



---

## Secure Socket Layer

Copyright © 1999-7 2010 Ericsson AB. All Rights Reserved.  
Secure Socket Layer 4.1.1  
December 7 2010

---

**Copyright © 1999-7 2010 Ericsson AB. All Rights Reserved.**

The contents of this file are subject to the Erlang Public License, Version 1.1, (the "License"); you may not use this file except in compliance with the License. You should have received a copy of the Erlang Public License along with this software. If not, it can be retrieved online at <http://www.erlang.org/>. Software distributed under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License. Ericsson AB. All Rights Reserved..

**December 7 2010**



# 1 User's Guide

---

The *SSL* application provides secure communication over sockets.

## 1.1 Transport Layer Security (TLS) and its predecessor, Secure Socket Layer (SSL)

The erlang *ssl* application currently supports SSL 3.0 and TLS 1.0 RFC 2246, and will in the future also support later versions of TLS. SSL 2.0 is not supported.

By default erlang *ssl* is run over the TCP/IP protocol even though you could plug in an other reliable transport protocol with the same API as *gen\_tcp*.

If a client and server wants to use an upgrade mechanism, such as defined by RFC2817, to upgrade a regular TCP/IP connection to a *ssl* connection the erlang *ssl* API supports this. This can be useful for things such as supporting HTTP and HTTPS on the same port and implementing virtual hosting.

### 1.1.1 Security overview

To achieve authentication and privacy the client and server will perform a TLS Handshake procedure before transmitting or receiving any data. During the handshake they agree on a protocol version and cryptographic algorithms, they generate shared secrets using public key cryptographics and optionally authenticate each other with digital certificates.

### 1.1.2 Data Privacy and Integrity

A *symmetric key* algorithm has one key only. The key is used for both encryption and decryption. These algorithms are fast compared to public key algorithms (using two keys, a public and a private one) and are therefore typically used for encrypting bulk data.

The keys for the symmetric encryption are generated uniquely for each connection and are based on a secret negotiated in the TLS handshake.

The TLS handshake protocol and data transfer is run on top of the TLS Record Protocol that uses a keyed-hash MAC (Message Authenticity Code), or HMAC, to protect the message's data integrity. From the TLS RFC "A Message Authentication Code is a one-way hash computed from a message and some secret data. It is difficult to forge without knowing the secret data. Its purpose is to detect if the message has been altered."

### 1.1.3 Digital Certificates

A certificate is similar to a driver's license, or a passport. The holder of the certificate is called the *subject*. The certificate is signed with the private key of the issuer of the certificate. A chain of trust is build by having the issuer in its turn being certified by an other certificate and so on until you reach the so called root certificate that is self signed i.e. issued by itself.

Certificates are issued by *certification authorities* (CAs) only. There are a handful of top CAs in the world that issue root certificates. You can examine the certificates of several of them by clicking through the menus of your web browser.

### 1.1.4 Authentication of Sender

Authentication of the sender is done by public key path validation as defined in RFC 3280. Simplified that means that each certificate in the certificate chain is issued by the one before, the certificates attributes are valid ones, and the root cert is a trusted cert that is present in the trusted certs database kept by the peer.

The server will always send a certificate chain as part of the TLS handshake, but the client will only send one if the server requests it. If the client does not have an appropriate certificate it may send an "empty" certificate to the server.

The client may choose to accept some path evaluation errors for instance a web browser may ask the user if they want to accept an unknown CA root certificate. The server, if it request a certificate, will on the other hand not accept any path validation errors. It is configurable if the server should accept or reject an "empty" certificate as response to a certificate request.

### 1.1.5 TLS Sessions

From the TLS RFC "A TLS session is an association between a client and a server. Sessions are created by the handshake protocol. Sessions define a set of cryptographic security parameters, which can be shared among multiple connections. Sessions are used to avoid the expensive negotiation of new security parameters for each connection."

Session data is by default kept by the ssl application in a memory storage hence session data will be lost at application restart or takeover. Users may define their own callback module to handle session data storage if persistent data storage is required. Session data will also be invalidated after 24 hours from it was saved, for security reasons. It is of course possible to configure the amount of time the session data should be saved.

Ssl clients will by default try to reuse an available session, ssl servers will by default agree to reuse sessions when clients ask to do so.

