crl_get_crt_serial

`int gnutls_x509_crl_get_crt_serial (`*gnutls_x509_crl_t* **`crl`**`,` *int* [Function]
`indx``,` *unsigned char ** **`serial`**`,` *size_t ** **`serial_size`**`,` *time_t ** **`t`**`)`

crl: should contain a gnutls_x509_crl_t structure

indx: the index of the certificate to extract (starting from 0)

serial: where the serial number will be copied

serial_size: initially holds the size of serial

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

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

Returns a negative value on failure.

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 zero to get the first one.

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

sizeof_oid: initially holds the size of 'oid'

This function will extract the requested OID of the name of the CRL issuer, specified by the given index.

If oid is null then only the size will be filled.

Returns GNUTLS_E_SHORT_MEMORY_BUFFER if the provided buffer is not long enough, and in that case the sizeof_oid will be updated with the required size. On success 0 is returned.

gnutls_x509_crl_get_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 zero to get the first one.

raw_flag: If non zero returns the raw DER data of the DN part.

buf: a pointer to a structure to hold the peer's name (may be null)

sizeof_buf: initially holds the size of `buf`

This function will extract the part of the name of the CRL issuer specified by the given OID. The output will be encoded as described in 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 zero, 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_issuer_dn

`int gnutls_x509_crl_get_issuer_dn (`*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`

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_next_update

`time_t gnutls_x509_crl_get_next_update (`*gnutls_x509_crl_t* `crl``)` [Function]

crl: should contain a gnutls_x509_crl_t structure

This function will return the time the next CRL will be issued. This field is optional in a CRL so it might be normal to get an error instead.

Returns (time_t)-1 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

This function will return a value of the gnutls_sign_algorithm_t enumeration that is the signature algorithm.

Returns a negative value on error.

gnutls_x509_crl_get_this_update

`time_t gnutls_x509_crl_get_this_update (`*gnutls_x509_crl_t* `crl``)` [Function]

crl: should contain a gnutls_x509_crl_t structure

This function will return the time this CRL was issued.

Returns (time_t)-1 on error.

gnutls_x509_crl_get_version

`int gnutls_x509_crl_get_version (`*gnutls_x509_crl_t* `crl``)` [Function]

crl: should contain a gnutls_x509_crl_t structure

This function will return the version of the specified CRL.

Returns a negative value 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

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 0 on success.

gnutls_x509_crl_init

`int gnutls_x509_crl_init (`*gnutls_x509_crl_t* * **`crl`**`)` [Function]

crl: The structure to be initialized

This function will initialize a CRL structure. CRL stands for Certificate Revocation List. A revocation list usually contains lists of certificate serial numbers that have been revoked by an Authority. The revocation lists are always signed with the authority's private key.

Returns 0 on success.

gnutls_x509_crl_set_crt_serial

`int gnutls_x509_crl_set_crt_serial (`*gnutls_x509_crl_t* **`crl`**`,` *const void* * **`serial`**`,` *size_t* **`serial_size`**`,` *time_t* **`revocation_time`**`)` [Function]

crl: should contain a gnutls_x509_crl_t structure

serial: The revoked certificate's serial number

serial_size: Holds the size of the serial field.

revocation_time: The time this certificate was revoked

This function will set a revoked certificate's serial number to the CRL.

Returns 0 on success, or a negative value in case of an error.

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: should contain a gnutls_x509_crt_t structure with the revoked certificate

revocation_time: The time this certificate was revoked

This function will set a revoked certificate's serial number to the CRL.

Returns 0 on success, or a negative value in case of an error.

gnutls_x509_crl_set_next_update

`int gnutls_x509_crl_set_next_update (`*gnutls_x509_crl_t* **`crl`**`,` *time_t* **`exp_time`**`)` [Function]

crl: should contain a gnutls_x509_crl_t structure

exp_time: The actual time

This function will set the time this CRL will be updated.

Returns 0 on success, or a negative value in case of an error.

gnutls_x509_crl_set_this_update

`int gnutls_x509_crl_set_this_update (`*gnutls_x509_crl_t* `crl`, *time_t* `act_time``)` [Function]

crl: should contain a gnutls_x509_crl_t structure

act_time: The actual time

This function will set the time this CRL was issued.

Returns 0 on success, or a negative value in case of an error.

gnutls_x509_crl_set_version

`int gnutls_x509_crl_set_version (`*gnutls_x509_crl_t* `crl`, *unsigned int* `version``)` [Function]

crl: should contain a gnutls_x509_crl_t structure

version: holds the version number. For CRLv1 crls must be 1.

This function will set the version of the CRL. This must be one for CRL version 1, and so on. The CRLs generated by gnutls should have a version number of 2.

Returns 0 on success.

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

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 0 on success.

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

This function is the same a `gnutls_x509_crl_sign2()` with no flags, and SHA1 as the hash algorithm.

Returns 0 on success.

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.

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 0 on success and a negative value in case of an error.

gnutls_x509_crq_deinit

`void gnutls_x509_crq_deinit (`*gnutls_x509_crq_t* **`crq`**`)` [Function]

crq: The structure to be initialized

This function will deinitialize a CRL 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: Holds the request

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

output_data: will contain a certificate request PEM or DER encoded

output_data_size: holds the size of output_data (and will be replaced by the actual size of parameters)

This function will export the certificate request to a PKCS10

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".

Return value: In case of failure a negative value will be returned, and 0 on success.

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 null terminated string

indx: In case multiple same OIDs exist in the attribute list, this specifies which to send. Use zero 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`

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

Returns 0 on success.

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 null terminated password

sizeof_pass: Initially holds the size of `pass`.

This function will return the challenge password in the request.

Returns 0 on success.

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 zero to get the first one.

raw_flag: If non zero returns the raw DER data of the DN part.

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

sizeof_buf: initially holds the size of `buf`

This function will extract the part of the name of the Certificate request subject, specified by the given OID. The output will be encoded as described in RFC2253. The output string will be ASCII or UTF-8 encoded, depending on the certificate data.

Some helper macros with popular OIDs can be found in gnutls/x509.h If raw flag is zero, 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. 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 zero to get the first one.

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

sizeof_oid: initially holds the size of `oid`

This function will extract the requested OID of the name of the Certificate request subject, specified by the given index.

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_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`

This function will copy the name of the Certificate request subject 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. On success 0 is returned.

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

This function will return the public key algorithm of a PKCS `\10` 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 value on error.

gnutls_x509_crq_get_version

`int gnutls_x509_crq_get_version (`*gnutls_x509_crq_t* **`crq`**`)` [Function]

crq: should contain a gnutls_x509_crq_t structure

This function will return the version of the specified Certificate request.

Returns a negative value on error.

gnutls_x509_crq_import

`int gnutls_x509_crq_import (`*gnutls_x509_crq_t* **`crq`**`, const` *gnutls_datum_t ** **`data`**`, gnutls_x509_crt_fmt_t` **`format`**`)` [Function]

crq: The structure to store the parsed certificate request.

data: The DER or PEM encoded certificate.

format: One of DER or PEM

This function will convert the given DER or PEM encoded Certificate to the native gnutls_x509_crq_t format. The output will be stored in `cert`.

If the Certificate is PEM encoded it should have a header of "NEW CERTIFICATE REQUEST".

Returns 0 on success.

gnutls_x509_crq_init

`int gnutls_x509_crq_init (`*gnutls_x509_crq_t ** **`crq`**`)` [Function]

crq: The structure to be initialized

This function will initialize a PKCS10 certificate request structure.

Returns 0 on success.

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 null terminated string

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

sizeof_buf: holds the size of `buf`

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

Returns 0 on success.

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 null terminated password

This function will set a challenge password to be used when revoking the request.

Returns 0 on success.

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 null terminated string

raw_flag: must be 0, or 1 if the data are DER encoded

data: a pointer to the input data

sizeof_data: holds the size of `data`

This function will set the part of the name of the Certificate request subject, specified by the given OID. The input string should be ASCII or UTF-8 encoded.

Some helper macros with popular OIDs can be found in gnutls/x509.h With this function you can only set the known OIDs. You can test for known OIDs using `gnutls_x509_dn_oid_known()`. For OIDs that are not known (by gnutls) you should properly DER encode your data, and call this function with raw_flag set.

Returns 0 on success.

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

This function will set the public parameters from the given private key to the request. Only RSA keys are currently supported.

Returns 0 on success.

gnutls_x509_crq_set_version

`int gnutls_x509_crq_set_version` (*gnutls_x509_crq_t* `crq`, *unsigned int* `version`) [Function]

crq: should contain a gnutls_x509_crq_t structure

version: holds the version number. For v1 Requests must be 1.

This function will set the version of the certificate request. For version 1 requests this must be one.

Returns 0 on success.

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. GNUTLS_DIG_SHA1 is the safe choice unless you know what you're doing.

flags: must be 0

This function will sign the certificate request with a private key. This must be the same key as the one used in `gnutls_x509_crt_set_key()` since a certificate request is self signed.

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

Returns 0 on success.

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

This function is the same a `gnutls_x509_crq_sign2()` with no flags, and SHA1 as the hash algorithm.

Returns 0 on success.

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

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 subject alternative name PKIX extension.

Returns non zero on success, and zero 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

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

A negative value 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

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 value 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: should contain a gnutls_x509_crt_t structure

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

This function will copy the CRL distribution points certificate extension, from the source to the destination certificate. This may be useful to copy from a CA certificate to issued ones.

Returns 0 on success.

gnutls_x509_crt_deinit

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

cert: The structure to be initialized

This function will deinitialize a CRL 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)

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".

Return value: In case of failure a negative value 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

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

Returns (time_t)-1 on error.

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

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

This function will return the X.509v3 certificate authority's key identifier. This is obtained by the X.509 Authority Key identifier extension field (2.5.29.35). Note that this function only returns the keyIdentifier field of the extension.

Returns 0 on success and a negative value in case of an error.

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 zero if the extension is marked as critical

This function will return certificates CA status, by reading the basicConstraints X.509 extension (2.5.29.19). If the certificate is a CA a positive value will be returned, or zero if the certificate does not have CA flag set.

A negative value 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 zero if the extension is marked as critical (may be null)

This function will return the CRL distribution points (2.5.29.31), contained in the given certificate.

`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 GNUTLS_CRL_REASON_PRIVILEGE_WITHDRAWN, GNUTLS_CRL_REASON_AA_COMPROMIS or zero for all possible reasons.

This is specified in X509v3 Certificate Extensions. GNUTLS will return the distribution point type, or a negative error code on error.

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 returns GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE;

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 ** **`sizeof_buf`**`)` [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 zero to get the first one.

raw_flag: If non zero returns the raw DER data of the DN part.

buf: a pointer where the DN part will be copied (may be null).

sizeof_buf: initially holds the size of `buf`

This function will extract the part of the name of the Certificate subject specified by the given OID. The output, if the raw flag is not used, will be encoded as described in 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 zero, 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. 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 ** **`sizeof_oid`**`)` [Function]

cert: should contain a gnutls_x509_crt_t structure

indx: This specifies which OID to return. Use zero 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`

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.

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_crt_get_dn

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

cert: should contain a gnutls_x509_crt_t structure

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

sizeof_buf: initially holds the size of `buf`

This function will copy the name of the Certificate in the provided buffer. The name will be in the form "C=xxxx,O=yyyy,CN=zzzz" as described in 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. 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

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

Returns (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 ** `sizeof_buf`*, 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 zero 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 zero if the extension is marked as critical

This function will return the extension specified by the OID in the certificate. The extensions will be returned as binary data DER encoded, in the provided buffer.

A negative value may be returned in case of parsing error. If the certificate does not contain the specified extension 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 * * `sizeof_oid`) [Function]

cert: should contain a gnutls_x509_crt_t structure

indx: Specifies which extension OID to send. Use zero to get the first one.

oid: a pointer to a structure to hold the OID (may be null)

sizeof_oid: initially holds the size of `oid`

This function will return the requested extension OID in the certificate. The extension OID will be stored as a string in the provided buffer.

A negative value may be returned in case of parsing error. If your 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 * * `sizeof_buf`) [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)

sizeof_buf: initially holds the size of `buf`

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 *sizeof_buf 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 * * `sizeof_buf`) [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 zero to get the first one.

raw_flag: If non zero returns the raw DER data of the DN part.

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

sizeof_buf: initially holds the size of `buf`

This function will extract the part of the name of the Certificate issuer specified by the given OID. The output, if the raw flag is not used, will be encoded as described in 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 zero, 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. 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 * * `sizeof_oid`)` [Function]

cert: should contain a gnutls_x509_crt_t structure

indx: This specifies which OID to return. Use zero 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`

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.

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_crt_get_issuer_dn

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

cert: should contain a gnutls_x509_crt_t structure

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

sizeof_buf: initially holds the size of `buf`

This function will copy the name of the Certificate issuer in the provided buffer. The name will be in the form "C=xxxx,O=yyyy,CN=zzzz" as described in 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. On success 0 is returned.

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)

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.

Return value: In case of failure a negative value 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 * *`sizeof_oid`*, unsigned int * *`critical`*)` [Function]

cert: should contain a gnutls_x509_crt_t structure

indx: This specifies which OID to return. Use zero 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`

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.

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_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 zero if the extension is marked as critical

This function will return certificate's key usage, by reading the keyUsage X.509 extension (2.5.29.15). The key usage value will ORed values of the: GNUTLS_KEY_DIGITAL_SIGNATURE, GNUTLS_KEY_NON_REPUDIATION, GNUTLS_KEY_KEY_ENCIPHERMENT, GNUTLS_KEY_DATA_ENCIPHERMENT, GNUTLS_KEY_KEY_AGREEMENT, GNUTLS_KEY_KEY_CERT_SIGN, GNUTLS_KEY_CRL_SIGN, GNUTLS_KEY_ENCIPHER_ONLY, GNUTLS_KEY_DECIPHER_ONLY.

A negative value may be returned 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

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 value 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

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.

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

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.

