GNUTLS

a Transport Layer Security Library
This is a Draft document
Applies to GnuTLS 1.0.17

by Nikos Mavroyanopoulos

Contents

Preface

Introduction

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.

This document tries to be self contained, although basic network programming and PKI knowlegde is assumed in most of it. [6] is a good introduction to Public Key Infrastructure.

Availability

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

Chapter 1

The Library

1.1 Description

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 1.0*[1] and *SSL 3.0* protocols, accompanied with the required framework for authentication and public key infrastructure. The library is available under the GNU Lesser GPL license[2]. 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 *TLS* **Extension mechanism**.
- Support for *TLS* **Compression Methods**.

Additionally *GnuTLS* provides a limited emulation API for the widely used OpenSSL[3] 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

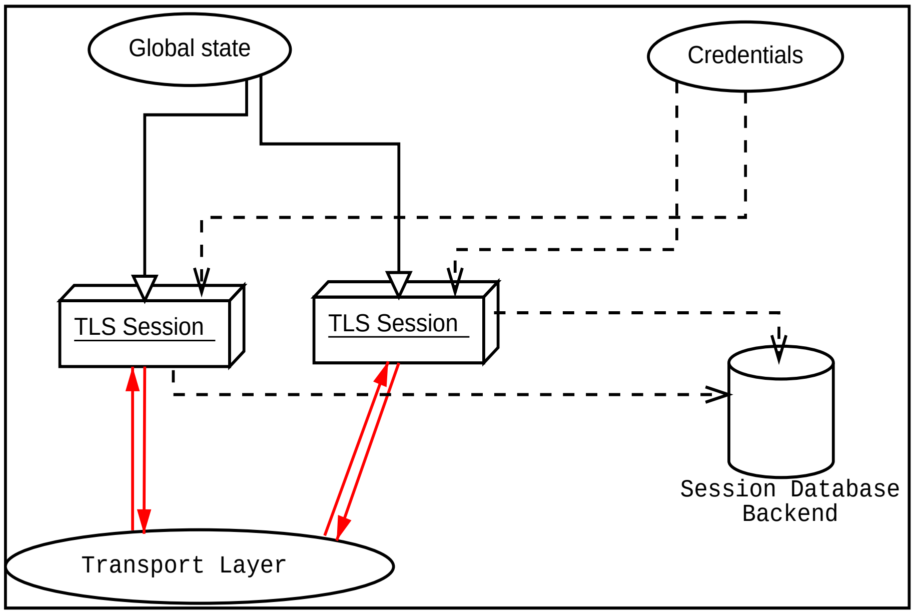
[1]See section 2 on page 5 for a more detailed description of the protocols.
[2]A copy of the license is included in the distribution
[3]http://www.openssl.org/

GnuTLS library. The 'Certificate part' consists of the certificate parsing, and verification functions which is partially implemented in the *GnuTLS* library. The Libtasn1[4] a library which offers ASN.1 parsing capabilities, is used for the X.509 certificate parsing functions, and Opencdk[5] is used for the OpenPGP key support in *GnuTLS*. The 'Crypto backend' is provided by the libgcrypt[6] 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.

1.2 General Idea

A brief description of how *GnuTLS* works internally is shown at the figure 1.2. This section may be easier to understand after having seen the examples on page 35.



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*.

[4]ftp://ftp.gnupg.org/gcrypt/alpha/gnutls/libtasn1/
[5]ftp://ftp.gnupg.org/gcrypt/alpha/gnutls/opencdk/
[6]ftp://ftp.gnupg.org/gcrypt/alpha/libgcrypt/

The credentials structure is used by some authentication methods, such as certificate authentication[7]. 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[8] to store the newly negotiated session. The session database is examined by the server just after having received the client hello[9], and if the session ID sent by the client, matches a stored session, the stored session will be retrieved, and the new session will be a resumed one, and will share the same session ID with the previous one.

1.3 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, ie 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() (see section 8.1.116 p.130) . 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() (see section 8.1.81 p.119) .

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. All the error codes are documented in appendix B on page 223.

[7]see section 4 on page 23

[8]see section 2.5 on 11 for information on initialization

[9]The first message in a *TLS* handshake

1.4 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() (see section 8.1.89 p.122) 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.

1.5 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() (see section 8.1.147 p.142)
- gnutls_transport_set_pull_function() (see section 8.1.146 p.142)

Other callback functions such as the one set by gnutls_srp_set_server_credentials_function() (see section 8.3.15 p.205) , may require more complicated input, including data to be allocated. These callbacks should allocate and free memory using the functions shown below.

- gnutls_malloc() (see section 8.1.102 p.126)
- gnutls_free() (see section 8.1.84 p.120)

Chapter 2

Introduction to *TLS*

TLS stands for 'Transport Layer Security' and is the successor of *SSL*, the Secure Sockets Layer protocol[1] designed by Netscape. *TLS 1.0* is an Internet protocol, defined by IETF[2], described in [4] and also in [16]. The protocol provides confidentiality, and authentication layers over any reliable transport layer. The description, below, refers to *TLS 1.0* but also applies to *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.

2.1 TLS layers

TLS 1.0 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 section 2.4 on page 9 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 section 2.5 on page 9 for more information about the handshake protocol. The protocol layering in TLS is shown at figure 2.1.

[1]described in [5]

[2]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.

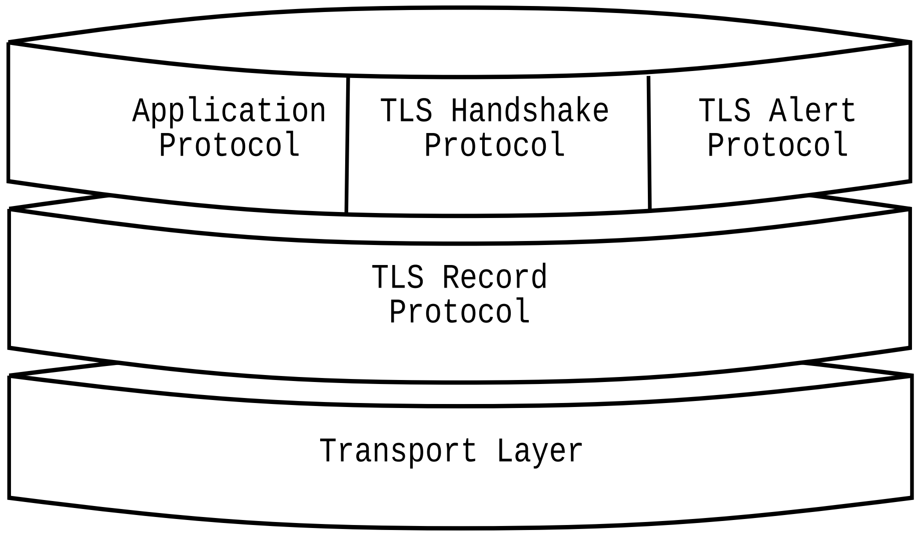


Figure 2.1: Layers in the TLS protocol

2.2 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() (see section 8.1.147 p.142)
- gnutls_transport_set_pull_function() (see section 8.1.146 p.142)
- gnutls_transport_set_ptr() (see section 8.1.145 p.142)

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.

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.

2.3 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() (see section 8.1.117 p.131) : to send a record packet (with application data).
- gnutls_record_recv() (see section 8.1.116 p.130) : to receive a record packet (with application data).

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.

Encryption algorithms used in the record layer

Confidentiality in the record layer is achieved by using symmetric block encryption algorithms like **3DES**, **AES**[3], or stream algorithms like **ARCFOUR_128**[4] See figure 2.2 for a complete list. 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 1.0* protocol, a random number of blocks will be appended to data, to prevent eavesdroppers from guessing the actual data size.

Compression algorithms used in the record layer

The TLS' record layer also supports compression. The algorithms implemented in *GnuTLS* can be found in figure 2.4. All the algorithms except for DEFLATE

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

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

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.

Figure 2.2: Supported cipher 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.
MAC_RMD160	RIPEMD is a cryptographic hash algorithm developed in the framework of the EU project RIPE. Outputs 160 bits of data.

Figure 2.3: Supported MAC algorithms

which is referenced in [7], should be considered as *GnuTLS*' extensions[5], 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 compressable 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 paper [7].

DEFLATE	Zlib compression, using the deflate algorithm.
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.

Figure 2.4: Supported compression algorithms

[5]You should use gnutls_handshake_set_private_extensions() (see section 8.1.93 p.123) to enable private extensions.

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* which is implemented in *GnuTLS*. For a detailed discussion see the archives of the TLS Working Group mailing list and the paper [12].

2.4 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() (see section 8.1.4 p.92) : to send an alert signal.
- gnutls_error_to_alert() (see section 8.1.82 p.120) : to map a gnutls error number to an alert signal.
- gnutls_alert_get() (see section 8.1.3 p.91) : returns the last received alert.
- gnutls_alert_get_name() (see section 8.1.2 p.91) : returns the name, in a character array, of the given alert.

2.5 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() (see section 8.1.53 p.110) : to set the priority of bulk cipher algorithms.
- gnutls_mac_set_priority() (see section 8.1.101 p.126) : to set the priority of MAC algorithms.
- gnutls_kx_set_priority() (see section 8.1.98 p.125) : to set the priority of key exchange algorithms.
- gnutls_compression_set_priority() (see section 8.1.57 p.111) : to set the priority of compression methods.
- gnutls_certificate_type_set_priority() (see section 8.1.46 p.108) : to set the priority of certificate types (ie. OpenPGP, X.509).
- gnutls_protocol_set_priority() (see section 8.1.112 p.129) : to set the priority of protocol versions (ie. *SSL 3.0*, *TLS 1.0*).
- gnutls_set_default_priority() (see section 8.1.138 p.139) : 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() (see section 8.1.59 p.112) : to set the appropriate credentials structures.
- gnutls_certificate_server_set_request() (see section 8.1.27 p.100) : to set whether client certificate is required or not.
- gnutls_handshake() (see section 8.1.94 p.124) : to initiate the handshake.

TLS cipher suites

The Handshake Protocol of *TLS 1.0* 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[6] 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

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

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 appendix C on page 227.

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() (see section 8.1.27 p.100) function. If a certificate is to be requested from the client during the handshake, the server will send a certificate request message that contains a list of acceptable certificate signers. The client may then send a certificate, signed by one of the server's acceptable signers. In *GnuTLS* the server's acceptable signers list is constructed using the trusted CA certificates in the credentials structure.

Resuming Sessions

The gnutls_handshake() (see section 8.1.94 p.124) 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 resume client (see section 6.3.5) 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.

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. Ie. copy and keep the necessary parameters. See the functions:

- gnutls_session_get_data() (see section 8.1.131 p.137)

- gnutls_session_get_id() (see section 8.1.132 p.137)
- gnutls_session_set_data() (see section 8.1.135 p.138)

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() (see section 8.1.65 p.114)
- gnutls_db_set_store_function() (see section 8.1.67 p.114)
- gnutls_db_set_retrieve_function() (see section 8.1.66 p.114)
- gnutls_db_set_ptr() (see section 8.1.64 p.113)

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() (see section 8.1.60 p.112) is provided for that reason.

2.6 TLS Extensions

A number of extensions to the *TLS* protocol have been proposed mainly in [2]. The extensions supported in *GnuTLS* are

- Maximum fragment length negotiation
- Server name indication

discussed in the subsections that follow.

Maximum fragment length negotiation

This extension allows a *TLS 1.0* 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() (see section 8.1.118 p.132) and the gnutls_record_get_max_size() (see section 8.1.115 p.130) functions.

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() (see section 8.1.130 p.137) and gnutls_server_name_get() (see section 8.1.129 p.136) can be used to enable this extension, or to retrieve the name sent by a client.

Chapter 3

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:

1. Certificate authentication

2. Anonymous authentication

3. SRP authentication

3.1 Certificate authentication

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 4.1 on page 23 for more information on X.509 protocols.

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 [11] proposal.

See 4.2 on page 26 for more information about the OpenPGP trust model. For a more detailed introduction to OpenPGP and GnuPG see [1].

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() (see section 8.1.38 p.104) or gnutls_certificate_set_openpgp_key_file() (see section **??** p.**??**) 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() (see section 8.1.28 p.100)
- gnutls_certificate_client_set_retrieve_function() (see section 8.1.16 p.96)

Certificate verification is possible by loading the trusted authorities into the credentials structure by using gnutls_certificate_set_x509_trust_file() (see section 8.1.41 p.106) or gnutls_certificate_set_openpgp_keyring_file() (see section **??** p.**??**) for openpgp keys. Note however that the peer's certificate is not automatically verified, you should call gnutls_certificate_verify_peers() (see section 8.1.48 p.108) , 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() (see section 8.1.25 p.99) to obtain the raw certificate of the peer and verify it using the functions discussed in section 4.1 on page 23.

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 in figure 3.1 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

[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

credentials structure. 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() (see section 8.1.76 p.117)
- gnutls_certificate_set_dh_params() (see section 8.1.30 p.101)
- gnutls_rsa_params_generate2() (see section 8.1.125 p.134)
- gnutls_certificate_set_rsa_export_params() (see section 8.1.32 p.102)

RSA	The RSA algorithm is used to encrypt a key and send it to the peer. The certificate must allow the key to be used for encryption.
RSA_EXPORT	The RSA algorithm is used to encrypt a key and send it to the peer. In the EXPORT algorithm, the server signs temporary RSA parameters of 512 bits – which are considered weak – and sends them to the client.
DHE_RSA	The RSA algorithm is used to sign Ephemeral Diffie Hellman parameters which are sent to the peer. The key in the certificate must allow the key to be used for signing. Note that key exchange algorithms which use Ephemeral Diffie Hellman parameters, offer perfect forward secrecy. That means that even if the private key used for signing is compromised, it cannot be used to reveal past session data.
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.

Figure 3.1: Key exchange algorithms for OpenPGP and X.509 certificates.

3.2 Anonymous authentication

The anonymous key exchange perform 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 in figure 3.2.

Note that the key exchange methods for anonymous authentication require Diffie Hellman parameters to be generated and associated with an anonymous credentials structure.

ANON_DH	This algorithm exchanges Diffie Hellman parameters.

Figure 3.2: Supported anonymous key exchange algorithms

3.3 Authentication using SRP

Authentication using the SRP[2] protocol is actually password authentication. 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 schemas, 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* "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. See [18] 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 [17]. The available key exchange methods are shown in figure 3.3.

SRP	Authentication using the SRP protocol.
SRP_DSS	Client authentication using the SRP protocol. Server is authenticated using a certificate with DSA parameters.
SRP_RSA	Client authentication using the SRP protocol. Server is authenticated using a certificate with RSA parameters.

Figure 3.3: Supported SRP key exchange algorithms

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() (see section 8.3.13 p.204) . Alternatively they could specify a callback function by using the function gnutls_srp_set_client_credentials_function() (see section 8.3.12 p.203) . 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.

[2]SRP stands for Secure Remote Password and is described in [19]. The SRP key exchange is an extension to the *TLS 1.0* protocol

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() (see section 8.3.14 p.204) . If a different password file format is to be used, then the function gnutls_srp_set_server_credentials_function() (see section 8.3.15 p.205) , should be called, in order to set an appropriate callback.

Some helper functions such as

- gnutls_srp_verifier() (see section 8.3.16 p.205)
- gnutls_srp_base64_encode() (see section 8.3.7 p.202)
- gnutls_srp_base64_decode() (see section 8.3.5 p.201)

are included in *GnuTLS*, and may 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 section 7.1 on page 85 for more information.

3.4 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() (see section 8.1.59 p.112) . A mapping is shown in figure 3.4.

Key exchange	**Client credentials**	**Server credentials**
KX_RSA	CRD_CERTIFICATE	CRD_CERTIFICATE
KX_DHE_RSA		
KX_DHE_DSS		
KX_RSA_EXPORT		
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

Figure 3.4: Key exchange algorithms and the corresponding credential types

3.5 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** and **gnutls_rsa_params**, and functions like gnutls_certificate_set_dh_params() (see section 8.1.30 p.101) and gnutls_certificate_set_rsa_export_params() (see section 8.1.32 p.102) 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() (see section 8.1.31 p.101) . An example is shown below.

```
#include <gnutls.h>

gnutls_rsa_params rsa_params;
gnutls_dh_params 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 session, gnutls_params_type 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 cert_cred;

   initialize_params();

   /* ...
    */

   gnutls_certificate_set_params_function( cert_cred, get_params);
}
```

Chapter 4

More on certificate authentication

4.1 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.

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 figure 4.1 for a typical example. The API for handling X.509 certificates is described at section 8.2 on page 144. Some examples are listed below.

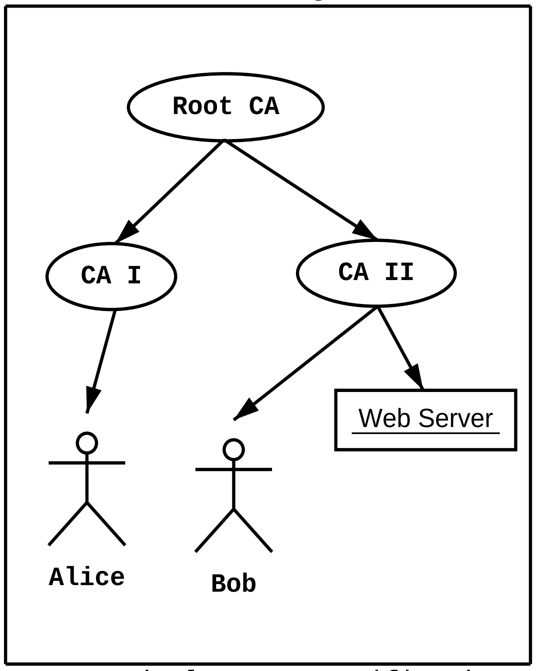
4.1.1 X.509 certificates

An X.509 certificate usually contains information about the certificate holder, the signer, a unique serial number, expiration dates and some other fields [8] 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

Figure 4.1: X.509 certification



Two typical X.509 Certification paths

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 6.5.2 on page 75.

4.1.2 Verifying X.509 certificate paths

Verifying certificate paths is important in X.509 authentication. For this purpose the function gnutls_x509_crt_verify() (see section 8.2.125 p.190) is provided. The output of this function is the bitwise OR of the elements of the "gnutls_certificate_status" enumeration. A detailed description of these elements can be found in figure 4.2. The function gnutls_certificate_verify_peers() (see section 8.1.48 p.108) is equivalent to the previous one, and will verify the peer's certificate in a TLS session.

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.

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.
CERT_SIGNER_NOT_FOUND	The certificate's issuer is not known.

Figure 4.2: X.509 certificate verification

It is required to verify if the certificate's owner is the one you expect. See [15] and section 6.3.3 on page 41 for an example.

4.1.3 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 [14]. Other certificate request's format such as PKIX's RFC2511 [13] 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 6.5.3 on page 78.

4.1.4 PKCS #12 structures

A PKCS #12 structure [10] usually contains a user's private keys and certificates. It is commonly used in browsers to export and import the user's identities.

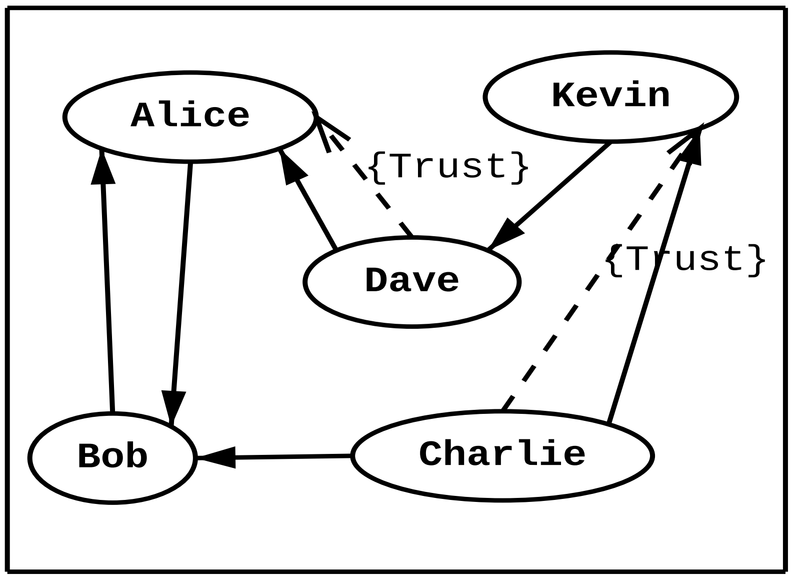
In *GnuTLS* the PKCS #12 structures are handled using the *gnutls_pkcs12_t* type. This is an abstract type that may hold several *gnutls_pkcs12_bag_t* types. The Bag types are the holders of the actual data, which may be certificates, private keys or encrypted data. An Bag of type encrypted should be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

An example of a PKCS #12 structure generation can be found at section 6.5.4 on page 80.

4.2 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.

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.



An example of the web of trust model

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

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

OpenPGP keys

In *GnuTLS* the OpenPGP key structures [3] 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*.

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[1] is recommended.

There are two verification functions in *GnuTLS*, The gnutls_openpgp_key_verify_ring() (see section 8.4.15 p.211) and the gnutls_openpgp_key_verify_trustdb() (see section 8.4.17 p.211) . 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 figure 4.3.
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.

Figure 4.3: OpenPGP key verification

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

Chapter 5

How to use *TLS* in application protocols

5.1 Introduction

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.

5.2 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[1] protocol [15].

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[2]. 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

[1]RFC2818

[2]see also the Server Name Indication extension on 2.6, page 12.

available privileged ports, this approach was soon obsoleted.

5.3 Upward negotiation

Other application protocols[3] 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 [9] 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

*** 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
```

[3]See LDAP, IMAP etc.

```
(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[4], or anything else!

[4]in SRP authentication

Chapter 6

How to use *GnuTLS* in applications

6.1 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.

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.

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() (see section 8.1.49 p.109)

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‘
```

6.2 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);
}
```

6.3 Client examples

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

6.3.1 Simple client example with X.509 certificate support

Let's assume now that we want to create a TCP client which communicates with servers that use X.509 or OpenPGP certificate authentication. The following client is a very simple *TLS* client, 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.
 */

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

/* Connects to the peer and returns a socket
 * descriptor.
 */
int tcp_connect( void)
{
   const char *PORT = "443";
   const char *SERVER = "127.0.0.1";
   int err, sd;
   struct sockaddr_in sa;

   /* connects to server
    */
   sd = socket(AF_INET, SOCK_STREAM, 0);

   memset(&sa, '\0', sizeof(sa));
   sa.sin_family = AF_INET;
   sa.sin_port = htons(atoi(PORT));
   inet_pton(AF_INET, SERVER, &sa.sin_addr);
   err = connect(sd, (SA *) & sa, sizeof(sa));
   if (err < 0) {
      fprintf(stderr, "Connect error\n");
      exit(1);
   }

   return sd;
}

/* closes the given socket descriptor.
 */
void tcp_close( int sd)
{
   shutdown(sd, SHUT_RDWR);     /* no more receptions */
   close(sd);
}

int main()
{
   int ret, sd, ii;
   gnutls_session session;
   char buffer[MAX_BUF + 1];
   gnutls_certificate_credentials 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)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;
}
```

6.3.2 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() (see section 8.1.94 p.124)

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

extern void print_x509_certificate_info(gnutls_session);

/* This function will print some details of the
 * given session.
 */
int print_info(gnutls_session session)
{
   const char *tmp;
   gnutls_credentials_type cred;
   gnutls_kx_algorithm 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_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;
}
```

6.3.3 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. The following function is an example on how to verify the peer's certificate chain. This is an advanced case. Things in a TLS session may be simplified by using gnutls_certificate_verify_peers2() (see section 8.1.47 p.108) .

