Secure Socket Layer

Secure Socket Layer 3.10.8
21 2017

21 2017

1 User's Guide

The *SSL* application provides secure communication over sockets.

This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/).

This product includes cryptographic software written by Eric Young (eay@cryptsoft.com).

This product includes software written by Tim Hudson (tjh@cryptsoft.com).

For full OpenSSL and SSLeay license texts, see *Licenses*.

1.1 The SSL Protocol

Here we provide a short introduction to the SSL protocol. We only consider those part of the protocol that are important from a programming point of view.

For a very good general introduction to SSL and TLS see the book .

Outline:

- Two types of connections - connection: handshake, data transfer, and shutdown - SSL/TLS protocol - server must have certificate - what the the server sends to the client - client may verify the server - server may ask client for certificate - what the client sends to the server - server may then verify the client - verification - certificate chains - root certificates - public keys - key agreement - purpose of certificate - references

1.1.1 SSL Connections

The SSL protocol is implemented on top of the TCP/IP protocol. From an endpoint view it also has the same type of connections as that protocol, almost always created by calls to socket interface functions *listen*, *accept* and *connect*. The endpoints are *servers* and *clients*.

A *serverlisten*s for connections on a specific address and port. This is done once. The server then *accept*s each connections on that same address and port. This is typically done indefinitely many times.

A *client* connects to a server on a specific address and port. For each purpose this is done once.

For a plain TCP/IP connection the establishment of a connection (through an accept or a connect) is followed by data transfer between the client and server, finally ended by a connection close.

An SSL connection also consists of data transfer and connection close, However, the data transfer contains encrypted data, and in order to establish the encryption parameters, the data transfer is preceded by an SSL *handshake*. In this handshake the server plays a dominant role, and the main instrument used in achieving a valid SSL connection is the server's *certificate*. We consider certificates in the next section, and the SSL handshake in a subsequent section.

1.1.2 Certificates

A certificate is similar to a driver's license, or a passport. The holder of the certificate is called the *subject*. First of all the certificate identifies the subject in terms of the name of the subject, its postal address, country name, company name (if applicable), etc.

Although a driver's license is always issued by a well-known and distinct authority, a certificate may have an *issuer* that is not so well-known. Therefore a certificate also always contains information on the issuer of the certificate. That information is of the same type as the information on the subject. The issuer of a certificate also signs the certificate with a *digital signature* (the signature is an inherent part of the certificate), which allow others to verify that the issuer really is the issuer of the certificate.

Now that a certificate can be checked by verifying the signature of the issuer, the question is how to trust the issuer. The answer to this question is to require that there is a certificate for the issuer as well. That issuer has in turn an issuer, which must also have a certificate, and so on. This *certificate chain* has to have en end, which then must be a certificate that is trusted by other means. We shall cover this problem of *authentication* in a subsequent section.

1.1.3 Encryption Algorithms

An encryption algorithm is a mathematical algorithm for encryption and decryption of messages (arrays of bytes, say). The algorithm as such is always required to be publicly known, otherwise its strength cannot be evaluated, and hence it cannot be used reliably. The secrecy of an encrypted message is not achieved by the secrecy of the algorithm used, but by the secrecy of the *keys* used as input to the encryption and decryption algorithms. For an account of cryptography in general see .

There are two classes of encryption algorithms: *symmetric key* algorithms and *public key* algorithms. Both types of algorithms are used in the SSL protocol.

In the sequel we assume holders of keys keep them secret (except public keys) and that they in that sense are trusted. How a holder of a secret key is proved to be the one it claims to be is a question of *authentication*, which, in the context of the SSL protocol, is described in a section further below.

Symmetric Key Algorithms

A *symmetric key* algorithm has one key only. The key is used for both encryption and decryption. Obviously the key of a symmetric key algorithm must always be kept secret by the users of the key. DES is an example of a symmetric key algorithm.

Symmetric key algorithms are fast compared to public key algorithms. They are therefore typically used for encrypting bulk data.

Public Key Algorithms

A *public key* algorithm has two keys. Any of the two keys can be used for encryption. A message encrypted with one of the keys, can only be decrypted with the other key. One of the keys is public (known to the world), while the other key is private (i.e. kept secret) by the owner of the two keys.

RSA is an example of a public key algorithm.

Public key algorithms are slow compared to symmetric key algorithms, and they are therefore seldom used for bulk data encryption. They are therefore only used in cases where the fact that one key is public and the other is private, provides features that cannot be provided by symmetric algorithms.

Digital Signature Algorithms

An interesting feature of a public key algorithm is that its public and private keys can both be used for encryption. Anyone can use the public key to encrypt a message, and send that message to the owner of the private key, and be sure of that only the holder of the private key can decrypt the message.

On the other hand, the owner of the private key can encrypt a message with the private key, thus obtaining an encrypted message that can decrypted by anyone having the public key.

The last approach can be used as a digital signature algorithm. The holder of the private key signs an array of bytes by performing a specified well-known *message digest algorithm* to compute a hash of the array, encrypts the hash value with its private key, an then presents the original array, the name of the digest algorithm, and the encryption of the hash value as a *signed array of bytes*.

Now anyone having the public key, can decrypt the encrypted hash value with that key, compute the hash with the specified digest algorithm, and check that the hash values compare equal in order to verify that the original array was indeed signed by the holder of the private key.

What we have accounted for so far is by no means all that can be said about digital signatures (see for further details).

Message Digests Algorithms

A message digest algorithm is a hash function that accepts an array bytes of arbitrary but finite length of input, and outputs an array of bytes of fixed length. Such an algorithm is also required to be very hard to invert.

MD5 (16 bytes output) and SHA1 (20 bytes output) are examples of message digest algorithms.

1.1.4 SSL Handshake

The main purpose of the handshake performed before an an SSL connection is established is to negotiate the encryption algorithm and key to be used for the bulk data transfer between the client and the server. We are writing *the* key, since the algorithm to choose for bulk encryption one of the symmetric algorithms.

There is thus only one key to agree upon, and obviously that key has to be kept secret between the client and the server. To obtain that the handshake has to be encrypted as well.

The SSL protocol requires that the server always sends its certificate to the client in the beginning of the handshake. The client then retrieves the server's public key from the certificate, which means that the client can use the server's public key to encrypt messages to the server, and the server can decrypt those messages with its private key. Similarly, the server can encrypt messages to the client with its private key, and the client can decrypt messages with the server's public key. It is thus is with the server's public and private keys that messages in the handshake are encrypted and decrypted, and hence the key agreed upon for symmetric encryption of bulk data can be kept secret (there are more things to consider to really keep it secret, see).

The above indicates that the server does not care who is connecting, and that only the client has the possibility to properly identify the server based on the server's certificate. That is indeed true in the minimal use of the protocol, but it is possible to instruct the server to request the certificate of the client, in order to have a means to identify the client, but it is by no means required to establish an SSL connection.

If a server request the client certificate, it verifies, as a part of the protocol, that the client really holds the private key of the certificate by sending the client a string of bytes to encrypt with its private key, which the server then decrypts with the client's public key, the result of which is compared with the original string of bytes (a similar procedure is always performed by the client when it has received the server's certificate).