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.

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 0 on success and a negative value in case of an 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

This function will return a value of the gnutls_sign_algorithm_t enumeration that is the signature algorithm.

Returns a negative value on error.

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

This function will return the alternative names, contained in the given certificate.

This is specified in X509v3 Certificate Extensions. GNUTLS will return the Alternative name (2.5.29.17), or a negative error code.

Returns GNUTLS_E_SHORT_MEMORY_BUFFER if &ret_size is not enough to hold the alternative name. In that case &ret_size will be updated. If everything was ok the type of alternative name is returned. 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 returns GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE;

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

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

This function will return the X.509v3 certificate's subject key identifier. This is obtained by the X.509 Subject Key identifier extension field (2.5.29.14).

Returns 0 on success and a negative value in case of an error.

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

This function will return the version of the specified Certificate.

Returns a negative value 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

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 0 on success.

gnutls_x509_crt_init

`int gnutls_x509_crt_init` (*gnutls_x509_crt_t* * `cert`) [Function]

cert: The structure to be initialized

This function will initialize an X.509 certificate structure.

Returns 0 on success.

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 zero or an OR'd sequence of gnutls_certificate_import_flags.

This function will convert the given PEM encoded certificate list to the native gnutls_x509_crt_t format. The output will be stored in `certs`. They will be automatically initialized.

If the Certificate is PEM encoded it should have a header of "X509 CERTIFICATE", or "CERTIFICATE".

Returns the number of certificates read or a negative error value.

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.

This function will try to verify the given certificate list and return its status. Note that expiration and activation dates are not checked by this function, you should check them using the appropriate functions.

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.

GNUTLS_CERT_INVALID: the certificate chain is not valid.

GNUTLS_CERT_REVOKED: a certificate in the chain has been revoked.

Returns 0 on success and a negative value in case of an error.

gnutls_x509_crt_set_activation_time

`int gnutls_x509_crt_set_activation_time (`*gnutls_x509_crt_t* `cert`, *time_t* `act_time``)` [Function]

cert: should contain a gnutls_x509_crt_t structure

act_time: The actual time

This function will set the time this Certificate was or will be activated.

Returns 0 on success, or a negative value in case of an error.

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: should contain a gnutls_x509_crt_t structure

id: The key ID

id_size: Holds the size of the serial field.

This function will set the X.509 certificate's authority key ID extension. Only the keyIdentifier field can be set with this function.

Returns 0 on success, or a negative value in case of an error.

gnutls_x509_crt_set_ca_status

`int gnutls_x509_crt_set_ca_status (`*gnutls_x509_crt_t* `crt`, *unsigned int* `ca`) [Function]

crt: should contain a gnutls_x509_crt_t structure

ca: true(1) or false(0). Depending on the Certificate authority status.

This function will set the basicConstraints certificate extension.

Returns 0 on success.

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: should contain a gnutls_x509_crt_t structure

type: is one of the gnutls_x509_subject_alt_name_t enumerations

data_string: The data to be set

reason_flags: revocation reasons

This function will set the CRL distribution points certificate extension.

Returns 0 on success.

gnutls_x509_crt_set_crq

`int gnutls_x509_crt_set_crq` (*gnutls_x509_crt_t* `crt`, *gnutls_x509_crq_t* `crq`) [Function]

crt: should contain a gnutls_x509_crt_t structure

crq: holds a certificate request

This function will set the name and public parameters from the given certificate request to the certificate. Only RSA keys are currently supported.

Returns 0 on success.

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: should contain a gnutls_x509_crt_t structure

oid: holds an Object Identifier in a null terminated string

raw_flag: must be 0, or 1 if the data are DER encoded

name: a pointer to the name

sizeof_name: holds the size of `name`

This function will set the part of the name of the Certificate subject, specified by the given OID. The input string should be ASCII or UTF-8 encoded.

Some helper macros with popular OIDs can be found in gnutls/x509.h With this function you can only set the known OIDs. You can test for known OIDs using `gnutls_x509_dn_oid_known()`. For OIDs that are not known (by gnutls) you should properly DER encode your data, and call this function with raw_flag set.

Returns 0 on success.

gnutls_x509_crt_set_expiration_time

`int gnutls_x509_crt_set_expiration_time` (*gnutls_x509_crt_t* `cert`, *time_t* `exp_time`) [Function]

cert: should contain a gnutls_x509_crt_t structure

exp_time: The actual time

This function will set the time this Certificate will expire.

Returns 0 on success, or a negative value in case of an error.

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: should contain a gnutls_x509_crt_t structure

oid: holds an Object Identified in null terminated string

buf: a pointer to a DER encoded data

sizeof_buf: holds the size of `buf`

critical: should be non zero if the extension is to be marked as critical

This function will set an the extension, by the specified OID, in the certificate. The extension data should be binary data DER encoded.

Returns 0 on success and a negative value in case of an error.

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: should contain a gnutls_x509_crt_t structure

oid: holds an Object Identifier in a null terminated string

raw_flag: must be 0, or 1 if the data are DER encoded

name: a pointer to the name

sizeof_name: holds the size of `name`

This function will set the part of the name of the Certificate issuer, specified by the given OID. The input string should be ASCII or UTF-8 encoded.

Some helper macros with popular OIDs can be found in gnutls/x509.h With this function you can only set the known OIDs. You can test for known OIDs using `gnutls_x509_dn_oid_known()`. For OIDs that are not known (by gnutls) you should properly DER encode your data, and call this function with raw_flag set.

Normally you do not need to call this function, since the signing operation will copy the signer's name as the issuer of the certificate.

Returns 0 on success.

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: should contain a gnutls_x509_crt_t structure

oid: a pointer to a null terminated string that holds the OID

critical: Whether this extension will be critical or not

This function will set the key purpose OIDs of the Certificate. These are stored in the Extended Key Usage extension (2.5.29.37) See the GNUTLS_KP_* definitions for human readable names.

Subsequent calls to this function will append OIDs to the OID list.

On success 0 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: should contain a gnutls_x509_crt_t structure

usage: an ORed sequence of the GNUTLS_KEY_* elements.

This function will set the keyUsage certificate extension.

Returns 0 on success.

gnutls_x509_crt_set_key

`int gnutls_x509_crt_set_key (`*gnutls_x509_crt_t* **`crt`**`,` [Function]
gnutls_x509_privkey_t **`key`**`)`

crt: should contain a gnutls_x509_crt_t structure

key: holds a private key

This function will set the public parameters from the given private key to the certificate. Only RSA keys are currently supported.

Returns 0 on success.

gnutls_x509_crt_set_serial

`int gnutls_x509_crt_set_serial (`*gnutls_x509_crt_t* **`cert`**`,` *const void* [Function]
`*` **`serial`**`,` *size_t* **`serial_size`**`)`

cert: should contain a gnutls_x509_crt_t structure

serial: The serial number

serial_size: Holds the size of the serial field.

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 0 on success, or a negative value in case of an error.

gnutls_x509_crt_set_subject_alternative_name

`int gnutls_x509_crt_set_subject_alternative_name` [Function]
`(`*gnutls_x509_crt_t* **`crt`**`,` *gnutls_x509_subject_alt_name_t* **`type`**`,` *const char* `*`
`data_string``)`

crt: should contain a gnutls_x509_crt_t structure

type: is one of the gnutls_x509_subject_alt_name_t enumerations

data_string: The data to be set

This function will set the subject alternative name certificate extension.

Returns 0 on success.

gnutls_x509_crt_set_subject_key_id

`int gnutls_x509_crt_set_subject_key_id (`*gnutls_x509_crt_t* **`cert`**`,` [Function]
const void `*` **`id`**`,` *size_t* **`id_size`**`)`

cert: should contain a gnutls_x509_crt_t structure

id: The key ID

id_size: Holds the size of the serial field.

This function will set the X.509 certificate's subject key ID extension.

Returns 0 on success, or a negative value in case of an error.

gnutls_x509_crt_set_version

`int gnutls_x509_crt_set_version (`*gnutls_x509_crt_t* **`crt`**, *unsigned int* **`version`**`)` [Function]

crt: should contain a gnutls_x509_crt_t structure

version: holds the version number. For X.509v1 certificates must be 1.

This function will set the version of the certificate. This must be one for X.509 version 1, and so on. Plain certificates without extensions must have version set to one.

Returns 0 on success.

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: should contain a gnutls_x509_crt_t structure

issuer: is the certificate of the certificate issuer

issuer_key: holds the issuer's private key

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

flags: must be 0

This function will sign the 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 0 on success.

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: should contain a gnutls_x509_crt_t structure

issuer: is the certificate of the certificate issuer

issuer_key: holds the issuer's private key

This function is the same a `gnutls_x509_crt_sign2()` with no flags, and SHA1 as the hash algorithm.

Returns 0 on success.

gnutls_x509_crt_to_xml

`int gnutls_x509_crt_to_xml (`*gnutls_x509_crt_t* **`cert`**, *gnutls_datum_t* ** `res`**, *int* **`detail`**`)` [Function]

cert: should contain a gnutls_x509_crt_t structure

res: The datum that will hold the result

detail: The detail level (must be GNUTLS_XML_SHOW_ALL or GNUTLS_XML_NORMAL)

This function will return the XML structures of the given X.509 certificate. The XML structures are allocated internally (with malloc) and stored into res.

NOTE: This function is currently not implemented. See the NEWS entry for version 1.3.5.

Returns a negative error code in case of an error.

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

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

In case of a verification failure 0 is returned, and 1 on success.

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.

This function will try to verify the given certificate and return its status. The verification output in this functions cannot be GNUTLS_CERT_NOT_VALID.

Returns 0 on success and a negative value in case of an error.

gnutls_x509_dn_oid_known

`int gnutls_x509_dn_oid_known (`*const char ** `oid``)` [Function]

oid: holds an Object Identifier in a null terminated string

This function will inform about known DN OIDs. This is useful since functions like `gnutls_x509_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

This function will copy a private key from source to destination key.

gnutls_x509_privkey_deinit

`void gnutls_x509_privkey_deinit (`*gnutls_x509_privkey_t* **`key`**`)` [Function]

key: The structure to be initialized

This function will deinitialize a private key structure.

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]

p: will hold the p

q: will hold the q

g: will hold the g

y: will hold the y

x: will hold the x

This function will export the DSA private key's parameters found in the given structure. The new parameters will be allocated using `gnutls_malloc()` and will be stored in the appropriate datum.

gnutls_x509_privkey_export_pkcs8

`int gnutls_x509_privkey_export_pkcs8 (`*gnutls_x509_privkey_t* **`key`**, *gnutls_x509_crt_fmt_t* **`format`**, *const char ** **`password`**, *unsigned int* **`flags`**, *void ** **`output_data`**, *size_t ** **`output_data_size`**`)` [Function]

key: Holds the key

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

password: the password that will be used to encrypt the key.

flags: an ORed sequence of gnutls_pkcs_encrypt_flags_t

output_data: will contain a private key PEM or DER encoded

output_data_size: holds the size of output_data (and will be replaced by the actual size of parameters)

This function will export the private key to a PKCS8 structure. Currently only RSA keys can be exported since there is no documented standard for other keys. 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.

Return value: In case of failure a negative value 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

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.

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)

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".

Return value: In case of failure a negative value will be returned, and 0 on success.

gnutls_x509_privkey_fix

`int gnutls_x509_privkey_fix (`*gnutls_x509_privkey_t* `key``)` [Function]

key: Holds the key

This function will recalculate the secondary parameters in a key. In RSA keys, this can be the coefficient and exponent1,2.

Return value: In case of failure a negative value will be returned, and 0 on success.

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 RSA or DSA.

bits: the size of the modulus

flags: unused for now. Must be 0.

This function will generate a random private key. Note that this function must be called on an empty private key.

Returns 0 on success or a negative value on error.

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)

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.

Return value: In case of failure a negative value will be returned, and 0 on success.

gnutls_x509_privkey_get_pk_algorithm

`int gnutls_x509_privkey_get_pk_algorithm (`*gnutls_x509_privkey_t* `key``)` [Function]

key: should contain a gnutls_x509_privkey_t structure

This function will return the public key algorithm of a private key.

Returns a member of the gnutls_pk_algorithm_t enumeration on success, or a negative value on error.

gnutls_x509_privkey_import_dsa_raw

`int gnutls_x509_privkey_import_dsa_raw` (*gnutls_x509_privkey_t* `key`, *const gnutls_datum_t * p*, *const gnutls_datum_t * q*, *const gnutls_datum_t * g*, *const gnutls_datum_t * y*, *const gnutls_datum_t * x*) [Function]

key: The structure to store the parsed key

p: holds the p

q: holds the q

g: holds the g

y: holds the y

x: holds the x

This function will convert the given DSA raw parameters to the native gnutls_x509_privkey_t format. The output will be stored in `key`.

gnutls_x509_privkey_import_pkcs8

`int gnutls_x509_privkey_import_pkcs8` (*gnutls_x509_privkey_t* `key`, *const gnutls_datum_t * data*, *gnutls_x509_crt_fmt_t format*, *const char * password*, *unsigned int flags*) [Function]

key: The structure to store the parsed key

data: The DER or PEM encoded key.

format: One of DER or PEM

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

flags: 0 if encrypted or GNUTLS_PKCS_PLAIN if not encrypted.

This function will convert the given DER or PEM encoded PKCS8 2.0 encrypted key to the native gnutls_x509_privkey_t format. The output will be stored in `key`. Currently only RSA 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 0 on success.

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

This function will convert the given RSA raw parameters to the native gnutls_x509_privkey_t format. The output will be stored in `key`.

gnutls_x509_privkey_import

`int gnutls_x509_privkey_import` (*gnutls_x509_privkey_t* **`key`**, *const gnutls_datum_t * * **`data`**, *gnutls_x509_crt_fmt_t* **`format`**) [Function]

key: The structure to store the parsed key

data: The DER or PEM encoded certificate.

format: One of DER or PEM

This function will convert the given DER or PEM encoded key to the native gnutls_x509_privkey_t format. The output will be stored in `key` .