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

/* All the available CRLs
 */
extern gnutls_x509_crl* crl_list;
extern int crl_list_size;

/* All the available trusted CAs
 */
extern gnutls_x509_crt* ca_list;
extern int ca_list_size;

static void verify_cert2(gnutls_x509_crt crt,
   gnutls_x509_crt issuer, gnutls_x509_crl * crl_list, int crl_list_size);
static void verify_last_cert(gnutls_x509_crt crt,
   gnutls_x509_crt *ca_list, int ca_list_size,
   gnutls_x509_crl * 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 session, const char* hostname,
   const gnutls_datum* cert_chain, int cert_chain_length)
```

```
{
   int i, ret;
   gnutls_x509_crt 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);
   }

   /* 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 crt,
   gnutls_x509_crt issuer, gnutls_x509_crl * 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 the trusted CA list.
 * Also checks the crl_list if the certificate is revoked.
 */
static void verify_last_cert(gnutls_x509_crt crt,
   gnutls_x509_crt *ca_list, int ca_list_size,
   gnutls_x509_crl * 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, 0, &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");
   }
}
```

6.3.4 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/mman.h>
#include <sys/stat.h>
#include <gnutls/gnutls.h>
#include <gnutls/x509.h>

/* A TLS client that loads the certificate and key.
```

```
 */

#define MAX_BUF 1024
#define SA struct sockaddr
#define MSG "GET / HTTP/1.0\r\n\r\n"

#define CERT_FILE "cert.pem"
#define KEY_FILE "key.pem"
#define CAFILE "ca.pem"

static int cert_callback(gnutls_session session,
   const gnutls_datum* req_ca_rdn, int nreqs,
   const gnutls_pk_algorithm* sign_algos, int sign_algos_length,
   gnutls_retr_st * st);

gnutls_x509_crt crt;
gnutls_x509_privkey key;

/* Helper functions to load a certificate and key
 * files into memory. They use mmap for simplicity.
 */
static gnutls_datum mmap_file( const char* file)
{
int fd;
gnutls_datum mmaped_file = { NULL, 0 };
struct stat stat_st;
void* ptr;

   fd = open( file, 0);
   if (fd==-1) return mmaped_file;

   fstat( fd, &stat_st);

   if ((ptr=mmap( NULL, stat_st.st_size, PROT_READ, MAP_SHARED, fd, 0)) == MAP_FAILED)
      return mmaped_file;

   mmaped_file.data = ptr;
   mmaped_file.size = stat_st.st_size;

   return mmaped_file;
}

static void munmap_file( gnutls_datum data)
{
   munmap( data.data, data.size);
}
```

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

   data = mmap_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);
   }

   munmap_file( data);

   data = mmap_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);
   }

   munmap_file( data);

}

int main()
{
   int ret, sd, ii;
   gnutls_session session;
   char buffer[MAX_BUF + 1];
```

```
gnutls_certificate_credentials 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)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 session,
                  const gnutls_datum* req_ca_rdn, int nreqs,
                  const gnutls_pk_algorithm* sign_algos, int sign_algos_length,
                  gnutls_retr_st * st)
{
   char issuer_dn[256];
   int i, ret;
   size_t len;
   gnutls_certificate_type 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;

}
```

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

/* Those functions are defined in other examples.
 */
extern void check_alert(gnutls_session 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 SA struct sockaddr
#define MSG "GET / HTTP/1.0\r\n\r\n"

int main()
{
   int ret;
   int sd, ii, alert;
   gnutls_session session;
   char buffer[MAX_BUF + 1];
   gnutls_certificate_credentials 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)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;
}
```

6.3.6 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 <gnutls/gnutls.h>
#include <gnutls/extra.h>

/* Those functions are defined in other examples.
 */
extern void check_alert(gnutls_session 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()
{
   int ret;
   int sd, ii;
   gnutls_session session;
   char buffer[MAX_BUF + 1];
   gnutls_srp_client_credentials srp_cred;
   gnutls_certificate_client_credentials 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_client_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)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, 0);

 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;
}
```

6.4 Server examples

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

6.4.1 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.
 */


#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 x509_cred;

gnutls_session initialize_tls_session()
{
   gnutls_session 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 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()
{
   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 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)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 (;;) {
         bzero(buffer, 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");
      gnutls_bye( session, GNUTLS_SHUT_WR); /* do not wait for
                               * the peer to close the connection.
                               */

      close(sd);
      gnutls_deinit(session);

   }
   close(listen_sd);

   gnutls_certificate_free_credentials(x509_cred);

   gnutls_global_deinit();

   return 0;

}
```

6.4.2 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 cert_cred;

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

#define TLS_SESSION_CACHE 50

gnutls_session initialize_tls_session()
{
   gnutls_session 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 dh_params;
/* Export-grade cipher suites require temporary RSA
 * keys.
 */
gnutls_rsa_params 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()
{
```

```
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 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)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 (;;) {
      bzero(buffer, 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");
      gnutls_bye(session, GNUTLS_SHUT_WR);      /* do not wait for
                                                   * the peer to close the connection.
                                                  */

      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 key, gnutls_datum 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 wrap_db_fetch(void *dbf, gnutls_datum key)
{
   gnutls_datum 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 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;

}
```

6.4.3 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 cred;
const int cert_type_priority[2] = { GNUTLS_CRT_OPENPGP, 0 };
gnutls_dh_params dh_params;

/* Defined in a previous example */
extern int generate_dh_params( void);
extern gnutls_session initialize_tls_session( void);

int main()
{
   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 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)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 (;;) {
   bzero(buffer, 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");
gnutls_bye( session, GNUTLS_SHUT_WR); /* do not wait for
                           * the peer to close the connection.
                           */

close(sd);
gnutls_deinit(session);
```

```
   }
   close(listen_sd);

   gnutls_certificate_free_credentials( cred);

   gnutls_global_deinit();

   return 0;

}
```

6.4.4 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 srp_cred;
gnutls_certificate_credentials cert_cred;

gnutls_session initialize_tls_session()
{
   gnutls_session 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()
{
   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 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)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 (;;) {
      bzero(buffer, 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");
   gnutls_bye( session, GNUTLS_SHUT_WR); /* do not wait for
                              * the peer to close the connection.
                              */

   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;
```

```
}
```

6.5 Miscellaneous examples

6.5.1 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 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));
   }
}
```

6.5.2 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];
unsigned char *_bin = bin;
char* print;

   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.
 */
static void print_x509_certificate_info(gnutls_session session)
{
   char serial[40];
   char dn[128];
   int i;
   size_t size;
   unsigned int algo, bits;
   time_t expiration_time, activation_time;
   const gnutls_datum *cert_list;
   int cert_list_size = 0;
   gnutls_x509_crt 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]);

   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);

   }
}
```

6.5.3 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 <gnutls/gnutls.h>
#include <gnutls/x509.h>
#include <time.h>

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

int main()
{
   gnutls_x509_crq crq;
   gnutls_x509_privkey key;
   unsigned char buffer[10*1024];
   int buffer_size = sizeof(buffer);
   int ret;

   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;
```

```
}
```

6.5.4 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 * cert, const gnutls_datum * pkcs8_key,
                 const char *password)
{
   gnutls_pkcs12 pkcs12;
   int ret, bag_index;
   gnutls_pkcs12_bag 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 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));
   exit(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));
   exit(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));
      exit(1);
   }

   fd = fopen(OUTFILE, "w");
   if (fd == NULL) {
      fprintf(stderr, "cannot open file\n");
      exit(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);

}
```

6.6 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.

Chapter 7

Included programs

7.1 The “srptool” program

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
  ```

7.2 The “gnutls-cli-debug” program

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.
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 RIPEMD160 MAC support (TLS extension)... 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
```

7.3 The “certtool” program

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 *doc/certtool.cfg*, in the distribution, for an example of a template file.

How to use certtool interactively:

- 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
```

Chapter 8

Function reference

8.1 *GnuTLS* library

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

8.1.1 _gnutls_deinit

void **_gnutls_deinit** (*gnutls_session* **session**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.

Description

This function clears all buffers associated with the *session*. The difference with **gnutls_deinit()** is that this function will not interfere with the session database.

8.1.2 gnutls_alert_get_name

*const char** **gnutls_alert_get_name** (*gnutls_alert_level* **alert**)

Arguments

- *gnutls_alert_level* **alert**: is an alert number *gnutls_session* structure.

Description

Returns a string that describes the given alert number or NULL. See **gnutls_alert_get()**.

8.1.3 gnutls_alert_get

gnutls_alert_description **gnutls_alert_get** (*gnutls_session* **session**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.

Description
Returns 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.

8.1.4 gnutls_alert_send

int **gnutls_alert_send** (*gnutls_session* **session**, *gnutls_alert_level* **level**, *gnutls_alert_description* **desc**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.
- *gnutls_alert_level* **level**: is the level of the alert
- *gnutls_alert_description* **desc**: is the alert description

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.

8.1.5 gnutls_anon_allocate_client_credentials

int **gnutls_anon_allocate_client_credentials** (*gnutls_anon_client_credentials ** **sc**)

Arguments

- *gnutls_anon_client_credentials ** **sc**: is a pointer to an *gnutls_anon_client_credentials* structure.

Description
This structure is complex enough to manipulate directly thus this helper function is provided in order to allocate it.

8.1.6 gnutls_anon_allocate_server_credentials

int **gnutls_anon_allocate_server_credentials** (*gnutls_anon_server_credentials ** **sc**)

Arguments

- *gnutls_anon_server_credentials ** **sc**: is a pointer to an *gnutls_anon_server_credentials* structure.

Description

This structure is complex enough to manipulate directly thus this helper function is provided in order to allocate it.

8.1.7 gnutls_anon_free_client_credentials

void **gnutls_anon_free_client_credentials** (*gnutls_anon_client_credentials* **sc**)

Arguments

- *gnutls_anon_client_credentials* **sc**: is an *gnutls_anon_client_credentials* structure.

Description

This structure is complex enough to manipulate directly thus this helper function is provided in order to free (deallocate) it.

8.1.8 gnutls_anon_free_server_credentials

void **gnutls_anon_free_server_credentials** (*gnutls_anon_server_credentials* **sc**)

Arguments

- *gnutls_anon_server_credentials* **sc**: is an *gnutls_anon_server_credentials* structure.

Description

This structure is complex enough to manipulate directly thus this helper function is provided in order to free (deallocate) it.

8.1.9 gnutls_anon_set_params_function

void **gnutls_anon_set_params_function** (*gnutls_anon_server_credentials* **res**, *gnutls_params_function** **func**)

Arguments

- *gnutls_anon_server_credentials* **res**: is a gnutls_certificate_credentials structure

- *gnutls_params_function** **func**: is the function to be called

Description
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.

8.1.10 gnutls_anon_set_server_dh_params

void **gnutls_anon_set_server_dh_params** (*gnutls_anon_server_credentials* **res**, *gnutls_dh_params* **dh_params**)

Arguments

- *gnutls_anon_server_credentials* **res**: is a gnutls_anon_server_credentials structure
- *gnutls_dh_params* **dh_params**: is a structure that holds diffie hellman parameters.

Description
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.

8.1.11 gnutls_auth_get_type

gnutls_credentials_type **gnutls_auth_get_type** (*gnutls_session* **session**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.

Description
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.

8.1.12 gnutls_bye

int **gnutls_bye** (*gnutls_session* **session**, *gnutls_close_request* **how**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.
- *gnutls_close_request* **how**: is an integer

Description

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

8.1.13 gnutls_certificate_activation_time_peers

time_t **gnutls_certificate_activation_time_peers** (*gnutls_session* **session**)

Arguments

- *gnutls_session* **session**: is a gnutls session

Description

This function will return the peer's certificate activation time. This is the creation time for openpgp keys.
Returns (time_t) -1 on error.

8.1.14 gnutls_certificate_allocate_credentials

int **gnutls_certificate_allocate_credentials** (*gnutls_certificate_credentials* * **res**)

Arguments

- *gnutls_certificate_credentials* * **res**: is a pointer to an *gnutls_certificate_credentials* structure.

Description

This structure is complex enough to manipulate directly thus this helper function is provided in order to allocate it.
Returns 0 on success.

8.1.15 gnutls_certificate_client_get_request_status

int **gnutls_certificate_client_get_request_status** (*gnutls_session* **session**)

Arguments

- *gnutls_session* **session**: is a gnutls session

Description
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.

8.1.16 gnutls_certificate_client_set_retrieve_function

void **gnutls_certificate_client_set_retrieve_function** (*gnutls_certificate_credentials* **cred**, *gnutls_certificate_client_retrieve_function ** **func**)
Arguments

- *gnutls_certificate_credentials* **cred**: is a *gnutls_certificate_credentials* structure.

- *gnutls_certificate_client_retrieve_function ** **func**: is the callback function

Description
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, const gnutls_datum* req_ca_dn, int nreqs, gnutls_pk_algorithm* 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.

8.1.17 gnutls_certificate_client_set_select_function

void **gnutls_certificate_client_set_select_function** (*gnutls_session* **session**, *gnutls_certificate_client_select_function ** **func**)
Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.

- *gnutls_certificate_client_select_function ** **func**: is the callback function

Description

This function sets a callback to be called while selecting the (client) certificate. The callback's function prototype is: int (*callback)(gnutls_session, const gnutls_datum *client_cert, int ncerts, const gnutls_datum* req_ca_dn, int nreqs);
client_cert contains **ncerts** gnutls_datum structures which hold the raw certificates (DER for X.509 or binary for OpenPGP), of the client.
req_ca_dn, is only used in X.509 certificates. Contains a list with the CA names that the server considers trusted. Normally we should send a certificate that is signed by one of these CAs. These names are DER encoded. To get a more meaningful value use the function **gnutls_x509_rdn_get()**.
This function specifies what we, in case of a client, are going to do when we have to send a certificate. If this callback function is not provided then gnutls will automatically try to find an appropriate certificate to send. The appropriate certificate is chosen based on the CAs sent by the server, and the requested public key 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 return the index of the certificate choosen by the user. The index is relative to the certificates in the callback's parameter. The value (-1) indicates that the user does not want to use client authentication.

8.1.18 gnutls_certificate_expiration_time_peers

time_t **gnutls_certificate_expiration_time_peers** (*gnutls_session* **session**)

Arguments

- *gnutls_session* **session**: is a gnutls session

Description

This function will return the peer's certificate expiration time.
Returns (time_t) -1 on error.

8.1.19 gnutls_certificate_free_ca_names

void **gnutls_certificate_free_ca_names** (*gnutls_certificate_credentials* **sc**)

Arguments

- *gnutls_certificate_credentials* **sc**: is an *gnutls_certificate_credentials* structure.

Description

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.

8.1.20 gnutls_certificate_free_cas

void **gnutls_certificate_free_cas** (*gnutls_certificate_credentials* **sc**)

Arguments

- *gnutls_certificate_credentials* **sc**: is an *gnutls_certificate_credentials* structure.

Description

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

8.1.21 gnutls_certificate_free_credentials

void **gnutls_certificate_free_credentials** (*gnutls_certificate_credentials* **sc**)

Arguments

- *gnutls_certificate_credentials* **sc**: is an *gnutls_certificate_credentials* structure.

Description

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).

8.1.22 gnutls_certificate_free_crls

void **gnutls_certificate_free_crls** (*gnutls_certificate_credentials* **sc**)

Arguments

- *gnutls_certificate_credentials* **sc**: is an *gnutls_certificate_credentials* structure.

Description

This function will delete all the CRLs associated with the given credentials.

8.1.23 gnutls_certificate_free_keys

void **gnutls_certificate_free_keys** (*gnutls_certificate_credentials* **sc**)

Arguments

- *gnutls_certificate_credentials* **sc**: is an *gnutls_certificate_credentials* structure.

Description

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.

8.1.24 gnutls_certificate_get_ours

*const gnutls_datum ** **gnutls_certificate_get_ours** (*gnutls_session* **session**)

Arguments

- *gnutls_session* **session**: is a gnutls session

Description

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.

8.1.25 gnutls_certificate_get_peers

*const gnutls_datum ** **gnutls_certificate_get_peers** (*gnutls_session* **session**, *unsigned int ** **list_size**)

Arguments

- *gnutls_session* **session**: is a gnutls session
- *unsigned int ** **list_size**: is the length of the certificate list

Description

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 raw encoded key is returned.
Returns NULL in case of an error, or if no certificate was sent.

8.1.26 gnutls_certificate_send_x509_rdn_sequence

void **gnutls_certificate_send_x509_rdn_sequence** (*gnutls_session* **session**, *int* **status**)

Arguments

- *gnutls_session* **session**: is a pointer to a *gnutls_session* structure.
- *int* **status**: is 0 or 1

Description

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.

8.1.27 gnutls_certificate_server_set_request

void **gnutls_certificate_server_set_request** (*gnutls_session* **session**, *gnutls_certificate_request* **req**)

Arguments

- *gnutls_session* **session**: is an *gnutls_session* structure.
- *gnutls_certificate_request* **req**: is one of GNUTLS_CERT_REQUEST, GNUTLS_CERT_REQUIRE

Description

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.

8.1.28 gnutls_certificate_server_set_retrieve_function

void **gnutls_certificate_server_set_retrieve_function** (*gnutls_certificate_credentials* **cred**, *gnutls_certificate_server_retrieve_function ** **func**)

Arguments

- *gnutls_certificate_credentials* **cred**: is a *gnutls_certificate_credentials* structure.
- *gnutls_certificate_server_retrieve_function ** **func**: is the callback function

Description

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, 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.

8.1.29 gnutls_certificate_server_set_select_function

void **gnutls_certificate_server_set_select_function** (*gnutls_session* **session**, *gnutls_certificate_server_select_function ** **func**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.
- *gnutls_certificate_server_select_function ** **func**: is the callback function

Description

This function sets a callback to be called while selecting the (server) certificate. The callback's function form is: int (*callback)(gnutls_session, gnutls_datum *server_cert, int ncerts);
server_cert contains **ncerts** gnutls_datum structures which hold the raw certificate (DER encoded in X.509) of the server.
This function specifies what we, in case of a server, are going to do when we have to send a certificate. If this callback function is not provided then gnutls will automatically try to find an appropriate certificate to send. (actually send the first in the list)
In case the callback returned a negative number then gnutls will not attempt to choose the appropriate certificate and the caller function will fail.
The callback function will only be called once per handshake. The callback function should return the index of the certificate choosen by the server. -1 indicates an error.

8.1.30 gnutls_certificate_set_dh_params

void **gnutls_certificate_set_dh_params** (*gnutls_certificate_credentials* **res**, *gnutls_dh_params* **dh_params**)

Arguments

- *gnutls_certificate_credentials* **res**: is a gnutls_certificate_credentials structure
- *gnutls_dh_params* **dh_params**: is a structure that holds diffie hellman parameters.

Description

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.

8.1.31 gnutls_certificate_set_params_function

void **gnutls_certificate_set_params_function** (*gnutls_certificate_credentials* **res**, *gnutls_params_function** **func**)

Arguments

- *gnutls_certificate_credentials* **res**: is a gnutls_certificate_credentials structure
- *gnutls_params_function** **func**: is the function to be called

Description

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.

8.1.32 gnutls_certificate_set_rsa_export_params

void **gnutls_certificate_set_rsa_export_params** (*gnutls_certificate_credentials* **res**, *gnutls_rsa_params* **rsa_params**)

Arguments

- *gnutls_certificate_credentials* **res**: is a gnutls_certificate_credentials structure
- *gnutls_rsa_params* **rsa_params**: is a structure that holds temporary RSA parameters.

Description

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

8.1.33 gnutls_certificate_set_verify_flags

void **gnutls_certificate_set_verify_flags** (*gnutls_certificate_credentials* **res**, *unsigned int* **flags**)

Arguments

- *gnutls_certificate_credentials* **res**: is a gnutls_certificate_credentials structure
- *unsigned int* **flags**: are the flagsis a structure that holds diffie hellman parameters.

Description

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.

8.1.34 gnutls_certificate_set_verify_limits

void **gnutls_certificate_set_verify_limits** (*gnutls_certificate_credentials* **res**, *unsigned int* **max_bits**, *unsigned int* **max_depth**)

Arguments

- *gnutls_certificate_credentials* **res**: is a gnutls_certificate_credentials structure
- *unsigned int* **max_bits**: is the number of bits of an acceptable certificate (default 8200)
- *unsigned int* **max_depth**: is maximum depth of the verification of a certificate chain (default 5)

Description
This function will set some upper limits for the default verification function (**gnutls_certificate_verify_peers()**) to avoid denial of service attacks.

8.1.35 gnutls_certificate_set_x509_crl_file

int **gnutls_certificate_set_x509_crl_file** (*gnutls_certificate_credentials* **res**, *const char ** **crlfile**, *gnutls_x509_crt_fmt* **type**)
Arguments

- *gnutls_certificate_credentials* **res**: is an *gnutls_certificate_credentials* structure.
- *const char ** **crlfile**: is a file containing the list of verified CRLs (DER or PEM list)
- *gnutls_x509_crt_fmt* **type**: is PEM or DER

Description
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_peers()**. This function may be called multiple times.
Returns the number of CRLs processed or a negative value on error.

8.1.36 gnutls_certificate_set_x509_crl_mem

int **gnutls_certificate_set_x509_crl_mem** (*gnutls_certificate_credentials* **res**, *const gnutls_datum ** **CRL**, *gnutls_x509_crt_fmt* **type**)
Arguments

- *gnutls_certificate_credentials* **res**: is an *gnutls_certificate_credentials* structure.
- *const gnutls_datum ** **CRL**: is a list of trusted CRLs. They should have been verified before.
- *gnutls_x509_crt_fmt* **type**: is DER or PEM

Description

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_peers()**. This function may be called multiple times.
Returns the number of CRLs processed or a negative value on error.

8.1.37 gnutls_certificate_set_x509_crl

int **gnutls_certificate_set_x509_crl** (*gnutls_certificate_credentials* **res**, *gnutls_x509_crl** **crl_list**, *int* **crl_list_size**)

Arguments

- *gnutls_certificate_credentials* **res**: is an *gnutls_certificate_credentials* structure.
- *gnutls_x509_crl** **crl_list**: is a list of trusted CRLs. They should have been verified before.
- *int* **crl_list_size**: holds the size of the crl_list

Description

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_peers()**. This function may be called multiple times.
Returns 0 on success.

8.1.38 gnutls_certificate_set_x509_key_file

int **gnutls_certificate_set_x509_key_file** (*gnutls_certificate_credentials* **res**, *const char ** **CERTFILE**, *const char ** **KEYFILE**, *gnutls_x509_crt_fmt* **type**)

Arguments

- *gnutls_certificate_credentials* **res**: is an *gnutls_certificate_credentials* structure.
- *const char ** **CERTFILE**: is a file that containing the certificate list (path) for the specified private key, in PKCS7 format, or a list of certificates
- *const char ** **KEYFILE**: is a file that contains the private key
- *gnutls_x509_crt_fmt* **type**: is PEM or DER

Description

This function sets a certificate/private key pair in the gnutls_certificate_credentials

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.

8.1.39 gnutls_certificate_set_x509_key_mem

int **gnutls_certificate_set_x509_key_mem** (*gnutls_certificate_credentials* **res**, *const gnutls_datum** **cert**, *const gnutls_datum** **key**, *gnutls_x509_crt_fmt* **type**)

Arguments

- *gnutls_certificate_credentials* **res**: is an *gnutls_certificate_credentials* structure.
- *const gnutls_datum** **cert**: contains a certificate list (path) for the specified private key
- *const gnutls_datum** **key**: is the private key
- *gnutls_x509_crt_fmt* **type**: is PEM or DER

Description
This function sets a certificate/private key pair in the gnutls_certificate_credentials 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.

8.1.40 gnutls_certificate_set_x509_key

int **gnutls_certificate_set_x509_key** (*gnutls_certificate_credentials* **res**, *gnutls_x509_crt ** **cert_list**, *int* **cert_list_size**, *gnutls_x509_privkey* **key**)

Arguments

- *gnutls_certificate_credentials* **res**: is an *gnutls_certificate_credentials* structure.
- *gnutls_x509_crt ** **cert_list**: contains a certificate list (path) for the specified private key

- *int* **cert_list_size**: holds the size of the certificate list
- *gnutls_x509_privkey* **key**: is a gnutls_x509_privkey key

Description

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

8.1.41 gnutls_certificate_set_x509_trust_file

int **gnutls_certificate_set_x509_trust_file** (*gnutls_certificate_credentials* **res**, *const char ** **cafile**, *gnutls_x509_crt_fmt* **type**)

Arguments

- *gnutls_certificate_credentials* **res**: is an *gnutls_certificate_credentials* structure.
- *const char ** **cafile**: is a file containing the list of trusted CAs (DER or PEM list)
- *gnutls_x509_crt_fmt* **type**: is PEM or DER

Description

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_peers()**. 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.