The way clients and servers *authenticate* each other, i.e. proves that their respective peers are what they claim to be, is the topic of the next section.

1.1.5 Authentication

As we have already seen the reception of a certificate from a peer is not enough to prove that the peer is authentic. More certificates are needed, and we have to consider how certificates are issued and on what grounds.

Certificates are issued by *certification authorities* (*CA*s) only. They issue certificates both for other CAs and ordinary users (which are not CAs).

Certain CAs are *top CAs*, i.e. they do not have a certificate issued by another CA. Instead they issue their own certificate, where the subject and issuer part of the certificate are identical (such a certificate is called a self-signed certificate). A top CA has to be well-known, and has to have a publicly available policy telling on what grounds it issues certificates.

There are a handful of top CAs in the world. You can examine the certificates of several of them by clicking through the menus of your web browser.

A top CA typically issues certificates for other CAs, called *intermediate CAs*, but possibly also to ordinary users. Thus the certificates derivable from a top CA constitute a tree, where the leaves of the tree are ordinary user certificates.

A *certificate chain* is an ordered sequence of certificates, `C1, C2, ..., Cn`, say, where `C1` is a top CA certificate, and where `Cn` is an ordinary user certificate, and where the holder of `C1` is the issuer of `C2`, the holder of `C2` is the issuer of `C3`, ..., and the holder of `Cn-1` is the issuer of `Cn`, the ordinary user certificate. The holders of `C2, C3, ..., Cn-1` are then intermediate CAs.

Now to verify that a certificate chain is unbroken we have to take the public key from each certificate `Ck`, and apply that key to decrypt the signature of certificate `Ck-1`, thus obtaining the message digest computed by the holder of the `Ck` certificate, compute the real message digest of the `Ck-1` certificate and compare the results. If they compare equal the link of the chain between `Ck` and `Ck-1` is considered to unbroken. This is done for each link k = 1, 2, ..., n-1. If all links are found to be unbroken, the user certificate `Cn` is considered authenticated.

Trusted Certificates

Now that there is a way to authenticate a certificate by checking that all links of a certificate chain are unbroken, the question is how you can be sure to trust the certificates in the chain, and in particular the top CA certificate of the chain.

To provide an answer to that question consider the perspective of a client, which have just received the certificate of the server. In order to authenticate the server the client has to construct a certificate chain and to prove that the chain is unbroken. The client has to have a set of CA certificates (top CA or intermediate CA certificates) not obtained from the server, but obtained by other means. Those certificates are kept `locally` by the client, and are trusted by the client.

More specifically, the client does not really have to have top CA certificates in its local storage. In order to authenticate a server it is sufficient for the client to posses the trusted certificate of the issuer of the server certificate.

Now that is not the whole story. A server can send an (incomplete) certificate chain to its client, and then the task of the client is to construct a certificate chain that begins with a trusted certificate and ends with the server's certificate. (A client can also send a chain to its server, provided the server requested the client's certificate.)

All this means that an unbroken certificate chain begins with a trusted certificate (top CA or not), and ends with the peer certificate. That is the end of the chain is obtained from the peer, but the beginning of the chain is obtained from local storage, which is considered trusted.

1.2 Using the SSL application

Here we provide an introduction to using the Erlang/OTP SSL application, which is accessed through the `ssl` interface module.

We also present example code in the Erlang module `client_server`, also provided in the directory `ssl-X.Y.Z/examples`, with source code in `src` and the compiled module in `ebin` of that directory.

1.2.1 The ssl Module

The `ssl` module provides the user interface to the Erlang/OTP SSL application. The interface functions provided are very similar to those provided by the `gen_tcp` and `inet` modules.

Servers use the interface functions `listen` and `accept`. The `listen` function specifies a TCP port to to listen to, and each call to the `accept` function establishes an incoming connection.

Clients use the `connect` function which specifies the address and port of a server to connect to, and a successful call establishes such a connection.

The `listen` and `connect` functions have almost all the options that the corresponding functions in `gen_tcp/` have, but there are also additional options specific to the SSL protocol.

The most important SSL specific option is the `cacertfile` option which specifies a local file containing trusted CA certificates which are and used for peer authentication. This option is used by clients and servers in case they want to authenticate their peers.

The `certfile` option specifies a local path to a file containing the certificate of the holder of the connection endpoint. In case of a server endpoint this option is mandatory since the contents of the sever certificate is needed in the the handshake preceding the establishment of a connection.

Similarly, the `keyfile` option points to a local file containing the private key of the holder of the endpoint. If the `certfile` option is present, this option has to be specified as well, unless the private key is provided in the same file as specified by the `certfile` option (a certificate and a private key can thus coexist in the same file).

The `verify` option specifies how the peer should be verified:

0
: Do not verify the peer,

1
: Verify peer,

2
: Verify peer, fail the verification if the peer has no certificate.

The `depth` option specifies the maximum length of the verification certificate chain. Depth = 0 means the peer certificate, depth = 1 the CA certificate, depth = 2 the next CA certificate etc. If the verification process does not find a trusted CA certificate within the maximum length, the verification fails.

The `ciphers` option specifies which ciphers to use (a string of colon separated cipher names). To obtain a list of available ciphers, evaluate the `ssl:ciphers/0` function (the SSL application has to be running).

1.2.2 A Client-Server Example

Here is a simple client server example.

1.3 PKIX Certificates

1.3.1 Introduction to Certificates

Certificates were originally defined by ITU (CCITT) and the latest definitions are described in , but those definitions are (as always) not working.

Working certificate definitions for the Internet Community are found in the the PKIX RFCs and . The parsing of certificates in the Erlang/OTP SSL application is based on those RFCS.

Certificates are defined in terms of ASN.1 (). For an introduction to ASN.1 see **ASN.1 Information Site**.

1.3.2 PKIX Certificates

Certificate handling is now handled by the `public_key` application.

DER encoded certificates returned by `ssl:peercert/1` can for example be decoded by the `public_key:pkix_decode_cert/2` function.

1.4 Creating Certificates

Here we consider the creation of example certificates.

1.4.1 The openssl Command

The `openssl` command is a utility that comes with the OpenSSL distribution. It provides a variety of subcommands. Each subcommand is invoked as

```

```

```
openssl subcmd <options and arguments>
```

where `subcmd` denotes the subcommand in question.

We shall use the following subcommands to create certificates for the purpose of testing Erlang/OTP SSL:

- *req* to create certificate requests and a self-signed certificates,
- *ca* to create certificates from certificate requests.

We create the following certificates:

- the *erlangCA* root certificate (a self-signed certificate),
- the *otpCA* certificate signed by the *erlangCA*,
- a client certificate signed by the *otpCA*, and
- a server certificate signed by the *otpCA*.

The openssl configuration file

An `openssl` configuration file consist of a number of sections, where each section starts with one line containing `[ section_name ]`, where `section_name` is the name of the section. The first section of the file is either unnamed, or is named `[ default ]`. For further details see the OpenSSL config(5) manual page.