If the key is PEM encoded it should have a header of "RSA PRIVATE KEY", or "DSA PRIVATE KEY".

Returns 0 on success.

gnutls_x509_privkey_init

`int gnutls_x509_privkey_init` (*gnutls_x509_privkey_t * **`key`**) [Function]

key: The structure to be initialized

This function will initialize an private key structure.

Returns 0 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)

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.

In case of failure a negative value will be returned, and 0 on success.

gnutls_x509_privkey_verify_data

`int gnutls_x509_privkey_verify_data (`*gnutls_x509_privkey_t* **`key`**, *unsigned int* **`flags`**, *const gnutls_datum_t ** **`data`**, *const gnutls_datum_t ** **`signature`**`)` [Function]

key: Holds the key

flags: should be 0 for now

data: holds the data to be signed

signature: contains the signature

This function will verify the given signed data, using the parameters in the private key.

In case of a verification failure 0 is returned, and 1 on success.

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 zero 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`

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 GNUTLS_E_SHORT_MEMORY_BUFFER and updates *sizeof_buf if the provided buffer is not long enough, and 0 on success.

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.

This function will return the specified Object identifier, of the RDN sequence.

Returns GNUTLS_E_SHORT_MEMORY_BUFFER and updates *sizeof_buf if the provided buffer is not long enough, and 0 on success.

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`

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.

If the provided buffer is not long enough, returns GNUTLS_E_SHORT_MEMORY_BUFFER and *sizeof_buf will be updated. On success 0 is returned.

9.3 GnuTLS-extra functions

These functions are only available in the GPL version of the library called `gnutls-extra`. The prototypes for this library lie in '`gnutls/extra.h`'.

gnutls_extra_check_version

`const char * gnutls_extra_check_version` (*const char * req_version*) [Function]

req_version: the version to check

Check that the version of the gnutls-extra library is at minimum the requested one and return the version string; return NULL if the condition is not satisfied. If a NULL is passed to this function, no check is done, but the version string is simply returned.

gnutls_global_init_extra

`int gnutls_global_init_extra` (*void*) [Function]

This function initializes the global state of gnutls-extra library to defaults. Returns zero on success.

Note that `gnutls_global_init()` has to be called before this function. If this function is not called then the gnutls-extra library will not be usable.

9.4 OpenPGP functions

The following functions are to be used for OpenPGP certificate handling. Their prototypes lie in '`gnutls/openpgp.h`'. You need to link with '`libgnutls-extra`' to be able to use these functions (see Section 9.3 [GnuTLS-extra functions], page 183).

gnutls_certificate_set_openpgp_key_file

`int gnutls_certificate_set_openpgp_key_file` (*gnutls_certificate_credentials_t* `res`, *const char * certfile*, *const char * keyfile*) [Function]

res: the destination context to save the data.

certfile: the file that contains the public key.

keyfile: the file that contains the secret key.

This funtion is used to load OpenPGP keys into the GnuTLS credentials structure. It doesn't matter whether the keys are armored or but, but the files should only contain one key which should not be encrypted.

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**)

res: the destination context to save the data.

cert: the datum that contains the public key.

key: the datum that contains the secret key.

This funtion is used to load OpenPGP keys into the GnuTLS credential structure. It doesn't matter whether the keys are armored or but, but the files should only contain one key which should not be encrypted.

gnutls_certificate_set_openpgp_keyring_file

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

c: A certificate credentials structure

file: filename of the keyring.

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

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**)

c: A certificate credentials structure

data: buffer with keyring data.

dlen: length of data buffer.

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.

gnutls_certificate_set_openpgp_keyserver

`int gnutls_certificate_set_openpgp_keyserver` [Function]
(*gnutls_certificate_credentials_t* **res**, *const char ** **keyserver**, *int* **port**)

res: the destination context to save the data.

keyserver: is the key server address

port: is the key server port to connect to

This funtion will set a key server for use with openpgp keys. This key server will only be used if the peer sends a key fingerprint instead of a key in the handshake. Using a key server may delay the handshake process.

gnutls_certificate_set_openpgp_key

`int gnutls_certificate_set_openpgp_key` [Function]
(*gnutls_certificate_credentials_t* **res**, *gnutls_openpgp_key_t* **key**, *gnutls_openpgp_privkey_t* **pkey**)

res: is an `gnutls_certificate_credentials_t` structure.

key: contains an openpgp public key

pkey: is an openpgp private key

This function sets a certificate/private key pair in the gnutls_certificate_credentials_t structure. This function may be called more than once (in case multiple keys/certificates exist for the server).

gnutls_certificate_set_openpgp_trustdb

`int gnutls_certificate_set_openpgp_trustdb` [Function]
(*gnutls_certificate_credentials_t* **res**, *const char * * **trustdb**)

res: the destination context to save the data.

trustdb: is the trustdb filename

This funtion will set a GnuPG trustdb which will be used in key verification functions. Only version 3 trustdb files are supported.

gnutls_openpgp_key_check_hostname

`int gnutls_openpgp_key_check_hostname` (*gnutls_openpgp_key_t* **key**, *const char * * **hostname**) [Function]

key: should contain an gnutls_openpgp_key_t structure

hostname: A null terminated string that contains a DNS name

This function will check if the given key's owner matches the given hostname. This is a basic implementation of the matching described in RFC2818 (HTTPS), which takes into account wildcards.

Returns non zero on success, and zero on failure.

gnutls_openpgp_key_deinit

`void gnutls_openpgp_key_deinit` (*gnutls_openpgp_key_t* **key**) [Function]

key: The structure to be initialized

This function will deinitialize a key structure.

gnutls_openpgp_key_export

`int gnutls_openpgp_key_export` (*gnutls_openpgp_key_t* **key**, *gnutls_openpgp_key_fmt_t* **format**, *void * * **output_data**, *size_t * **output_data_size**) [Function]

key: Holds the key.

format: One of gnutls_openpgp_key_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)

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 0 on success.

gnutls_openpgp_key_get_creation_time

`time_t gnutls_openpgp_key_get_creation_time` [Function]
(*gnutls_openpgp_key_t* `key`)

key: the structure that contains the OpenPGP public key.

Returns the timestamp when the OpenPGP key was created.

gnutls_openpgp_key_get_expiration_time

`time_t gnutls_openpgp_key_get_expiration_time` [Function]
(*gnutls_openpgp_key_t* `key`)

key: the structure that contains the OpenPGP public key.

Returns the time when the OpenPGP key expires. A value of '0' means that the key doesn't expire at all.

gnutls_openpgp_key_get_fingerprint

`int gnutls_openpgp_key_get_fingerprint` (*gnutls_openpgp_key_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.

fprlen: the integer to save the length of the fingerprint.

Returns the fingerprint of the OpenPGP key. Depends on the algorithm, the fingerprint can be 16 or 20 bytes.

gnutls_openpgp_key_get_id

`int gnutls_openpgp_key_get_id` (*gnutls_openpgp_key_t* `key`, *unsigned char* `keyid`[*8*]) [Function]

key: the structure that contains the OpenPGP public key.

Returns the 64-bit keyID of the OpenPGP key.

gnutls_openpgp_key_get_key_usage

`int gnutls_openpgp_key_get_key_usage` (*gnutls_openpgp_key_t* `key`, *unsigned int * key_usage*) [Function]

key: should contain a gnutls_openpgp_key_t structure

key_usage: where the key usage bits will be stored

This function will return certificate's key usage, by checking the key algorithm. The key usage value will ORed values of the: GNUTLS_KEY_DIGITAL_SIGNATURE, GNUTLS_KEY_KEY_ENCIPHERMENT.

A negative value may be returned in case of parsing error.

gnutls_openpgp_key_get_name

`int gnutls_openpgp_key_get_name (`*gnutls_openpgp_key_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

sizeof_buf: holds the size of 'buf'

Extracts the userID from the parsed OpenPGP key.

Returns 0 on success, and GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE if the index of the ID does not exist.

gnutls_openpgp_key_get_pk_algorithm

`int gnutls_openpgp_key_get_pk_algorithm (`*gnutls_openpgp_key_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

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_PKAlgorithm enumeration on success, or a negative value on error.

gnutls_openpgp_key_get_version

`int gnutls_openpgp_key_get_version (`*gnutls_openpgp_key_t* `key`)` [Function]

key: the structure that contains the OpenPGP public key.

Extract the version of the OpenPGP key.

gnutls_openpgp_key_import

`int gnutls_openpgp_key_import (`*gnutls_openpgp_key_t* `key`, *const gnutls_datum_t * * `data`, *gnutls_openpgp_key_fmt_t* `format`)` [Function]

key: The structure to store the parsed key.

data: The RAW or BASE64 encoded key.

format: One of gnutls_openpgp_key_fmt_t elements.

This function will convert the given RAW or Base64 encoded key to the native gnutls_openpgp_key_t format. The output will be stored in 'key'.

Returns 0 on success.

gnutls_openpgp_key_init

`int gnutls_openpgp_key_init (`*gnutls_openpgp_key_t * key*`)` [Function]

key: The structure to be initialized

This function will initialize an OpenPGP key structure.

Returns 0 on success.

gnutls_openpgp_key_to_xml

`int gnutls_openpgp_key_to_xml (`*gnutls_openpgp_key_t key*, *gnutls_datum_t * xmlkey*, *int ext*`)` [Function]

xmlkey: he datum struct to store the XML result.

ext: extension mode (1/0), 1 means include key signatures and key data.

This function will return the all OpenPGP key information encapsulated as a XML string.

gnutls_openpgp_key_verify_ring

`int gnutls_openpgp_key_verify_ring (`*gnutls_openpgp_key_t key*, *gnutls_openpgp_keyring_t keyring*, *unsigned int flags*, *unsigned int * verify*`)` [Function]

key: the structure that holds the key.

keyring: holds the keyring to check against

flags: unused (should be 0)

verify: will hold the certificate verification output.

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

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

GNUTLS_CERT_INVALID: A signature on the key is invalid.

GNUTLS_CERT_REVOKED: The key has been revoked.

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

Returns 0 on success.

gnutls_openpgp_key_verify_self

`int gnutls_openpgp_key_verify_self (`*gnutls_openpgp_key_t key*, *unsigned int flags*, *unsigned int * verify*`)` [Function]

key: the structure that holds the key.

flags: unused (should be 0)

verify: will hold the key verification output.

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

GNUTLS_CERT_INVALID: The self signature on the key is invalid.

Returns 0 on success.

gnutls_openpgp_key_verify_trustdb

`int gnutls_openpgp_key_verify_trustdb (`*gnutls_openpgp_key_t* **`key`**, *gnutls_openpgp_trustdb_t* **`trustdb`**, *unsigned int* **`flags`**, *unsigned int ** **`verify`**`)` [Function]

key: the structure that holds the key.

trustdb: holds the trustdb to check against

flags: unused (should be 0)

verify: will hold the certificate verification output.

Checks if the key is revoked or disabled, in the trustdb. The verification output will be put in `verify` and will be one or more of the gnutls_certificate_status_t enumerated elements bitwise or'd.

GNUTLS_CERT_INVALID: A signature on the key is invalid.

GNUTLS_CERT_REVOKED: The key has been revoked.

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

Returns 0 on success.

gnutls_openpgp_keyring_check_id

`int gnutls_openpgp_keyring_check_id (`*gnutls_openpgp_keyring_t* **`ring`**, *const unsigned char* **`keyid`**`[8]`, *unsigned int* **`flags`**`)` [Function]

ring: holds the keyring to check against

flags: unused (should be 0)

Check if a given key ID exists in the keyring.

Returns 0 on success (if keyid exists) and a negative error code on failure.

gnutls_openpgp_keyring_deinit

`void gnutls_openpgp_keyring_deinit (`*gnutls_openpgp_keyring_t* **`keyring`**`)` [Function]

keyring: The structure to be initialized

This function will deinitialize a CRL structure.

gnutls_openpgp_keyring_import

`int gnutls_openpgp_keyring_import (`*gnutls_openpgp_keyring_t* **`keyring`**, *const gnutls_datum_t ** **`data`**, *gnutls_openpgp_key_fmt_t* **`format`**`)` [Function]

keyring: The structure to store the parsed key.

data: The RAW or BASE64 encoded keyring.

format: One of gnutls_openpgp_keyring_fmt elements.

This function will convert the given RAW or Base64 encoded keyring to the native gnutls_openpgp_keyring_t format. The output will be stored in 'keyring'.

Returns 0 on success.

gnutls_openpgp_keyring_init

`int gnutls_openpgp_keyring_init (`*gnutls_openpgp_keyring_t* * `keyring`) [Function]

keyring: The structure to be initialized

This function will initialize an OpenPGP keyring structure.

Returns 0 on success.

gnutls_openpgp_privkey_deinit

`void gnutls_openpgp_privkey_deinit (`*gnutls_openpgp_privkey_t* `key`) [Function]

key: The structure to be initialized

This function will deinitialize a key structure.

gnutls_openpgp_privkey_get_pk_algorithm

`int gnutls_openpgp_privkey_get_pk_algorithm` [Function]
(*gnutls_openpgp_privkey_t* `key`, *unsigned int* * `bits`)

key: is an OpenPGP key

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

This function will return the public key algorithm of an OpenPGP certificate.

If bits is non null, it should have enough size to hold the parameters size in bits. For RSA the bits returned is the modulus. For DSA the bits returned are of the public exponent.

Returns a member of the GNUTLS_PKAlgorithm enumeration on success, or a negative value on error.

gnutls_openpgp_privkey_import

`int gnutls_openpgp_privkey_import (`*gnutls_openpgp_privkey_t* `key`, [Function]
const gnutls_datum_t * `data`, *gnutls_openpgp_key_fmt_t* `format`, *const char* * `pass`, *unsigned int* `flags`)

key: The structure to store the parsed key.

data: The RAW or BASE64 encoded key.

format: One of gnutls_openpgp_key_fmt_t elements.

pass: Unused for now

flags: should be zero

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 0 on success.

gnutls_openpgp_privkey_init

`int gnutls_openpgp_privkey_init (`*gnutls_openpgp_privkey_t* * `key`) [Function]

key: The structure to be initialized

This function will initialize an OpenPGP key structure.