8.1.42 gnutls_certificate_set_x509_trust_mem

int **gnutls_certificate_set_x509_trust_mem** (*gnutls_certificate_credentials* **res**, *const gnutls_datum ** **ca**, *gnutls_x509_crt_fmt* **type**)

Arguments

- *gnutls_certificate_credentials* **res**: is an *gnutls_certificate_credentials* structure.
- *const gnutls_datum ** **ca**: is a list of trusted CAs or a DER certificate
- *gnutls_x509_crt_fmt* **type**: is DER or PEM

Description

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_peers()**. 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.

8.1.43 gnutls_certificate_set_x509_trust

int **gnutls_certificate_set_x509_trust** (*gnutls_certificate_credentials* **res**, *gnutls_x509_crt ** **ca_list**, *int* **ca_list_size**)

Arguments

- *gnutls_certificate_credentials* **res**: is an *gnutls_certificate_credentials* structure.
- *gnutls_x509_crt ** **ca_list**: is a list of trusted CAs
- *int* **ca_list_size**: holds the size of the CA list

Description

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_peers()**. 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.

8.1.44 gnutls_certificate_type_get_name

*const char ** **gnutls_certificate_type_get_name** (*gnutls_certificate_type* **type**)

Arguments

- *gnutls_certificate_type* **type**: is a certificate type

Description

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

8.1.45 gnutls_certificate_type_get

gnutls_certificate_type **gnutls_certificate_type_get** (*gnutls_session* **session**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.

Description

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

8.1.46 gnutls_certificate_type_set_priority

int **gnutls_certificate_type_set_priority** (*gnutls_session* **session**, *const int ** **list**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.
- *const int ** **list**: is a 0 terminated list of gnutls_certificate_type elements.

Description

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.

8.1.47 gnutls_certificate_verify_peers2

int **gnutls_certificate_verify_peers2** (*gnutls_session* **session**, *unsigned int ** **status**)

Arguments

- *gnutls_session* **session**: is a gnutls session
- *unsigned int ** **status**: is the output of the verification

Description

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 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.

8.1.48 gnutls_certificate_verify_peers

int **gnutls_certificate_verify_peers** (*gnutls_session* **session**)

Arguments

- *gnutls_session* **session**: is a gnutls session

Description
This function will try to verify the peer's certificate and return its status (trusted, invalid etc.). 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.
The return value should be one or more of the gnutls_certificate_status enumerated elements bitwise or'd, or a negative error code on error.
This is the same as **gnutls_x509_verify_certificate()** and uses the loaded CAs in the credentials as trusted CAs.

8.1.49 gnutls_check_version

*const char ** **gnutls_check_version** (*const char ** **req_version**)
Arguments

- *const char ** **req_version**: the version to check

Description
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.

8.1.50 gnutls_cipher_get_key_size

size_t **gnutls_cipher_get_key_size** (*gnutls_cipher_algorithm* **algorithm**)
Arguments

- *gnutls_cipher_algorithm* **algorithm**: is an encryption algorithm

Description
Returns the length (in bytes) of the given cipher's key size. Returns 0 if the given cipher is invalid.

8.1.51 gnutls_cipher_get_name

*const char ** **gnutls_cipher_get_name** (*gnutls_cipher_algorithm* **algorithm**)
Arguments

- *gnutls_cipher_algorithm* **algorithm**: is an encryption algorithm

Description

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

8.1.52 gnutls_cipher_get

gnutls_cipher_algorithm **gnutls_cipher_get** (*gnutls_session* **session**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.

Description

Returns the currently used cipher.

8.1.53 gnutls_cipher_set_priority

int **gnutls_cipher_set_priority** (*gnutls_session* **session**, *const int ** **list**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.
- *const int ** **list**: is a 0 terminated list of gnutls_cipher_algorithm elements.

Description

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.

8.1.54 gnutls_cipher_suite_get_name

*const char ** **gnutls_cipher_suite_get_name** (*gnutls_kx_algorithm* **kx_algorithm**, *gnutls_cipher_algorithm* **cipher_algorithm**, *gnutls_mac_algorithm* **mac_algorithm**)

Arguments

- *gnutls_kx_algorithm* **kx_algorithm**: is a Key exchange algorithm
- *gnutls_cipher_algorithm* **cipher_algorithm**: is a cipher algorithm
- *gnutls_mac_algorithm* **mac_algorithm**: is a MAC algorithm

Description

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.

8.1.55 gnutls_compression_get_name

*const char ** **gnutls_compression_get_name** (*gnutls_compression_method* **algorithm**)

Arguments

- *gnutls_compression_method* **algorithm**: is a Compression algorithm

Description

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

8.1.56 gnutls_compression_get

gnutls_compression_method **gnutls_compression_get** (*gnutls_session* **session**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.

Description

Returns the currently used compression method.

8.1.57 gnutls_compression_set_priority

int **gnutls_compression_set_priority** (*gnutls_session* **session**, *const int ** **list**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.
- *const int ** **list**: is a 0 terminated list of gnutls_compression_method elements.

Description

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.

8.1.58 gnutls_credentials_clear

void **gnutls_credentials_clear** (*gnutls_session* **session**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.

Description
Clears all the credentials previously set in this session.

8.1.59 gnutls_credentials_set

int **gnutls_credentials_set** (*gnutls_session* **session**, *gnutls_credentials_type* **type**, *void** **cred**)
Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.
- *gnutls_credentials_type* **type**: is the type of the credentials
- *void** **cred**: is a pointer to a structure.

Description
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 in case of a client. In case of a server it should be gnutls_anon_server_credentials.
For GNUTLS_CRD_SRP cred should be gnutls_srp_client_credentials in case of a client, and gnutls_srp_server_credentials, in case of a server.
For GNUTLS_CRD_CERTIFICATE cred should be gnutls_certificate_credentials.

8.1.60 gnutls_db_check_entry

int **gnutls_db_check_entry** (*gnutls_session* **session**, *gnutls_datum* **session_entry**)
Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.
- *gnutls_datum* **session_entry**: is the session data (not key)

Description
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.

8.1.61 gnutls_db_get_ptr

*void** **gnutls_db_get_ptr** (*gnutls_session* **session**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.

Description

Returns the pointer that will be sent to db store, retrieve and delete functions, as the first argument.

8.1.62 gnutls_db_remove_session

void **gnutls_db_remove_session** (*gnutls_session* **session**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.

Description

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.

8.1.63 gnutls_db_set_cache_expiration

void **gnutls_db_set_cache_expiration** (*gnutls_session* **session**, *int* **seconds**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.
- *int* **seconds**: is the number of seconds.

Description

Sets the expiration time for resumed sessions. The default is 3600 (one hour) at the time writing this.

8.1.64 gnutls_db_set_ptr

void **gnutls_db_set_ptr** (*gnutls_session* **session**, *void** **ptr**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.
- *void** **ptr**: is the pointer

Description

Sets the pointer that will be provided to db store, retrieve and delete functions, as the first argument.

8.1.65 gnutls_db_set_remove_function

void **gnutls_db_set_remove_function** (*gnutls_session* **session**, *gnutls_db_remove_func* **rem_func**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.
- *gnutls_db_remove_func* **rem_func**: is the function.

Description

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_function()** will be null unless **gnutls_db_set_ptr()** has been called.

8.1.66 gnutls_db_set_retrieve_function

void **gnutls_db_set_retrieve_function** (*gnutls_session* **session**, *gnutls_db_retr_func* **retr_func**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.
- *gnutls_db_retr_func* **retr_func**: is the function.

Description

Sets the function that will be used to retrieve data from the resumed sessions database. This function must return a gnutls_datum containing the data on success, or a gnutls_datum containing null and 0 on failure.
The datum's data must be allocated using the function **gnutls_malloc()**.
The first argument to **store_function()** will be null unless **gnutls_db_set_ptr()** has been called.

8.1.67 gnutls_db_set_store_function

void **gnutls_db_set_store_function** (*gnutls_session* **session**, *gnutls_db_store_func* **store_func**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.

- *gnutls_db_store_func* **store_func**: is the function

Description
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_function()** will be null unless **gnutls_db_set_ptr()** has been called.

8.1.68 gnutls_deinit

void **gnutls_deinit** (*gnutls_session* **session**)
Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.

Description
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.

8.1.69 gnutls_dh_get_peers_public_bits

int **gnutls_dh_get_peers_public_bits** (*gnutls_session* **session**)
Arguments

- *gnutls_session* **session**: is a gnutls session

Description
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.

8.1.70 gnutls_dh_get_prime_bits

int **gnutls_dh_get_prime_bits** (*gnutls_session* **session**)
Arguments

- *gnutls_session* **session**: is a gnutls session

Description
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.

8.1.71 gnutls_dh_get_secret_bits

int **gnutls_dh_get_secret_bits** (*gnutls_session* **session**)

Arguments

- *gnutls_session* **session**: is a gnutls session

Description

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.

8.1.72 gnutls_dh_params_cpy

int **gnutls_dh_params_cpy** (*gnutls_dh_params* **dst**, *gnutls_dh_params* **src**)

Arguments

- *gnutls_dh_params* **dst**: Is the destination structure, which should be initialized.
- *gnutls_dh_params* **src**: Is the source structure

Description

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

8.1.73 gnutls_dh_params_deinit

void **gnutls_dh_params_deinit** (*gnutls_dh_params* **dh_params**)

Arguments

- *gnutls_dh_params* **dh_params**: Is a structure that holds the prime numbers

Description

This function will deinitialize the DH parameters structure.

8.1.74 gnutls_dh_params_export_pkcs3

int **gnutls_dh_params_export_pkcs3** (*gnutls_dh_params* **params**, *gnutls_x509_crt_fmt* **format**, *unsigned char** **params_data**, *size_t** **params_data_size**)

Arguments

- *gnutls_dh_params* **params**: Holds the DH parameters
- *gnutls_x509_crt_fmt* **format**: the format of output params. One of PEM or DER.

- *unsigned char** **params_data**: will contain a PKCS3 DHParams structure PEM or DER encoded
- *size_t** **params_data_size**: holds the size of params_data (and will be replaced by the actual size of parameters)

Description

This function will export the given 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.

8.1.75 gnutls_dh_params_export_raw

int **gnutls_dh_params_export_raw** (*gnutls_dh_params* **params**, *gnutls_datum ** **prime**, *gnutls_datum ** **generator**, *unsigned int ** **bits**)

Arguments

- *gnutls_dh_params* **params**: Holds the DH parameters
- *gnutls_datum ** **prime**: will hold the new prime
- *gnutls_datum ** **generator**: will hold the new generator
- *unsigned int ** **bits**: if non null will hold is the prime's number of bits

Description

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.

8.1.76 gnutls_dh_params_generate2

int **gnutls_dh_params_generate2** (*gnutls_dh_params* **params**, *unsigned int* **bits**)

Arguments

- *gnutls_dh_params* **params**: Is the structure that the DH parameters will be stored
- *unsigned int* **bits**: is the prime's number of bits

Description

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.

8.1.77 gnutls_dh_params_import_pkcs3

int **gnutls_dh_params_import_pkcs3** (*gnutls_dh_params* **params**, *const gnutls_datum ** **pkcs3_params**, *gnutls_x509_crt_fmt* **format**)

Arguments

- *gnutls_dh_params* **params**: A structure where the parameters will be copied to
- *const gnutls_datum ** **pkcs3_params**: should contain a PKCS3 DHParams structure PEM or DER encoded
- *gnutls_x509_crt_fmt* **format**: the format of params. PEM or DER.

Description

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.

8.1.78 gnutls_dh_params_import_raw

int **gnutls_dh_params_import_raw** (*gnutls_dh_params* **dh_params**, *const gnutls_datum ** **prime**, *const gnutls_datum** **generator**)

Arguments

- *gnutls_dh_params* **dh_params**: Is a structure that will hold the prime numbers
- *const gnutls_datum ** **prime**: holds the new prime
- *const gnutls_datum** **generator**: holds the new generator

Description

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.

8.1.79 gnutls_dh_params_init

int **gnutls_dh_params_init** (*gnutls_dh_params ** **dh_params**)

Arguments

- *gnutls_dh_params ** **dh_params**: Is a structure that will hold the prime numbers

Description

This function will initialize the DH parameters structure.

8.1.80 gnutls_dh_set_prime_bits

void **gnutls_dh_set_prime_bits** (*gnutls_session* **session**, *unsigned int* **bits**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.

- *unsigned int* **bits**: is the number of bits

Description

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.

8.1.81 gnutls_error_is_fatal

int **gnutls_error_is_fatal** (*int* **error**)

Arguments

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

Description

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.

8.1.82 gnutls_error_to_alert

int **gnutls_error_to_alert** (*int* **err**, *int** **level**)

Arguments

- *int* **err**: is a negative integer
- *int** **level**: the alert level will be stored there

Description

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.

8.1.83 gnutls_fingerprint

int **gnutls_fingerprint** (*gnutls_digest_algorithm* **algo**, *const gnutls_datum** **data**, *void** **result**, *size_t** **result_size**)

Arguments

- *gnutls_digest_algorithm* **algo**: is a digest algorithm
- *const gnutls_datum** **data**: is the data
- *void** **result**: is the place where the result will be copied (may be null).
- *size_t** **result_size**: should hold the size of the result. The actual size of the returned result will also be copied there.

Description

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.

8.1.84 gnutls_free

void **gnutls_free** (*void** **ptr**)

Arguments

- *void** **ptr**:

Description

This function will free data pointed by ptr.
The deallocation function used is the one set by **gnutls_global_set_mem_functions()**.

8.1.85 gnutls_global_deinit

void **gnutls_global_deinit** (**void**)

Arguments

- **void**:

Description

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

8.1.86 gnutls_global_init

int **gnutls_global_init** (**void**)

Arguments

- **void**:

Description

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.

8.1.87 gnutls_global_set_log_function

void **gnutls_global_set_log_function** (*gnutls_log_func* **log_func**)

Arguments

- *gnutls_log_func* **log_func**: it's a log function

Description

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*);

8.1.88 gnutls_global_set_log_level

void **gnutls_global_set_log_level** (*int* **level**)

Arguments

- *int* **level**: it's an integer from 0 to 9.

Description

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.

8.1.89 gnutls_global_set_mem_functions

void **gnutls_global_set_mem_functions** (*void ** **(*gnutls_alloc_func**)

Arguments

- *void ** **(*gnutls_alloc_func**:

Description

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.

8.1.90 gnutls_handshake_get_last_in

gnutls_handshake_description **gnutls_handshake_get_last_in** (*gnutls_session* **session**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.

Description

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.

8.1.91 gnutls_handshake_get_last_out

gnutls_handshake_description **gnutls_handshake_get_last_out** (*gnutls_session* **session**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.

Description

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.

8.1.92 gnutls_handshake_set_max_packet_length

void **gnutls_handshake_set_max_packet_length** (*gnutls_session* **session**, *int* **max**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.
- *int* **max**: is the maximum number.

Description

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.

8.1.93 gnutls_handshake_set_private_extensions

void **gnutls_handshake_set_private_extensions** (*gnutls_session* **session**, *int* **allow**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.
- *int* **allow**: is an integer (0 or 1)

Description

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.

8.1.94 gnutls_handshake

int **gnutls_handshake** (*gnutls_session* **session**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.

Description

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_D
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.

8.1.95 gnutls_init

int **gnutls_init** (*gnutls_session ** **session**, *gnutls_connection_end* **con_end**)

Arguments

- *gnutls_session ** **session**: is a pointer to a *gnutls_session* structure.
- *gnutls_connection_end* **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.

Description

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.

8.1.96 gnutls_kx_get_name

*const char ** **gnutls_kx_get_name** (*gnutls_kx_algorithm* **algorithm**)

Arguments

- *gnutls_kx_algorithm* **algorithm**: is a key exchange algorithm

Description

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

8.1.97 gnutls_kx_get

gnutls_kx_algorithm **gnutls_kx_get** (*gnutls_session* **session**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.

Description

Returns the key exchange algorithm used in the last handshake.

8.1.98 gnutls_kx_set_priority

int **gnutls_kx_set_priority** (*gnutls_session* **session**, *const int ** **list**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.
- *const int ** **list**: is a 0 terminated list of gnutls_kx_algorithm elements.

Description

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.

8.1.99 gnutls_mac_get_name

*const char ** **gnutls_mac_get_name** (*gnutls_mac_algorithm* **algorithm**)

Arguments

- *gnutls_mac_algorithm* **algorithm**: is a MAC algorithm

Description

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

8.1.100 gnutls_mac_get

gnutls_mac_algorithm **gnutls_mac_get** (*gnutls_session* **session**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.

Description

Returns the currently used mac algorithm.

8.1.101 gnutls_mac_set_priority

int **gnutls_mac_set_priority** (*gnutls_session* **session**, *const int ** **list**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.
- *const int ** **list**: is a 0 terminated list of gnutls_mac_algorithm elements.

Description

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.

8.1.102 gnutls_malloc

*void** **gnutls_malloc** (*size_t* **s**)

Arguments

- *size_t* **s**:

Description

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()**.

8.1.103 gnutls_openpgp_send_key

void **gnutls_openpgp_send_key** (*gnutls_session* **session**, *gnutls_openpgp_key_status* **status**)

Arguments

- *gnutls_session* **session**: is a pointer to a *gnutls_session* structure.
- *gnutls_openpgp_key_status* **status**: is one of OPENPGP_KEY, or OPENPGP_KEY_FINGERPRINT

Description

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.

8.1.104 gnutls_pem_base64_decode_alloc

int **gnutls_pem_base64_decode_alloc** (*const char** **header**, *const gnutls_datum ** **b64_data**, *gnutls_datum** **result**)

Arguments

- *const char** **header**: The PEM header (eg. CERTIFICATE)
- *const gnutls_datum ** **b64_data**: contains the encoded data
- *gnutls_datum** **result**: the place where decoded data lie

Description

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.

8.1.105 gnutls_pem_base64_decode

int **gnutls_pem_base64_decode** (*const char** **header**, *const gnutls_datum ** **b64_data**, *unsigned char** **result**, *size_t** **result_size**)

Arguments

- *const char** **header**: A null terminated string with the PEM header (eg. CERTIFICATE)
- *const gnutls_datum ** **b64_data**: contain the encoded data
- *unsigned char** **result**: the place where decoded data will be copied
- *size_t** **result_size**: holds the size of the result

Description

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.

8.1.106 gnutls_pem_base64_encode_alloc

int **gnutls_pem_base64_encode_alloc** (*const char** **msg**, *const gnutls_datum ** **data**, *gnutls_datum** **result**)

Arguments

- *const char** **msg**: is a message to be put in the encoded header
- *const gnutls_datum ** **data**: contains the raw data
- *gnutls_datum** **result**: will hold the newly allocated encoded data

Description

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.

8.1.107 gnutls_pem_base64_encode

int **gnutls_pem_base64_encode** (*const char** **msg**, *const gnutls_datum ** **data**, *char** **result**, *size_t** **result_size**)

Arguments

- *const char** **msg**: is a message to be put in the header
- *const gnutls_datum ** **data**: contain the raw data
- *char** **result**: the place where base64 data will be copied
- *size_t** **result_size**: holds the size of the result

Description

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.

8.1.108 gnutls_perror

void **gnutls_perror** (*int* **error**)

Arguments

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

Description

This function is like **perror()**. The only difference is that it accepts an error number returned by a gnutls function.

8.1.109 gnutls_pk_algorithm_get_name

*const char ** **gnutls_pk_algorithm_get_name** (*gnutls_pk_algorithm* **algorithm**)

Arguments

- *gnutls_pk_algorithm* **algorithm**: is a pk algorithm

Description

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

8.1.110 gnutls_protocol_get_name

*const char ** **gnutls_protocol_get_name** (*gnutls_protocol_version* **version**)

Arguments

- *gnutls_protocol_version* **version**: is a (gnutls) version number

Description

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

8.1.111 gnutls_protocol_get_version

gnutls_protocol_version **gnutls_protocol_get_version** (*gnutls_session* **session**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.

Description

Returns the version of the currently used protocol.

8.1.112 gnutls_protocol_set_priority

int **gnutls_protocol_set_priority** (*gnutls_session* **session**, *const int ** **list**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.
- *const int ** **list**: is a 0 terminated list of gnutls_protocol_version elements.

Description

Sets the priority on the protocol versions supported by gnutls. This function actually enables or disables protocols. Newer protocol versions always have highest priority.

8.1.113 gnutls_record_check_pending

size_t **gnutls_record_check_pending** (*gnutls_session* **session**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.

Description

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).

8.1.114 gnutls_record_get_direction

int **gnutls_record_get_direction** (*gnutls_session* **session**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.

Description

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.

8.1.115 gnutls_record_get_max_size

size_t **gnutls_record_get_max_size** (*gnutls_session* **session**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.

Description

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.

8.1.116 gnutls_record_recv

ssize_t **gnutls_record_recv** (*gnutls_session* **session**, *void ** **data**, *size_t* **sizeofdata**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.
- *void ** **data**: contains the data to send
- *size_t* **sizeofdata**: is the length of the data

Description

This function has the similar semantics to **send()**. The only difference is that it accepts a GNUTLS session.
If the 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.
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.

8.1.117 gnutls_record_send

ssize_t **gnutls_record_send** (*gnutls_session* **session**, *const void ** **data**, *size_t* **sizeofdata**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.
- *const void ** **data**: contains the data to send
- *size_t* **sizeofdata**: is the length of the data

Description

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 EINTR is returned by the internal push function (the default is **recv()**) then GNUTLS_E_INTERRUPTED will be returned. If GNUTLS_E_INTERRUPTED or GNUTLS_E_AGAIN is returned, you must call this function again, with the same parameters; cf. **gnutls_record_get_direction()**. Alternatively you could provide a NULL pointer for data, and 0 for size. Otherwise the write operation will be corrupted and the connection will be terminated.
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.

8.1.118 gnutls_record_set_max_size

ssize_t **gnutls_record_set_max_size** (*gnutls_session* **session**, *size_t* **size**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.
- *size_t* **size**: is the new size

Description

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.

8.1.119 gnutls_rehandshake

int **gnutls_rehandshake** (*gnutls_session* **session**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.

Description

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.

8.1.120 gnutls_rsa_export_get_modulus_bits

int **gnutls_rsa_export_get_modulus_bits** (*gnutls_session* **session**)

Arguments

- *gnutls_session* **session**: is a gnutls session

Description
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.

8.1.121 gnutls_rsa_params_cpy

int **gnutls_rsa_params_cpy** (*gnutls_rsa_params* **dst**, *gnutls_rsa_params* **src**)
Arguments

- *gnutls_rsa_params* **dst**: Is the destination structure, which should be initialized.
- *gnutls_rsa_params* **src**: Is the source structure

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

8.1.122 gnutls_rsa_params_deinit

void **gnutls_rsa_params_deinit** (*gnutls_rsa_params* **rsa_params**)
Arguments

- *gnutls_rsa_params* **rsa_params**: Is a structure that holds the parameters

Description
This function will deinitialize the RSA parameters structure.

8.1.123 gnutls_rsa_params_export_pkcs1

int **gnutls_rsa_params_export_pkcs1** (*gnutls_rsa_params* **params**, *gnutls_x509_crt_fmt* **format**, *unsigned char** **params_data**, *size_t** **params_data_size**)
Arguments

- *gnutls_rsa_params* **params**: Holds the RSA parameters
- *gnutls_x509_crt_fmt* **format**: the format of output params. One of PEM or DER.
- *unsigned char** **params_data**: will contain a PKCS1 RSAPublicKey structure PEM or DER encoded
- *size_t** **params_data_size**: holds the size of params_data (and will be replaced by the actual size of parameters)

Description

This function will export the given RSA parameters to a PKCS1 RSAPublicKey structure. If the buffer provided is not long enough to hold the output, then GNUTLS_E_SHORT_MEMORY_BUFFER will be returned.
If the structure is PEM encoded, it will have a header of "BEGIN RSA PRIVATE KEY".
In case of failure a negative value will be returned, and 0 on success.