The required sections for the subcommands we are going to use are as follows:

subcommand	required/default section	override command line option	configuration file option
req	[req]	-	`-config FILE`
ca	[ca]	`-name section`	`-config FILE`

Table 4.1: openssl subcommands to use

Creating the Erlang root CA

The Erlang root CA is created with the command

```
openssl req -new -x509 -config /some/path/req.cnf \\
               -keyout /some/path/key.pem -out /some/path/cert.pem
```

where the option `-new` indicates that we want to create a new certificate request and the option `-x509` implies that a self-signed certificate is created.

Creating the OTP CA

The OTP CA is created by first creating a certificate request with the command

```
openssl req -new -config /some/path/req.cnf \\
        -keyout /some/path/key.pem -out /some/path/req.pem
```

and the ask the Erlang CA to sign it:

```
openssl ca -batch -notext -config /some/path/req.cnf \\
        -extensions ca_cert -in /some/path/req.pem -out /some/path/cert.pem
```

where the option `-extensions` refers to a section in the configuration file saying that it should create a CA certificate, and not a plain user certificate.

The `client` and `server` certificates are created similarly, except that the option `-extensions` then has the value `user_cert`.

1.4.2 An Example

The following module `create_certs` is used by the Erlang/OTP SSL application for generating certificates to be used in tests. The source code is also found in `ssl-X.Y.Z/examples/certs/src`.

The purpose of the `create_certs:all/1` function is to make it possible to provide from the `erl` command line, the full path name of the `openssl` command.

Note that the module creates temporary OpenSSL configuration files for the `req` and `ca` subcommands.

1.5 Using SSL for Erlang Distribution

This chapter describes how the Erlang distribution can use SSL to get additional verification and security.

1.5.1 Introduction

The Erlang distribution can in theory use almost any connection based protocol as bearer. A module that implements the protocol specific parts of connection setup is however needed. The default distribution module is `inet_tcp_dist` which is included in the Kernel application. When starting an Erlang node distributed, `net_kernel` uses this module to setup listen ports and connections.

In the SSL application there is an additional distribution module, `inet_ssl_dist` which can be used as an alternative. All distribution connections will be using SSL and all participating Erlang nodes in a distributed system must use this distribution module.

The security depends on how the connections are set up, one can use key files or certificates to just get a crypted connection. One can also make the SSL package verify the certificates of other nodes to get additional security. Cookies are however always used as they can be used to differentiate between two different Erlang networks.

Setting up Erlang distribution over SSL involves some simple but necessary steps:

- Building boot scripts including the SSL application
- Specifying the distribution module for net_kernel
- Specifying security options and other SSL options

The rest of this chapter describes the above mentioned steps in more detail.

1.5.2 Building boot scripts including the SSL application

Boot scripts are built using the `systools` utility in the SASL application. Refer to the SASL documentations for more information on systools. This is only an example of what can be done.

The simplest boot script possible includes only the Kernel and STDLIB applications. Such a script is located in the Erlang distributions bin directory. The source for the script can be found under the Erlang installation top directory under `releases/<OTP version>start_clean.rel`. Copy that script to another location (and preferably another name) and add the SSL application with its current version number after the STDLIB application.

An example .rel file with SSL added may look like this:

```
{release, {"OTP  APN 181 01","P7A"}, {erts, "5.0"},
 [{kernel,"2.5"},
  {stdlib,"1.8.1"},
  {ssl,"2.2.1"}]}.
```

Note that the version numbers surely will differ in your system. Whenever one of the applications included in the script is upgraded, the script has to be changed.

Assuming the above .rel file is stored in a file `start_ssl.rel` in the current directory, a boot script can be built like this:

```
1> systools:make_script("start_ssl",[]).
```

There will now be a file `start_ssl.boot` in the current directory. To test the boot script, start Erlang with the `-boot` command line parameter specifying this boot script (with its full path but without the `.boot` suffix), in Unix it could look like this:

```
$ erl -boot /home/me/ssl/start_ssl
Erlang (BEAM) emulator version 5.0

Eshell V5.0  (abort with ^G)
1> whereis(ssl_server).
<0.32.0>
```

The `whereis` function call verifies that the SSL application is really started.

As an alternative to building a bootscript, one can explicitly add the path to the ssl `ebin` directory on the command line. This is done with the command line option `-pa`. This works as the ssl application really need not be started for the distribution to come up, a primitive version of the ssl server is started by the distribution module itself, so as long as the primitive code server can reach the code, the distribution will start. The `-pa` method is only recommended for testing purposes.

1.5.3 Specifying distribution module for net_kernel

The distribution module for SSL is named `inet_ssl_dist` and is specified on the command line whit the `-proto_dist` option. The argument to `-proto_dist` should be the module name without the `_dist` suffix, so this distribution module is specified with `-proto_dist inet_ssl` on the command line.

Extending the command line from above gives us the following:

```
$ erl -boot /home/me/ssl/start_ssl -proto_dist inet_ssl
```

For the distribution to actually be started, we need to give the emulator a name as well:

```
$ erl -boot /home/me/ssl/start_ssl -proto_dist inet_ssl -sname ssl_test
Erlang (BEAM) emulator version 5.0 [source]

Eshell V5.0  (abort with ^G)
(ssl_test@myhost)1>
```

Note however that a node started in this way will refuse to talk to other nodes, as no certificates or key files are supplied (see below).

When the SSL distribution starts, the OTP system is in its early boot stage, why neither `application` nor `code` are usable. As SSL needs to start a port program in this early stage, it tries to determine the path to that program from the primitive code loaders code path. If this fails, one need to specify the directory where the port program resides. This can be done either with an environment variable `ERL_SSL_PORTPROGRAM_DIR` or with the command line option `-ssl_portprogram_dir`. The value should be the directory where the `ssl_esock` port program is located. Note that this option is never needed in a normal Erlang installation.

1.5.4 Specifying security options and other SSL options

For SSL to work, you either need certificate files or a key file. Certificate files can be specified both when working as client and as server (connecting or accepting).

On the `erl` command line one can specify options that the ssl distribution will add when creation a socket. It is mandatory to specify at least a key file or client and server certificates. One can specify any *SSL option* on the command line, but must not specify any socket options (like packet size and such). The SSL options are listed in the Reference Manual. The only difference between the options in the reference manual and the ones that can be specified to the distribution on the command line is that `certfile` can (and usually needs to) be specified as `client_certfile` and `server_certfile`. The `client_certfile` is used when the distribution initiates a connection to another node and the `server_cerfile` is used when accepting a connection from a remote node.

The command line argument for specifying the SSL options is named `-ssl_dist_opt` and should be followed by an even number of SSL options/option values. The `-ssl_dist_opt` argument can be repeated any number of times.

An example command line would now look something like this (line breaks in the command are for readability, they should not be there when typed):

```
$ erl -boot /home/me/ssl/start_ssl -proto_dist inet_ssl
  -ssl_dist_opt client_certfile "/home/me/ssl/erlclient.pem"
  -ssl_dist_opt server_certfile "/home/me/ssl/erlserver.pem"
  -ssl_dist_opt verify 1 depth 1
  -sname ssl_test
Erlang (BEAM) emulator version 5.0 [source]

Eshell V5.0  (abort with ^G)
(ssl_test@myhost)1>
```

A node started in this way will be fully functional, using SSL as the distribution protocol.