Returns 0 on success.

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

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.

gnutls_openpgp_trustdb_deinit

`void gnutls_openpgp_trustdb_deinit (`*gnutls_openpgp_trustdb_t* `trustdb`) [Function]

trustdb: The structure to be initialized

This function will deinitialize a CRL structure.

gnutls_openpgp_trustdb_import_file

`int gnutls_openpgp_trustdb_import_file (`*gnutls_openpgp_trustdb_t* `trustdb`, *const char* * `file`) [Function]

trustdb: The structure to store the parsed key.

file: The file that holds the trustdb.

This function will convert the given RAW or Base64 encoded trustdb to the native gnutls_openpgp_trustdb_t format. The output will be stored in 'trustdb'.

Returns 0 on success.

gnutls_openpgp_trustdb_init

`int gnutls_openpgp_trustdb_init (`*gnutls_openpgp_trustdb_t* * `trustdb`) [Function]

trustdb: The structure to be initialized

This function will initialize an OpenPGP trustdb structure.

Returns 0 on success.

9.5 TLS Inner Application (TLS/IA) functions

The following functions are used for TLS Inner Application (TLS/IA). Their prototypes lie in '`gnutls/extra.h`'. You need to link with '`libgnutls-extra`' to be able to use these functions (see Section 9.3 [GnuTLS-extra functions], page 183).

The typical control flow in an TLS/IA client (that would not require an Application Phase for resumed sessions) would be similar to the following:

```
int client_avp (gnuls_session_t *session, void *ptr,
                const char *last, size_t lastlen,
char **new, size_t *newlen)
{
...
}
...
int main ()
{
  gnutls_ia_client_credentials_t iacred;
...
  gnutls_init (&session, GNUTLS_CLIENT);
...
  /* Enable TLS/IA. */
  gnutls_ia_allocate_client_credentials(&iacred);
  gnutls_ia_set_client_avp_function(iacred, client_avp);
  gnutls_credentials_set (session, GNUTLS_CRD_IA, iacred);
...
  ret = gnutls_handshake (session);
  // Error handling...
...
  if (gnutls_ia_handshake_p (session))
    {
      ret = gnutls_ia_handshake (session);
      // Error handling...
...
```

See below for detailed descriptions of all the functions used above.

The function `client_avp` would have to be implemented by your application. The function is responsible for handling the AVP data. See `gnutls_ia_set_client_avp_function` below for more information on how that function should be implemented.

The control flow in a typical server is similar to the above, use `gnutls_ia_server_credentials_t` instead of `gnutls_ia_client_credentials_t`, and replace the call to the client functions with the corresponding server functions.

gnutls_ia_allocate_client_credentials

`int gnutls_ia_allocate_client_credentials` [Function]
(*gnutls_ia_client_credentials_t* * *sc*)

sc: is a pointer to an `gnutls_ia_server_credentials_t` structure.

This structure is complex enough to manipulate directly thus this helper function is provided in order to allocate it.

Adding this credential to a session will enable TLS/IA, and will require an Application Phase after the TLS handshake (if the server support TLS/IA). Use `gnutls_ia_require_inner_phase()` to toggle the TLS/IA mode.

Returns 0 on success.

gnutls_ia_allocate_server_credentials

`int gnutls_ia_allocate_server_credentials` [Function]
(*gnutls_ia_server_credentials_t* * **sc**)

sc: is a pointer to an `gnutls_ia_server_credentials_t` structure.

This structure is complex enough to manipulate directly thus this helper function is provided in order to allocate it.

Adding this credential to a session will enable TLS/IA, and will require an Application Phase after the TLS handshake (if the client support TLS/IA). Use `gnutls_ia_require_inner_phase()` to toggle the TLS/IA mode.

Returns 0 on success.

gnutls_ia_enable

`void gnutls_ia_enable` (*gnutls_session_t* **session**, *int* **allow_skip_on_resume**) [Function]

session: is a `gnutls_session_t` structure.

allow_skip_on_resume: non-zero if local party allows to skip the TLS/IA application phases for a resumed session.

Specify whether we must advertise support for the TLS/IA extension during the handshake.

At the client side, we always advertise TLS/IA if gnutls_ia_enable was called before the handshake; at the server side, we also require that the client has advertised that it wants to run TLS/IA before including the advertisement, as required by the protocol.

Similarly, at the client side we always advertise that we allow TLS/IA to be skipped for resumed sessions if `allow_skip_on_resume` is non-zero; at the server side, we also require that the session is indeed resumable and that the client has also advertised that it allows TLS/IA to be skipped for resumed sessions.

After the TLS handshake, call `gnutls_ia_handshake_p()` to find out whether both parties agreed to do a TLS/IA handshake, before calling `gnutls_ia_handshake()` or one of the lower level gnutls_ia_* functions. Specify whether we must advertise support for the TLS/IA extension during the handshake.

At the client side, we always advertise TLS/IA if gnutls_ia_enable was called before the handshake; at the server side, we also require that the client has advertised that it wants to run TLS/IA before including the advertisement, as required by the protocol.

Similarly, at the client side we always advertise that we allow TLS/IA to be skipped for resumed sessions if `allow_skip_on_resume` is non-zero; at the server side, we also

require that the session is indeed resumable and that the client has also advertised that it allows TLS/IA to be skipped for resumed sessions.

After the TLS handshake, call `gnutls_ia_handshake_p()` to find out whether both parties agreed to do a TLS/IA handshake, before calling `gnutls_ia_handshake()` or one of the lower level gnutls_ia_* functions.

gnutls_ia_endphase_send

`int gnutls_ia_endphase_send (`*gnutls_session_t* **`session`**`, int` **`final_p`**`)` [Function]

session: is a `gnutls_session_t` structure.

final_p: Set iff this should signal the final phase.

Send a TLS/IA end phase message.

In the client, this should only be used to acknowledge an end phase message sent by the server.

In the server, this can be called instead of `gnutls_ia_send()` if the server wishes to end an application phase.

Return value: Return 0 on success, or an error code.

gnutls_ia_extract_inner_secret

`void gnutls_ia_extract_inner_secret (`*gnutls_session_t* **`session`**`, char * buffer`**`)` [Function]

session: is a `gnutls_session_t` structure.

buffer: pre-allocated buffer to hold 48 bytes of inner secret.

Copy the 48 bytes large inner secret into the specified buffer

This function is typically used after the TLS/IA handshake has concluded. The TLS/IA inner secret can be used as input to a PRF to derive session keys. Do not use the inner secret directly as a session key, because for a resumed session that does not include an application phase, the inner secret will be identical to the inner secret in the original session. It is important to include, for example, the client and server randomness when deriving a sesssion key from the inner secret.

gnutls_ia_free_client_credentials

`void gnutls_ia_free_client_credentials (`*gnutls_ia_client_credentials_t* **`sc`**`)` [Function]

sc: is an `gnutls_ia_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_ia_free_server_credentials

`void gnutls_ia_free_server_credentials (`*gnutls_ia_server_credentials_t* **`sc`**`)` [Function]

sc: is an `gnutls_ia_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_ia_generate_challenge

`int gnutls_ia_generate_challenge (`*gnutls_session_t* **`session`**, *size_t* **`buffer_size`**, *char ** **`buffer`**`)` [Function]

session: is a `gnutls_session_t` structure.

buffer_size: size of output buffer.

buffer: pre-allocated buffer to contain `buffer_size` bytes of output.

Generate an application challenge that the client cannot control or predict, based on the TLS/IA inner secret.

Return value: Returns 0 on success, or an negative error code.

gnutls_ia_get_client_avp_ptr

`void * gnutls_ia_get_client_avp_ptr (`*gnutls_ia_client_credentials_t* **`cred`**`)` [Function]

cred: is a `gnutls_ia_client_credentials_t` structure.

Returns the pointer that will be provided to the TLS/IA callback function as the first argument.

gnutls_ia_get_server_avp_ptr

`void * gnutls_ia_get_server_avp_ptr (`*gnutls_ia_server_credentials_t* **`cred`**`)` [Function]

cred: is a `gnutls_ia_client_credentials_t` structure.

Returns the pointer that will be provided to the TLS/IA callback function as the first argument.

gnutls_ia_handshake_p

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

session: is a `gnutls_session_t` structure.

Predicate to be used after `gnutls_handshake()` to decide whether to invoke `gnutls_ia_handshake()`. Usable by both clients and servers.

Return value: non-zero if TLS/IA handshake is expected, zero otherwise.

gnutls_ia_handshake

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

session: is a `gnutls_session_t` structure.

Perform a TLS/IA handshake. This should be called after `gnutls_handshake()` iff `gnutls_ia_handshake_p()`.

Return 0 on success, or an error code.

gnutls_ia_permute_inner_secret

`int gnutls_ia_permute_inner_secret (`*gnutls_session_t* **`session`**, *size_t* **`session_keys_size`**, *const char ** **`session_keys`**`)` [Function]

session: is a `gnutls_session_t` structure.

session_keys_size: Size of generated session keys (0 if none).

session_keys: Generated session keys, used to permute inner secret (NULL if none).

Permute the inner secret using the generated session keys.

This can be called in the TLS/IA AVP callback to mix any generated session keys with the TLS/IA inner secret.

Return value: Return zero on success, or a negative error code.

gnutls_ia_recv

`ssize_t gnutls_ia_recv (`*gnutls_session_t* **`session`**, *char ** **`data`**, *size_t* **`sizeofdata`**`)` [Function]

session: is a `gnutls_session_t` structure.

data: the buffer that the data will be read into, must hold >= 12 bytes.

sizeofdata: the number of requested bytes, must be >= 12.

Receive TLS/IA data. This function has the similar semantics with `recv()`. The only difference is that is accepts a GNUTLS session, and uses different error codes.

If the server attempt to finish an application phase, this function will return `GNUTLS_E_WARNING_IA_IPHF_RECEIVED` or `GNUTLS_E_WARNING_IA_FPHF_RECEIVED`. The caller should then invoke `gnutls_ia_verify_endphase()`, and if it runs the client side, also send an endphase message of its own using gnutls_ia_endphase_send.

If EINTR is returned by the internal push function (the default is `code{recv()}`) 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.

Returns the number of bytes received. A negative error code is returned in case of an error. The `GNUTLS_E_WARNING_IA_IPHF_RECEIVED` and `GNUTLS_E_WARNING_IA_FPHF_RECEIVED` errors are returned when an application phase finished message has been sent by the server.

gnutls_ia_send

`ssize_t gnutls_ia_send (`*gnutls_session_t* **`session`**, *const char ** **`data`**, *size_t* **`sizeofdata`**`)` [Function]

session: is a `gnutls_session_t` structure.

data: contains the data to send

sizeofdata: is the length of the data

Send TLS/IA application payload data. This function has the similar semantics with `send()`. The only difference is that is accepts a GNUTLS session, and uses different error codes.

The TLS/IA protocol is synchronous, so you cannot send more than one packet at a time. The client always send the first packet.

To finish an application phase in the server, use `gnutls_ia_endphase_send()`. The client cannot end an application phase unilaterally; rather, a client is required to respond with an endphase of its own if gnutls_ia_recv indicates that the server has sent one.

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.

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

gnutls_ia_set_client_avp_function

`void gnutls_ia_set_client_avp_function` [Function]
(*gnutls_ia_client_credentials_t* ***cred***, *gnutls_ia_avp_func* ***avp_func***)

cred: is a `gnutls_ia_client_credentials_t` structure.

avp_func: is the callback function

Set the TLS/IA AVP callback handler used for the session.

The AVP callback is called to process AVPs received from the server, and to get a new AVP to send to the server.

The callback's function form is: int (*avp_func) (gnutls_session_t session, void *ptr, const char *last, size_t lastlen, char **next, size_t *nextlen);

The `session` parameter is the `gnutls_session_t` structure corresponding to the current session. The `ptr` parameter is the application hook pointer, set through `gnutls_ia_set_client_avp_ptr()`. The AVP received from the server is present in `last` of `lastlen` size, which will be `NULL` on the first invocation. The newly allocated output AVP to send to the server should be placed in *`next` of *`nextlen` size.

The callback may invoke `gnutls_ia_permute_inner_secret()` to mix any generated session keys with the TLS/IA inner secret.

Return 0 (`GNUTLS_IA_APPLICATION_PAYLOAD`) on success, or a negative error code to abort the TLS/IA handshake.

Note that the callback must use allocate the `next` parameter using `gnutls_malloc()`, because it is released via `gnutls_free()` by the TLS/IA handshake function.

gnutls_ia_set_client_avp_ptr

`void gnutls_ia_set_client_avp_ptr (`*gnutls_ia_client_credentials_t* ***cred***, *void * **ptr***`)` [Function]

cred: is a `gnutls_ia_client_credentials_t` structure.

ptr: is the pointer

Sets the pointer that will be provided to the TLS/IA callback function as the first argument.

gnutls_ia_set_server_avp_function

`void gnutls_ia_set_server_avp_function` [Function]
(*gnutls_ia_server_credentials_t* **cred**, *gnutls_ia_avp_func* **avp_func**)

cred: is a `gnutls_ia_server_credentials_t` structure.

Set the TLS/IA AVP callback handler used for the session.

The callback's function form is: int (*avp_func) (gnutls_session_t session, void *ptr, const char *last, size_t lastlen, char **next, size_t *nextlen);

The `session` parameter is the `gnutls_session_t` structure corresponding to the current session. The `ptr` parameter is the application hook pointer, set through `gnutls_ia_set_server_avp_ptr()`. The AVP received from the client is present in `last` of `lastlen` size. The newly allocated output AVP to send to the client should be placed in *`next` of *`nextlen` size.