8.1.124 gnutls_rsa_params_export_raw

int **gnutls_rsa_params_export_raw** (*gnutls_rsa_params* **params**, *gnutls_datum ** **m**, *gnutls_datum ** **e**, *gnutls_datum ** **d**, *gnutls_datum ** **p**, *gnutls_datum** **q**, *gnutls_datum** **u**, *unsigned int ** **bits**)

Arguments

- *gnutls_rsa_params* **params**: a structure that holds the rsa parameters
- *gnutls_datum ** **m**: will hold the modulus
- *gnutls_datum ** **e**: will hold the public exponent
- *gnutls_datum ** **d**: will hold the private exponent
- *gnutls_datum ** **p**: will hold the first prime (p)
- *gnutls_datum** **q**: will hold the second prime (q)
- *gnutls_datum** **u**: will hold the coefficient
- *unsigned int ** **bits**: if non null will hold the prime's number of bits

Description

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

8.1.125 gnutls_rsa_params_generate2

int **gnutls_rsa_params_generate2** (*gnutls_rsa_params* **params**, *unsigned int* **bits**)

Arguments

- *gnutls_rsa_params* **params**: The structure where the parameters will be stored
- *unsigned int* **bits**: is the prime's number of bits

Description

This function will generate new temporary RSA parameters for use in RSA-EXPORT ciphersuites. This function is normally slow.
Note that if the parameters are to be used in export cipher suites the bits value should be 512 or less. Also note that the generation of new RSA parameters is only useful to servers. Clients use the parameters sent by the server, thus it's no use calling this in client side.

8.1.126 gnutls_rsa_params_import_pkcs1

int **gnutls_rsa_params_import_pkcs1** (*gnutls_rsa_params* **params**, *const gnutls_datum* * **pkcs1_params**, *gnutls_x509_crt_fmt* **format**)

Arguments

- *gnutls_rsa_params* **params**: A structure where the parameters will be copied to
- *const gnutls_datum* * **pkcs1_params**: should contain a PKCS1 RSAPublicKey structure PEM or DER encoded
- *gnutls_x509_crt_fmt* **format**: the format of params. PEM or DER.

Description

This function will extract the RSAPublicKey found in a PKCS1 formatted structure.
If the structure is PEM encoded, it should have a header of "BEGIN RSA PRIVATE KEY".
In case of failure a negative value will be returned, and 0 on success.

8.1.127 gnutls_rsa_params_import_raw

int **gnutls_rsa_params_import_raw** (*gnutls_rsa_params* **rsa_params**, *const gnutls_datum* * **m**, *const gnutls_datum* * **e**, *const gnutls_datum* * **d**, *const gnutls_datum* * **p**, *const gnutls_datum* * **q**, *const gnutls_datum* * **u**)

Arguments

- *gnutls_rsa_params* **rsa_params**: Is a structure will hold the parameters
- *const gnutls_datum* * **m**: holds the modulus
- *const gnutls_datum* * **e**: holds the public exponent
- *const gnutls_datum* * **d**: holds the private exponent
- *const gnutls_datum* * **p**: holds the first prime (p)
- *const gnutls_datum* * **q**: holds the second prime (q)

- *const gnutls_datum ** **u**: holds the coefficient

Description

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

8.1.128 gnutls_rsa_params_init

int **gnutls_rsa_params_init** (*gnutls_rsa_params ** **rsa_params**)

Arguments

- *gnutls_rsa_params ** **rsa_params**: Is a structure that will hold the parameters

Description

This function will initialize the temporary RSA parameters structure.

8.1.129 gnutls_server_name_get

int **gnutls_server_name_get** (*gnutls_session* **session**, *void ** **data**, *size_t ** **data_length**, *unsigned int ** **type**, *unsigned int* **indx**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.
- *void ** **data**: will hold the data
- *size_t ** **data_length**: will hold the data length. Must hold the maximum size of data.
- *unsigned int ** **type**: will hold the server name indicator type
- *unsigned int* **indx**: is the index of the server_name

Description

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.

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.

8.1.130 gnutls_server_name_set

int **gnutls_server_name_set** (*gnutls_session* **session**, *gnutls_server_name_type* **type**, *const void ** **name**, *size_t* **name_length**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.
- *gnutls_server_name_type* **type**: specifies the indicator type
- *const void ** **name**: is a string that contains the server name.
- *size_t* **name_length**: holds the length of name

Description

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.

8.1.131 gnutls_session_get_data

int **gnutls_session_get_data** (*gnutls_session* **session**, *void** **session_data**, *size_t ** **session_data_size**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.
- *void** **session_data**: is a pointer to space to hold the session.
- *size_t ** **session_data_size**: is the session_data's size, or it will be set by the function.

Description

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.

8.1.132 gnutls_session_get_id

int **gnutls_session_get_id** (*gnutls_session* **session**, *void** **session_id**, *size_t ** **session_id_size**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.
- *void** **session_id**: is a pointer to space to hold the session id.
- *size_t* * **session_id_size**: is the session id's size, or it will be set by the function.

Description

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.

8.1.133 gnutls_session_get_ptr

*void** **gnutls_session_get_ptr** (*gnutls_session* **session**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.

Description

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

8.1.134 gnutls_session_is_resumed

int **gnutls_session_is_resumed** (*gnutls_session* **session**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.

Description

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

8.1.135 gnutls_session_set_data

int **gnutls_session_set_data** (*gnutls_session* **session**, *const void** **session_data**, *size_t* **session_data_size**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.
- *const void** **session_data**: is a pointer to space to hold the session.
- *size_t* **session_data_size**: is the session's size

Description

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.

8.1.136 gnutls_session_set_ptr

void **gnutls_session_set_ptr** (*gnutls_session* **session**, *void** **ptr**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.
- *void** **ptr**: is the user pointer

Description

This function will set (assosiate) the user given pointer to the session structure. This is pointer can be accessed with **gnutls_session_get_ptr()**.

8.1.137 gnutls_set_default_export_priority

int **gnutls_set_default_export_priority** (*gnutls_session* **session**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.

Description

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.

8.1.138 gnutls_set_default_priority

int **gnutls_set_default_priority** (*gnutls_session* **session**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.

Description

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_256_CBC, AES_128_CBC, 3DES_CBC, and ARCFOUR_128 for ciphers.

8.1.139 gnutls_sign_algorithm_get_name

*const char ** **gnutls_sign_algorithm_get_name** (*gnutls_sign_algorithm* **algorithm**)

Arguments

- *gnutls_sign_algorithm* **algorithm**: is a sign algorithm

Description

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

8.1.140 gnutls_strerror

*const char** **gnutls_strerror** (*int* **error**)

Arguments

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

Description

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.

8.1.141 gnutls_transport_get_ptr2

void **gnutls_transport_get_ptr2** (*gnutls_session* **session**, *gnutls_transport_ptr ** **recv_ptr**, *gnutls_transport_ptr ** **send_ptr**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.
- *gnutls_transport_ptr ** **recv_ptr**: will hold the value for the pull function
- *gnutls_transport_ptr ** **send_ptr**: will hold the value for the push function

Description

Used to get the arguments of the transport functions (like PUSH and PULL). These should have been set using **gnutls_transport_set_ptr2()**.

8.1.142 gnutls_transport_get_ptr

gnutls_transport_ptr **gnutls_transport_get_ptr** (*gnutls_session* **session**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.

Description

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

8.1.143 gnutls_transport_set_lowat

void **gnutls_transport_set_lowat** (*gnutls_session* **session**, *int* **num**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.
- *int* **num**: is the low water value.

Description

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.

8.1.144 gnutls_transport_set_ptr2

void **gnutls_transport_set_ptr2** (*gnutls_session* **session**, *gnutls_transport_ptr* **recv_ptr**, *gnutls_transport_ptr* **send_ptr**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.
- *gnutls_transport_ptr* **recv_ptr**: is the value for the pull function
- *gnutls_transport_ptr* **send_ptr**: is the value for the push function

Description

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.

8.1.145 gnutls_transport_set_ptr

void **gnutls_transport_set_ptr** (*gnutls_session* **session**, *gnutls_transport_ptr* **ptr**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.
- *gnutls_transport_ptr* **ptr**: is the value.

Description

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.

8.1.146 gnutls_transport_set_pull_function

void **gnutls_transport_set_pull_function** (*gnutls_session* **session**, *gnutls_pull_func* **pull_func**)

Arguments

- *gnutls_session* **session**: gnutls session
- *gnutls_pull_func* **pull_func**: it's a function like read

Description

This is the function where you set a function for gnutls to receive data. Normally, if you use berkeley style sockets, you may not use this function since the default (recv(2)) will probably be ok. This function should be called once and after **gnutls_global_init()**. PULL_FUNC is of the form, ssize_t (*gnutls_pull_func)(gnutls_transport_ptr, const void*, size_t);

8.1.147 gnutls_transport_set_push_function

void **gnutls_transport_set_push_function** (*gnutls_session* **session**, *gnutls_push_func* **push_func**)

Arguments

- *gnutls_session* **session**: gnutls session
- *gnutls_push_func* **push_func**: it's a function like write

Description

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 may not 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.

This function should be called once and after **gnutls_global_init()**. PUSH_FUNC is of the form, ssize_t (*gnutls_push_func)(gnutls_transport_ptr, const void*, size_t);

8.2 *GnuTLS* X.509 certificate handling

The following functions are to be used for X.509 certificate handling. Their prototypes lie in "gnutls/x509.h".

8.2.1 gnutls_pkcs12_bag_decrypt

int **gnutls_pkcs12_bag_decrypt** (*gnutls_pkcs12_bag* **bag**, *const char** **pass**)

Arguments

- *gnutls_pkcs12_bag* **bag**: The bag
- *const char** **pass**: The password used for encryption. This can only be ASCII.

Description

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

8.2.2 gnutls_pkcs12_bag_deinit

void **gnutls_pkcs12_bag_deinit** (*gnutls_pkcs12_bag* **bag**)

Arguments

- *gnutls_pkcs12_bag* **bag**: The structure to be initialized

Description

This function will deinitialize a PKCS12 Bag structure.

8.2.3 gnutls_pkcs12_bag_encrypt

int **gnutls_pkcs12_bag_encrypt** (*gnutls_pkcs12_bag* **bag**, *const char** **pass**, *unsigned int* **flags**)

Arguments

- *gnutls_pkcs12_bag* **bag**: The bag
- *const char** **pass**: The password used for encryption. This can only be ASCII.
- *unsigned int* **flags**: should be one of gnutls_pkcs_encrypt_flags elements bitwise or'd

Description

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

8.2.4 gnutls_pkcs12_bag_get_count

int **gnutls_pkcs12_bag_get_count** (*gnutls_pkcs12_bag* **bag**)

Arguments

- *gnutls_pkcs12_bag* **bag**: The bag

Description

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

8.2.5 gnutls_pkcs12_bag_get_data

int **gnutls_pkcs12_bag_get_data** (*gnutls_pkcs12_bag* **bag**, *int* **indx**, *gnutls_datum * **data**)

Arguments

- *gnutls_pkcs12_bag* **bag**: The bag
- *int* **indx**: The element of the bag to get the data from
- *gnutls_datum ** **data**: where the bag's data will be. Should be treated as constant.

Description

This function will return the bag's data. The data is a constant that is stored into the bag. Should not be accessed after the bag is deleted.
Returns 0 on success and a negative error code on error.

8.2.6 gnutls_pkcs12_bag_get_friendly_name

int **gnutls_pkcs12_bag_get_friendly_name** (*gnutls_pkcs12_bag* **bag**, *int* **indx**, *char *** **name**)

Arguments

- *gnutls_pkcs12_bag* **bag**: The bag
- *int* **indx**: The bag's element to add the id
- *char *** **name**: will hold a pointer to the name (to be treated as const)

Description

This function will return the friendly name, of the specified bag element. The key ID is usually used to distinguish the local private key and the certificate pair.
Returns 0 on success, or a negative value on error.

8.2.7 gnutls_pkcs12_bag_get_key_id

int **gnutls_pkcs12_bag_get_key_id** (*gnutls_pkcs12_bag* **bag**, *int* **indx**, *gnutls_datum** **id**)

Arguments

- *gnutls_pkcs12_bag* **bag**: The bag
- *int* **indx**: The bag's element to add the id
- *gnutls_datum** **id**: where the ID will be copied (to be treated as const)

Description

This function will return the key ID, of the specified bag element. The key ID is usually used to distinguish the local private key and the certificate pair.
Returns 0 on success, or a negative value on error.

8.2.8 gnutls_pkcs12_bag_get_type

gnutls_pkcs12_bag_type **gnutls_pkcs12_bag_get_type** (*gnutls_pkcs12_bag* **bag**, *int* **indx**)

Arguments

- *gnutls_pkcs12_bag* **bag**: The bag
- *int* **indx**: The element of the bag to get the type

Description

This function will return the bag's type. One of the gnutls_pkcs12_bag_type enumerations.

8.2.9 gnutls_pkcs12_bag_init

int **gnutls_pkcs12_bag_init** (*gnutls_pkcs12_bag ** **bag**)

Arguments

- *gnutls_pkcs12_bag ** **bag**: The structure to be initialized

Description

This function will initialize a PKCS12 bag structure. PKCS12 Bags usually contain private keys, lists of X.509 Certificates and X.509 Certificate revocation lists.
Returns 0 on success.

8.2.10 gnutls_pkcs12_bag_set_crl

int **gnutls_pkcs12_bag_set_crl** (*gnutls_pkcs12_bag* **bag**, *gnutls_x509_crl* **crl**)

Arguments

- *gnutls_pkcs12_bag* **bag**: The bag
- *gnutls_x509_crl* **crl**: the CRL to be copied.

Description

This function will insert the given CRL into the bag. This is just a wrapper over **gnutls_pkcs12_bag_set_data()**.
Returns the index of the added bag on success, or a negative value on failure.

8.2.11 gnutls_pkcs12_bag_set_crt

int **gnutls_pkcs12_bag_set_crt** (*gnutls_pkcs12_bag* **bag**, *gnutls_x509_crt* **crt**)

Arguments

- *gnutls_pkcs12_bag* **bag**: The bag
- *gnutls_x509_crt* **crt**: the certificate to be copied.

Description

This function will insert the given certificate into the bag. This is just a wrapper over **gnutls_pkcs12_bag_set_data()**.
Returns the index of the added bag on success, or a negative value on failure.

8.2.12 gnutls_pkcs12_bag_set_data

int **gnutls_pkcs12_bag_set_data** (*gnutls_pkcs12_bag* **bag**, *gnutls_pkcs12_bag_type* **type**, *const gnutls_datum** **data**)

Arguments

- *gnutls_pkcs12_bag* **bag**: The bag
- *gnutls_pkcs12_bag_type* **type**: The data's type
- *const gnutls_datum** **data**: the data to be copied.

Description

This function will insert the given data of the given type into the bag.
Returns the index of the added bag on success, or a negative value on error.

8.2.13 gnutls_pkcs12_bag_set_friendly_name

int **gnutls_pkcs12_bag_set_friendly_name** (*gnutls_pkcs12_bag* **bag**, *int* **indx**, *const char** **name**)

Arguments

- *gnutls_pkcs12_bag* **bag**: The bag
- *int* **indx**: The bag's element to add the id
- *const char** **name**: the name

Description

This function will add the given key friendly name, to the specified, by the index, bag element. The name will be encoded as a 'Friendly name' bag attribute, which is usually used to set a user name to the local private key and the certificate pair.
Returns 0 on success, or a negative value on error.

8.2.14 gnutls_pkcs12_bag_set_key_id

int **gnutls_pkcs12_bag_set_key_id** (*gnutls_pkcs12_bag* **bag**, *int* **indx**, *const gnutls_datum** **id**)

Arguments

- *gnutls_pkcs12_bag* **bag**: The bag
- *int* **indx**: The bag's element to add the id
- *const gnutls_datum** **id**: the ID

Description

This function will add the given key ID, to the specified, by the index, bag element. The key ID will be encoded as a 'Local key identifier' bag attribute, which is usually used to distinguish the local private key and the certificate pair.
Returns 0 on success, or a negative value on error.

8.2.15 gnutls_pkcs12_deinit

void **gnutls_pkcs12_deinit** (*gnutls_pkcs12* **pkcs12**)

Arguments

- *gnutls_pkcs12* **pkcs12**: The structure to be initialized

Description

This function will deinitialize a PKCS12 structure.

8.2.16 gnutls_pkcs12_export

int **gnutls_pkcs12_export** (*gnutls_pkcs12* **pkcs12**, *gnutls_x509_crt_fmt* **format**, *void** **output_data**, *size_t** **output_data_size**)

Arguments

- *gnutls_pkcs12* **pkcs12**: Holds the pkcs12 structure
- *gnutls_x509_crt_fmt* **format**: the format of output params. One of PEM or DER.
- *void** **output_data**: will contain a structure PEM or DER encoded
- *size_t** **output_data_size**: holds the size of output_data (and will be replaced by the actual size of parameters)

Description

This function will export the pkcs12 structure to DER or PEM format.
If the buffer provided is not long enough to hold the output, then GNUTLS_E_SHORT_MEMORY_BUFFER will be returned.
If the structure is PEM encoded, it will have a header of "BEGIN PKCS12".
In case of failure a negative value will be returned, and 0 on success.

8.2.17 gnutls_pkcs12_generate_mac

int **gnutls_pkcs12_generate_mac** (*gnutls_pkcs12* **pkcs12**, *const char** **pass**)

Arguments

- *gnutls_pkcs12* **pkcs12**:
- *const char** **pass**: The password for the MAC

Description

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

8.2.18 gnutls_pkcs12_get_bag

int **gnutls_pkcs12_get_bag** (*gnutls_pkcs12* **pkcs12**, *int* **indx**, *gnutls_pkcs12_bag* **bag**)

Arguments

- *gnutls_pkcs12* **pkcs12**:
- *int* **indx**: contains the index of the bag to extract
- *gnutls_pkcs12_bag* **bag**: An initialized bag, where the contents of the bag will be copied

Description

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.

8.2.19 gnutls_pkcs12_import

int **gnutls_pkcs12_import** (*gnutls_pkcs12* **pkcs12**, *const gnutls_datum ** **data**, *gnutls_x509_crt_fmt* **format**, *unsigned int* **flags**)

Arguments

- *gnutls_pkcs12* **pkcs12**: The structure to store the parsed PKCS12.
- *const gnutls_datum ** **data**: The DER or PEM encoded PKCS12.
- *gnutls_x509_crt_fmt* **format**: One of DER or PEM
- *unsigned int* **flags**: an ORed sequence of gnutls_privkey_pkcs8_flags

Description

This function will convert the given DER or PEM encoded PKCS12 to the native gnutls_pkcs12 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.

8.2.20 gnutls_pkcs12_init

int **gnutls_pkcs12_init** (*gnutls_pkcs12 ** **pkcs12**)

Arguments

- *gnutls_pkcs12 ** **pkcs12**: The structure to be initialized

Description

This function will initialize a PKCS12 structure. PKCS12 structures usually contain lists of X.509 Certificates and X.509 Certificate revocation lists.
Returns 0 on success.

8.2.21 gnutls_pkcs12_set_bag

int **gnutls_pkcs12_set_bag** (*gnutls_pkcs12* **pkcs12**, *gnutls_pkcs12_bag* **bag**)

Arguments

- *gnutls_pkcs12* **pkcs12**:
- *gnutls_pkcs12_bag* **bag**: An initialized bag

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

8.2.22 gnutls_pkcs12_verify_mac

int **gnutls_pkcs12_verify_mac** (*gnutls_pkcs12* **pkcs12**, *const char** **pass**)
Arguments

- *gnutls_pkcs12* **pkcs12**:
- *const char** **pass**: The password for the MAC

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

8.2.23 gnutls_pkcs7_deinit

void **gnutls_pkcs7_deinit** (*gnutls_pkcs7* **pkcs7**)
Arguments

- *gnutls_pkcs7* **pkcs7**: The structure to be initialized

Description
This function will deinitialize a PKCS7 structure.

8.2.24 gnutls_pkcs7_delete_crl

int **gnutls_pkcs7_delete_crl** (*gnutls_pkcs7* **pkcs7**, *int* **indx**)
Arguments

- *gnutls_pkcs7* **pkcs7**:
- *int* **indx**: the index of the crl to delete

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

8.2.25 gnutls_pkcs7_delete_crt

int **gnutls_pkcs7_delete_crt** (*gnutls_pkcs7* **pkcs7**, *int* **indx**)
Arguments

- *gnutls_pkcs7* **pkcs7**:
- *int* **indx**: the index of the certificate to delete

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

8.2.26 gnutls_pkcs7_export

int **gnutls_pkcs7_export** (*gnutls_pkcs7* **pkcs7**, *gnutls_x509_crt_fmt* **format**, *void** **output_data**, *size_t** **output_data_size**)

Arguments

- *gnutls_pkcs7* **pkcs7**: Holds the pkcs7 structure
- *gnutls_x509_crt_fmt* **format**: the format of output params. One of PEM or DER.
- *void** **output_data**: will contain a structure PEM or DER encoded
- *size_t** **output_data_size**: holds the size of output_data (and will be replaced by the actual size of parameters)

Description
This function will export the pkcs7 structure to DER or PEM format.
If the buffer provided is not long enough to hold the output, then GNUTLS_E_SHORT_MEMORY_BUFFE
will be returned.
If the structure is PEM encoded, it will have a header of "BEGIN PKCS7".
In case of failure a negative value will be returned, and 0 on success.

8.2.27 gnutls_pkcs7_get_crl_count

int **gnutls_pkcs7_get_crl_count** (*gnutls_pkcs7* **pkcs7**)

Arguments

- *gnutls_pkcs7* **pkcs7**:

Description
This function will return the number of certifcates in the PKCS7 or RFC2630 crl set.
Returns a negative value on failure.

8.2.28 gnutls_pkcs7_get_crl_raw

int **gnutls_pkcs7_get_crl_raw** (*gnutls_pkcs7* **pkcs7**, *int* **indx**, *void** **crl**, *size_t** **crl_size**)

Arguments

- *gnutls_pkcs7* **pkcs7**:
- *int* **indx**: contains the index of the crl to extract
- *void** **crl**: the contents of the crl will be copied there (may be null)
- *size_t** **crl_size**: should hold the size of the crl

Description
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 GNUTLS_E_SHORT_MEMORY_BUFFER is returned.
After the last crl has been read GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE will be returned.

8.2.29 gnutls_pkcs7_get_crt_count

int **gnutls_pkcs7_get_crt_count** (*gnutls_pkcs7* **pkcs7**)
Arguments

- *gnutls_pkcs7* **pkcs7**:

Description
This function will return the number of certifcates in the PKCS7 or RFC2630 certificate set.
Returns a negative value on failure.

8.2.30 gnutls_pkcs7_get_crt_raw

int **gnutls_pkcs7_get_crt_raw** (*gnutls_pkcs7* **pkcs7**, *int* **indx**, *void** **certificate**, *size_t** **certificate_size**)
Arguments

- *gnutls_pkcs7* **pkcs7**:
- *int* **indx**: contains the index of the certificate to extract
- *void** **certificate**: the contents of the certificate will be copied there (may be null)
- *size_t** **certificate_size**: should hold the size of the certificate

Description
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 GNUTLS_E_SHORT_MEMORY_BUFFER is returned.
After the last certificate has been read GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE will be returned.

8.2.31 gnutls_pkcs7_import

int **gnutls_pkcs7_import** (*gnutls_pkcs7* **pkcs7**, *const gnutls_datum ** **data**, *gnutls_x509_crt_fmt* **format**)

Arguments

- *gnutls_pkcs7* **pkcs7**: The structure to store the parsed PKCS7.
- *const gnutls_datum ** **data**: The DER or PEM encoded PKCS7.
- *gnutls_x509_crt_fmt* **format**: One of DER or PEM

Description

This function will convert the given DER or PEM encoded PKCS7 to the native gnutls_pkcs7 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.