1.5.5 Setting up environment to always use SSL

A convenient way to specify arguments to Erlang is to use the `ERL_FLAGS` environment variable. All the flags needed to use SSL distribution can be specified in that variable and will then be interpreted as command line arguments for all subsequent invocations of Erlang.

In a Unix (Bourne) shell it could look like this (line breaks for readability):

```
$ ERL_FLAGS="-boot \\"/home/me/ssl/start_ssl\\" -proto_dist inet_ssl
  -ssl_dist_opt client_certfile \\"/home/me/ssl/erlclient.pem\\"
  -ssl_dist_opt server_certfile \\"/home/me/ssl/erlserver.pem\\"
  -ssl_dist_opt verify 1 -ssl_dist_opt depth 1"
$ export ERL_FLAGS
$ erl -sname ssl_test
```

```
Erlang (BEAM) emulator version 5.0 [source]

Eshell V5.0  (abort with ^G)
(ssl_test@myhost)1> init:get_arguments().
[{root,["/usr/local/erlang"]},
 {progname,["erl "]},
 {sname,["ssl_test"]},
 {boot,["/home/me/ssl/start_ssl"]},
 {proto_dist,["inet_ssl"]},
 {ssl_dist_opt,["client_certfile","/home/me/ssl/erlclient.pem"]},
 {ssl_dist_opt,["server_certfile","/home/me/ssl/erlserver.pem"]},
 {ssl_dist_opt,["verify","1"]},
 {ssl_dist_opt,["depth","1"]},
 {home,["/home/me"]}]
```

The `init:get_arguments()` call verifies that the correct arguments are supplied to the emulator.

1.6 Licenses

This chapter contains in extenso versions of the OpenSSL and SSLeay licenses.

1.6.1 OpenSSL License

```
/* ====================================================================
 * Copyright (c) 1998-2002 The OpenSSL Project.  All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in
 *    the documentation and/or other materials provided with the
 *    distribution.
 *
 * 3. All advertising materials mentioning features or use of this
 *    software must display the following acknowledgment:
 *    "This product includes software developed by the OpenSSL Project
 *    for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
 *
 * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
 *    endorse or promote products derived from this software without
 *    prior written permission. For written permission, please contact
 *    openssl-core@openssl.org.
 *
 * 5. Products derived from this software may not be called "OpenSSL"
 *    nor may "OpenSSL" appear in their names without prior written
 *    permission of the OpenSSL Project.
 *
 * 6. Redistributions of any form whatsoever must retain the following
 *    acknowledgment:
 *    "This product includes software developed by the OpenSSL Project
 *    for use in the OpenSSL Toolkit (http://www.openssl.org/)"
 *
 * THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
 * EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
```

```
 * PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE OpenSSL PROJECT OR
 * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
 * NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
 * STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
 * OF THE POSSIBILITY OF SUCH DAMAGE.
 * ====================================================================
 *
 * This product includes cryptographic software written by Eric Young
 * (eay@cryptsoft.com).  This product includes software written by Tim
 * Hudson (tjh@cryptsoft.com).
 *
 */
```

1.6.2 SSLeay License

```
/* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
 * All rights reserved.
 *
 * This package is an SSL implementation written
 * by Eric Young (eay@cryptsoft.com).
 * The implementation was written so as to conform with Netscapes SSL.
 *
 * This library is free for commercial and non-commercial use as long as
 * the following conditions are aheared to.  The following conditions
 * apply to all code found in this distribution, be it the RC4, RSA,
 * lhash, DES, etc., code; not just the SSL code.  The SSL documentation
 * included with this distribution is covered by the same copyright terms
 * except that the holder is Tim Hudson (tjh@cryptsoft.com).
 *
 * Copyright remains Eric Young's, and as such any Copyright notices in
 * the code are not to be removed.
 * If this package is used in a product, Eric Young should be given attribution
 * as the author of the parts of the library used.
 * This can be in the form of a textual message at program startup or
 * in documentation (online or textual) provided with the package.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:
 *    "This product includes cryptographic software written by
 *     Eric Young (eay@cryptsoft.com)"
 *    The word 'cryptographic' can be left out if the rouines from the library
 *    being used are not cryptographic related :-).
 * 4. If you include any Windows specific code (or a derivative thereof) from
 *    the apps directory (application code) you must include an acknowledgement:
 *    "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
 *
 * THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
```

```
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * The licence and distribution terms for any publically available version or
 * derivative of this code cannot be changed.  i.e. this code cannot simply be
 * copied and put under another distribution licence
 * [including the GNU Public Licence.]
 */
```

2 Reference Manual

The *SSL* application provides secure communication over sockets.

This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/).

This product includes cryptographic software written by Eric Young (eay@cryptsoft.com).

This product includes software written by Tim Hudson (tjh@cryptsoft.com).

For full OpenSSL and SSLeay license texts, see *Licenses*.

ssl

Application

The Secure Socket Layer (SSL) application provides secure socket communication over TCP/IP.

Warning

In previous versions of Erlang/OTP SSL it was advised, as a work-around, to set the operating system environment variable `SSL_CERT_FILE` to point at a file containing CA certificates. That variable is no longer needed, and is not recognised by Erlang/OTP SSL any more.

However, the OpenSSL package does interpret that environment variable. Hence a setting of that variable might have unpredictable effects on the Erlang/OTP SSL application. It is therefore adviced to not used that environment variable at all.

Environment

The following application environment configuration parameters are defined for the SSL application. Refer to application(3) for more information about configuration parameters.

Note that the environment parameters can be set on the command line, for instance,

`erl ... -ssl protocol_version '[sslv2,sslv3]' ....`

`ephemeral_rsa = true | false <optional>`

Enables all SSL servers (those that listen and accept) to use ephemeral RSA key generation when a clients connect with weak handshake cipher specifications, that need equally weak ciphers from the server (i.e. obsolete restrictions on export ciphers). Default is `false`.

`debug = true | false <optional>`

Causes debug information to be written to standard output. Default is `false`.

`debugdir = path() | false <optional>`

Causes debug information output controlled by `debug` and `msgdebug` to be printed to a file named `ssl_esock.<pid>.log` in the directory specified by `debugdir`, where `<pid>` is the operating system specific textual representation of the process identifier of the external port program of the SSL application. Default is `false`, i.e. no log file is produced.

`msgdebug = true | false <optional>`

Sets `debug = true` and causes also the contents of low level messages to be printed to standard output. Default is `false`.

`port_program = string() | false <optional>`

Name of port program. The default is `ssl_esock`.

`protocol_version = [sslv2|sslv3|tlsv1] <optional>`.

Name of protocols to use. If this option is not set, all protocols are assumed, i.e. the default value is `[sslv2, sslv3, tlsv1]`.

`proxylsport = integer() | false <optional>`

Define the port number of the listen port of the SSL port program. Almost never is this option needed.

`proxylsbacklog = integer() | false <optional>`

Set the listen queue size of the listen port of the SSL port program. The default is 128.

OpenSSL libraries

The current implementation of the Erlang SSL application is based on the *OpenSSL* package version 0.9.7 or higher. There are source and binary releases on the web.