The AVP callback is called to process incoming AVPs from the client, and to get a new AVP to send to the client. It can also be used to instruct the TLS/IA handshake to do go into the Intermediate or Final phases. It return a negative error code, or an `gnutls_ia_apptype_t` message type.

The callback may invoke `gnutls_ia_permute_inner_secret()` to mix any generated session keys with the TLS/IA inner secret.

Specifically, return `GNUTLS_IA_APPLICATION_PAYLOAD` (0) to send another AVP to the client, return `GNUTLS_IA_INTERMEDIATE_PHASE_FINISHED` (1) to indicate that an IntermediatePhaseFinished message should be sent, and return `GNUTLS_IA_FINAL_PHASE_FINISHED` (2) to indicate that an FinalPhaseFinished message should be sent. In the last two cases, the contents of the `next` and `nextlen` parameter is not used.

Note that the callback must use allocate the `next` parameter using `gnutls_malloc()`, because it is released via `gnutls_free()` by the TLS/IA handshake function.

gnutls_ia_set_server_avp_ptr

`void gnutls_ia_set_server_avp_ptr` (*gnutls_ia_server_credentials_t* **cred**, *void * **ptr***) [Function]

cred: is a `gnutls_ia_client_credentials_t` structure.

ptr: is the pointer

Sets the pointer that will be provided to the TLS/IA callback function as the first argument.

gnutls_ia_verify_endphase

`int gnutls_ia_verify_endphase` (*gnutls_session_t* **session**, *const char * **checksum***) [Function]

session: is a `gnutls_session_t` structure.

checksum: 12-byte checksum data, received from `gnutls_ia_recv()`.

Verify TLS/IA end phase checksum data. If verification fails, the `GNUTLS_A_INNER_APPLICATION_VERIFICATION` alert is sent to the other sie.

This function is called when `gnutls_ia_recv()` return `GNUTLS_E_WARNING_IA_IPHF_RECEIVED` or `GNUTLS_E_WARNING_IA_FPHF_RECEIVED`.

Return value: Return 0 on successful verification, or an error code. If the checksum verification of the end phase message fails, `GNUTLS_E_IA_VERIFY_FAILED` is returned.

9.6 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:`
: Function was interrupted.

`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_BASE64_DECODING_ERROR:`
: Base64 decoding error.

`GNUTLS_E_BASE64_ENCODING_ERROR:`
: Base64 encoding 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_COMPRESSION_FAILED:`
: Compression of the TLS record packet has failed.

`GNUTLS_E_CONSTRAINT_ERROR:`
: Some constraint limits were reached.

`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_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_HASH_FAILED:`
: Hashing has failed.

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

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

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

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

`GNUTLS_E_INIT_LIBEXTRA:`
: The initialization of GnuTLS-extra has failed.

`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_LIBRARY_VERSION_MISMATCH:`
: The GnuTLS library version does not match the GnuTLS-extra library version.

`GNUTLS_E_LZO_INIT_FAILED:`
: The initialization of LZO has failed.

`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_TRUSTDB_VERSION_UNSUPPORTED:`
: The specified GnuPG TrustDB version is not supported. TrustDB v4 is supported.

`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_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_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_TOO_MANY_EMPTY_PACKETS:`
Too many empty record 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_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_UNSUPPORTED_CERTIFICATE_TYPE:`
The certificate type 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_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.

10 Certificate to XML convertion functions

This appendix contains some example output of the XML convertion functions:

- [gnutls_x509_crt_to_xml], page 175
- [gnutls_openpgp_key_to_xml], page 188

10.1 An X.509 certificate

```
<?xml version="1.0" encoding="UTF-8"?>

<gnutls:x509:certificate version="1.1">
 <certificate type="SEQUENCE">
  <tbsCertificate type="SEQUENCE">
    <version type="INTEGER" encoding="HEX">02</version>
    <serialNumber type="INTEGER" encoding="HEX">01</serialNumber>
    <signature type="SEQUENCE">
      <algorithm type="OBJECT ID">1.2.840.113549.1.1.4</algorithm>
      <parameters type="ANY">
        <md5WithRSAEncryption encoding="HEX">0500</md5WithRSAEncryption>
      </parameters>
    </signature>
    <issuer type="CHOICE">
      <rdnSequence type="SEQUENCE OF">
        <unnamed1 type="SET OF">
          <unnamed1 type="SEQUENCE">
            <type type="OBJECT ID">2.5.4.6</type>
            <value type="ANY">
              <X520countryName>GR</X520countryName>
            </value>
          </unnamed1>
        </unnamed1>
        <unnamed2 type="SET OF">
          <unnamed1 type="SEQUENCE">
            <type type="OBJECT ID">2.5.4.8</type>
            <value type="ANY">
              <X520StateOrProvinceName>Attiki</X520StateOrProvinceName>
            </value>
          </unnamed1>
        </unnamed2>
        <unnamed3 type="SET OF">
          <unnamed1 type="SEQUENCE">
            <type type="OBJECT ID">2.5.4.7</type>
            <value type="ANY">
              <X520LocalityName>Athina</X520LocalityName>
            </value>
          </unnamed1>
        </unnamed3>
        <unnamed4 type="SET OF">
          <unnamed1 type="SEQUENCE">
            <type type="OBJECT ID">2.5.4.10</type>
            <value type="ANY">
              <X520OrganizationName>GNUTLS</X520OrganizationName>
            </value>
          </unnamed1>
        </unnamed4>
        <unnamed5 type="SET OF">
          <unnamed1 type="SEQUENCE">
```

```
        <type type="OBJECT ID">2.5.4.11</type>
        <value type="ANY">
          <X520OrganizationalUnitName>GNUTLS dev.</X520OrganizationalUnitName>
        </value>
      </unnamed1>
    </unnamed5>
    <unnamed6 type="SET OF">
      <unnamed1 type="SEQUENCE">
        <type type="OBJECT ID">2.5.4.3</type>
        <value type="ANY">
          <X520CommonName>GNUTLS TEST CA</X520CommonName>
        </value>
      </unnamed1>
    </unnamed6>
    <unnamed7 type="SET OF">
      <unnamed1 type="SEQUENCE">
        <type type="OBJECT ID">1.2.840.113549.1.9.1</type>
        <value type="ANY">
          <Pkcs9email>gnutls-dev@gnupg.org</Pkcs9email>
        </value>
      </unnamed1>
    </unnamed7>
  </rdnSequence>
</issuer>
<validity type="SEQUENCE">
  <notBefore type="CHOICE">
    <utcTime type="TIME">010707101845Z</utcTime>
  </notBefore>
  <notAfter type="CHOICE">
    <utcTime type="TIME">020707101845Z</utcTime>
  </notAfter>
</validity>
<subject type="CHOICE">
  <rdnSequence type="SEQUENCE OF">
    <unnamed1 type="SET OF">
      <unnamed1 type="SEQUENCE">
        <type type="OBJECT ID">2.5.4.6</type>
        <value type="ANY">
          <X520countryName>GR</X520countryName>
        </value>
      </unnamed1>
    </unnamed1>
    <unnamed2 type="SET OF">
      <unnamed1 type="SEQUENCE">
        <type type="OBJECT ID">2.5.4.8</type>
        <value type="ANY">
          <X520StateOrProvinceName>Attiki</X520StateOrProvinceName>
        </value>
      </unnamed1>
    </unnamed2>
    <unnamed3 type="SET OF">
      <unnamed1 type="SEQUENCE">
        <type type="OBJECT ID">2.5.4.7</type>
        <value type="ANY">
          <X520LocalityName>Athina</X520LocalityName>
        </value>
      </unnamed1>
    </unnamed3>
```

```
    <unnamed4 type="SET OF">
      <unnamed1 type="SEQUENCE">
        <type type="OBJECT ID">2.5.4.10</type>
        <value type="ANY">
          <X520OrganizationName>GNUTLS</X520OrganizationName>
        </value>
      </unnamed1>
    </unnamed4>
    <unnamed5 type="SET OF">
      <unnamed1 type="SEQUENCE">
        <type type="OBJECT ID">2.5.4.11</type>
        <value type="ANY">
          <X520OrganizationalUnitName>GNUTLS dev.</X520OrganizationalUnitName>
        </value>
      </unnamed1>
    </unnamed5>
    <unnamed6 type="SET OF">
      <unnamed1 type="SEQUENCE">
        <type type="OBJECT ID">2.5.4.3</type>
        <value type="ANY">
          <X520CommonName>localhost</X520CommonName>
        </value>
      </unnamed1>
    </unnamed6>
    <unnamed7 type="SET OF">
      <unnamed1 type="SEQUENCE">
        <type type="OBJECT ID">1.2.840.113549.1.9.1</type>
        <value type="ANY">
          <Pkcs9email>root@localhost</Pkcs9email>
        </value>
      </unnamed1>
    </unnamed7>
  </rdnSequence>
</subject>
<subjectPublicKeyInfo type="SEQUENCE">
  <algorithm type="SEQUENCE">
    <algorithm type="OBJECT ID">1.2.840.113549.1.1.1</algorithm>
    <parameters type="ANY">
      <rsaEncryption encoding="HEX">0500</rsaEncryption>
    </parameters>
  </algorithm>
  <subjectPublicKey type="BIT STRING" encoding="HEX" length="1120">
  30818902818100D00B49EBB226D951F5CC57072199DDF287683D2DA1A0E
  FCC96BFF73164777C78C3991E92EDA66584E7B97BAB4BE68D595D225557
  E01E7E57B5C35C04B491948C5C427AD588D8C6989764996D6D44E17B65C
  CFC86F3B4842DE559B730C1DE3AEF1CE1A328AFF8A357EBA911E1F7E8FC
  1598E21E4BF721748C587F50CF46157D950203010001</subjectPublicKey>
</subjectPublicKeyInfo>
<extensions type="SEQUENCE OF">
  <unnamed1 type="SEQUENCE">
    <extnID type="OBJECT ID">2.5.29.35</extnID>
    <critical type="BOOLEAN">FALSE</critical>
    <extnValue type="SEQUENCE">
      <keyIdentifier type="OCTET STRING" encoding="HEX">
      EFEE94ABC8CA577F5313DB76DC1A950093BAF3C9</keyIdentifier>
    </extnValue>
  </unnamed1>
  <unnamed2 type="SEQUENCE">
```

```
        <extnID type="OBJECT ID">2.5.29.37</extnID>
        <critical type="BOOLEAN">FALSE</critical>
        <extnValue type="SEQUENCE OF">
          <unnamed1 type="OBJECT ID">1.3.6.1.5.5.7.3.1</unnamed1>
          <unnamed2 type="OBJECT ID">1.3.6.1.5.5.7.3.2</unnamed2>
          <unnamed3 type="OBJECT ID">1.3.6.1.4.1.311.10.3.3</unnamed3>
          <unnamed4 type="OBJECT ID">2.16.840.1.113730.4.1</unnamed4>
        </extnValue>
      </unnamed2>
      <unnamed3 type="SEQUENCE">
        <extnID type="OBJECT ID">2.5.29.19</extnID>
        <critical type="BOOLEAN">TRUE</critical>
        <extnValue type="SEQUENCE">
          <cA type="BOOLEAN">FALSE</cA>
        </extnValue>
      </unnamed3>
    </extensions>
  </tbsCertificate>
  <signatureAlgorithm type="SEQUENCE">
    <algorithm type="OBJECT ID">1.2.840.113549.1.1.4</algorithm>
    <parameters type="ANY">
      <md5WithRSAEncryption encoding="HEX">0500</md5WithRSAEncryption>
    </parameters>
  </signatureAlgorithm>
  <signature type="BIT STRING" encoding="HEX" length="1024">
  B73945273AF2A395EC54BF5DC669D953885A9D811A3B92909D24792D36A44EC
  27E1C463AF8738BEFD29B311CCE8C6D9661BEC30911DAABB39B8813382B32D2
  E259581EBCD26C495C083984763966FF35D1DEFE432891E610C85072578DA74
  23244A8F5997B41A1F44E61F4F22C94375775055A5E72F25D5E4557467A91BD
  4251</signature>
 </certificate>
</gnutls:x509:certificate>
```