8.2.32 gnutls_pkcs7_init

int **gnutls_pkcs7_init** (*gnutls_pkcs7 ** **pkcs7**)

Arguments

- *gnutls_pkcs7 ** **pkcs7**: The structure to be initialized

Description

This function will initialize a PKCS7 structure. PKCS7 structures usually contain lists of X.509 Certificates and X.509 Certificate revocation lists.
Returns 0 on success.

8.2.33 gnutls_pkcs7_set_crl_raw

int **gnutls_pkcs7_set_crl_raw** (*gnutls_pkcs7* **pkcs7**, *const gnutls_datum** **crl**)

Arguments

- *gnutls_pkcs7* **pkcs7**:
- *const gnutls_datum** **crl**: the DER encoded crl to be added

Description

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

8.2.34 gnutls_pkcs7_set_crl

int **gnutls_pkcs7_set_crl** (*gnutls_pkcs7* **pkcs7**, *gnutls_x509_crl* **crl**)

Arguments

- *gnutls_pkcs7* **pkcs7**:
- *gnutls_x509_crl* **crl**: the DER encoded crl to be added

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

8.2.35 gnutls_pkcs7_set_crt_raw

int **gnutls_pkcs7_set_crt_raw** (*gnutls_pkcs7* **pkcs7**, *const gnutls_datum** **crt**)
Arguments

- *gnutls_pkcs7* **pkcs7**:
- *const gnutls_datum** **crt**: the DER encoded certificate to be added

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

8.2.36 gnutls_pkcs7_set_crt

int **gnutls_pkcs7_set_crt** (*gnutls_pkcs7* **pkcs7**, *gnutls_x509_crt* **crt**)
Arguments

- *gnutls_pkcs7* **pkcs7**:
- *gnutls_x509_crt* **crt**: the certificate to be copied.

Description
This function will add a parsed certificate to the PKCS7 or RFC2630 certificate set. This is a wrapper function over **gnutls_pkcs7_set_crt_raw()** .
Returns 0 on success.

8.2.37 gnutls_x509_crl_check_issuer

int **gnutls_x509_crl_check_issuer** (*gnutls_x509_crl* **cert**, *gnutls_x509_crt* **issuer**)
Arguments

- *gnutls_x509_crl* **cert**:
- *gnutls_x509_crt* **issuer**: is the certificate of a possible issuer

Description

This function will check if the given CRL was issued by the given issuer certificate. It will return true (1) if the given CRL was issued by the given issuer, and false (0) if not.
A negative value is returned in case of an error.

8.2.38 gnutls_x509_crl_deinit

void **gnutls_x509_crl_deinit** (*gnutls_x509_crl* **crl**)

Arguments

- *gnutls_x509_crl* **crl**: The structure to be initialized

Description

This function will deinitialize a CRL structure.

8.2.39 gnutls_x509_crl_export

int **gnutls_x509_crl_export** (*gnutls_x509_crl* **crl**, *gnutls_x509_crt_fmt* **format**, *void** **output_data**, *size_t** **output_data_size**)

Arguments

- *gnutls_x509_crl* **crl**: Holds the revocation list
- *gnutls_x509_crt_fmt* **format**: the format of output params. One of PEM or DER.
- *void** **output_data**: will contain a private key PEM or DER encoded
- *size_t** **output_data_size**: holds the size of output_data (and will be replaced by the actual size of parameters)

Description

This function will export the revocation list to DER or PEM format.
If the buffer provided is not long enough to hold the output, then GNUTLS_E_SHORT_MEMORY_BUFFE
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.

8.2.40 gnutls_x509_crl_get_crt_count

int **gnutls_x509_crl_get_crt_count** (*gnutls_x509_crl* **crl**)

Arguments

- *gnutls_x509_crl* **crl**: should contain a gnutls_x509_crl structure

Description

This function will return the number of revoked certificates in the given CRL. Returns a negative value on failure.

8.2.41 gnutls_x509_crl_get_crt_serial

int **gnutls_x509_crl_get_crt_serial** (*gnutls_x509_crl* **crl**, *int* **index**, *unsigned char ** **serial**, *size_t ** **serial_size**, *time_t ** **time**)

Arguments

- *gnutls_x509_crl* **crl**: should contain a gnutls_x509_crl structure
- *int* **index**: the index of the certificate to extract (starting from 0)
- *unsigned char ** **serial**: where the serial number will be copied
- *size_t ** **serial_size**: initially holds the size of serial
- *time_t ** **time**: if non null, will hold the time this certificate was revoked

Description

This function will return the serial number of the specified, by the index, revoked certificate.
Returns a negative value on failure.

8.2.42 gnutls_x509_crl_get_dn_oid

int **gnutls_x509_crl_get_dn_oid** (*gnutls_x509_crl* **crl**, *int* **indx**, *void ** **oid**, *size_t ** **sizeof_oid**)

Arguments

- *gnutls_x509_crl* **crl**: should contain a gnutls_x509_crl structure
- *int* **indx**: Specifies which DN OID to send. Use zero to get the first one.
- *void ** **oid**: a pointer to a structure to hold the name (may be null)
- *size_t ** **sizeof_oid**: initially holds the size of 'oid'

Description

This function will extract the requested OID of the name of the CRL issuer, specified by the given index.
If oid is null then only the size will be filled.
Returns GNUTLS_E_SHORT_MEMORY_BUFFER if the provided buffer is not long enough, and in that case the sizeof_oid will be updated with the required size. On success 0 is returned.

8.2.43 gnutls_x509_crl_get_issuer_dn_by_oid

int **gnutls_x509_crl_get_issuer_dn_by_oid** (*gnutls_x509_crl* **crl**, *const char ** **oid**, *int* **indx**, *unsigned int* **raw_flag**, *void ** **buf**, *size_t ** **sizeof_buf**)

Arguments

- *gnutls_x509_crl* **crl**: should contain a gnutls_x509_crl structure
- *const char ** **oid**: holds an Object Identified in null terminated string
- *int* **indx**: In case multiple same OIDs exist in the RDN, this specifies which to send. Use zero to get the first one.
- *unsigned int* **raw_flag**: If non zero returns the raw DER data of the DN part.
- *void ** **buf**: a pointer to a structure to hold the peer's name (may be null)
- *size_t ** **sizeof_buf**: initially holds the size of **buf**

Description

This function will extract the part of the name of the CRL issuer specified by the given OID. The output will be encoded as described in RFC2253. The output string will be ASCII or UTF-8 encoded, depending on the certificate data.
Some helper macros with popular OIDs can be found in gnutls/x509.h If raw flag is 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.

8.2.44 gnutls_x509_crl_get_issuer_dn

int **gnutls_x509_crl_get_issuer_dn** (*gnutls_x509_crl* **crl**, *char ** **buf**, *size_t ** **sizeof_buf**)

Arguments

- *gnutls_x509_crl* **crl**: should contain a gnutls_x509_crl structure
- *char ** **buf**: a pointer to a structure to hold the peer's name (may be null)
- *size_t ** **sizeof_buf**: initially holds the size of **buf**

Description

This function will copy the name of the CRL issuer in the provided buffer. The name will be in the form "C=xxxx,O=yyyy,CN=zzzz" as described in RFC2253.

The output string will be ASCII or UTF-8 encoded, depending on the certificate data.
If buf is null then only the size will be filled.
Returns GNUTLS_E_SHORT_MEMORY_BUFFER if the provided buffer is not long enough, and in that case the sizeof_buf will be updated with the required size, and 0 on success.

8.2.45 gnutls_x509_crl_get_next_update

time_t **gnutls_x509_crl_get_next_update** (*gnutls_x509_crl* **crl**)

Arguments

- *gnutls_x509_crl* **crl**: should contain a gnutls_x509_crl structure

Description

This function will return the time the next CRL will be issued. This field is optional in a CRL so it might be normal to get an error instead.
Returns (time_t)-1 on error.

8.2.46 gnutls_x509_crl_get_signature_algorithm

int **gnutls_x509_crl_get_signature_algorithm** (*gnutls_x509_crl* **crl**)

Arguments

- *gnutls_x509_crl* **crl**: should contain a gnutls_x509_crl structure

Description

This function will return a value of the gnutls_sign_algorithm enumeration that is the signature algorithm.
Returns a negative value on error.

8.2.47 gnutls_x509_crl_get_this_update

time_t **gnutls_x509_crl_get_this_update** (*gnutls_x509_crl* **crl**)

Arguments

- *gnutls_x509_crl* **crl**: should contain a gnutls_x509_crl structure

Description

This function will return the time this CRL was issued.
Returns (time_t)-1 on error.

8.2.48 gnutls_x509_crl_get_version

int **gnutls_x509_crl_get_version** (*gnutls_x509_crl* **crl**)

Arguments

- *gnutls_x509_crl* **crl**: should contain a gnutls_x509_crl structure

Description

This function will return the version of the specified CRL.
Returns a negative value on error.

8.2.49 gnutls_x509_crl_import

int **gnutls_x509_crl_import** (*gnutls_x509_crl* **crl**, *const gnutls_datum ** **data**, *gnutls_x509_crt_fmt* **format**)

Arguments

- *gnutls_x509_crl* **crl**: The structure to store the parsed CRL.
- *const gnutls_datum ** **data**: The DER or PEM encoded CRL.
- *gnutls_x509_crt_fmt* **format**: One of DER or PEM

Description

This function will convert the given DER or PEM encoded CRL to the native gnutls_x509_crl 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.

8.2.50 gnutls_x509_crl_init

int **gnutls_x509_crl_init** (*gnutls_x509_crl ** **crl**)

Arguments

- *gnutls_x509_crl ** **crl**: The structure to be initialized

Description

This function will initialize a CRL structure. CRL stands for Certificate Revocation List. A revocation list usually contains lists of certificate serial numbers that have been revoked by an Authority. The revocation lists are always signed with the authority's private key.
Returns 0 on success.

8.2.51 gnutls_x509_crl_set_crt_serial

int **gnutls_x509_crl_set_crt_serial** (*gnutls_x509_crl* **crl**, *const void** **serial**, *size_t* **serial_size**, *time_t* **revocation_time**)

Arguments

- *gnutls_x509_crl* **crl**: should contain a gnutls_x509_crl structure

- *const void** **serial**: The revoked certificate's serial number
- *size_t* **serial_size**: Holds the size of the serial field.
- *time_t* **revocation_time**: The time this certificate was revoked

Description

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.

8.2.52 gnutls_x509_crl_set_crt

int **gnutls_x509_crl_set_crt** (*gnutls_x509_crl* **crl**, *gnutls_x509_crt* **crt**, *time_t* **revocation_time**)

Arguments

- *gnutls_x509_crl* **crl**: should contain a gnutls_x509_crl structure
- *gnutls_x509_crt* **crt**: should contain a gnutls_x509_crt structure with the revoked certificate
- *time_t* **revocation_time**: The time this certificate was revoked

Description

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.

8.2.53 gnutls_x509_crl_set_next_update

int **gnutls_x509_crl_set_next_update** (*gnutls_x509_crl* **crl**, *time_t* **exp_time**)

Arguments

- *gnutls_x509_crl* **crl**: should contain a gnutls_x509_crl structure
- *time_t* **exp_time**: The actual time

Description

This function will set the time this CRL will be updated.
Returns 0 on success, or a negative value in case of an error.

8.2.54 gnutls_x509_crl_set_this_update

int **gnutls_x509_crl_set_this_update** (*gnutls_x509_crl* **crl**, *time_t* **act_time**)

Arguments

- *gnutls_x509_crl* **crl**: should contain a gnutls_x509_crl structure

- *time_t* **act_time**: The actual time

Description
This function will set the time this CRL was issued.
Returns 0 on success, or a negative value in case of an error.

8.2.55 gnutls_x509_crl_set_version

int **gnutls_x509_crl_set_version** (*gnutls_x509_crl* **crl**, *unsigned int* **version**)
Arguments

- *gnutls_x509_crl* **crl**: should contain a gnutls_x509_crl structure
- *unsigned int* **version**: holds the version number. For CRLv1 crls must be 1.

Description
This function will set the version of the CRL. This must be one for CRL version 1, and so on. The CRLs generated by gnutls should have a version number of 2.
Returns 0 on success.

8.2.56 gnutls_x509_crl_sign

int **gnutls_x509_crl_sign** (*gnutls_x509_crl* **crl**, *gnutls_x509_crt* **issuer**, *gnutls_x509_privkey* **issuer_key**)
Arguments

- *gnutls_x509_crl* **crl**: should contain a gnutls_x509_crl structure
- *gnutls_x509_crt* **issuer**: is the certificate of the certificate issuer
- *gnutls_x509_privkey* **issuer_key**: holds the issuer's private key

Description
This function will sign the CRL with the issuer's private key, and will copy the issuer's information into the CRL.
This must be the last step in a certificate CRL since all the previously set parameters are now signed.
Returns 0 on success.

8.2.57 gnutls_x509_crl_verify

int **gnutls_x509_crl_verify** (*gnutls_x509_crl* **crl**, *gnutls_x509_crt ** **CA_list**, *int* **CA_list_length**, *unsigned int* **flags**, *unsigned int ** **verify**)
Arguments

- *gnutls_x509_crl* **crl**: is the crl to be verified
- *gnutls_x509_crt ** **CA_list**: is a certificate list that is considered to be trusted one
- *int* **CA_list_length**: holds the number of CA certificates in CA_list
- *unsigned int* **flags**: Flags that may be used to change the verification algorithm. Use OR of the gnutls_certificate_verify_flags enumerations.
- *unsigned int ** **verify**: will hold the crl verification output.

Description

This function will try to verify the given crl and return its status. See **gnutls_x509_crt_list_verify()** for a detailed description of return values.
Returns 0 on success and a negative value in case of an error.

8.2.58 gnutls_x509_crq_deinit

void **gnutls_x509_crq_deinit** (*gnutls_x509_crq* **crq**)

Arguments

- *gnutls_x509_crq* **crq**: The structure to be initialized

Description

This function will deinitialize a CRL structure.

8.2.59 gnutls_x509_crq_export

int **gnutls_x509_crq_export** (*gnutls_x509_crq* **crq**, *gnutls_x509_crt_fmt* **format**, *void** **output_data**, *size_t** **output_data_size**)

Arguments

- *gnutls_x509_crq* **crq**: Holds the request
- *gnutls_x509_crt_fmt* **format**: the format of output params. One of PEM or DER.
- *void** **output_data**: will contain a certificate request PEM or DER encoded
- *size_t** **output_data_size**: holds the size of output_data (and will be replaced by the actual size of parameters)

Description

This function will export the certificate request to a PKCS10
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 NEW CERTIFICATE REQUEST".
In case of failure a negative value will be returned, and 0 on success.

8.2.60 gnutls_x509_crq_get_challenge_password

int **gnutls_x509_crq_get_challenge_password** (*gnutls_x509_crq* **crq**, *char** **pass**, *size_t** **sizeof_pass**)

Arguments

- *gnutls_x509_crq* **crq**: should contain a gnutls_x509_crq structure
- *char** **pass**: will hold a null terminated password
- *size_t** **sizeof_pass**: Initially holds the size of **pass**.

Description

This function will return the challenge password in the request.
Returns 0 on success.

8.2.61 gnutls_x509_crq_get_dn_by_oid

int **gnutls_x509_crq_get_dn_by_oid** (*gnutls_x509_crq* **crq**, *const char** **oid**, *int* **indx**, *unsigned int* **raw_flag**, *void ** **buf**, *size_t ** **sizeof_buf**)

Arguments

- *gnutls_x509_crq* **crq**: should contain a gnutls_x509_crq structure
- *const char** **oid**: holds an Object Identified in null terminated string
- *int* **indx**: In case multiple same OIDs exist in the RDN, this specifies which to send. Use zero to get the first one.
- *unsigned int* **raw_flag**: If non zero returns the raw DER data of the DN part.
- *void ** **buf**: a pointer to a structure to hold the name (may be null)
- *size_t ** **sizeof_buf**: initially holds the size of **buf**

Description

This function will extract the part of the name of the Certificate request subject, specified by the given OID. The output will be encoded as described in RFC2253. The output string will be ASCII or UTF-8 encoded, depending on the certificate data.
Some helper macros with popular OIDs can be found in gnutls/x509.h If raw flag is 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.

8.2.62 gnutls_x509_crq_get_dn_oid

int **gnutls_x509_crq_get_dn_oid** (*gnutls_x509_crq* **crq**, *int* **indx**, *void ** **oid**, *size_t ** **sizeof_oid**)

Arguments

- *gnutls_x509_crq* **crq**: should contain a gnutls_x509_crq structure
- *int* **indx**: Specifies which DN OID to send. Use zero to get the first one.
- *void ** **oid**: a pointer to a structure to hold the name (may be null)
- *size_t ** **sizeof_oid**: initially holds the size of **oid**

Description

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

8.2.63 gnutls_x509_crq_get_dn

int **gnutls_x509_crq_get_dn** (*gnutls_x509_crq* **crq**, *char ** **buf**, *size_t ** **sizeof_buf**)

Arguments

- *gnutls_x509_crq* **crq**: should contain a gnutls_x509_crq structure
- *char ** **buf**: a pointer to a structure to hold the name (may be null)
- *size_t ** **sizeof_buf**: initially holds the size of **buf**

Description

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.

8.2.64 gnutls_x509_crq_get_pk_algorithm

int **gnutls_x509_crq_get_pk_algorithm** (*gnutls_x509_crq* **crq**, *unsigned int** **bits**)

Arguments

- *gnutls_x509_crq* **crq**: should contain a gnutls_x509_crq structure
- *unsigned int** **bits**: if bits is non null it will hold the size of the parameters' in bits

Description

This function will return the public key algorithm of a 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 enumeration on success, or a negative value on error.

8.2.65 gnutls_x509_crq_get_version

int **gnutls_x509_crq_get_version** (*gnutls_x509_crq* **crq**)

Arguments

- *gnutls_x509_crq* **crq**: should contain a gnutls_x509_crq structure

Description

This function will return the version of the specified Certificate request.
Returns a negative value on error.

8.2.66 gnutls_x509_crq_import

int **gnutls_x509_crq_import** (*gnutls_x509_crq* **crq**, *const gnutls_datum ** **data**, *gnutls_x509_crt_fmt* **format**)

Arguments

- *gnutls_x509_crq* **crq**: The structure to store the parsed certificate request.
- *const gnutls_datum ** **data**: The DER or PEM encoded certificate.
- *gnutls_x509_crt_fmt* **format**: One of DER or PEM

Description

This function will convert the given DER or PEM encoded Certificate to the native gnutls_x509_crq 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.

8.2.67 gnutls_x509_crq_init

int **gnutls_x509_crq_init** (*gnutls_x509_crq ** **crq**)

Arguments

- *gnutls_x509_crq ** **crq**: The structure to be initialized

Description

This function will initialize a PKCS10 certificate request structure.
Returns 0 on success.

8.2.68 gnutls_x509_crq_set_challenge_password

int **gnutls_x509_crq_set_challenge_password** (*gnutls_x509_crq* **crq**, *const char** **pass**)

Arguments

- *gnutls_x509_crq* **crq**: should contain a gnutls_x509_crq structure
- *const char** **pass**: holds a null terminated password

Description

This function will set a challenge password to be used when revoking the request.
Returns 0 on success.

8.2.69 gnutls_x509_crq_set_dn_by_oid

int **gnutls_x509_crq_set_dn_by_oid** (*gnutls_x509_crq* **crq**, *const char** **oid**, *unsigned int* **raw_flag**, *const void ** **data**, *unsigned int* **sizeof_data**)

Arguments

- *gnutls_x509_crq* **crq**: should contain a gnutls_x509_crq structure
- *const char** **oid**: holds an Object Identifier in a null terminated string
- *unsigned int* **raw_flag**: must be 0, or 1 if the data are DER encoded
- *const void ** **data**: a pointer to the input data
- *unsigned int* **sizeof_data**: holds the size of **data**

Description

This function will set the part of the name of the Certificate request subject, specified by the given OID. The input string should be ASCII or UTF-8 encoded. Some helper macros with popular OIDs can be found in gnutls/x509.h With this function you can only set the known OIDs. You can test for known OIDs using **gnutls_x509_dn_oid_known()**. For OIDs that are not known (by gnutls) you should properly DER encode your data, and call this function with raw_flag set.
Returns 0 on success.

8.2.70 gnutls_x509_crq_set_key

int **gnutls_x509_crq_set_key** (*gnutls_x509_crq* **crq**, *gnutls_x509_privkey* **key**)

Arguments

- *gnutls_x509_crq* **crq**: should contain a gnutls_x509_crq structure
- *gnutls_x509_privkey* **key**: holds a private key

Description

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.

8.2.71 gnutls_x509_crq_set_version

int **gnutls_x509_crq_set_version** (*gnutls_x509_crq* **crq**, *unsigned int* **version**)

Arguments

- *gnutls_x509_crq* **crq**: should contain a gnutls_x509_crq structure
- *unsigned int* **version**: holds the version number. For v1 Requests must be 1.

Description

This function will set the version of the certificate request. For version 1 requests this must be one.
Returns 0 on success.

8.2.72 gnutls_x509_crq_sign

int **gnutls_x509_crq_sign** (*gnutls_x509_crq* **crq**, *gnutls_x509_privkey* **key**)

Arguments

- *gnutls_x509_crq* **crq**: should contain a gnutls_x509_crq structure
- *gnutls_x509_privkey* **key**: holds a private key

Description

This function will sign the certificate request with a private key. This must be the same key as the one used in **gnutls_x509_crt_set_key()** since a certificate request is self signed.
This must be the last step in a certificate request generation since all the previously set parameters are now signed.
Returns 0 on success.

8.2.73 gnutls_x509_crt_check_hostname

int **gnutls_x509_crt_check_hostname** (*gnutls_x509_crt* **cert**, *const char ** **hostname**)

Arguments

- *gnutls_x509_crt* **cert**: should contain an gnutls_x509_crt structure
- *const char ** **hostname**: A null terminated string that contains a DNS name

Description

This function will check if the given certificate's subject matches the given hostname. This is a basic implementation of the matching described in RFC2818 (HTTPS), which takes into account wildcards, and the subject alternative name PKIX extension.
Returns non zero on success, and zero on failure.

8.2.74 gnutls_x509_crt_check_issuer

int **gnutls_x509_crt_check_issuer** (*gnutls_x509_crt* **cert**, *gnutls_x509_crt* **issuer**)

Arguments

- *gnutls_x509_crt* **cert**: is the certificate to be checked
- *gnutls_x509_crt* **issuer**: is the certificate of a possible issuer

Description

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.

8.2.75 gnutls_x509_crt_check_revocation

int **gnutls_x509_crt_check_revocation** (*gnutls_x509_crt* **cert**, *gnutls_x509_crl ** **crl_list**, *int* **crl_list_length**)

Arguments

- *gnutls_x509_crt* **cert**: should contain a gnutls_x509_crt structure
- *gnutls_x509_crl ** **crl_list**: should contain a list of gnutls_x509_crl structures
- *int* **crl_list_length**: the length of the crl_list

Description

This function will return check if the given certificate is revoked. It is assumed that the CRLs have been verified before.
Returns 0 if the certificate is NOT revoked, and 1 if it is. A negative value is returned on error.

8.2.76 gnutls_x509_crt_cpy_crl_dist_points

int **gnutls_x509_crt_cpy_crl_dist_points** (*gnutls_x509_crt* **dst**, *gnutls_x509_crt* **src**)

Arguments

- *gnutls_x509_crt* **dst**: should contain a gnutls_x509_crt structure
- *gnutls_x509_crt* **src**: the certificate where the dist points will be copied from

Description

This function will copy the CRL distribution points certificate extension, from the source to the destination certificate. This may be useful to copy from a CA certificate to issued ones.
Returns 0 on success.