Source releases of OpenSSL can be downloaded from the **OpenSSL** project home page, or mirror sites listed there.

The same URL also contains links to some compiled binaries and libraries of OpenSSL (see the `Related/Binaries` menu) of which the **Shining Light Productions Win32 and OpenSSL** pages are of interest for the Win32 user.

For some Unix flavours there are binary packages available on the net.

If you cannot find a suitable binary OpenSSL package, you have to fetch an OpenSSL source release and compile it.

You then have to compile and install the libraries `libcrypto.so` and `libssl.so` (Unix), or the libraries `libeay32.dll` and `ssleay32.dll` (Win32).

For Unix The `ssl_esock` port program is delivered linked to OpenSSL libraries in `/usr/local/lib`, but the default dynamic linking will also accept libraries in `/lib` and `/usr/lib`.

If that is not applicable to the particular Unix operating system used, the example `Makefile` in the SSL `priv/obj` directory, should be used as a guide to relinking the final version of the port program.

For `Win32` it is only required that the libraries can be found from the `PATH` environment variable, or that they reside in the appropriate `SYSTEM32` directory; hence no particular relinking is need. Hence no example `Makefile` for Win32 is provided.

Restrictions

Users must be aware of export restrictions and patent rights concerning cryptographic software.

SEE ALSO

application(3)

ssl

Erlang module

This module contains interface functions to the Secure Socket Layer.

General

There is a new implementation of ssl available in this module but until it is 100 % complete, so that it can replace the old implementation in all aspects it will be described here *new ssl API*

The reader is advised to also read the `ssl(6)` manual page describing the SSL application.

Warning:

It is strongly advised to seed the random generator after the ssl application has been started (see `seed/1` below), and before any connections are established. Although the port program interfacing to the ssl libraries does a "random" seeding of its own in order to make everything work properly, that seeding is by no means random for the world since it has a constant value which is known to everyone reading the source code of the port program.

Common data types

The following datatypes are used in the functions below:

- `options() = [option()]`
- `option() = socketoption() | ssloption()`
- `socketoption() = {mode, list} | {mode, binary} | binary | {packet, packettype()} | {header, integer()} | {nodelay, boolean()} | {active, activetype()} | {backlog, integer()} | {ip, ipaddress()} | {port, integer()}`
- `ssloption() = {verify, code()} | {depth, depth()} | {certfile, path()} | {keyfile, path()} | {password, string()} | {cacertfile, path()} | {ciphers, string()}`
- `packettype()` (see inet(3))
- `activetype()` (see inet(3))
- `reason() = atom() | {atom(), string()}`
- `bytes() = [byte()]`
- `string() = [byte()]`
- `byte() = 0 | 1 | 2 | ... | 255`
- `code() = 0 | 1 | 2`
- `depth() = byte()`
- `address() = hostname() | ipstring() | ipaddress()`
- `ipaddress() = ipstring() | iptuple()`
- `hostname() = string()`
- `ipstring() = string()`
- `iptuple() = {byte(), byte(), byte(), byte()}`
- `sslsocket()`
- `protocol() = sslv2 | sslv3 | tlsv1`

•

The socket option {backlog, integer()} is for listen/2 only, and the option {port, integer()} is for connect/3/4 only.

The following socket options are set by default: {mode, list}, {packet, 0}, {header, 0}, {nodelay, false}, {active, true}, {backlog, 5}, {ip, {0,0,0,0}}, and {port, 0}.

Note that the options {mode, binary} and binary are equivalent. Similarly {mode, list} and the absence of option binary are equivalent.

The ssl options are for setting specific SSL parameters as follows:

- {verify, code()} Specifies type of verification: 0 = do not verify peer; 1 = verify peer, 2 = verify peer, fail if no peer certificate. The default value is 0.
- {depth, depth()} Specifies the maximum verification depth, i.e. how far in a chain of certificates the verification process can proceed before the verification is considered to fail.

  Peer certificate = 0, CA certificate = 1, higher level CA certificate = 2, etc. The value 2 thus means that a chain can at most contain peer cert, CA cert, next CA cert, and an additional CA cert.

  The default value is 1.
- {certfile, path()} Path to a file containing the user's certificate. chain of PEM encoded certificates.
- {keyfile, path()} Path to file containing user's private PEM encoded key.
- {password, string()} String containing the user's password. Only used if the private keyfile is password protected.
- {cacertfile, path()} Path to file containing PEM encoded CA certificates (trusted certificates used for verifying a peer certificate).
- {ciphers, string()} String of ciphers as a colon separated list of ciphers. The function ciphers/0 can be used to find all available ciphers.

The type sslsocket() is opaque to the user.

The owner of a socket is the one that created it by a call to transport_accept/[1,2], connect/[3,4], or listen/2.

When a socket is in active mode (the default), data from the socket is delivered to the owner of the socket in the form of messages:

- {ssl, Socket, Data}
- {ssl_closed, Socket}
- {ssl_error, Socket, Reason}

A Timeout argument specifies a timeout in milliseconds. The default value for a Timeout argument is infinity.

Functions listed below may return the value {error, closed}, which only indicates that the SSL socket is considered closed for the operation in question. It is for instance possible to have {error, closed} returned from an call to send/2, and a subsequent call to recv/3 returning {ok, Data}.

Hence a return value of {error, closed} must not be interpreted as if the socket was completely closed. On the contrary, in order to free all resources occupied by an SSL socket, close/1 must be called, or else the process owning the socket has to terminate.

For each SSL socket there is an Erlang process representing the socket. When a socket is opened, that process links to the calling client process. Implementations that want to detect abnormal exits from the socket process by receiving {'EXIT', Pid, Reason} messages, should use the function pid/1 to retrieve the process identifier from the socket, in order to be able to match exit messages properly.

Exports

`ciphers() -> {ok, string()} | {error, enotstarted}`

Returns a string consisting of colon separated cipher designations that are supported by the current SSL library implementation.

The SSL application has to be started to return the string of ciphers.

`close(Socket) -> ok | {error, Reason}`

Types:

Socket = sslsocket()

Closes a socket returned by `transport_accept/[1,2]`, `connect/[3,4]`, or `listen/2`

`connect(Address, Port, Options) -> {ok, Socket} | {error, Reason}`
`connect(Address, Port, Options, Timeout) -> {ok, Socket} | {error, Reason}`

Types:

Address = address()

Port = integer()

Options = [connect_option()]

connect_option() = {mode, list} | {mode, binary} | binary | {packet, packettype()} | {header, integer()} | {nodelay, boolean()} | {active, activetype()} | {ip, ipaddress()} | {port, integer()} | {verify, code()} | {depth, depth()} | {certfile, path()} | {keyfile, path()} | {password, string()} | {cacertfile, path()} | {ciphers, string()}

Timeout = integer()

Socket = sslsocket()

Connects to `Port` at `Address`. If the optional `Timeout` argument is specified, and a connection could not be established within the given time, `{error, timeout}` is returned. The default value for `Timeout` is `infinity`.

The `ip` and `port` options are for binding to a particular *local* address and port, respectively.

`connection_info(Socket) -> {ok, {Protocol, Cipher}} | {error, Reason}`

Types:

Socket = sslsocket()

Protocol = protocol()

Cipher = string()

Gets the chosen protocol version and cipher for an established connection (accepted och connected).