10.2 An OpenPGP key

```
<?xml version="1.0"?>

<gnutls:openpgp:key version="1.0">
 <OPENPGPKEY>
  <MAINKEY>
    <KEYID>BD572CDCCCC07C3</KEYID>
    <FINGERPRINT>BE615E88D6CFF27225B8A2E7BD572CDCCCC07C35</FINGERPRINT>
    <PKALGO>DSA</PKALGO>
    <KEYLEN>1024</KEYLEN>
    <CREATED>1011533164</CREATED>
    <REVOKED>0</REVOKED>
    <KEY ENCODING="HEX"/>
    <DSA-P>0400E72E76B62EEFA9A3BD5940932924180500C02D7029D6CA2066E
    FC34C86038627C643EB1A652A7AF1D37CF46FC505AC1E0C699B37895B4BCB
    3E53541FFDA4766D6168C2B8AAFD6AB22466D06D18034D5DAC698E6993BA5
    B350FF822E1CD8702A75114E8B73A6B09CB3B93CE44DBB516C9BB5F95BB66
    6188602A0A1447236C0658F</DSA-P>
    <DSA-Q>00A08F5B5E78D85F792CC2072F9474645726FB4D9373</DSA-Q>
    <DSA-G>03FE3578D689D6606E9118E9F9A7042B963CF23F3D8F1377A273C0
    F0974DBF44B3CABCBE14DD64412555863E39A9C627662D77AC36662AE4497
    92C3262D3F12E9832A7565309D67BA0AE4DF25F5EDA0937056AD5BE89F406
    9EBD7EC76CE432441DF5D52FFFD06D39E5F61E36947B698A77CB62AB81E4A
    4122BF9050671D9946C865E</DSA-G>
```

```
      <DSA-Y>0400D061437A964DDE318818C2B24DE008E60096B60DB8A684B85A
      838D119FC930311889AD57A3B927F448F84EB253C623EDA73B42FF78BCE63
      A6A531D75A64CE8540513808E9F5B10CE075D3417B801164918B131D3544C
      8765A8ECB9971F61A09FC73D509806106B5977D211CB0E1D04D0ED96BCE89
      BAE8F73D800B052139CBF8D</DSA-Y>
    </MAINKEY>
    <USERID>
      <NAME>OpenCDK test key (Only intended for test purposes!)</NAME>
      <EMAIL>opencdk@foo-bar.org</EMAIL>
      <PRIMARY>0</PRIMARY>
      <REVOKED>0</REVOKED>
    </USERID>
    <SIGNATURE>
      <VERSION>4</VERSION>
      <SIGCLASS>19</SIGCLASS>
      <EXPIRED>0</EXPIRED>
      <PKALGO>DSA</PKALGO>
      <MDALGO>SHA1</MDALGO>
      <CREATED>1011533164</CREATED>
      <KEYID>BD572CDCCCC07C3</KEYID>
    </SIGNATURE>
    <SUBKEY>
      <KEYID>FCB0CF3A5261E06</KEYID>
      <FINGERPRINT>297B48ACC09C0FF683CA1ED1FCB0CF3A5261E067</FINGERPRINT>
      <PKALGO>ELG</PKALGO>
      <KEYLEN>1024</KEYLEN>
      <CREATED>1011533167</CREATED>
      <REVOKED>0</REVOKED>
      <KEY ENCODING="HEX"/>
      <ELG-P>0400E20156526069D067D24F4D71E6D38658E08BE3BF246C1ADCE0
      8DB69CD8D459C1ED335738410798755AFDB79F1797CF022E70C7960F12CA6
      896D27CFD24A11CD316DDE1FBCC1EA615C5C31FEC656E467078C875FC509B
      1ECB99C8B56C2D875C50E2018B5B0FA378606EB6425A2533830F55FD21D64
      9015615D49A1D09E9510F5F</ELG-P>
      <ELG-G>000305</ELG-G>
      <ELG-Y>0400D0BDADE40432758675C87D0730C360981467BAE1BEB6CC105A
      3C1F366BFDBEA12E378456513238B8AD414E52A2A9661D1DF1DB6BB5F33F6
      906166107556C813224330B30932DB7C8CC8225672D7AE24AF2469750E539
      B661EA6475D2E03CD8D3838DC4A8AC4AFD213536FE3E96EC9D0AEA65164B5
      76E01B37A8DCA89F2B257D0</ELG-Y>
    </SUBKEY>
    <SIGNATURE>
      <VERSION>4</VERSION>
      <SIGCLASS>24</SIGCLASS>
      <EXPIRED>0</EXPIRED>
      <PKALGO>DSA</PKALGO>
      <MDALGO>SHA1</MDALGO>
      <CREATED>1011533167</CREATED>
      <KEYID>BD572CDCCCC07C3</KEYID>
    </SIGNATURE>
  </OPENPGPKEY>
</gnutls:openpgp:key>
```

11 All the supported ciphersuites in GnuTLS

TLS_RSA_NULL_MD5	0x00 0x01	RFC 2246
TLS_ANON_DH_3DES_EDE_CBC_SHA	0x00 0x1B	RFC 2246
TLS_ANON_DH_ARCFOUR_MD5	0x00 0x18	RFC 2246
TLS_ANON_DH_AES_128_CBC_SHA	0x00 0x34	RFC 2246
TLS_ANON_DH_AES_256_CBC_SHA	0x00 0x3A	RFC 2246
TLS_RSA_ARCFOUR_SHA	0x00 0x05	RFC 2246
TLS_RSA_ARCFOUR_MD5	0x00 0x04	RFC 2246
TLS_RSA_3DES_EDE_CBC_SHA	0x00 0x0A	RFC 2246
TLS_RSA_EXPORT_ARCFOUR_40_MD5	0x00 0x03	RFC 2246
TLS_DHE_DSS_3DES_EDE_CBC_SHA	0x00 0x13	RFC 2246
TLS_DHE_RSA_3DES_EDE_CBC_SHA	0x00 0x16	RFC 2246
TLS_RSA_AES_128_CBC_SHA	0x00 0x2F	RFC 3268
TLS_RSA_AES_256_CBC_SHA	0x00 0x35	RFC 3268
TLS_DHE_DSS_AES_256_CBC_SHA	0x00 0x38	RFC 3268
TLS_DHE_DSS_AES_128_CBC_SHA	0x00 0x32	RFC 3268
TLS_DHE_RSA_AES_256_CBC_SHA	0x00 0x39	RFC 3268
TLS_DHE_RSA_AES_128_CBC_SHA	0x00 0x33	RFC 3268
TLS_SRP_SHA_3DES_EDE_CBC_SHA	0x00 0x50	draft-ietf-tls-srp
TLS_SRP_SHA_AES_128_CBC_SHA	0x00 0x53	draft-ietf-tls-srp
TLS_SRP_SHA_AES_256_CBC_SHA	0x00 0x56	draft-ietf-tls-srp
TLS_SRP_SHA_RSA_3DES_EDE_CBC_SHA	0x00 0x51	draft-ietf-tls-srp
TLS_SRP_SHA_DSS_3DES_EDE_CBC_SHA	0x00 0x52	draft-ietf-tls-srp

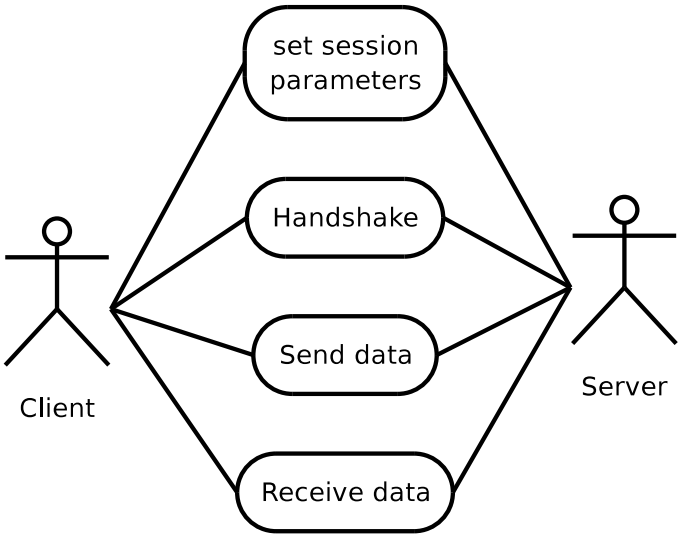
TLS_SRP_SHA_RSA_AES_128_CBC_SHA	0x00 0x54	draft-ietf-tls-srp
TLS_SRP_SHA_DSS_AES_128_CBC_SHA	0x00 0x55	draft-ietf-tls-srp
TLS_SRP_SHA_RSA_AES_256_CBC_SHA	0x00 0x57	draft-ietf-tls-srp
TLS_SRP_SHA_DSS_AES_256_CBC_SHA	0x00 0x58	draft-ietf-tls-srp
TLS_DHE_DSS_ARCFOUR_SHA	0x00 0x66	draft-ietf-tls-56-bit-ciphersuites
TLS_PSK_ARCFOUR_SHA	0x00 0x8A	draft-ietf-tls-psk
TLS_PSK_3DES_EDE_CBC_SHA	0x00 0x8B	draft-ietf-tls-psk
TLS_PSK_AES_128_CBC_SHA	0x00 0x8C	draft-ietf-tls-psk
TLS_PSK_AES_256_CBC_SHA	0x00 0x8D	draft-ietf-tls-psk

12 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.

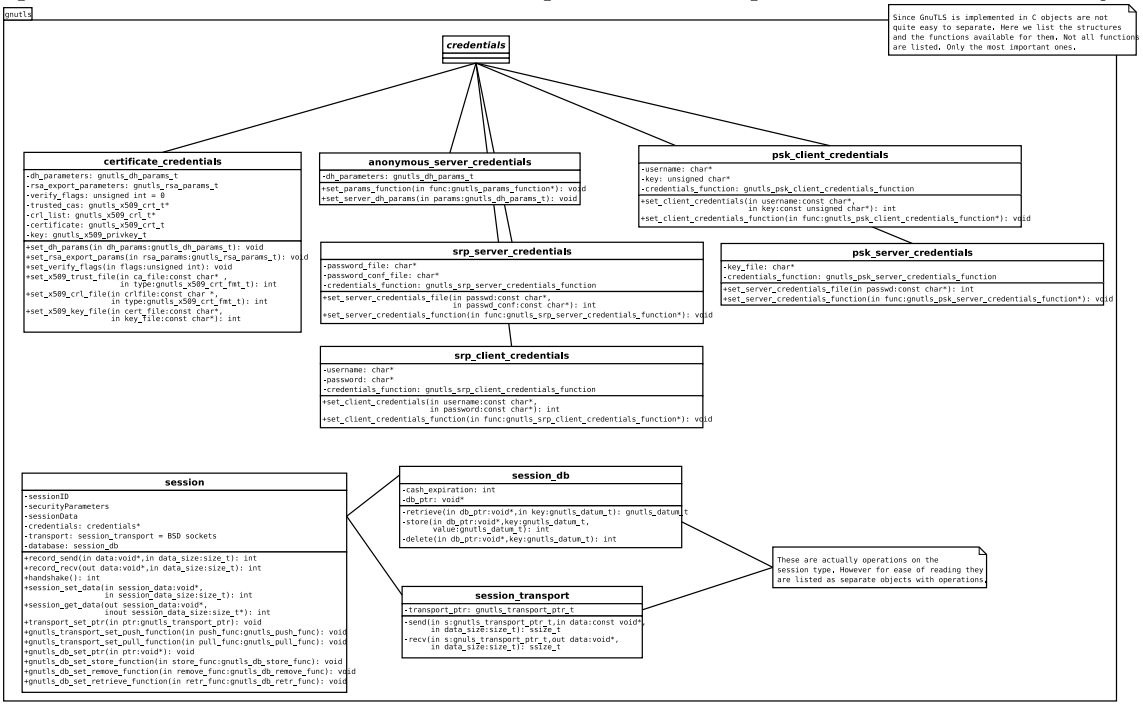
12.1 The TLS protocol

The main needs for the TLS protocol to be used are shown in the image below.



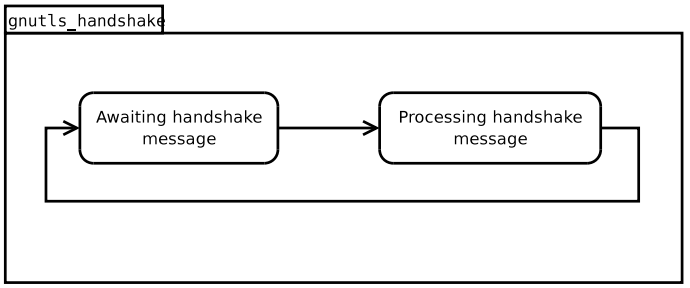
This is being accomplished by the following object diagram. Note that since GnuTLS is being developed in C object are just structures with attributes. The

operations listed are functions that require the first parameter to be that object.



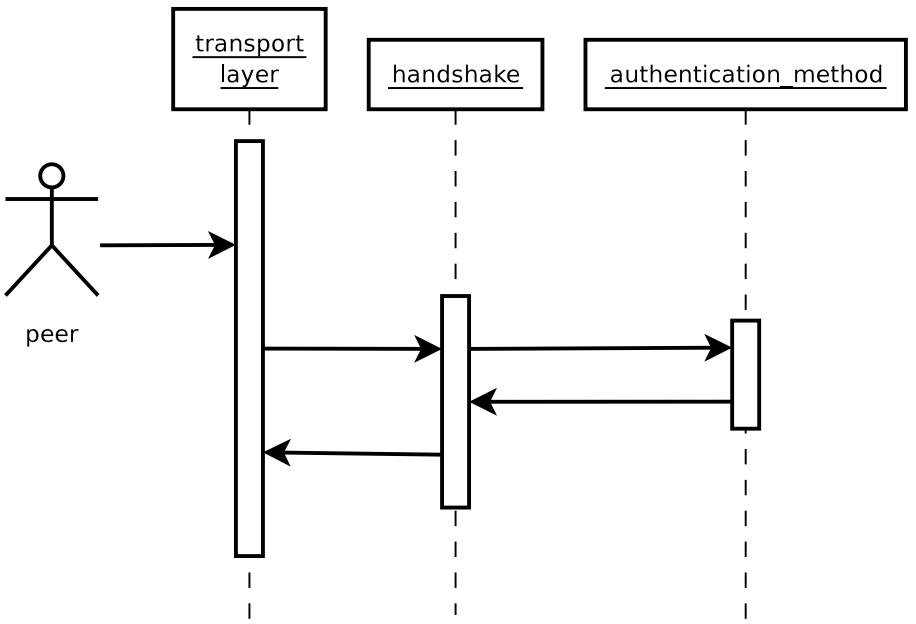
12.2 TLS Handshake protocol

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 the following figure.



Also the way the input is processed varies per ciphersuite. Several implementations of the internal handlers are available and [gnutls_handshake], page 118 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.