8.2.77 gnutls_x509_crt_deinit

void **gnutls_x509_crt_deinit** (*gnutls_x509_crt* **cert**)

Arguments

- *gnutls_x509_crt* **cert**: The structure to be initialized

Description

This function will deinitialize a CRL structure.

8.2.78 gnutls_x509_crt_export

int **gnutls_x509_crt_export** (*gnutls_x509_crt* **cert**, *gnutls_x509_crt_fmt* **format**, *void** **output_data**, *size_t** **output_data_size**)

Arguments

- *gnutls_x509_crt* **cert**: Holds the certificate
- *gnutls_x509_crt_fmt* **format**: the format of output params. One of PEM or DER.
- *void** **output_data**: will contain a certificate PEM or DER encoded

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

Description

This function will export the certificate to DER or PEM format.
If the buffer provided is not long enough to hold the output, then GNUTLS_E_SHORT_MEMORY_BUFFER will be returned.
If the structure is PEM encoded, it will have a header of "BEGIN CERTIFICATE".
In case of failure a negative value will be returned, and 0 on success.

8.2.79 gnutls_x509_crt_get_activation_time

time_t **gnutls_x509_crt_get_activation_time** (*gnutls_x509_crt* **cert**)

Arguments

- *gnutls_x509_crt* **cert**: should contain a gnutls_x509_crt structure

Description

This function will return the time this Certificate was or will be activated.
Returns (time_t)-1 on error.

8.2.80 gnutls_x509_crt_get_authority_key_id

int **gnutls_x509_crt_get_authority_key_id** (*gnutls_x509_crt* **cert**, *void** **ret**, *size_t** **ret_size**, *unsigned int** **critical**)

Arguments

- *gnutls_x509_crt* **cert**: should contain a gnutls_x509_crt structure
- *void** **ret**:
- *size_t** **ret_size**:
- *unsigned int** **critical**: will be non zero if the extension is marked as critical (may be null)

Description

This function will return the X.509v3 certificate authority's key identifier. This is obtained by the X.509 Authority Key identifier extension field (2.5.29.35). Note that this function only returns the keyIdentifier field of the extension.
Returns 0 on success and a negative value in case of an error.

8.2.81 gnutls_x509_crt_get_ca_status

int **gnutls_x509_crt_get_ca_status** (*gnutls_x509_crt* **cert**, *unsigned int** **critical**)

Arguments

- *gnutls_x509_crt* **cert**: should contain a gnutls_x509_crt structure
- *unsigned int** **critical**: will be non zero if the extension is marked as critical

Description

This function will return certificates CA status, by reading the basicConstraints X.509 extension (2.5.29.19). If the certificate is a CA a positive value will be returned, or 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.

8.2.82 gnutls_x509_crt_get_crl_dist_points

int **gnutls_x509_crt_get_crl_dist_points** (*gnutls_x509_crt* **cert**, *unsigned int* **seq**, *void ** **ret**, *size_t ** **ret_size**, *unsigned int** **reason_flags**, *unsigned int ** **critical**)

Arguments

- *gnutls_x509_crt* **cert**: should contain a gnutls_x509_crt structure
- *unsigned int* **seq**: specifies the sequence number of the distribution point (0 for the first one, 1 for the second etc.)
- *void ** **ret**: is the place where the distribution point will be copied to
- *size_t ** **ret_size**: holds the size of ret.
- *unsigned int** **reason_flags**: Revocation reasons flags.
- *unsigned int ** **critical**: will be non zero if the extension is marked as critical (may be null)

Description

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_I GNUTLS_CRL_REASON_PRIVILEGE_WITHDRAWN, GNUTLS_CRL_REASON_AA_COMPROMISE 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 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.
If the certificate does not have an Alternative name with the specified sequence number then returns GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE;

8.2.83 gnutls_x509_crt_get_dn_by_oid

int **gnutls_x509_crt_get_dn_by_oid** (*gnutls_x509_crt* **cert**, *const char** **oid**, *int* **indx**, *unsigned int* **raw_flag**, *void ** **buf**, *size_t ** **sizeof_buf**)

Arguments

- *gnutls_x509_crt* **cert**: should contain a gnutls_x509_crt structure
- *const char** **oid**: holds an Object Identified in null terminated string
- *int* **indx**: In case multiple same OIDs exist in the RDN, this specifies which to send. Use zero to get the first one.
- *unsigned int* **raw_flag**: If non zero returns the raw DER data of the DN part.
- *void ** **buf**: a pointer to a structure to hold the name (may be null)
- *size_t ** **sizeof_buf**: initially holds the size of **buf**

Description

This function will extract the part of the name of the Certificate subject, specified by the given OID. The output 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.

8.2.84 gnutls_x509_crt_get_dn_oid

int **gnutls_x509_crt_get_dn_oid** (*gnutls_x509_crt* **cert**, *int* **indx**, *void ** **oid**, *size_t ** **sizeof_oid**)

Arguments

- *gnutls_x509_crt* **cert**: should contain a gnutls_x509_crt structure
- *int* **indx**: This specifies which OID to return. Use zero to get the first one.

- *void ** **oid**: a pointer to a buffer to hold the OID (may be null)
- *size_t ** **sizeof_oid**: initially holds the size of **oid**

Description

This function will extract the OIDs of the name of the Certificate subject specified by the given index.
If oid is null then only the size will be filled.
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.

8.2.85 gnutls_x509_crt_get_dn

int **gnutls_x509_crt_get_dn** (*gnutls_x509_crt* **cert**, *char ** **buf**, *size_t ** **sizeof_buf**)

Arguments

- *gnutls_x509_crt* **cert**: should contain a gnutls_x509_crt structure
- *char ** **buf**: a pointer to a structure to hold the name (may be null)
- *size_t ** **sizeof_buf**: initially holds the size of **buf**

Description

This function will copy the name of the Certificate in the provided buffer. The name will be in the form "C=xxxx,O=yyyy,CN=zzzz" as described in RFC2253. The output string will be ASCII or UTF-8 encoded, depending on the certificate data.
If **buf** is null then only the size will be filled.
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.

8.2.86 gnutls_x509_crt_get_expiration_time

time_t **gnutls_x509_crt_get_expiration_time** (*gnutls_x509_crt* **cert**)

Arguments

- *gnutls_x509_crt* **cert**: should contain a gnutls_x509_crt structure

Description

This function will return the time this Certificate was or will be expired.
Returns (time_t)-1 on error.

8.2.87 gnutls_x509_crt_get_extension_by_oid

int **gnutls_x509_crt_get_extension_by_oid** (*gnutls_x509_crt* **cert**, *const char** **oid**, *int* **indx**, *void** **buf**, *size_t ** **sizeof_buf**, *unsigned int ** **critical**)

Arguments

- *gnutls_x509_crt* **cert**: should contain a gnutls_x509_crt structure
- *const char** **oid**: holds an Object Identified in null terminated string
- *int* **indx**: In case multiple same OIDs exist in the extensions, this specifies which to send. Use zero to get the first one.
- *void** **buf**: a pointer to a structure to hold the name (may be null)
- *size_t ** **sizeof_buf**: initially holds the size of **buf**
- *unsigned int ** **critical**: will be non zero if the extension is marked as critical

Description

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

8.2.88 gnutls_x509_crt_get_extension_oid

int **gnutls_x509_crt_get_extension_oid** (*gnutls_x509_crt* **cert**, *int* **indx**, *void** **oid**, *size_t ** **sizeof_oid**)

Arguments

- *gnutls_x509_crt* **cert**: should contain a gnutls_x509_crt structure
- *int* **indx**: Specifies which extension OID to send. Use zero to get the first one.
- *void** **oid**: a pointer to a structure to hold the OID (may be null)
- *size_t ** **sizeof_oid**: initially holds the size of **oid**

Description

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

8.2.89 gnutls_x509_crt_get_fingerprint

int **gnutls_x509_crt_get_fingerprint** (*gnutls_x509_crt* **cert**, *gnutls_digest_algorithm* **algo**, *void ** **buf**, *size_t ** **sizeof_buf**)

Arguments

- *gnutls_x509_crt* **cert**: should contain a gnutls_x509_crt structure
- *gnutls_digest_algorithm* **algo**: is a digest algorithm
- *void ** **buf**: a pointer to a structure to hold the fingerprint (may be null)
- *size_t ** **sizeof_buf**: initially holds the size of **buf**

Description

This function will calculate and copy the certificate's fingerprint in the provided buffer.
If the buffer is null then only the size will be filled.
Returns GNUTLS_E_SHORT_MEMORY_BUFFER if the provided buffer is not long enough, and in that case the sizeof_buf will be updated with the required size. On success 0 is returned.

8.2.90 gnutls_x509_crt_get_issuer_dn_by_oid

int **gnutls_x509_crt_get_issuer_dn_by_oid** (*gnutls_x509_crt* **cert**, *const char** **oid**, *int* **indx**, *unsigned int* **raw_flag**, *void ** **buf**, *size_t ** **sizeof_buf**)

Arguments

- *gnutls_x509_crt* **cert**: should contain a gnutls_x509_crt structure
- *const char** **oid**: holds an Object Identified in null terminated string
- *int* **indx**: In case multiple same OIDs exist in the RDN, this specifies which to send. Use zero to get the first one.
- *unsigned int* **raw_flag**: If non zero returns the raw DER data of the DN part.
- *void ** **buf**: a pointer to a structure to hold the name (may be null)
- *size_t ** **sizeof_buf**: initially holds the size of **buf**

Description

This function will extract the part of the name of the Certificate issuer specified by the given OID. The output 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.

8.2.91 gnutls_x509_crt_get_issuer_dn_oid

int **gnutls_x509_crt_get_issuer_dn_oid** (*gnutls_x509_crt* **cert**, *int* **indx**, *void ** **oid**, *size_t ** **sizeof_oid**)

Arguments

- *gnutls_x509_crt* **cert**: should contain a gnutls_x509_crt structure
- *int* **indx**: This specifies which OID to return. Use zero to get the first one.
- *void ** **oid**: a pointer to a buffer to hold the OID (may be null)
- *size_t ** **sizeof_oid**: initially holds the size of **oid**

Description
This function will extract the OIDs of the name of the Certificate issuer specified by the given index.
If **oid** is null then only the size will be filled.
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.

8.2.92 gnutls_x509_crt_get_issuer_dn

int **gnutls_x509_crt_get_issuer_dn** (*gnutls_x509_crt* **cert**, *char ** **buf**, *size_t ** **sizeof_buf**)

Arguments

- *gnutls_x509_crt* **cert**: should contain a gnutls_x509_crt structure
- *char ** **buf**: a pointer to a structure to hold the name (may be null)
- *size_t ** **sizeof_buf**: initially holds the size of **buf**

Description
This function will copy the name of the Certificate issuer in the provided buffer. The name will be in the form "C=xxxx,O=yyyy,CN=zzzz" as described in RFC2253. The output string will be ASCII or UTF-8 encoded, depending on

the certificate data.
If **buf** is null then only the size will be filled.
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.

8.2.93 gnutls_x509_crt_get_key_id

int **gnutls_x509_crt_get_key_id** (*gnutls_x509_crt* **crt**, *unsigned int* **flags**, *unsigned char** **output_data**, *size_t** **output_data_size**)

Arguments

- *gnutls_x509_crt* **crt**: Holds the certificate
- *unsigned int* **flags**: should be 0 for now
- *unsigned char** **output_data**: will contain the key ID
- *size_t** **output_data_size**: holds the size of output_data (and will be replaced by the actual size of parameters)

Description

This function will return a unique ID the depends on the public key parameters. This ID can be used in checking whether a certificate corresponds to the given private key.
If the buffer provided is not long enough to hold the output, then GNUTLS_E_SHORT_MEMORY_BUFFE
will be returned. The output will normally be a SHA-1 hash output, which is 20 bytes.
In case of failure a negative value will be returned, and 0 on success.

8.2.94 gnutls_x509_crt_get_key_purpose_oid

int **gnutls_x509_crt_get_key_purpose_oid** (*gnutls_x509_crt* **cert**, *int* **indx**, *void ** **oid**, *size_t ** **sizeof_oid**, *unsigned int** **critical**)

Arguments

- *gnutls_x509_crt* **cert**: should contain a gnutls_x509_crt structure
- *int* **indx**: This specifies which OID to return. Use zero to get the first one.
- *void ** **oid**: a pointer to a buffer to hold the OID (may be null)
- *size_t ** **sizeof_oid**: initially holds the size of **oid**
- *unsigned int** **critical**:

Description

This function will extract the key purpose OIDs of the Certificate specified by the given index. These are stored in the Extended Key Usage extension (2.5.29.37) See the GNUTLS_KP_* definitions for human readable names.
If **oid** is null then only the size will be filled.
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.

8.2.95 gnutls_x509_crt_get_key_usage

int **gnutls_x509_crt_get_key_usage** (*gnutls_x509_crt* **cert**, *unsigned int ** **key_usage**, *unsigned int ** **critical**)

Arguments

- *gnutls_x509_crt* **cert**: should contain a gnutls_x509_crt structure
- *unsigned int ** **key_usage**: where the key usage bits will be stored
- *unsigned int ** **critical**: will be non zero if the extension is marked as critical

Description

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

8.2.96 gnutls_x509_crt_get_pk_algorithm

int **gnutls_x509_crt_get_pk_algorithm** (*gnutls_x509_crt* **cert**, *unsigned int** **bits**)

Arguments

- *gnutls_x509_crt* **cert**: should contain a gnutls_x509_crt structure
- *unsigned int** **bits**: if bits is non null it will hold the size of the parameters' in bits

Description

This function will return the public key algorithm of an X.509 certificate.

If bits is non null, it should have enough size to hold the parameters size in bits. For RSA the bits returned is the modulus. For DSA the bits returned are of the public exponent.
Returns a member of the gnutls_pk_algorithm enumeration on success, or a negative value on error.

8.2.97 gnutls_x509_crt_get_pk_dsa_raw

int **gnutls_x509_crt_get_pk_dsa_raw** (*gnutls_x509_crt* **crt**, *gnutls_datum ** **p**, *gnutls_datum ** **q**, *gnutls_datum ** **g**, *gnutls_datum ** **y**)

Arguments

- *gnutls_x509_crt* **crt**: Holds the certificate
- *gnutls_datum ** **p**: will hold the p
- *gnutls_datum ** **q**: will hold the q
- *gnutls_datum ** **g**: will hold the g
- *gnutls_datum ** **y**: will hold the y

Description
This function will export the DSA private key's parameters found in the given certificate. The new parameters will be allocated using **gnutls_malloc()** and will be stored in the appropriate datum.

8.2.98 gnutls_x509_crt_get_pk_rsa_raw

int **gnutls_x509_crt_get_pk_rsa_raw** (*gnutls_x509_crt* **crt**, *gnutls_datum ** **m**, *gnutls_datum ** **e**)

Arguments

- *gnutls_x509_crt* **crt**: Holds the certificate
- *gnutls_datum ** **m**: will hold the modulus
- *gnutls_datum ** **e**: will hold the public exponent

Description
This function will export the RSA private key's parameters found in the given structure. The new parameters will be allocated using **gnutls_malloc()** and will be stored in the appropriate datum.

8.2.99 gnutls_x509_crt_get_serial

int **gnutls_x509_crt_get_serial** (*gnutls_x509_crt* **cert**, *void** **result**, *size_t** **result_size**)

Arguments

- *gnutls_x509_crt* **cert**: should contain a gnutls_x509_crt structure
- *void** **result**: The place where the serial number will be copied
- *size_t** **result_size**: Holds the size of the result field.

Description

This function will return the X.509 certificate's serial number. This is obtained by the X509 Certificate serialNumber field. Serial is not always a 32 or 64bit number. Some CAs use large serial numbers, thus it may be wise to handle it as something opaque.
Returns 0 on success and a negative value in case of an error.

8.2.100 gnutls_x509_crt_get_signature_algorithm

int **gnutls_x509_crt_get_signature_algorithm** (*gnutls_x509_crt* **cert**)

Arguments

- *gnutls_x509_crt* **cert**: should contain a gnutls_x509_crt structure

Description

This function will return a value of the gnutls_sign_algorithm enumeration that is the signature algorithm.
Returns a negative value on error.

8.2.101 gnutls_x509_crt_get_subject_alt_name

int **gnutls_x509_crt_get_subject_alt_name** (*gnutls_x509_crt* **cert**, *unsigned int* **seq**, *void ** **ret**, *size_t ** **ret_size**, *unsigned int ** **critical**)

Arguments

- *gnutls_x509_crt* **cert**: should contain a gnutls_x509_crt structure
- *unsigned int* **seq**: specifies the sequence number of the alt name (0 for the first one, 1 for the second etc.)
- *void ** **ret**: is the place where the alternative name will be copied to
- *size_t ** **ret_size**: holds the size of ret.
- *unsigned int ** **critical**: will be non zero if the extension is marked as critical (may be null)

Description

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, or the type of alternative name if everything was ok. The type is one of the enumerated gnutls_x509_subject_alt_name.

If the certificate does not have an Alternative name with the specified sequence number then returns GNUTLS_E_REQUESTED_DATA_NOT_AVAILABLE;

8.2.102 gnutls_x509_crt_get_subject_key_id

int **gnutls_x509_crt_get_subject_key_id** (*gnutls_x509_crt* **cert**, *void** **ret**, *size_t** **ret_size**, *unsigned int** **critical**)

Arguments

- *gnutls_x509_crt* **cert**: should contain a gnutls_x509_crt structure
- *void** **ret**:
- *size_t** **ret_size**:
- *unsigned int** **critical**: will be non zero if the extension is marked as critical (may be null)

Description

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

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

8.2.103 gnutls_x509_crt_get_version

int **gnutls_x509_crt_get_version** (*gnutls_x509_crt* **cert**)

Arguments

- *gnutls_x509_crt* **cert**: should contain a gnutls_x509_crt structure

Description

This function will return the version of the specified Certificate.

Returns a negative value on error.

8.2.104 gnutls_x509_crt_import

int **gnutls_x509_crt_import** (*gnutls_x509_crt* **cert**, *const gnutls_datum ** **data**, *gnutls_x509_crt_fmt* **format**)

Arguments

- *gnutls_x509_crt* **cert**: The structure to store the parsed certificate.
- *const gnutls_datum ** **data**: The DER or PEM encoded certificate.
- *gnutls_x509_crt_fmt* **format**: One of DER or PEM

Description

This function will convert the given DER or PEM encoded Certificate to the native gnutls_x509_crt 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.

8.2.105 gnutls_x509_crt_init

int **gnutls_x509_crt_init** (*gnutls_x509_crt ** **cert**)

Arguments

- *gnutls_x509_crt ** **cert**: The structure to be initialized

Description

This function will initialize an X.509 certificate structure.
Returns 0 on success.

8.2.106 gnutls_x509_crt_list_verify

int **gnutls_x509_crt_list_verify** (*gnutls_x509_crt** **cert_list**, *int* **cert_list_length**, *gnutls_x509_crt ** **CA_list**, *int* **CA_list_length**, *gnutls_x509_crl** **CRL_list**, *int* **CRL_list_length**, *unsigned int* **flags**, *unsigned int ** **verify**)

Arguments

- *gnutls_x509_crt** **cert_list**: is the certificate list to be verified
- *int* **cert_list_length**: holds the number of certificate in cert_list
- *gnutls_x509_crt ** **CA_list**: is the CA list which will be used in verification
- *int* **CA_list_length**: holds the number of CA certificate in CA_list
- *gnutls_x509_crl** **CRL_list**: holds a list of CRLs.
- *int* **CRL_list_length**: the length of CRL list.
- *unsigned int* **flags**: Flags that may be used to change the verification algorithm. Use OR of the gnutls_certificate_verify_flags enumerations.
- *unsigned int ** **verify**: will hold the certificate verification output.

Description

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 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.

8.2.107 gnutls_x509_crt_set_activation_time

int **gnutls_x509_crt_set_activation_time** (*gnutls_x509_crt* **cert**, *time_t* **act_time**)

Arguments

- *gnutls_x509_crt* **cert**: should contain a gnutls_x509_crt structure
- *time_t* **act_time**: The actual time

Description

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.

8.2.108 gnutls_x509_crt_set_authority_key_id

int **gnutls_x509_crt_set_authority_key_id** (*gnutls_x509_crt* **cert**, *const void** **id**, *size_t* **id_size**)

Arguments

- *gnutls_x509_crt* **cert**: should contain a gnutls_x509_crt structure
- *const void** **id**: The key ID
- *size_t* **id_size**: Holds the size of the serial field.

Description

This function will set the X.509 certificate's authority key ID extension. Only the keyIdentifier field can be set with this function.
Returns 0 on success, or a negative value in case of an error.

8.2.109 gnutls_x509_crt_set_ca_status

int **gnutls_x509_crt_set_ca_status** (*gnutls_x509_crt* **crt**, *unsigned int* **ca**)

Arguments

- *gnutls_x509_crt* **crt**: should contain a gnutls_x509_crt structure
- *unsigned int* **ca**: true(1) or false(0). Depending on the Certificate authority status.

Description

This function will set the basicConstraints certificate extension.
Returns 0 on success.

8.2.110 gnutls_x509_crt_set_crl_dist_points

int **gnutls_x509_crt_set_crl_dist_points** (*gnutls_x509_crt* **crt**, *gnutls_x509_subject_alt_name* **type**, *const void** **data_string**, *unsigned int* **reason_flags**)

Arguments

- *gnutls_x509_crt* **crt**: should contain a gnutls_x509_crt structure
- *gnutls_x509_subject_alt_name* **type**: is one of the gnutls_x509_subject_alt_name enumerations
- *const void** **data_string**: The data to be set
- *unsigned int* **reason_flags**: revocation reasons

Description

This function will set the CRL distribution points certificate extension.
Returns 0 on success.

8.2.111 gnutls_x509_crt_set_crq

int **gnutls_x509_crt_set_crq** (*gnutls_x509_crt* **crt**, *gnutls_x509_crq* **crq**)

Arguments

- *gnutls_x509_crt* **crt**: should contain a gnutls_x509_crt structure
- *gnutls_x509_crq* **crq**: holds a certificate request

Description

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.

8.2.112 gnutls_x509_crt_set_dn_by_oid

int **gnutls_x509_crt_set_dn_by_oid** (*gnutls_x509_crt* **crt**, *const char** **oid**, *unsigned int* **raw_flag**, *const void ** **name**, *unsigned int* **sizeof_name**)

Arguments

- *gnutls_x509_crt* **crt**: should contain a gnutls_x509_crt structure
- *const char** **oid**: holds an Object Identifier in a null terminated string
- *unsigned int* **raw_flag**: must be 0, or 1 if the data are DER encoded
- *const void ** **name**: a pointer to the name
- *unsigned int* **sizeof_name**: holds the size of **name**

Description

This function will set the part of the name of the Certificate subject, specified by the given OID. The input string should be ASCII or UTF-8 encoded.
Some helper macros with popular OIDs can be found in gnutls/x509.h With this function you can only set the known OIDs. You can test for known OIDs using **gnutls_x509_dn_oid_known()**. For OIDs that are not known (by gnutls) you should properly DER encode your data, and call this function with raw_flag set.
Returns 0 on success.

8.2.113 gnutls_x509_crt_set_expiration_time

int **gnutls_x509_crt_set_expiration_time** (*gnutls_x509_crt* **cert**, *time_t* **exp_time**)

Arguments

- *gnutls_x509_crt* **cert**: should contain a gnutls_x509_crt structure
- *time_t* **exp_time**: The actual time

Description

This function will set the time this Certificate will expire.
Returns 0 on success, or a negative value in case of an error.