`controlling_process(Socket, NewOwner) -> ok | {error, Reason}`

Types:

Socket = sslsocket()

NewOwner = pid()

Assigns a new controlling process to `Socket`. A controlling process is the owner of a socket, and receives all messages from the socket.

`format_error(ErrorCode) -> string()`

Types:

ErrorCode = term()

Returns a diagnostic string describing an error.

getopts(Socket, OptionsTags) -> {ok, Options} | {error, Reason}

Types:

Socket = sslsocket()

OptionTags = [optiontag()]()

Returns the options the tags of which are `OptionTags` for for the socket `Socket`.

listen(Port, Options) -> {ok, ListenSocket} | {error, Reason}

Types:

Port = integer()

Options = [listen_option()]

listen_option() = {mode, list} | {mode, binary} | binary | {packet, packettype()} | {header, integer()} | {active, activetype()} | {backlog, integer()} | {ip, ipaddress()} | {verify, code()} | {depth, depth()} | {certfile, path()} | {keyfile, path()} | {password, string()} | {cacertfile, path()} | {ciphers, string()}

ListenSocket = sslsocket()

Sets up a socket to listen on port `Port` at the local host. If `Port` is zero, `listen/2` picks an available port number (use `port/1` to retrieve it).

The listen queue size defaults to 5. If a different value is wanted, the option `{backlog, Size}` should be added to the list of options.

An empty `Options` list is considered an error, and `{error, enooptions}` is returned.

The returned `ListenSocket` can only be used in calls to `transport_accept/[1,2]`.

peercert(Socket) -> {ok, Cert} | {error, Reason}

Types:

Socket = sslsocket()

Cert = binary()()

Subject = term()()

Returns the DER encoded peer certificate, the certificate can be decoded with `public_key:pkix_decode_cert/2`.

peername(Socket) -> {ok, {Address, Port}} | {error, Reason}

Types:

Socket = sslsocket()

Address = ipaddress()

Port = integer()

Returns the address and port number of the peer.

pid(Socket) -> pid()

Types:

Socket = sslsocket()

Returns the pid of the socket process. The returned pid should only be used for receiving exit messages.

recv(Socket, Length) -> {ok, Data} | {error, Reason}
recv(Socket, Length, Timeout) -> {ok, Data} | {error, Reason}

Types:

Socket = sslsocket()

Length = integer() >= 0

Timeout = integer()

Data = bytes() | binary()

Receives data on socket `Socket` when the socket is in passive mode, i.e. when the option `{active, false}` has been specified.

A notable return value is `{error, closed}` which indicates that the socket is closed.

A positive value of the `Length` argument is only valid when the socket is in raw mode (option `{packet, 0}` is set, and the option `binary` is *not* set); otherwise it should be set to 0, whence all available bytes are returned.

If the optional `Timeout` parameter is specified, and no data was available within the given time, `{error, timeout}` is returned. The default value for `Timeout` is `infinity`.

seed(Data) -> ok | {error, Reason}

Types:

Data = iolist() | binary()

Seeds the ssl random generator.

It is strongly advised to seed the random generator after the ssl application has been started, and before any connections are established. Although the port program interfacing to the OpenSSL libraries does a "random" seeding of its own in order to make everything work properly, that seeding is by no means random for the world since it has a constant value which is known to everyone reading the source code of the seeding.

A notable return value is `{error, edata}}` indicating that `Data` was not a binary nor an iolist.

send(Socket, Data) -> ok | {error, Reason}

Types:

Socket = sslsocket()

Data = iolist() | binary()

Writes `Data` to `Socket`.

A notable return value is `{error, closed}` indicating that the socket is closed.

setopts(Socket, Options) -> ok | {error, Reason}

Types:

Socket = sslsocket()

Options = [socketoption]()

Sets options according to `Options` for the socket `Socket`.

ssl_accept(Socket) -> ok | {error, Reason}
ssl_accept(Socket, Timeout) -> ok | {error, Reason}

Types:

Socket = sslsocket()

Timeout = integer()

Reason = atom()

The `ssl_accept` function establish the SSL connection on the server side. It should be called directly after `transport_accept`, in the spawned server-loop.

Note that the ssl connection is not complete until `ssl_accept` has returned `true`, and if an error is returned, the socket is unavailable and for instance `close/1` will crash.

`sockname(Socket) -> {ok, {Address, Port}} | {error, Reason}`

Types:

Socket = sslsocket()

Address = ipaddress()

Port = integer()

Returns the local address and port number of the socket `Socket`.

`transport_accept(Socket) -> {ok, NewSocket} | {error, Reason}`
`transport_accept(Socket, Timeout) -> {ok, NewSocket} | {error, Reason}`

Types:

Socket = NewSocket = sslsocket()

Timeout = integer()

Reason = atom()

Accepts an incoming connection request on a listen socket. `ListenSocket` must be a socket returned from `listen/2`. The socket returned should be passed to `ssl_accept` to complete ssl handshaking and establishing the connection.

> **Warning:**
>
> The socket returned can only be used with `ssl_accept`, no traffic can be sent or received before that call.

The accepted socket inherits the options set for `ListenSocket` in `listen/2`.

The default value for `Timeout` is `infinity`. If `Timeout` is specified, and no connection is accepted within the given time, `{error, timeout}` is returned.

`version() -> {ok, {SSLVsn, CompVsn, LibVsn}}`

Types:

SSLVsn = CompVsn = LibVsn = string()()

Returns the SSL application version (`SSLVsn`), the library version used when compiling the SSL application port program (`CompVsn`), and the actual library version used when dynamically linking in runtime (`LibVsn`).

If the SSL application has not been started, `CompVsn` and `LibVsn` are empty strings.

ERRORS

The possible error reasons and the corresponding diagnostic strings returned by `format_error/1` are either the same as those defined in the `inet(3)` reference manual, or as follows:

`closed`

Connection closed for the operation in question.

`ebadsocket`

Connection not found (internal error).

`ebadstate`

Connection not in connect state (internal error).

`ebrokertype`

Wrong broker type (internal error).

`ecacertfile`

Own CA certificate file is invalid.

`ecertfile`

Own certificate file is invalid.

`echaintoolong`

The chain of certificates provided by peer is too long.

`ecipher`

Own list of specified ciphers is invalid.

`ekeyfile`

Own private key file is invalid.

`ekeymismatch`

Own private key does not match own certificate.

`enoissuercert`

Cannot find certificate of issuer of certificate provided by peer.

`enoservercert`

Attempt to do accept without having set own certificate.

`enotlistener`

Attempt to accept on a non-listening socket.

`enoproxysocket`

No proxy socket found (internal error).

`enooptions`

The list of options is empty.

`enotstarted`

The SSL application has not been started.

`eoptions`

Invalid list of options.

`epeercert`

Certificate provided by peer is in error.

`epeercertexpired`

Certificate provided by peer has expired.

`epeercertinvalid`

Certificate provided by peer is invalid.

`eselfsignedcert`

Certificate provided by peer is self signed.

`esslaccept`

Server SSL handshake procedure between client and server failed.