12.3 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 only to implement the functions as seen in the figure below.

mod_auth_st

```
generate_server_certificate(in session:gnutls_session_t,out data:opaque**): int
generate_client_certificate(in session:gnutls_session_t,out data:opaque**): int
generate_server_kx(in session:gnutls_session_t,out data:opaque**): int
generate_client_kx(in session:gnutls_session_t,out data:opaque**): int
generate_client_cert_vrfy(in session:gnutls_session_t,out data:opaque**): int
generate_server_certificate_request(in session:gnutls_session_t,
                                    out data:opaque**): int
process_server_certificate(in session:gnutls_session_t,in data:opaque*,
                           in data_size:size_t): int
process_client_certificate(in session:gnutls_session_t,in data:opaque*,
                           in data_size:size_t): int
process_server_kx(in session:gnutls_session_t,in data:opaque*,
                  in data_size:size_t): int
process_client_kx(in session:gnutls_session_t,in data:opaque*,
                  in data_size:size_t): int
process_client_cert_vrfy(in session:gnutls_session_t,in data:opaque*,
                         in data_size:size_t): int
process_server_certificate_request(in session:gnutls_session_t,
                                   in data:opaque*,in data_size:size_t): int
```

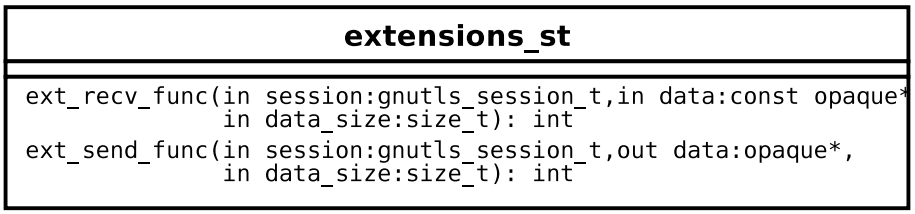
The functions that need to be implemented are the ones responsible for interpreting the handshake protocol messages. It is common for such functions to read data from one or

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.

12.4 TLS Extension handling

As with authentication methods, the TLS extensions handlers can be implemented using the following interface.



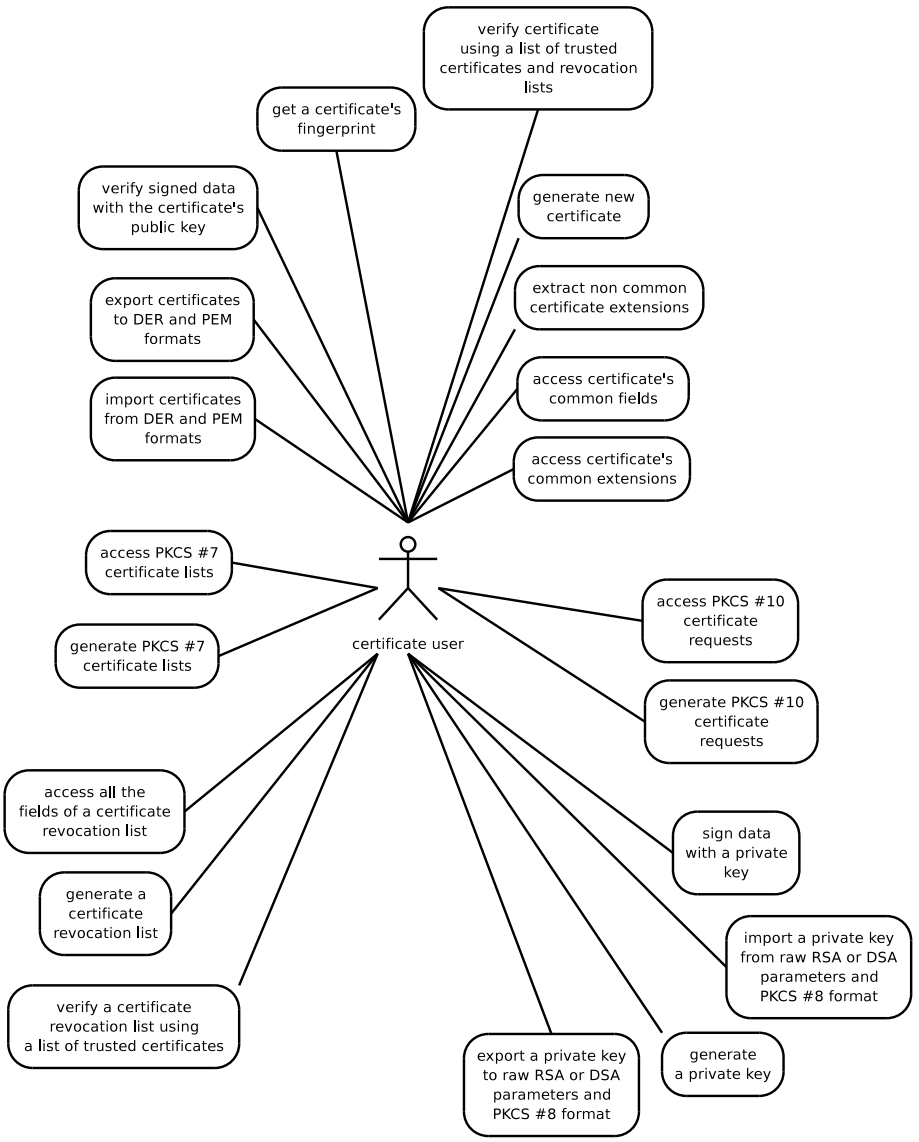
Here there are two functions, one for receiving the extension data and one for sending. These functions have to check internally whether they operate in client or server side.

A simple example of an extension handler can be seen in `ext_srp.c` After implementing these functions, together with the extension number they handle, they have to be registered in `gnutls_extensions.c` in the `_gnutls_extensions` structure.

[1] such as the `gnutls_certificate_credentials_t` structures

12.5 Certificate handling

What is provided by the certificate handling functions is summarized in the following diagram.



Appendix A Copying This Manual

A.1 GNU Free Documentation License

Version 1.2, November 2002

Copyright © 2000,2001,2002 Free Software Foundation, Inc.
51 Franklin St, Fifth Floor, Boston, MA 02110-1301, USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. 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.

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 for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

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.

A.1.1 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 1.2
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.

Concept Index

A

Alert protocol 8
Anonymous authentication 14

C

Callback functions 4
Certificate authentication 18
Certificate requests 21
Certificate to XML convertion 205
certtool 90
Ciphersuites 210
Client Certificate authentication 9
Compression algorithms 7

D

Digital signatures 23

E

Error codes 199
Example programs 28

F

FDL, GNU Free Documentation License 217
Function reference 94

G

gnutls-cli 87
gnutls-cli-debug 88
GnuTLS-extra functions 183
gnutls-serv 89

H

Handshake protocol 8

I

Inner Application (TLS/IA) functions 192
Internal architecture 212

M

Maximum fragment length 11

O

OpenPGP functions 183
OpenPGP Keys 12, 21
OpenPGP Server 67
OpenSSL 86

P

PCT 11
PKCS #10 21
PKCS #12 21
PSK authentication 15

R

Record protocol 6
Resuming sessions 10

S

Server name indication 11
SRP authentication 14
srptool 87
SSL 2 11
Symmetric encryption algorithms 7

T

TLS Extensions 10, 11
TLS Inner Application (TLS/IA) functions 192
TLS Layers 5
Transport protocol 6

V

Verifying certificate paths 19

X

X.509 certificates 12, 18
X.509 Functions 141

Function and Data Index

gnutls_alert_get 94
gnutls_alert_get_name 94
gnutls_alert_send 94
gnutls_alert_send_appropriate 94
gnutls_anon_allocate_client_credentials .. 95
gnutls_anon_allocate_server_credentials .. 95
gnutls_anon_free_client_credentials 95
gnutls_anon_free_server_credentials 95
gnutls_anon_set_params_function 96
gnutls_anon_set_server_dh_params 96
gnutls_anon_set_server_params_function ... 96
gnutls_auth_client_get_type 96
gnutls_auth_get_type 96
gnutls_auth_server_get_type 97
gnutls_bye 97
gnutls_certificate_activation_time_peers 97
gnutls_certificate_allocate_credentials .. 97
gnutls_certificate_client_get_request_status 98
gnutls_certificate_client_set_retrieve_function 98
gnutls_certificate_expiration_time_peers 98
gnutls_certificate_free_ca_names 99
gnutls_certificate_free_cas 99
gnutls_certificate_free_credentials 99
gnutls_certificate_free_crls 99
gnutls_certificate_free_keys 99
gnutls_certificate_get_ours 100
gnutls_certificate_get_peers 100
gnutls_certificate_send_x509_rdn_sequence 100
gnutls_certificate_server_set_request ... 100
gnutls_certificate_server_set_retrieve_function 101
gnutls_certificate_set_dh_params 101
gnutls_certificate_set_openpgp_key 185
gnutls_certificate_set_openpgp_key_file 183
gnutls_certificate_set_openpgp_key_mem .. 184
gnutls_certificate_set_openpgp_keyring_file 184
gnutls_certificate_set_openpgp_keyring_mem 184
gnutls_certificate_set_openpgp_keyserver 184
gnutls_certificate_set_openpgp_trustdb .. 185
gnutls_certificate_set_params_function .. 101
gnutls_certificate_set_rsa_export_params 101
gnutls_certificate_set_verify_flags 102
gnutls_certificate_set_verify_limits 102
gnutls_certificate_set_x509_crl 103
gnutls_certificate_set_x509_crl_file 102
gnutls_certificate_set_x509_crl_mem 102
gnutls_certificate_set_x509_key 104
gnutls_certificate_set_x509_key_file 103
gnutls_certificate_set_x509_key_mem 103
gnutls_certificate_set_x509_simple_pkcs12_file 104
gnutls_certificate_set_x509_trust 105
gnutls_certificate_set_x509_trust_file .. 105
gnutls_certificate_set_x509_trust_mem ... 105
gnutls_certificate_type_get 106
gnutls_certificate_type_get_name 106
gnutls_certificate_type_set_priority 106
gnutls_certificate_verify_flags 20
gnutls_certificate_verify_peers 107
gnutls_certificate_verify_peers2 106
gnutls_check_version 107
gnutls_cipher_get 107
gnutls_cipher_get_key_size 107
gnutls_cipher_get_name 107
gnutls_cipher_set_priority 108
gnutls_cipher_suite_get_name 108
gnutls_compression_get 108
gnutls_compression_get_name 108
gnutls_compression_set_priority 108
gnutls_credentials_clear 109
gnutls_credentials_set 109
gnutls_db_check_entry 109
gnutls_db_get_ptr 110
gnutls_db_remove_session 110
gnutls_db_set_cache_expiration 110
gnutls_db_set_ptr 110
gnutls_db_set_remove_function 110
gnutls_db_set_retrieve_function 111
gnutls_db_set_store_function 111
gnutls_deinit 111
gnutls_dh_get_group 111
gnutls_dh_get_peers_public_bits 112
gnutls_dh_get_prime_bits 112
gnutls_dh_get_pubkey 112
gnutls_dh_get_secret_bits 112
gnutls_dh_params_cpy 112
gnutls_dh_params_deinit 112
gnutls_dh_params_export_pkcs3 113
gnutls_dh_params_export_raw 113
gnutls_dh_params_generate2 113
gnutls_dh_params_import_pkcs3 114
gnutls_dh_params_import_raw 114
gnutls_dh_params_init 114
gnutls_dh_set_prime_bits 114
gnutls_error_is_fatal 115
gnutls_error_to_alert 115
gnutls_extra_check_version 183
gnutls_fingerprint 115
gnutls_free 115
gnutls_global_deinit 116

gnutls_global_init 116
gnutls_global_init_extra 183
gnutls_global_set_log_function 116
gnutls_global_set_log_level 116
gnutls_global_set_mem_functions 117
gnutls_handshake 118
gnutls_handshake_get_last_in 117
gnutls_handshake_get_last_out 117
gnutls_handshake_set_max_packet_length .. 117
gnutls_handshake_set_private_extensions 118
gnutls_hex_decode 118
gnutls_hex_encode 119
gnutls_ia_allocate_client_credentials ... 192
gnutls_ia_allocate_server_credentials ... 193
gnutls_ia_enable 193
gnutls_ia_endphase_send 194
gnutls_ia_extract_inner_secret 194
gnutls_ia_free_client_credentials 194
gnutls_ia_free_server_credentials 194
gnutls_ia_generate_challenge 195
gnutls_ia_get_client_avp_ptr 195
gnutls_ia_get_server_avp_ptr 195
gnutls_ia_handshake 195
gnutls_ia_handshake_p 195
gnutls_ia_permute_inner_secret 196
gnutls_ia_recv 196
gnutls_ia_send 196
gnutls_ia_set_client_avp_function 197
gnutls_ia_set_client_avp_ptr 197
gnutls_ia_set_server_avp_function 198
gnutls_ia_set_server_avp_ptr 198
gnutls_ia_verify_endphase 198
gnutls_init 119
gnutls_kx_get 119
gnutls_kx_get_name 119
gnutls_kx_set_priority 119
gnutls_mac_get 120
gnutls_mac_get_name 120
gnutls_mac_set_priority 120
gnutls_malloc 120
gnutls_openpgp_key_check_hostname 185
gnutls_openpgp_key_deinit 185
gnutls_openpgp_key_export 185
gnutls_openpgp_key_get_creation_time 186
gnutls_openpgp_key_get_expiration_time .. 186
gnutls_openpgp_key_get_fingerprint 186
gnutls_openpgp_key_get_id 186
gnutls_openpgp_key_get_key_usage 186
gnutls_openpgp_key_get_name 187
gnutls_openpgp_key_get_pk_algorithm 187
gnutls_openpgp_key_get_version 187
gnutls_openpgp_key_import 187
gnutls_openpgp_key_init 188
gnutls_openpgp_key_to_xml 188
gnutls_openpgp_key_verify_ring 188
gnutls_openpgp_key_verify_self 188
gnutls_openpgp_key_verify_trustdb 189
gnutls_openpgp_keyring_check_id 189
gnutls_openpgp_keyring_deinit 189
gnutls_openpgp_keyring_import 189
gnutls_openpgp_keyring_init 190
gnutls_openpgp_privkey_deinit 190
gnutls_openpgp_privkey_get_pk_algorithm 190
gnutls_openpgp_privkey_import 190
gnutls_openpgp_privkey_init 191
gnutls_openpgp_send_key 120
gnutls_openpgp_set_recv_key_function 191
gnutls_openpgp_trustdb_deinit 191
gnutls_openpgp_trustdb_import_file 191
gnutls_openpgp_trustdb_init 191
gnutls_pem_base64_decode 121
gnutls_pem_base64_decode_alloc 121
gnutls_pem_base64_encode 121
gnutls_pem_base64_encode_alloc 121
gnutls_perror 122
gnutls_pk_algorithm_get_name 122
gnutls_pkcs12_bag_decrypt 141
gnutls_pkcs12_bag_deinit 141
gnutls_pkcs12_bag_encrypt 141
gnutls_pkcs12_bag_get_count 141
gnutls_pkcs12_bag_get_data 142
gnutls_pkcs12_bag_get_friendly_name 142
gnutls_pkcs12_bag_get_key_id 142
gnutls_pkcs12_bag_get_type 142
gnutls_pkcs12_bag_init 143
gnutls_pkcs12_bag_set_crl 143
gnutls_pkcs12_bag_set_crt 143
gnutls_pkcs12_bag_set_data 143
gnutls_pkcs12_bag_set_friendly_name 143
gnutls_pkcs12_bag_set_key_id 144
gnutls_pkcs12_deinit 144
gnutls_pkcs12_export 144
gnutls_pkcs12_generate_mac 144
gnutls_pkcs12_get_bag 145
gnutls_pkcs12_import 145
gnutls_pkcs12_init 145
gnutls_pkcs12_set_bag 145
gnutls_pkcs12_verify_mac 145
gnutls_pkcs7_deinit 146
gnutls_pkcs7_delete_crl 146
gnutls_pkcs7_delete_crt 146
gnutls_pkcs7_export 146
gnutls_pkcs7_get_crl_count 146
gnutls_pkcs7_get_crl_raw 147
gnutls_pkcs7_get_crt_count 147
gnutls_pkcs7_get_crt_raw 147
gnutls_pkcs7_import 147
gnutls_pkcs7_init 148
gnutls_pkcs7_set_crl 148
gnutls_pkcs7_set_crl_raw 148
gnutls_pkcs7_set_crt 148
gnutls_pkcs7_set_crt_raw 148
gnutls_prf 123
gnutls_prf_raw 122