8.2.114 gnutls_x509_crt_set_issuer_dn_by_oid

int **gnutls_x509_crt_set_issuer_dn_by_oid** (*gnutls_x509_crt* **crt**, *const char** **oid**, *unsigned int* **raw_flag**, *const void ** **name**, *unsigned int* **sizeof_name**)

Arguments

- *gnutls_x509_crt* **crt**: should contain a gnutls_x509_crt structure
- *const char** **oid**: holds an Object Identifier in a null terminated string

- *unsigned int* **raw_flag**: must be 0, or 1 if the data are DER encoded
- *const void ** **name**: a pointer to the name
- *unsigned int* **sizeof_name**: holds the size of **name**

Description

This function will set the part of the name of the Certificate issuer, specified by the given OID. The input string should be ASCII or UTF-8 encoded.
Some helper macros with popular OIDs can be found in gnutls/x509.h With this function you can only set the known OIDs. You can test for known OIDs using **gnutls_x509_dn_oid_known()**. For OIDs that are not known (by gnutls) you should properly DER encode your data, and call this function with raw_flag set.
Normally you do not need to call this function, since the signing operation will copy the signer's name as the issuer of the certificate.
Returns 0 on success.

8.2.115 gnutls_x509_crt_set_key_purpose_oid

int **gnutls_x509_crt_set_key_purpose_oid** (*gnutls_x509_crt* **cert**, *const void ** **oid**, *unsigned int* **critical**)

Arguments

- *gnutls_x509_crt* **cert**: should contain a gnutls_x509_crt structure
- *const void ** **oid**: a pointer to a null terminated string that holds the OID
- *unsigned int* **critical**: Whether this extension will be critical or not

Description

This function will set the key purpose OIDs of the Certificate. These are stored in the Extended Key Usage extension (2.5.29.37) See the GNUTLS_KP_* definitions for human readable names.
Subsequent calls to this function will append OIDs to the OID list.
On success 0 is returned.

8.2.116 gnutls_x509_crt_set_key_usage

int **gnutls_x509_crt_set_key_usage** (*gnutls_x509_crt* **crt**, *unsigned int* **usage**)

Arguments

- *gnutls_x509_crt* **crt**: should contain a gnutls_x509_crt structure
- *unsigned int* **usage**: an ORed sequence of the GNUTLS_KEY_* elements.

Description

This function will set the keyUsage certificate extension.
Returns 0 on success.

8.2.117 gnutls_x509_crt_set_key

int **gnutls_x509_crt_set_key** (*gnutls_x509_crt* **crt**, *gnutls_x509_privkey* **key**)

Arguments

- *gnutls_x509_crt* **crt**: should contain a gnutls_x509_crt structure
- *gnutls_x509_privkey* **key**: holds a private key

Description

This function will set the public parameters from the given private key to the certificate. Only RSA keys are currently supported.
Returns 0 on success.

8.2.118 gnutls_x509_crt_set_serial

int **gnutls_x509_crt_set_serial** (*gnutls_x509_crt* **cert**, *const void** **serial**, *size_t* **serial_size**)

Arguments

- *gnutls_x509_crt* **cert**: should contain a gnutls_x509_crt structure
- *const void** **serial**: The serial number
- *size_t* **serial_size**: Holds the size of the serial field.

Description

This function will set the X.509 certificate's serial number. Serial is not always a 32 or 64bit number. Some CAs use large serial numbers, thus it may be wise to handle it as something opaque.
Returns 0 on success, or a negative value in case of an error.

8.2.119 gnutls_x509_crt_set_subject_alternative_name

int **gnutls_x509_crt_set_subject_alternative_name** (*gnutls_x509_crt* **crt**, *gnutls_x509_subject_alt_nam* **type**, *const char** **data_string**)

Arguments

- *gnutls_x509_crt* **crt**: should contain a gnutls_x509_crt structure
- *gnutls_x509_subject_alt_name* **type**: is one of the gnutls_x509_subject_alt_name enumerations
- *const char** **data_string**: The data to be set

Description

This function will set the subject alternative name certificate extension.
Returns 0 on success.

8.2.120 gnutls_x509_crt_set_subject_key_id

int **gnutls_x509_crt_set_subject_key_id** (*gnutls_x509_crt* **cert**, *const void** **id**, *size_t* **id_size**)

Arguments

- *gnutls_x509_crt* **cert**: should contain a gnutls_x509_crt structure
- *const void** **id**: The key ID
- *size_t* **id_size**: Holds the size of the serial field.

Description

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.

8.2.121 gnutls_x509_crt_set_version

int **gnutls_x509_crt_set_version** (*gnutls_x509_crt* **crt**, *unsigned int* **version**)

Arguments

- *gnutls_x509_crt* **crt**: should contain a gnutls_x509_crt structure
- *unsigned int* **version**: holds the version number. For X.509v1 certificates must be 1.

Description

This function will set the version of the certificate. This must be one for X.509 version 1, and so on. Plain certificates without extensions must have version set to one.
Returns 0 on success.

8.2.122 gnutls_x509_crt_sign

int **gnutls_x509_crt_sign** (*gnutls_x509_crt* **crt**, *gnutls_x509_crt* **issuer**, *gnutls_x509_privkey* **issuer_key**)

Arguments

- *gnutls_x509_crt* **crt**: should contain a gnutls_x509_crt structure
- *gnutls_x509_crt* **issuer**: is the certificate of the certificate issuer
- *gnutls_x509_privkey* **issuer_key**: holds the issuer's private key

Description

This function will sign the certificate with the issuer's private key, and will copy the issuer's information into the certificate.
This must be the last step in a certificate generation since all the previously set parameters are now signed.
Returns 0 on success.

8.2.123 gnutls_x509_crt_to_xml

int **gnutls_x509_crt_to_xml** (*gnutls_x509_crt* **cert**, *gnutls_datum** **res**, *int* **detail**)

Arguments

- *gnutls_x509_crt* **cert**: should contain a gnutls_x509_crt structure
- *gnutls_datum** **res**: The datum that will hold the result
- *int* **detail**: The detail level (must be GNUTLS_XML_SHOW_ALL or GNUTLS_XML_NORMAL)

Description

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. Returns a negative error code in case of an error.

8.2.124 gnutls_x509_crt_verify_data

int **gnutls_x509_crt_verify_data** (*gnutls_x509_crt* **crt**, *unsigned int* **flags**, *const gnutls_datum** **data**, *const gnutls_datum** **signature**)

Arguments

- *gnutls_x509_crt* **crt**: Holds the certificate
- *unsigned int* **flags**: should be 0 for now
- *const gnutls_datum** **data**: holds the data to be signed
- *const gnutls_datum** **signature**: contains the signature

Description

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.

8.2.125 gnutls_x509_crt_verify

int **gnutls_x509_crt_verify** (*gnutls_x509_crt* *** **CA_list**, *int* **CA_list_length**, *unsigned int* **flags**, *unsigned int ** **verify**)

Arguments

- *gnutls_x509_crt* **cert**: is the certificate to be verified
- *gnutls_x509_crt ** **CA_list**: is one certificate that is considered to be trusted one

- *int* **CA_list_length**: holds the number of CA certificate in CA_list
- *unsigned int* **flags**: Flags that may be used to change the verification algorithm. Use OR of the gnutls_certificate_verify_flags enumerations.
- *unsigned int ** **verify**: will hold the certificate verification output.

Description

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.

8.2.126 gnutls_x509_dn_oid_known

int **gnutls_x509_dn_oid_known** (*const char** **oid**)

Arguments

- *const char** **oid**: holds an Object Identifier in a null terminated string

Description

This function will inform about known DN OIDs. This is useful since functions like **gnutls_x509_crt_set_dn_by_oid()** use the information on known OIDs to properly encode their input. Object Identifiers that are not known are not encoded by these functions, and their input is stored directly into the ASN.1 structure. In that case of unknown OIDs, you have the responsibility of DER encoding your data.
Returns 1 on known OIDs and 0 otherwise.

8.2.127 gnutls_x509_privkey_cpy

int **gnutls_x509_privkey_cpy** (*gnutls_x509_privkey* **dst**, *gnutls_x509_privkey* **src**)

Arguments

- *gnutls_x509_privkey* **dst**: The destination key, which should be initialized.
- *gnutls_x509_privkey* **src**: The source key

Description

This function will copy a private key from source to destination key.

8.2.128 gnutls_x509_privkey_deinit

void **gnutls_x509_privkey_deinit** (*gnutls_x509_privkey* **key**)

Arguments

- *gnutls_x509_privkey* **key**: The structure to be initialized

Description

This function will deinitialize a private key structure.

8.2.129 gnutls_x509_privkey_export_dsa_raw

int **gnutls_x509_privkey_export_dsa_raw** (*gnutls_x509_privkey* **key**, *gnutls_datum * * **p**, *gnutls_datum * * **q**, *gnutls_datum * * **g**, *gnutls_datum * * **y**, *gnutls_datum** **x**)

Arguments

- *gnutls_x509_privkey* **key**:
- *gnutls_datum * * **p**: will hold the p
- *gnutls_datum * * **q**: will hold the q
- *gnutls_datum * * **g**: will hold the g
- *gnutls_datum * * **y**: will hold the y
- *gnutls_datum** **x**: will hold the x

Description

This function will export the DSA private key's parameters found in the given structure. The new parameters will be allocated using **gnutls_malloc()** and will be stored in the appropriate datum.

8.2.130 gnutls_x509_privkey_export_pkcs8

int **gnutls_x509_privkey_export_pkcs8** (*gnutls_x509_privkey* **key**, *gnutls_x509_crt_fmt* **format**, *const char ** **password**, *unsigned int* **flags**, *void ** **output_data**, *size_t ** **output_data_size**)

Arguments

- *gnutls_x509_privkey* **key**: Holds the key
- *gnutls_x509_crt_fmt* **format**: the format of output params. One of PEM or DER.
- *const char ** **password**: the password that will be used to encrypt the key.
- *unsigned int* **flags**: an ORed sequence of gnutls_pkcs_encrypt_flags
- *void ** **output_data**: will contain a private key PEM or DER encoded
- *size_t ** **output_data_size**: holds the size of output_data (and will be replaced by the actual size of parameters)

Description

This function will export the private key to a PKCS8 structure. Currently only RSA keys can be exported. 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 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.
In case of failure a negative value will be returned, and 0 on success.

8.2.131 gnutls_x509_privkey_export_rsa_raw

int **gnutls_x509_privkey_export_rsa_raw** (*gnutls_x509_privkey* **key**, *gnutls_datum * * **m**, *gnutls_datum * * **e**, *gnutls_datum * * **d**, *gnutls_datum * * **p**, *gnutls_datum** **q**, *gnutls_datum** **u**)

Arguments

- *gnutls_x509_privkey* **key**:
- *gnutls_datum * * **m**: will hold the modulus
- *gnutls_datum * * **e**: will hold the public exponent
- *gnutls_datum * * **d**: will hold the private exponent
- *gnutls_datum * * **p**: will hold the first prime (p)
- *gnutls_datum** **q**: will hold the second prime (q)
- *gnutls_datum** **u**: will hold the coefficient

Description

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

8.2.132 gnutls_x509_privkey_export

int **gnutls_x509_privkey_export** (*gnutls_x509_privkey* **key**, *gnutls_x509_crt_fmt* **format**, *void** **output_data**, *size_t** **output_data_size**)

Arguments

- *gnutls_x509_privkey* **key**: Holds the key
- *gnutls_x509_crt_fmt* **format**: the format of output params. One of PEM or DER.

- *void** **output_data**: will contain a private key PEM or DER encoded
- *size_t** **output_data_size**: holds the size of output_data (and will be replaced by the actual size of parameters)

Description

This function will export the private key to a PKCS1 structure for RSA keys, or an integer sequence for DSA keys. The DSA keys are in the same format with the parameters used by openssl.
If the buffer provided is not long enough to hold the output, then GNUTLS_E_SHORT_MEMORY_BUFFE
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.

8.2.133 gnutls_x509_privkey_generate

int **gnutls_x509_privkey_generate** (*gnutls_x509_privkey* **key**, *gnutls_pk_algorithm* **algo**, *unsigned int* **bits**, *unsigned int* **flags**)

Arguments

- *gnutls_x509_privkey* **key**: should contain a gnutls_x509_privkey structure
- *gnutls_pk_algorithm* **algo**: is one of RSA or DSA.
- *unsigned int* **bits**: the size of the modulus
- *unsigned int* **flags**: unused for now. Must be 0.

Description

This function will generate a random private key. Note that this function must be called on an empty private key.
Returns 0 on success or a negative value on error.

8.2.134 gnutls_x509_privkey_get_key_id

int **gnutls_x509_privkey_get_key_id** (*gnutls_x509_privkey* **key**, *unsigned int* **flags**, *unsigned char** **output_data**, *size_t** **output_data_size**)

Arguments

- *gnutls_x509_privkey* **key**: Holds the key
- *unsigned int* **flags**: should be 0 for now
- *unsigned char** **output_data**: will contain the key ID
- *size_t** **output_data_size**: holds the size of output_data (and will be replaced by the actual size of parameters)

Description
This function will return a unique ID the depends on the public key parameters. This ID can be used in checking whether a certificate corresponds to the given key.
If the buffer provided is not long enough to hold the output, then GNUTLS_E_SHORT_MEMORY_BUFFER will be returned. The output will normally be a SHA-1 hash output, which is 20 bytes.
In case of failure a negative value will be returned, and 0 on success.

8.2.135 gnutls_x509_privkey_get_pk_algorithm

int **gnutls_x509_privkey_get_pk_algorithm** (*gnutls_x509_privkey* **key**)

Arguments

- *gnutls_x509_privkey* **key**: should contain a gnutls_x509_privkey structure

Description
This function will return the public key algorithm of a private key.
Returns a member of the gnutls_pk_algorithm enumeration on success, or a negative value on error.

8.2.136 gnutls_x509_privkey_import_dsa_raw

int **gnutls_x509_privkey_import_dsa_raw** (*gnutls_x509_privkey* **key**, *const gnutls_datum** **p**, *const gnutls_datum** **q**, *const gnutls_datum** **g**, *const gnutls_datum** **y**, *const gnutls_datum** **x**)

Arguments

- *gnutls_x509_privkey* **key**: The structure to store the parsed key
- *const gnutls_datum** **p**: holds the p
- *const gnutls_datum** **q**: holds the q
- *const gnutls_datum** **g**: holds the g
- *const gnutls_datum** **y**: holds the y
- *const gnutls_datum** **x**: holds the x

Description
This function will convert the given DSA raw parameters to the native gnutls_x509_privkey format. The output will be stored in **key**.

8.2.137 gnutls_x509_privkey_import_pkcs8

int **gnutls_x509_privkey_import_pkcs8** (*gnutls_x509_privkey* **key**, *const gnutls_datum ** **data**, *gnutls_x509_crt_fmt* **format**, *const char ** **password**, *unsigned int* **flags**)

Arguments

- *gnutls_x509_privkey* **key**: The structure to store the parsed key
- *const gnutls_datum ** **data**: The DER or PEM encoded key.
- *gnutls_x509_crt_fmt* **format**: One of DER or PEM
- *const char ** **password**: the password to decrypt the key (if it is encrypted).
- *unsigned int* **flags**: use 0.

Description

This function will convert the given DER or PEM encoded PKCS8 2.0 encrypted key to the native gnutls_x509_privkey 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.
Returns 0 on success.

8.2.138 gnutls_x509_privkey_import_rsa_raw

int **gnutls_x509_privkey_import_rsa_raw** (*gnutls_x509_privkey* **key**, *const gnutls_datum** **m**, *const gnutls_datum** **e**, *const gnutls_datum** **d**, *const gnutls_datum** **p**, *const gnutls_datum** **q**, *const gnutls_datum** **u**)

Arguments

- *gnutls_x509_privkey* **key**: The structure to store the parsed key
- *const gnutls_datum** **m**: holds the modulus
- *const gnutls_datum** **e**: holds the public exponent
- *const gnutls_datum** **d**: holds the private exponent
- *const gnutls_datum** **p**: holds the first prime (p)
- *const gnutls_datum** **q**: holds the second prime (q)
- *const gnutls_datum** **u**: holds the coefficient

Description

This function will convert the given RSA raw parameters to the native gnutls_x509_privkey format. The output will be stored in **key**.

8.2.139 gnutls_x509_privkey_import

int **gnutls_x509_privkey_import** (*gnutls_x509_privkey* **key**, *const gnutls_datum * * **data**, *gnutls_x509_crt_fmt* **format**)

Arguments

- *gnutls_x509_privkey* **key**: The structure to store the parsed key
- *const gnutls_datum * * **data**: The DER or PEM encoded certificate.
- *gnutls_x509_crt_fmt* **format**: One of DER or PEM

Description

This function will convert the given DER or PEM encoded key to the native gnutls_x509_privkey 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.

8.2.140 gnutls_x509_privkey_init

int **gnutls_x509_privkey_init** (*gnutls_x509_privkey * * **key**)

Arguments

- *gnutls_x509_privkey * * **key**: The structure to be initialized

Description

This function will initialize an private key structure.
Returns 0 on success.

8.2.141 gnutls_x509_privkey_sign_data

int **gnutls_x509_privkey_sign_data** (*gnutls_x509_privkey* **key**, *gnutls_digest_algorithm* **digest**, *unsigned int* **flags**, *const gnutls_datum** **data**, *void** **signature**, *size_t** **signature_size**)

Arguments

- *gnutls_x509_privkey* **key**: Holds the key
- *gnutls_digest_algorithm* **digest**: should be MD5 or SHA1
- *unsigned int* **flags**: should be 0 for now

- *const gnutls_datum** **data**: holds the data to be signed
- *void** **signature**: will contain the signature
- *size_t** **signature_size**: holds the size of signature (and will be replaced by the new size)

Description
This function will sign the given data using a signature algorithm supported by the private key. Signature algorithms are always used together with a hash functions. Different hash functions may be used for the RSA algorithm, but only SHA-1 for the DSA keys.
If the buffer provided is not long enough to hold the output, then GNUTLS_E_SHORT_MEMORY_BUFFE
will be returned.
In case of failure a negative value will be returned, and 0 on success.

8.2.142 gnutls_x509_privkey_verify_data

int **gnutls_x509_privkey_verify_data** (*gnutls_x509_privkey* **key**, *unsigned int* **flags**, *const gnutls_datum** **data**, *const gnutls_datum** **signature**)
Arguments

- *gnutls_x509_privkey* **key**: Holds the key
- *unsigned int* **flags**: should be 0 for now
- *const gnutls_datum** **data**: holds the data to be signed
- *const gnutls_datum** **signature**: contains the signature

Description
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.

8.2.143 gnutls_x509_rdn_get_by_oid

int **gnutls_x509_rdn_get_by_oid** (*const gnutls_datum ** **idn**, *const char ** **oid**, *int* **indx**, *unsigned int* **raw_flag**, *void ** **buf**, *size_t ** **sizeof_buf**)
Arguments

- *const gnutls_datum ** **idn**: should contain a DER encoded RDN sequence
- *const char ** **oid**: an Object Identifier
- *int* **indx**: In case multiple same OIDs exist in the RDN indicates which to send. Use 0 for the first one.

- *unsigned int* **raw_flag**: If non zero then the raw DER data are returned.
- *void ** **buf**: a pointer to a structure to hold the peer's name
- *size_t ** **sizeof_buf**: holds the size of **buf**

Description

This function will return the name of the given Object identifier, of the RDN sequence. The name will be encoded using the rules from RFC2253.
Returns GNUTLS_E_SHORT_MEMORY_BUFFER if the provided buffer is not long enough, and 0 on success.

8.2.144 gnutls_x509_rdn_get_oid

int **gnutls_x509_rdn_get_oid** (*const gnutls_datum ** **idn**, *int* **indx**, *void ** **buf**, *size_t ** **sizeof_buf**)

Arguments

- *const gnutls_datum ** **idn**: should contain a DER encoded RDN sequence
- *int* **indx**: Indicates which OID to return. Use 0 for the first one.
- *void ** **buf**:
- *size_t ** **sizeof_buf**:

Description

This function will return the specified Object identifier, of the RDN sequence.
Returns GNUTLS_E_SHORT_MEMORY_BUFFER if the provided buffer is not long enough, and 0 on success.

8.2.145 gnutls_x509_rdn_get

int **gnutls_x509_rdn_get** (*const gnutls_datum ** **idn**, *char ** **buf**, *size_t ** **sizeof_buf**)

Arguments

- *const gnutls_datum ** **idn**: should contain a DER encoded RDN sequence
- *char ** **buf**: a pointer to a structure to hold the peer's name
- *size_t ** **sizeof_buf**: holds the size of **buf**

Description

This function will return the name of the given RDN sequence. The name will be in the form "C=xxxx,O=yyyy,CN=zzzz" as described in RFC2253.
Returns GNUTLS_E_SHORT_MEMORY_BUFFER if the provided buffer is not long enough, and 0 on success.

8.3 *GnuTLS-extra* library

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".

8.3.1 gnutls_global_init_extra

int **gnutls_global_init_extra** (**void**)

Arguments

- **void**:

Description

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.

8.3.2 gnutls_srp_allocate_client_credentials

int **gnutls_srp_allocate_client_credentials** (*gnutls_srp_client_credentials ** **sc**)

Arguments

- *gnutls_srp_client_credentials ** **sc**: is a pointer to an *gnutls_srp_server_credentials* structure.

Description

This structure is complex enough to manipulate directly thus this helper function is provided in order to allocate it.
Returns 0 on success.

8.3.3 gnutls_srp_allocate_server_credentials

int **gnutls_srp_allocate_server_credentials** (*gnutls_srp_server_credentials ** **sc**)

Arguments

- *gnutls_srp_server_credentials ** **sc**: is a pointer to an *gnutls_srp_server_credentials* structure.

Description

This structure is complex enough to manipulate directly thus this helper function is provided in order to allocate it.
Returns 0 on success.

8.3.4 gnutls_srp_base64_decode_alloc

int **gnutls_srp_base64_decode_alloc** (*const gnutls_datum ** **b64_data**, *gnutls_datum** **result**)

Arguments

- *const gnutls_datum ** **b64_data**: contains the encoded data
- *gnutls_datum** **result**: the place where decoded data lie

Description

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.

8.3.5 gnutls_srp_base64_decode

int **gnutls_srp_base64_decode** (*const gnutls_datum ** **b64_data**, *char** **result**, *int** **result_size**)

Arguments

- *const gnutls_datum ** **b64_data**: contain the encoded data
- *char** **result**: the place where decoded data will be copied
- *int** **result_size**: holds the size of the result

Description

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.

8.3.6 gnutls_srp_base64_encode_alloc

int **gnutls_srp_base64_encode_alloc** (*const gnutls_datum ** **data**, *gnutls_datum** **result**)

Arguments

- *const gnutls_datum ** **data**: contains the raw data
- *gnutls_datum** **result**: will hold the newly allocated encoded data

Description

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.

8.3.7 gnutls_srp_base64_encode

int **gnutls_srp_base64_encode** (*const gnutls_datum ** **data**, *char** **result**, *int** **result_size**)

Arguments

- *const gnutls_datum ** **data**: contain the raw data
- *char** **result**: the place where base64 data will be copied
- *int** **result_size**: holds the size of the result

Description

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.

8.3.8 gnutls_srp_free_client_credentials

void **gnutls_srp_free_client_credentials** (*gnutls_srp_client_credentials* **sc**)

Arguments

- *gnutls_srp_client_credentials* **sc**: is an *gnutls_srp_client_credentials* structure.

Description

This structure is complex enough to manipulate directly thus this helper function is provided in order to free (deallocate) it.

8.3.9 gnutls_srp_free_server_credentials

void **gnutls_srp_free_server_credentials** (*gnutls_srp_server_credentials* **sc**)

Arguments

- *gnutls_srp_server_credentials* **sc**: is an *gnutls_srp_server_credentials* structure.

Description

This structure is complex enough to manipulate directly thus this helper function is provided in order to free (deallocate) it.

8.3.10 gnutls_srp_server_get_username

*const char ** **gnutls_srp_server_get_username** (*gnutls_session* **session**)

Arguments

- *gnutls_session* **session**: is a gnutls session

Description

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.