`esslconnect`

Client SSL handshake procedure between client and server failed.

`esslerrssl`

SSL protocol failure. Typically because of a fatal alert from peer.

`ewantconnect`

Protocol wants to connect, which is not supported in this version of the SSL application.

`ex509lookup`

Protocol wants X.509 lookup, which is not supported in this version of the SSL application.

`{badcall, Call}`

Call not recognized for current mode (active or passive) and state of socket.

`{badcast, Cast}`

Call not recognized for current mode (active or passive) and state of socket.

`{badinfo, Info}`

Call not recognized for current mode (active or passive) and state of socket.

SEE ALSO

gen_tcp(3), inet(3) public_key(3)

new_ssl

Erlang module

This module contains interface functions to the Secure Socket Layer.

NEW SSL

This manual page describes functions that are defined in the ssl module and represents the new ssl implementation that coexists with the old one, as the new implementation is not yet complete enough to replace the old one.

The new implementation can be accessed by providing the option {ssl_imp, new} to the ssl:connect and ssl:listen functions.

The new implementation is Erlang based and all logic is in Erlang and only payload encryption calculations are done in C via the crypto application. The main reason for making a new implementation is that the old solution was very crippled as the control of the ssl-socket was deep down in openssl making it hard if not impossible to support all inet options, ipv6 and upgrade of a tcp connection to a ssl connection. This version has a few limitations that will be removed before the ssl-4.0 release. Main differences and limitations are listed below.

- New ssl requires the crypto application.
- The option reuseaddr is supported and the default value is false as in gen_tcp. Old ssl is patched to accept that the option is set to true to provide a smoother migration between the versions. In old ssl the option is hard coded to true.
- ssl:version/0 is replaced by ssl:versions/0
- ssl:ciphers/0 is replaced by ssl:cipher_suites/0
- ssl:pid/1 is a meaningless function in new ssl and will be deprecated in ssl-4.0 until it is removed it will return a valid but meaningless pid.
- New API functions are ssl:shutdown/2, ssl:cipher_suites/[0,1] and ssl:versions/0
- Diffie-Hellman keyexchange is not supported yet.
- CRL and policy certificate extensions are not supported yet.
- Supported SSL/TLS-versions are SSL-3.0 and TLS-1.0
- For security reasons sslv2 is not supported.

COMMON DATA TYPES

The following data types are used in the functions below:

`boolean() = true | false`

`property() = atom()`

`option() = socketoption() | ssloption() | transportoption()`

`socketoption() = [{property(), term()}] - defaults to [{mode,list},{packet, 0},{header, 0},{active, true}].`

For valid options see *inet(3)* and *gen_tcp(3)* .

`ssloption() = {verify, verify_type()} | {fail_if_no_peer_cert, boolean()} {depth, integer()} | {certfile, path()} | {keyfile, path()} | {password, string()} | {cacertfile, path()} | {ciphers, ciphers()} | {ssl_imp, ssl_imp()} | {reuse_sessions, boolean()} | {reuse_session, fun()}`

`transportoption() = {CallbackModule, DataTag, ClosedTag} - defaults to {gen_tcp, tcp, tcp_closed}. Ssl may be run over any reliable transport protocol that has an equivalent API to gen_tcp's.`

    CallbackModule = atom()

    DataTag = atom() - tag used in socket data message.

    ClosedTag = atom() - tag used in socket close message.

`verify_type() = verify_none | verify_peer`

`path() = string() - representing a file path.`

`host() = hostname() | ipaddress()`

`hostname() = string()`

`ip_address() = {N1,N2,N3,N4} % IPv4 | {K1,K2,K3,K4,K5,K6,K7,K8} % IPv6`

`sslsocket() - opaque to the user.`

`protocol() = sslv3 | tlsv1`

`ciphers() = [ciphersuite()] | sting() (according to old API)`

`ciphersuite() = {key_exchange(), cipher(), hash(), exportable()}`

`key_exchange() = rsa | dh_dss | dh_rsa | dh_anon | dhe_dss | dhe_rsa | krb5 | KeyExchange_export`

`cipher() = rc4_128 | idea_cbc | des_cbc | '3des_ede_cbc' des40_cbc | dh_dss | aes_128_cbc | aes_256_cbc | rc2_cbc_40 | rc4_40`

`hash() = md5 | sha`

`exportable() = export | no_export | ignore`

`ssl_imp() = new | old - default is old.`

SSL OPTION DESCRIPTIONS

{verify, verify_type()}

If `verify_none` is specified x509-certificate path validation errors at the client side will not automatically cause the connection to fail, as it will if the verify type is `verify_peer`. See also the option verify_fun. Servers only do the path validation if `verify_peer` is set to true, as it then will send a certificate request to the client (this message is not sent if the verify option is `verify_none`) and you may then also want to specify the option `fail_if_no_peer_cert`.

{fail_if_no_peer_cert, boolean()}

Used together with {verify, verify_peer} by a ssl server. If set to true, the server will fail if the client does not have a certificate to send, e.i sends a empty certificate, if set to false it will only fail if the client sends a invalid certificate (an empty certificate is considered valid).

{verify_fun, fun(ErrorList) -> boolean()}

Used by the ssl client to determine if x509-certificate path validations errors are acceptable or if the connection should fail. Defaults to:

```
fun(ErrorList) ->
 case lists:foldl(fun({bad_cert,unknown_ca}, Acc) ->
        Acc;
       (Other, Acc) ->
     [Other | Acc]
    end, [], ErrorList) of
     [] ->
```

```
  true;
     [_|_] ->
  false
 end
end
```

I.e. by default if the only error found was that the CA-certificate holder was unknown this will be accepted. Possible errors in the error list are: {bad_cert, cert_expired}, {bad_cert, invalid_issuer}, {bad_cert, invalid_signature}, {bad_cert, name_not_permitted}, {bad_cert, unknown_ca}, {bad_cert, cert_expired}, {bad_cert, invalid_issuer}, {bad_cert, invalid_signature}, {bad_cert, name_not_permitted}, {bad_cert, cert_revoked} (not implemented yet), {bad_cert, unknown_critical_extension} or {bad_cert, term()} (Will be relevant later when an option is added for the user to be able to verify application specific extensions.)

{depth, integer()}
: Specifies the maximum verification depth, i.e. how far in a chain of certificates the verification process can proceed before the verification is considered to fail. Peer certificate = 0, CA certificate = 1, higher level CA certificate = 2, etc. The value 2 thus means that a chain can at most contain peer cert, CA cert, next CA cert, and an additional CA cert. The default value is 1.

{certfile, path()}
: Path to a file containing the user's certificate. Optional for clients but note that some servers requires that the client can certify itself.

{keyfile, path()}
: Path to file containing user's private PEM encoded key. As PEM-files may contain several entries this option defaults to the same file as given by certfile option.

{password, string()}
: String containing the user's password. Only used if the private keyfile is password protected.

{cacertfile, path()}
: Path to file containing PEM encoded CA certificates (trusted certificates used for verifying a peer certificate). May be omitted if you do not want to verify the peer.

{ciphers, ciphers()}
: The function `ciphers_suites/0` can be used to find all available ciphers.