## 1.2 Using the SSL API

### 1.2.1 General information

To see relevant version information for ssl you can call `ssl:versions/0`

To see all supported cipher suites call `ssl:cipher_suites/0`. Note that available cipher suites for a connection will depend on your certificate. It is also possible to specify a specific cipher suite(s) that you want your connection to use. Default is to use the strongest available.

### 1.2.2 Setting up connections

Here follows some small example of how to set up client/server connections using the erlang shell. The returned value of the `sslsocket` has been abbreviated with `[ . . . ]` as it can be fairly large and is opaque.

#### Minimal example

##### Note:

The minimal setup is not the most secure setup of ssl.

Start server side

```
1 server> ssl:start().
ok
```

## 1.2 Using the SSL API

---

Create a ssl listen socket

```
2 server> {ok, ListenSocket} =  
ssl:listen(9999, [{certfile, "cert.pem"}, {keyfile, "key.pem"}, {reuseaddr, true}]).  
{ok, {sslsocket, [...]}}
```

Do a transport accept on the ssl listen socket

```
3 server> {ok, Socket} = ssl:transport_accept(ListenSocket).  
{ok, {sslsocket, [...]}}
```

Start client side

```
1 client> ssl:start().  
ok
```

```
2 client> {ok, Socket} = ssl:connect("localhost", 9999, [], infinity).  
{ok, {sslsocket, [...]}}
```

Do the ssl handshake

```
4 server> ok = ssl:ssl_accept(Socket).  
ok
```

Send a message over ssl

```
5 server> ssl:send(Socket, "foo").  
ok
```

Flush the shell message queue to see that we got the message sent on the server side

```
3 client> flush().  
Shell got {ssl, {sslsocket, [...]}, "foo"}  
ok
```

## Upgrade example

### Note:

To upgrade a TCP/IP connection to a ssl connection the client and server have to agree to do so. Agreement may be accomplished by using a protocol such the one used by HTTP specified in RFC 2817.

Start server side

```
1 server> ssl:start().  
ok
```

Create a normal tcp listen socket

```
2 server> {ok, ListenSocket} = gen_tcp:listen(9999, [{reuseaddr, true}]).
{ok, #Port<0.475>}
```

Accept client connection

```
3 server> {ok, Socket} = gen_tcp:accept(ListenSocket).
{ok, #Port<0.476>}
```

Start client side

```
1 client> ssl:start().
ok
```

```
2 client> {ok, Socket} = gen_tcp:connect("localhost", 9999, [], infinity).
```

Make sure active is set to false before trying to upgrade a connection to a ssl connection, otherwise ssl handshake messages may be delivered to the wrong process.

```
4 server> inet:setopts(Socket, [{active, false}]).
ok
```

Do the ssl handshake.

```
5 server> {ok, SSLSocket} = ssl:ssl_accept(Socket, [{cacertfile, "cacerts.pem"},
{certfile, "cert.pem"}, {keyfile, "key.pem"}]).
{ok, {sslsocket, [...]}}
```

Upgrade to a ssl connection. Note that the client and server must agree upon the upgrade and the server must call ssl:accept/2 before the client calls ssl:connect/3.

```
3 client> {ok, SSLSocket} = ssl:connect(Socket, [{cacertfile, "cacerts.pem"},
{certfile, "cert.pem"}, {keyfile, "key.pem"}], infinity).
{ok, {sslsocket, [...]}}
```

Send a message over ssl

```
4 client> ssl:send(SSLSocket, "foo").
ok
```

Set active true on the ssl socket

```
4 server> ssl:setopts(SSLSocket, [{active, true}]).
ok
```

Flush the shell message queue to see that we got the message sent on the client side

```
5 server> flush().
Shell got {ssl, {sslsocket, [...]}, "foo"}
ok
```

### 1.3 Using SSL for Erlang Distribution

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

#### Note:

Note this documentation is written for the old ssl implementation and will be updated for the new one once this functionality is supported by the new implementation.

#### 1.3.1 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 encrypted 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.

#### 1.3.2 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 bootscrip, 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.

### 1.3.3 Specifying distribution module for `net_kernel`

The distribution module for SSL is named `inet_ssl_dist` and is specified on the command line with 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.

### 1.3.4 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_certfile` 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.

### 1.3.5 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.

## 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 ([ey@cryptsoft.com](mailto:ey@cryptsoft.com)).

This product includes software written by Tim Hudson ([tjh@cryptsoft.com](mailto:tjh@cryptsoft.com)).