8.3.11 gnutls_srp_server_set_select_function

void **gnutls_srp_server_set_select_function** (*gnutls_session* **session**, *gnutls_srp_server_select_function ** **func**)

Arguments

- *gnutls_session* **session**: is a *gnutls_session* structure.
- *gnutls_srp_server_select_function ** **func**: is the callback function

Description

This function sets a callback to assist in selecting the proper password file, in case there are more than one. The callback's function form is: int (*callback)(gnutls_session, const char** pfiles, const char** pconffiles, int npfiles);
pfiles contains **npfiles** char* structures which hold the password file name. **pconffiles** contain the corresponding conf files.
This function specifies what we, in case of a server, are going to do when we have to use a password file. If this callback function is not provided then gnutls will automatically select the first password file
In case the callback returned a negative number then gnutls will terminate this handshake.
The callback function will only be called once per handshake. The callback function should return the index of the password file that will be used by the server. -1 indicates an error.

8.3.12 gnutls_srp_set_client_credentials_function

void **gnutls_srp_set_client_credentials_function** (*gnutls_srp_client_credentials* **cred**, *gnutls_srp_client_credentials_function ** **func**)

Arguments

- *gnutls_srp_client_credentials* **cred**: is a *gnutls_srp_server_credentials* structure.

- *gnutls_srp_client_credentials_function ** **func**: is the callback function

Description

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, 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.

8.3.13 gnutls_srp_set_client_credentials

int **gnutls_srp_set_client_credentials** (*gnutls_srp_client_credentials* **res**, *char ** **username**, *char ** **password**)

Arguments

- *gnutls_srp_client_credentials* **res**: is an *gnutls_srp_client_credentials* structure.
- *char ** **username**: is the user's userid
- *char ** **password**: is the user's password

Description

This function sets the username and password, in a gnutls_srp_client_credentials 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.

8.3.14 gnutls_srp_set_server_credentials_file

int **gnutls_srp_set_server_credentials_file** (*gnutls_srp_server_credentials* **res**, *const char ** **password_file**, *const char ** **password_conf_file**)

Arguments

- *gnutls_srp_server_credentials* **res**: is an *gnutls_srp_server_credentials* structure.

- *const char ** **password_file**: is the SRP password file (tpasswd)
- *const char ** **password_conf_file**: is the SRP password conf file (tpasswd.conf)

Description

This function sets the password files, in a gnutls_srp_server_credentials structure. Those password files hold usernames and verifiers and will be used for SRP authentication.
Returns 0 on success.

8.3.15 gnutls_srp_set_server_credentials_function

void **gnutls_srp_set_server_credentials_function** (*gnutls_srp_server_credentials* **cred**, *gnutls_srp_server_credentials_function ** **func**)

Arguments

- *gnutls_srp_server_credentials* **cred**: is a *gnutls_srp_server_credentials* structure.
- *gnutls_srp_server_credentials_function ** **func**: is the callback function

Description

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, const char* username, gnutls_datum* salt, gnutls_datum *verifier, gnutls_datum* g, gnutls_datum* 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.

8.3.16 gnutls_srp_verifier

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

Arguments

- *const char** **username**: is the user's name
- *const char** **password**: is the user's password

- *const gnutls_datum* * **salt**: should be some randomly generated bytes
- *const gnutls_datum** **generator**: is the generator of the group
- *const gnutls_datum** **prime**: is the group's prime
- *gnutls_datum* * **res**: where the verifier will be stored.

Description

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.

8.4 *GnuTLS* OpenPGP key handling

The following functions are to be used for OpenPGP certificate handling. Their prototypes lie in "gnutls/openpgp.h".

8.4.1 gnutls_openpgp_key_check_hostname

int **gnutls_openpgp_key_check_hostname** (*gnutls_openpgp_key* **key**, *const char* * **hostname**)

Arguments

- *gnutls_openpgp_key* **key**: should contain an gnutls_openpgp_key structure
- *const char* * **hostname**: A null terminated string that contains a DNS name

Description

This function will check if the given key's owner matches the given hostname. This is a basic implementation of the matching described in RFC2818 (HTTPS), which takes into account wildcards.
Returns non zero on success, and zero on failure.

8.4.2 gnutls_openpgp_key_deinit

void **gnutls_openpgp_key_deinit** (*gnutls_openpgp_key* **key**)

Arguments

- *gnutls_openpgp_key* **key**: The structure to be initialized

Description

This function will deinitialize a key structure.

8.4.3 gnutls_openpgp_key_export

int **gnutls_openpgp_key_export** (*gnutls_openpgp_key* **key**, *gnutls_openpgp_key_fmt* **format**, *void** **output_data**, *size_t** **output_data_size**)

Arguments

- *gnutls_openpgp_key* **key**: Holds the key.
- *gnutls_openpgp_key_fmt* **format**: One of gnutls_openpgp_key_fmt elements.
- *void** **output_data**: will contain the key base64 encoded or raw
- *size_t** **output_data_size**: holds the size of output_data (and will be replaced by the actual size of parameters)

Description

This function will convert the given key to RAW or Base64 format. If the buffer provided is not long enough to hold the output, then GNUTLS_E_SHORT_MEMORY_BUFFER will be returned.
Returns 0 on success.

8.4.4 gnutls_openpgp_key_get_creation_time

time_t **gnutls_openpgp_key_get_creation_time** (*gnutls_openpgp_key* **key**)

Arguments

- *gnutls_openpgp_key* **key**: the structure that contains the OpenPGP public key.

Description

Returns the timestamp when the OpenPGP key was created.

8.4.5 gnutls_openpgp_key_get_expiration_time

time_t **gnutls_openpgp_key_get_expiration_time** (*gnutls_openpgp_key* **key**)

Arguments

- *gnutls_openpgp_key* **key**: the structure that contains the OpenPGP public key.

Description

Returns the time when the OpenPGP key expires. A value of '0' means that the key doesn't expire at all.

8.4.6 gnutls_openpgp_key_get_fingerprint

int **gnutls_openpgp_key_get_fingerprint** (*gnutls_openpgp_key* **key**, *void ** **fpr**, *size_t ** **fprlen**)

Arguments

- *gnutls_openpgp_key* **key**: the raw data that contains the OpenPGP public key.
- *void ** **fpr**: the buffer to save the fingerprint.
- *size_t ** **fprlen**: the integer to save the length of the fingerprint.

Description

Returns the fingerprint of the OpenPGP key. Depends on the algorithm, the fingerprint can be 16 or 20 bytes.

8.4.7 gnutls_openpgp_key_get_id

int **gnutls_openpgp_key_get_id** (*gnutls_openpgp_key* **key**, *unsigned char* **keyid[8]**)

Arguments

- *gnutls_openpgp_key* **key**: the structure that contains the OpenPGP public key.
- *unsigned char* **keyid[8]**:

Description

Returns the 64-bit keyID of the OpenPGP key.

8.4.8 gnutls_openpgp_key_get_key_usage

int **gnutls_openpgp_key_get_key_usage** (*gnutls_openpgp_key* **key**, *unsigned int ** **key_usage**)

Arguments

- *gnutls_openpgp_key* **key**: should contain a gnutls_openpgp_key structure
- *unsigned int ** **key_usage**: where the key usage bits will be stored

Description

This function will return certificate's key usage, by checking the key algorithm.
The key usage value will ORed values of the: GNUTLS_KEY_DIGITAL_SIGNATURE, GNUTLS_KEY_KEY_ENCIPHERMENT.
A negative value may be returned in case of parsing error.

8.4.9 gnutls_openpgp_key_get_name

int **gnutls_openpgp_key_get_name** (*gnutls_openpgp_key* **key**, *int* **idx**, *char ** **buf**, *size_t ** **sizeof_buf**)

Arguments

- *gnutls_openpgp_key* **key**: the structure that contains the OpenPGP public key.
- *int* **idx**: the index of the ID to extract
- *char ** **buf**: a pointer to a structure to hold the name
- *size_t ** **sizeof_buf**: holds the size of 'buf'

Description

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.

8.4.10 gnutls_openpgp_key_get_pk_algorithm

int **gnutls_openpgp_key_get_pk_algorithm** (*gnutls_openpgp_key* **key**, *unsigned int ** **bits**)

Arguments

- *gnutls_openpgp_key* **key**: is an OpenPGP key
- *unsigned int ** **bits**: if bits is non null it will hold the size of the parameters' in bits

Description

This function will return the public key algorithm of an OpenPGP certificate.
If bits is non null, it should have enough size to hold the parameters size in bits. For RSA the bits returned is the modulus. For DSA the bits returned are of the public exponent.
Returns a member of the GNUTLS_PKAlgorithm enumeration on success, or a negative value on error.

8.4.11 gnutls_openpgp_key_get_version

int **gnutls_openpgp_key_get_version** (*gnutls_openpgp_key* **key**)

Arguments

- *gnutls_openpgp_key* **key**: the structure that contains the OpenPGP public key.

Description

Extract the version of the OpenPGP key.

8.4.12 gnutls_openpgp_key_import

int **gnutls_openpgp_key_import** (*gnutls_openpgp_key* **key**, *const gnutls_datum * * **data**, *gnutls_openpgp_key_fmt* **format**)

Arguments

- *gnutls_openpgp_key* **key**: The structure to store the parsed key.
- *const gnutls_datum * * **data**: The RAW or BASE64 encoded key.
- *gnutls_openpgp_key_fmt* **format**: One of gnutls_openpgp_key_fmt elements.

Description

This function will convert the given RAW or Base64 encoded key to the native gnutls_openpgp_key format. The output will be stored in 'key'.
Returns 0 on success.

8.4.13 gnutls_openpgp_key_init

int **gnutls_openpgp_key_init** (*gnutls_openpgp_key * * **key**)

Arguments

- *gnutls_openpgp_key * * **key**: The structure to be initialized

Description

This function will initialize an OpenPGP key structure.
Returns 0 on success.

8.4.14 gnutls_openpgp_key_to_xml

int **gnutls_openpgp_key_to_xml** (*gnutls_openpgp_key* **key**, *gnutls_datum * * **xmlkey**, *int* **ext**)

Arguments

- *gnutls_openpgp_key* **key**:
- *gnutls_datum * * **xmlkey**: he datum struct to store the XML result.
- *int* **ext**: extension mode (1/0), 1 means include key signatures and key data.

Description

This function will return the all OpenPGP key information encapsulated as a XML string.

8.4.15 gnutls_openpgp_key_verify_ring

int **gnutls_openpgp_key_verify_ring** (*gnutls_openpgp_key* **key**, *gnutls_openpgp_keyring* **keyring**, *unsigned int* **flags**, *unsigned int ** **verify**)

Arguments

- *gnutls_openpgp_key* **key**: the structure that holds the key.
- *gnutls_openpgp_keyring* **keyring**: holds the keyring to check against
- *unsigned int* **flags**: unused (should be 0)
- *unsigned int ** **verify**: will hold the certificate verification output.

Description

Verify all signatures in the key, using the given set of keys (keyring).
The key verification output will be put in **verify** and will be one or more of the gnutls_certificate_status enumerated elements bitwise or'd.
GNUTLS_CERT_INVALID: A signature on the key is invalid.
GNUTLS_CERT_REVOKED: The key has been revoked.

NOTE

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.

8.4.16 gnutls_openpgp_key_verify_self

int **gnutls_openpgp_key_verify_self** (*gnutls_openpgp_key* **key**, *unsigned int* **flags**, *unsigned int ** **verify**)

Arguments

- *gnutls_openpgp_key* **key**: the structure that holds the key.
- *unsigned int* **flags**: unused (should be 0)
- *unsigned int ** **verify**: will hold the key verification output.

Description

Verifies the self signature in the key. The key verification output will be put in **verify** and will be one or more of the gnutls_certificate_status enumerated elements bitwise or'd.
GNUTLS_CERT_INVALID: The self signature on the key is invalid.
Returns 0 on success.

8.4.17 gnutls_openpgp_key_verify_trustdb

int **gnutls_openpgp_key_verify_trustdb** (*gnutls_openpgp_key* **key**, *gnutls_openpgp_trustdb* **trustdb**, *unsigned int* **flags**, *unsigned int ** **verify**)

Arguments

- *gnutls_openpgp_key* **key**: the structure that holds the key.
- *gnutls_openpgp_trustdb* **trustdb**: holds the trustdb to check against
- *unsigned int* **flags**: unused (should be 0)
- *unsigned int ** **verify**: will hold the certificate verification output.

Description
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 enumerated elements bitwise or'd.
GNUTLS_CERT_INVALID: A signature on the key is invalid.
GNUTLS_CERT_REVOKED: The key has been revoked.
NOTE
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.

8.4.18 gnutls_openpgp_keyring_check_id

int **gnutls_openpgp_keyring_check_id** (*gnutls_openpgp_keyring* **ring**, *const unsigned char* **keyid[8]**, *unsigned int* **flags**)
Arguments

- *gnutls_openpgp_keyring* **ring**: holds the keyring to check against
- *const unsigned char* **keyid[8]**:
- *unsigned int* **flags**: unused (should be 0)

Description
Check if a given key ID exists in the keyring.
Returns 0 on success (if keyid exists) and a negative error code on failure.

8.4.19 gnutls_openpgp_keyring_deinit

void **gnutls_openpgp_keyring_deinit** (*gnutls_openpgp_keyring* **keyring**)
Arguments

- *gnutls_openpgp_keyring* **keyring**: The structure to be initialized

Description
This function will deinitialize a CRL structure.

8.4.20 gnutls_openpgp_keyring_import

int **gnutls_openpgp_keyring_import** (*gnutls_openpgp_keyring* **keyring**, *const gnutls_datum ** **data**, *gnutls_openpgp_key_fmt* **format**)

Arguments

- *gnutls_openpgp_keyring* **keyring**: The structure to store the parsed key.
- *const gnutls_datum ** **data**: The RAW or BASE64 encoded keyring.
- *gnutls_openpgp_key_fmt* **format**: One of gnutls_openpgp_keyring_fmt elements.

Description

This function will convert the given RAW or Base64 encoded keyring to the native gnutls_openpgp_keyring format. The output will be stored in 'keyring'.
Returns 0 on success.

8.4.21 gnutls_openpgp_keyring_init

int **gnutls_openpgp_keyring_init** (*gnutls_openpgp_keyring ** **keyring**)

Arguments

- *gnutls_openpgp_keyring ** **keyring**: The structure to be initialized

Description

This function will initialize an OpenPGP keyring structure.
Returns 0 on success.

8.4.22 gnutls_openpgp_privkey_deinit

void **gnutls_openpgp_privkey_deinit** (*gnutls_openpgp_privkey* **key**)

Arguments

- *gnutls_openpgp_privkey* **key**: The structure to be initialized

Description

This function will deinitialize a key structure.

8.4.23 gnutls_openpgp_privkey_get_pk_algorithm

int **gnutls_openpgp_privkey_get_pk_algorithm** (*gnutls_openpgp_privkey* **key**, *unsigned int ** **bits**)

Arguments

- *gnutls_openpgp_privkey* **key**: is an OpenPGP key

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

Description

This function will return the public key algorithm of an OpenPGP certificate.
If bits is non null, it should have enough size to hold the parameters size in bits. For RSA the bits returned is the modulus. For DSA the bits returned are of the public exponent.
Returns a member of the GNUTLS_PKAlgorithm enumeration on success, or a negative value on error.

8.4.24 gnutls_openpgp_privkey_import

int **gnutls_openpgp_privkey_import** (*gnutls_openpgp_privkey* **key**, *const gnutls_datum ** **data**, *gnutls_openpgp_key_fmt* **format**, *const char** **pass**, *unsigned int* **flags**)

Arguments

- *gnutls_openpgp_privkey* **key**: The structure to store the parsed key.
- *const gnutls_datum ** **data**: The RAW or BASE64 encoded key.
- *gnutls_openpgp_key_fmt* **format**: One of gnutls_openpgp_key_fmt elements.
- *const char** **pass**: Unused for now
- *unsigned int* **flags**: should be zero

Description

This function will convert the given RAW or Base64 encoded key to the native gnutls_openpgp_privkey format. The output will be stored in 'key'.
Returns 0 on success.

8.4.25 gnutls_openpgp_privkey_init

int **gnutls_openpgp_privkey_init** (*gnutls_openpgp_privkey ** **key**)

Arguments

- *gnutls_openpgp_privkey ** **key**: The structure to be initialized

Description

This function will initialize an OpenPGP key structure.
Returns 0 on success.

8.4.26 gnutls_openpgp_trustdb_deinit

void **gnutls_openpgp_trustdb_deinit** (*gnutls_openpgp_trustdb* **trustdb**)

Arguments

- *gnutls_openpgp_trustdb* **trustdb**: The structure to be initialized

Description
This function will deinitialize a CRL structure.

8.4.27 gnutls_openpgp_trustdb_import_file

int **gnutls_openpgp_trustdb_import_file** (*gnutls_openpgp_trustdb* **trustdb**, *const char ** **file**)
Arguments

- *gnutls_openpgp_trustdb* **trustdb**: The structure to store the parsed key.
- *const char ** **file**: The file that holds the trustdb.

Description
This function will convert the given RAW or Base64 encoded trustdb to the native gnutls_openpgp_trustdb format. The output will be stored in 'trustdb'.
Returns 0 on success.

8.4.28 gnutls_openpgp_trustdb_init

int **gnutls_openpgp_trustdb_init** (*gnutls_openpgp_trustdb ** **trustdb**)
Arguments

- *gnutls_openpgp_trustdb ** **trustdb**: The structure to be initialized

Description
This function will initialize an OpenPGP trustdb structure.
Returns 0 on success.

Appendix A

Certificate to XML convertion functions

This appendix contains some example output of the XML convertion functions:

- gnutls_x509_crt_to_xml() (see section 8.2.123 p.190)
- gnutls_openpgp_key_to_xml() (see section 8.4.14 p.210)

A.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">30818902818100D
      </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">EFEE94ABC8CA577F5313DB76DC
        </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">B73945273AF2A395EC54BF5DC669D953885A9
 </certificate>
</gnutls:x509:certificate>
```

A.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>0400E72E76B62EEFA9A3BD594093292418050C02D7029D6CA2066EFC34C86038627C643EB1A
    <DSA-Q>00A08F5B5E78D85F792CC2072F9474645726FB4D9373</DSA-Q>
    <DSA-G>03FE3578D689D6606E9118E9F9A7042B963CF23F3D8F1377A273C0F0974DBF44B3CABCBE14D
    <DSA-Y>0400D061437A964DDE318818C2B24DE008E60096B60DB8A684B85A838D119FC930311889AD5
  </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>0400E20156526069D067D24F4D71E6D38658E08BE3BF246C1ADCE08DB69CD8D459C1ED33573
    <ELG-G>000305</ELG-G>
    <ELG-Y>0400D0BDADE40432758675C87D0730C360981467BAE1BEB6CC105A3C1F366BFDBEA12E37845
  </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>
```

Appendix B

Error codes and descriptions

Error code	Description
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.

continued on next page

continued from previous page

Error code	Description
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_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.

continued on next page

continued from previous page

Error code	Description
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_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 SRP password file.
GNUTLS_E_SRP_PWD_PARSING_ERROR	Parsing error in SRP password file.
GNUTLS_E_SUCCESS	Success.
GNUTLS_E_TOO_MANY_EMPTY_PACKETS	Too many empty record packets have been received.

continued on next page

continued from previous page

Error code	Description
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_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.

Table B.1: The error codes table

Appendix C

All the supported ciphersuites in *GnuTLS*

Cipher suite	TLS value	defined at
TLS_RSA_NULL_MD5	0x00 0x01	RFC2246
TLS_ANON_DH_3DES_EDE_CBC_SHA	0x00 0x1B	RFC2246
TLS_ANON_DH_ARCFOUR_MD5	0x00 0x18	RFC2246
TLS_ANON_DH_AES_128_CBC_SHA	0x00 0x34	RFC2246
TLS_ANON_DH_AES_256_CBC_SHA	0x00 0x3A	RFC2246
TLS_RSA_ARCFOUR_SHA	0x00 0x05	RFC2246
TLS_RSA_ARCFOUR_MD5	0x00 0x04	RFC2246
TLS_RSA_3DES_EDE_CBC_SHA	0x00 0x0A	RFC2246
TLS_RSA_EXPORT_ARCFOUR_40_MD5	0x00 0x03	RFC2246
TLS_DHE_DSS_3DES_EDE_CBC_SHA	0x00 0x13	RFC2246
TLS_DHE_RSA_3DES_EDE_CBC_SHA	0x00 0x16	RFC2246
TLS_RSA_AES_128_CBC_SHA	0x00 0x2F	RFC3268
TLS_RSA_AES_128_CBC_SHA	0x00 0x35	RFC3268
TLS_DHE_DSS_AES_256_CBC_SHA	0x00 0x38	RFC3268
TLS_DHE_DSS_AES_128_CBC_SHA	0x00 0x32	RFC3268
TLS_DHE_RSA_AES_256_CBC_SHA	0x00 0x39	RFC3268
TLS_DHE_RSA_AES_128_CBC_SHA	0x00 0x33	RFC3268
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

continued on next page

continued from previous page

Cipher suite	TLS value	defined at
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_3DES_EDE_CBC_RMD	0x00 0x72	draft-ietf-tls-openpgp-keys
TLS_DHE_RSA_3DES_EDE_CBC_RMD	0x00 0x77	draft-ietf-tls-openpgp-keys
TLS_DHE_DSS_AES_256_CBC_RMD	0x00 0x73	draft-ietf-tls-openpgp-keys
TLS_DHE_DSS_AES_128_CBC_RMD	0x00 0x74	draft-ietf-tls-openpgp-keys
TLS_DHE_RSA_AES_128_CBC_RMD	0x00 0x78	draft-ietf-tls-openpgp-keys
TLS_DHE_RSA_AES_256_CBC_RMD	0x00 0x79	draft-ietf-tls-openpgp-keys
TLS_RSA_3DES_EDE_CBC_RMD	0x00 0x7C	draft-ietf-tls-openpgp-keys
TLS_RSA_AES_128_CBC_RMD	0x00 0x7D	draft-ietf-tls-openpgp-keys
TLS_RSA_AES_256_CBC_RMD	0x00 0x7E	draft-ietf-tls-openpgp-keys
TLS_DHE_DSS_ARCFOUR_SHA	0x00 0x66	draft-ietf-tls-56-bit-ciphersuites

Table C.1: The ciphersuites table

Appendix D

GNU Free Documentation License

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 noncommercially. 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

4. MODIFICATIONS

Index

Bibliography

[1] Mike Ashley. The gnu privacy handbook, 2002. Available from http://www.gnupg.org/gph/en/manual.pdf.

[2] 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.

[3] 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.

[4] Tim Dierks and Christopher Allen. The tls protocol version 1.0, January 1999. Available from http://kaizi.viagenie.qc.ca/ietf/rfc/rfc2246.txt.

[5] 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.

[6] Peter Gutmann. Everything you never wanted to know about pki but were forced to find out, 2002. Available from http://www.cs.auckland.ac.nz/~ pgut001/pubs/pkitutorial.pdf.

[7] Scott Hollenbeck. Transport layer security protocol compression methods, January 2004. Internet draft, work in progress. Available from http://kaizi.viagenie.qc.ca/ietf/rfc/rfc3749.txt.

[8] 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.

[9] 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.

[10] RSA Laboratories. Pkcs 12 v1.0: Personal information exchange syntax, June 1999.

[11] Nikos Mavroyanopoulos. 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.

[12] Bodo Moeller. Security of cbc ciphersuites in ssl/tls: Problems and countermeasures, 2002. Available from http://www.openssl.org/˜ bodo/tls-cbc.txt.

[13] 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.

[14] 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.

[15] Eric Rescola. Http over tls, May 2000. Available from http://kaizi.viagenie.qc.ca/ietf/rfc/rfc2818.txt.

[16] Eric Rescola. Ssl and tls: Designing and building secure systems, 2001.

[17] David Taylor, Trevor Perrin, Tom Wu, and Nikos Mavroyanopoulos. Using srp for tls authentication, June 2004. Internet draft, work in progress. Available from http://www.normos.org/ietf/draft/draft-ietf-tls-srp-07.txt.

[18] Tom Wu. The stanford srp authentication project. Available at http://srp.stanford.edu/.

[19] Tom Wu. The srp authentication and key exchange system, September 2000. Available from http://kaizi.viagenie.qc.ca/ietf/rfc/rfc2945.txt.