{ssl_imp, ssl_imp()}
: Specify which ssl implementation you want to use.

{reuse_sessions, boolean()}
: Specifies if ssl sessions should be reused when possible.

{reuse_session, fun(SuggestedSessionId, PeerCert, Compression, CipherSuite) -> boolean()}
: Enables the ssl server to have a local policy for deciding if a session should be reused or not, only meaning full if `reuse_sessions` is set to true. SuggestedSessionId is a binary(), PeerCert is a DER encoded certificate, Compression is an enumeration integer and CipherSuite of type ciphersuite().

General

When a ssl socket is in active mode (the default), data from the socket is delivered to the owner of the socket in the form of messages:

- {ssl, Socket, Data}
- {ssl_closed, Socket}
- {ssl_error, Socket, Reason}

A `Timeout` argument specifies a timeout in milliseconds. The default value for a `Timeout` argument is `infinity`.

Exports

`cipher_suites() ->`

cipher_suites(Type) -> ciphers()

Types:

Type = erlang | openssl

Returns a list of supported cipher suites. cipher_suites() is equivalent to cipher_suites(erlang). Type openssl is provided for backwards compatibility with old ssl that used openssl.

connect(Socket, SslOptions) ->
connect(Socket, SslOptions, Timeout) -> {ok, SslSocket} | {error, Reason}

Types:

Socket = socket()

SslOptions = [ssloption()]

Timeout = integer() | infinity

SslSocket = sslsocket()

Reason = term()

Upgrades a gen_tcp, or equivalent, connected socket to a ssl socket e.i performs the client-side ssl handshake.

connect(Host, Port, Options) ->
connect(Host, Port, Options, Timeout) -> {ok, SslSocket} | {error, Reason}

Types:

Host = host()

Port = integer()

Options = [option()]

Timeout = integer() | infinity

SslSocket = sslsocket()

Reason = term()

Opens an ssl connection to Host, Port.

close(SslSocket) -> ok | {error, Reason}

Types:

SslSocket = sslsocket()

Reason = term()

Close a ssl connection.

controlling_process(SslSocket, NewOwner) -> ok | {error, Reason}

Types:

SslSocket = sslsocket()

NewOwner = pid()

Reason = term()

Assigns a new controlling process to the ssl-socket. A controlling process is the owner of a ssl-socket, and receives all messages from the socket.

connection_info(SslSocket) -> {ok, {ProtocolVersion, CipherSuite}} | {error, Reason}

Types:

CipherSuite = ciphersuite()

ProtocolVersion = protocol()

Returns the negotiated protocol version and cipher suite.

`getopts(Socket) ->`
`getopts(Socket, OptionNames) -> {ok, [socketoption()]} | {error, Reason}`

Types:

Socket = sslsocket()

OptionNames = [property()]

Get the value of the specified socket options, if no options are specified all options are returned.

`listen(Port, Options) -> {ok, ListenSocket} | {error, Reason}`

Types:

Port = integer()

Options = options()

ListenSocket = sslsocket()

Creates a ssl listen socket.

`peercert(Socket) -> {ok, Cert} | {error, Reason}`

Types:

Socket = sslsocket()

Cert = binary()

Subject = term()

The peer certificate is returned as a DER encoded binary. The certificate can be decoded with `public_key:pkix_decode_cert/2`.

`peername(Socket) -> {ok, {Address, Port}} | {error, Reason}`

Types:

Socket = sslsocket()

Address = ipaddress()

Port = integer()

Returns the address and port number of the peer.

`recv(Socket, Length) ->`
`recv(Socket, Length, Timeout) -> {ok, Data} | {error, Reason}`

Types:

Socket = sslsocket()

Length = integer()

Timeout = integer()

Data = [char()] | binary()

This function receives a packet from a socket in passive mode. A closed socket is indicated by a return value `{error, closed}`.

The `Length` argument is only meaningful when the socket is in `raw` mode and denotes the number of bytes to read. If `Length` = 0, all available bytes are returned. If `Length` > 0, exactly `Length` bytes are returned, or an error; possibly discarding less than `Length` bytes of data when the socket gets closed from the other side.

The optional `Timeout` parameter specifies a timeout in milliseconds. The default value is `infinity`.

`send(Socket, Data) -> ok | {error, Reason}`

Types:

Socket = sslsocket()

Data = iolist() | binary()

Writes `Data` to `Socket`.

A notable return value is `{error, closed}` indicating that the socket is closed.

`setopts(Socket, Options) -> ok | {error, Reason}`

Types:

Socket = sslsocket()

Options = [socketoption]()

Sets options according to `Options` for the socket `Socket`.

`shutdown(Socket, How) -> ok | {error, Reason}`

Types:

Socket = sslsocket()

How = read | write | read_write

Reason = reason()

Immediately close a socket in one or two directions.

`How == write` means closing the socket for writing, reading from it is still possible.

To be able to handle that the peer has done a shutdown on the write side, the `{exit_on_close, false}` option is useful.

`ssl_accept(ListenSocket) ->`
`ssl_accept(ListenSocket, Timeout) -> ok | {error, Reason}`

Types:

ListenSocket = sslsocket()

Timeout = integer()

Reason = term()

The `ssl_accept` function establish the SSL connection on the server side. It should be called directly after `transport_accept`, in the spawned server-loop.

`ssl_accept(ListenSocket, SslOptions) ->`
`ssl_accept(ListenSocket, SslOptions, Timeout) -> {ok, Socket} | {error, Reason}`

Types:

ListenSocket = socket()

SslOptions = ssloptions()

Timeout = integer()

Reason = term()

Upgrades a gen_tcp, or equivalent, socket to a ssl socket e.i performs the ssl server-side handshake.

`sockname(Socket) -> {ok, {Address, Port}} | {error, Reason}`

Types:

Socket = sslsocket()

Address = ipaddress()

Port = integer()

Returns the local address and port number of the socket `Socket`.

`start() ->`
`start(Type) -> ok | {error, Reason}`

Types:

Type = permanent | transient | temporary

Starts the Ssl application. Default type is temporary. *application(3)*

`stop() -> ok`

Stops the Ssl application. *application(3)*

`transport_accept(Socket) ->`
`transport_accept(Socket, Timeout) -> {ok, NewSocket} | {error, Reason}`

Types:

Socket = NewSocket = sslsocket()

Timeout = integer()

Reason = reason()

Accepts an incoming connection request on a listen socket. `ListenSocket` must be a socket returned from `listen/2`. The socket returned should be passed to `ssl_accept` to complete ssl handshaking and establishing the connection.

> **Warning:**
>
> The socket returned can only be used with `ssl_accept`, no traffic can be sent or received before that call.

The accepted socket inherits the options set for `ListenSocket` in `listen/2`.

The default value for `Timeout` is `infinity`. If `Timeout` is specified, and no connection is accepted within the given time, `{error, timeout}` is returned.

`versions() -> [{SslAppVer, SupportedSslVer, AvailableSslVsn}]`

Types:

SslAppVer = string()

SupportedSslVer = [protocol()]

AvailableSslVsn = [protocol()]

Returns version information relevant for the ssl application.

SEE ALSO

inet(3) and *gen_tcp(3)*