gnutls_protocol_get_name 123
gnutls_protocol_get_version 123
gnutls_protocol_set_priority 124
gnutls_psk_allocate_client_credentials .. 124
gnutls_psk_allocate_server_credentials .. 124
gnutls_psk_free_client_credentials 124
gnutls_psk_free_server_credentials 124
gnutls_psk_server_get_username 125
gnutls_psk_set_client_credentials 125
gnutls_psk_set_client_credentials_function
..... 125
gnutls_psk_set_params_function 125
gnutls_psk_set_server_credentials_file .. 126
gnutls_psk_set_server_credentials_function
..... 126
gnutls_psk_set_server_dh_params 126
gnutls_psk_set_server_params_function ... 126
gnutls_record_check_pending 127
gnutls_record_get_direction 127
gnutls_record_get_max_size 127
gnutls_record_recv 127
gnutls_record_send 128
gnutls_record_set_max_size 128
gnutls_rehandshake 129
gnutls_rsa_export_get_modulus_bits 129
gnutls_rsa_export_get_pubkey 129
gnutls_rsa_params_cpy 129
gnutls_rsa_params_deinit 129
gnutls_rsa_params_export_pkcs1 130
gnutls_rsa_params_export_raw 130
gnutls_rsa_params_generate2 130
gnutls_rsa_params_import_pkcs1 131
gnutls_rsa_params_import_raw 131
gnutls_rsa_params_init 131
gnutls_server_name_get 131
gnutls_server_name_set 132
gnutls_session_get_client_random 132
gnutls_session_get_data 133
gnutls_session_get_data2 132
gnutls_session_get_id 133
gnutls_session_get_master_secret 133
gnutls_session_get_ptr 133
gnutls_session_get_server_random 134
gnutls_session_is_resumed 134
gnutls_session_set_data 134
gnutls_session_set_ptr 134
gnutls_set_default_export_priority 134
gnutls_set_default_priority 135
gnutls_sign_algorithm_get_name 135
gnutls_srp_allocate_client_credentials .. 135
gnutls_srp_allocate_server_credentials .. 135
gnutls_srp_base64_decode 136
gnutls_srp_base64_decode_alloc 136
gnutls_srp_base64_encode 136
gnutls_srp_base64_encode_alloc 136
gnutls_srp_free_client_credentials 137
gnutls_srp_free_server_credentials 137
gnutls_srp_server_get_username 137
gnutls_srp_set_client_credentials 138
gnutls_srp_set_client_credentials_function
..... 137
gnutls_srp_set_server_credentials_file .. 138
gnutls_srp_set_server_credentials_function
..... 138
gnutls_srp_verifier 139
gnutls_strerror 139
gnutls_transport_get_ptr 139
gnutls_transport_get_ptr2 139
gnutls_transport_set_lowat 140
gnutls_transport_set_ptr 140
gnutls_transport_set_ptr2 140
gnutls_transport_set_pull_function 140
gnutls_transport_set_push_function 141
gnutls_x509_crl_check_issuer 148
gnutls_x509_crl_deinit 149
gnutls_x509_crl_export 149
gnutls_x509_crl_get_crt_count 149
gnutls_x509_crl_get_crt_serial 149
gnutls_x509_crl_get_dn_oid 150
gnutls_x509_crl_get_issuer_dn 150
gnutls_x509_crl_get_issuer_dn_by_oid 150
gnutls_x509_crl_get_next_update 151
gnutls_x509_crl_get_signature_algorithm
..... 151
gnutls_x509_crl_get_this_update 151
gnutls_x509_crl_get_version 151
gnutls_x509_crl_import 151
gnutls_x509_crl_init 152
gnutls_x509_crl_set_crt 152
gnutls_x509_crl_set_crt_serial 152
gnutls_x509_crl_set_next_update 152
gnutls_x509_crl_set_this_update 153
gnutls_x509_crl_set_version 153
gnutls_x509_crl_sign 153
gnutls_x509_crl_sign2 153
gnutls_x509_crl_verify 154
gnutls_x509_crq_deinit 154
gnutls_x509_crq_export 154
gnutls_x509_crq_get_attribute_by_oid 154
gnutls_x509_crq_get_challenge_password .. 155
gnutls_x509_crq_get_dn 156
gnutls_x509_crq_get_dn_by_oid 155
gnutls_x509_crq_get_dn_oid 156
gnutls_x509_crq_get_pk_algorithm 156
gnutls_x509_crq_get_version 157
gnutls_x509_crq_import 157
gnutls_x509_crq_init 157
gnutls_x509_crq_set_attribute_by_oid 157
gnutls_x509_crq_set_challenge_password .. 157
gnutls_x509_crq_set_dn_by_oid 158
gnutls_x509_crq_set_key 158
gnutls_x509_crq_set_version 158
gnutls_x509_crq_sign 159
gnutls_x509_crq_sign2 158
gnutls_x509_crt_check_hostname 159
gnutls_x509_crt_check_issuer 159

gnutls_x509_crt_check_revocation 160
gnutls_x509_crt_cpy_crl_dist_points 160
gnutls_x509_crt_deinit 160
gnutls_x509_crt_export 160
gnutls_x509_crt_get_activation_time 161
gnutls_x509_crt_get_authority_key_id 161
gnutls_x509_crt_get_ca_status 161
gnutls_x509_crt_get_crl_dist_points 161
gnutls_x509_crt_get_dn 163
gnutls_x509_crt_get_dn_by_oid 162
gnutls_x509_crt_get_dn_oid 162
gnutls_x509_crt_get_expiration_time 163
gnutls_x509_crt_get_extension_by_oid 163
gnutls_x509_crt_get_extension_oid 164
gnutls_x509_crt_get_fingerprint 164
gnutls_x509_crt_get_issuer_dn 165
gnutls_x509_crt_get_issuer_dn_by_oid 164
gnutls_x509_crt_get_issuer_dn_oid 165
gnutls_x509_crt_get_key_id 166
gnutls_x509_crt_get_key_purpose_oid 166
gnutls_x509_crt_get_key_usage 166
gnutls_x509_crt_get_pk_algorithm 167
gnutls_x509_crt_get_pk_dsa_raw 167
gnutls_x509_crt_get_pk_rsa_raw 167
gnutls_x509_crt_get_serial 168
gnutls_x509_crt_get_signature_algorithm 168
gnutls_x509_crt_get_subject_alt_name 168
gnutls_x509_crt_get_subject_key_id 169
gnutls_x509_crt_get_version 169
gnutls_x509_crt_import 169
gnutls_x509_crt_init 169
gnutls_x509_crt_list_import 169
gnutls_x509_crt_list_verify 170
gnutls_x509_crt_set_activation_time 171
gnutls_x509_crt_set_authority_key_id 171
gnutls_x509_crt_set_ca_status 171
gnutls_x509_crt_set_crl_dist_points 171
gnutls_x509_crt_set_crq 172
gnutls_x509_crt_set_dn_by_oid 172
gnutls_x509_crt_set_expiration_time 172
gnutls_x509_crt_set_extension_by_oid 172
gnutls_x509_crt_set_issuer_dn_by_oid 173
gnutls_x509_crt_set_key 174
gnutls_x509_crt_set_key_purpose_oid 173
gnutls_x509_crt_set_key_usage 173
gnutls_x509_crt_set_serial 174
gnutls_x509_crt_set_subject_alternative_name 174
gnutls_x509_crt_set_subject_key_id 174
gnutls_x509_crt_set_version 175
gnutls_x509_crt_sign 175
gnutls_x509_crt_sign2 175
gnutls_x509_crt_to_xml 175
gnutls_x509_crt_verify 176
gnutls_x509_crt_verify_data 176
gnutls_x509_dn_oid_known 176
gnutls_x509_privkey_cpy 177
gnutls_x509_privkey_deinit 177
gnutls_x509_privkey_export 178
gnutls_x509_privkey_export_dsa_raw 177
gnutls_x509_privkey_export_pkcs8 177
gnutls_x509_privkey_export_rsa_raw 178
gnutls_x509_privkey_fix 179
gnutls_x509_privkey_generate 179
gnutls_x509_privkey_get_key_id 179
gnutls_x509_privkey_get_pk_algorithm 179
gnutls_x509_privkey_import 181
gnutls_x509_privkey_import_dsa_raw 180
gnutls_x509_privkey_import_pkcs8 180
gnutls_x509_privkey_import_rsa_raw 180
gnutls_x509_privkey_init 181
gnutls_x509_privkey_sign_data 181
gnutls_x509_privkey_verify_data 182
gnutls_x509_rdn_get 182
gnutls_x509_rdn_get_by_oid 182
gnutls_x509_rdn_get_oid 182

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/`.

[RFC2246]
Tim Dierks and Christopher Allen, "The TLS Protocol Version 1.0", January 1999, Available from `http://kaizi.viagenie.qc.ca/ietf/rfc/rfc2246.txt`.

[RFC4346]
Tim Dierks and Eric Rescorla, "The TLS Protocol Version 1.1", Match 2006, Available from `http://kaizi.viagenie.qc.ca/ietf/rfc/rfc4346.txt`.

[RFC2440]
Jon Callas, Lutz Donnerhacke, Hal Finney and Rodney Thayer, "OpenPGP Message Format", November 1998, Available from `http://kaizi.viagenie.qc.ca/ietf/rfc/rfc2440.txt`.

[RFC2511]
Michael Myers, Carlisle Adams, Dave Solo and David Kemp, "Internet X.509 Certificate Request Message Format", March 1999, Available from `http://kaizi.viagenie.qc.ca/ietf/rfc/rfc2511.txt`.

[RFC2817]
Rohit Khare and Scott Lawrence, "Upgrading to TLS Within HTTP/1.1", May 2000, Available from `http://kaizi.viagenie.qc.ca/ietf/rfc/rfc2817.txt`

[RFC2818]
Eric Rescola, "HTTP Over TLS", May 2000, Available from `http://kaizi.viagenie.qc.ca/ietf/rfc/rfc2818.txt`.

[RFC2945]
Tom Wu, "The SRP Authentication and Key Exchange System", September 2000, Available from `http://kaizi.viagenie.qc.ca/ietf/rfc/rfc2945.txt`.

[RFC2986]
Magnus Nystrom and Burt Kaliski, "PKCS 10 v1.7: Certification Request Syntax Specification", November 2000, Available from `http://kaizi.viagenie.qc.ca/ietf/rfc/rfc2986.txt`.

[RFC3280]
Russell Housley, Tim Polk, Warwick Ford and David Solo, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", April 2002, Available from `http://kaizi.viagenie.qc.ca/ietf/rfc/rfc3280.txt`.

[RFC3749]
Scott Hollenbeck, "Transport Layer Security Protocol Compression Methods", May 2004, Available from `http://kaizi.viagenie.qc.ca/ietf/rfc/rfc3749.txt`.

[PKCS12] RSA Laboratories, "PKCS 12 v1.0: Personal Information Exchange Syntax", June 1999, Available from `http://www.rsa.com`.

[RESCOLA]
Eric Rescola, "SSL and TLS: Designing and Building Secure Systems", 2001

[SSL3] Alan Freier, Philip Karlton and Paul Kocher, "The SSL Protocol Version 3.0", November 1996, Available from `http://wp.netscape.com/eng/ssl3/draft302.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://kaizi.viagenie.qc.ca/ietf/rfc/rfc3546.txt`.

[TLSPGP] Nikos Mavrogiannopoulos, "Using OpenPGP keys for TLS authentication", April 2004, Internet draft, work in progress. Available from `http://www.normos.org/ietf/draft/draft-ietf-tls-openpgp-keys-05.txt`.

[TLSSRP] David Taylor, Trevor Perrin, Tom Wu and Nikos Mavrogiannopoulos, "Using SRP for TLS Authentication", August 2005, Internet draft, work in progress. Available from `http://www.normos.org/ietf/draft/draft-ietf-tls-srp-08.txt`.

[TLSPSK] Pasi Eronen and Hannes Tschofenig, "Pre-shared key Ciphersuites for TLS", December 2005, Available from `http://kaizi.viagenie.qc.ca/ietf/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/`.