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

## ssl

---

Application

### 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 '[sslv3, tlsv1]' ....
```

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

Protocol that will be supported by started clients and servers. If this option is not set it will default to all protocols currently supported by the erlang ssl application. Note that this option may be overridden by the version option to [ssl:connect/2,3](#) and [ssl:listen/2](#).

```
session_lifetime = integer() <optional>
```

The lifetime of session data in seconds.

```
session_cb = atom() <optional>
```

Name of session cache callback module that implements the `ssl_session_cache_api` behavior, defaults to `ssl_session_cache.erl`.

```
session_cb_init_args = list() <optional>
```

List of arguments to the init function in session cache callback module, defaults to `[]`.

### SEE ALSO

[application\(3\)](#)

## ssl

---

Erlang module

This module contains interface functions to the Secure Socket Layer.

## SSL

- ssl requires the crypto and public\_key applications.
- Supported SSL/TLS-versions are SSL-3.0 and TLS-1.0
- For security reasons sslv2 is not supported.
- Ephemeral Diffie-Hellman cipher suites are supported but not Diffie Hellman Certificates cipher suites.
- Export cipher suites are not supported as the U.S. lifted its export restrictions in early 2000.
- CRL and policy certificate extensions are not supported yet.

## 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()`} | {`verify_fun`, {`fun()`, `term()`}} | {`fail_if_no_peer_cert`, `boolean()`} | {`depth`, `integer()`} | {`cert`, `der_bin()`} | {`certfile`, `path()`} | {`key`, `der_bin()`} | {`keyfile`, `path()`} | {`password`, `string()`} | {`cacerts`, [`der_bin()`]} | {`cacertfile`, `path()`} | {`dh`, `der_bin()`} | {`dhfile`, `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.

`der_bin()` = `binary()` - Asn1 DER encoded entity as an erlang binary.

`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()] | string() (according to old API)
ciphersuite() = {key_exchange(), cipher(), hash()}
key_exchange() = rsa | dhe_dss | dhe_rsa | dh_anon
cipher() = rc4_128 | des_cbc | '3des_edc_cbc' | aes_128_cbc | aes_256_cbc
hash() = md5 | sha
ssl_imp() = new | old - default is new.

```

## SSL OPTION DESCRIPTIONS - COMMON for SERVER and CLIENT

Options described here are options that have the same meaning in the client and the server.

{cert, der\_bin()}

The DER encoded users certificate. If this option is supplied it will override the certfile option.

{certfile, path()}

Path to a file containing the user's certificate.

{key, der\_bin()}

The DER encoded users private key. If this option is supplied it will override the keyfile option.

{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.

{cacerts, [der\_bin()]}

The DER encoded trusted certificates. If this option is supplied it will override the cacertfile option.

{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 cipher suites that should be supported. The function `cipher_suites/0` can be used to find all available ciphers. Additionally some anonymous cipher suites (`{dh_anon, rc4_128, md5}`, `{dh_anon, des_cbc, sha}`, `{dh_anon, '3des_edc_cbc', sha}`, `{dh_anon, aes_128_cbc, sha}`, `{dh_anon, aes_256_cbc, sha}`) are supported for testing purposes and will only work if explicitly enabled by this option and they are supported/enabled by the peer also.

{ssl\_imp, ssl\_imp()}

Specify which ssl implementation you want to use. Defaults to new.

{secure\_renegotiate, boolean()}

Specifies if to reject renegotiation attempt that does not live up to RFC 5746. By default `secure_renegotiate` is set to false i.e. secure renegotiation will be used if possible but it will fallback to unsecure renegotiation if the peer does not support RFC 5746.

{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.

{verify\_fun, {Verifyfun :: fun(), InitialUserState :: term()}}

The verification fun should be defined as:

```
fun(OtpCert :: #'OTPCertificate'{}, Event :: {bad_cert, Reason :: atom()}) |
```

```
{extension, #'Extension'{}}, InitialUserState :: term()) ->
{valid, UserState :: term()} | {valid_peer, UserState :: term()} |
{fail, Reason :: term()} | {unknown, UserState :: term()}.
```

The verify fun will be called during the X509-path validation when an error or an extension unknown to the ssl application is encountered. Additionally it will be called when a certificate is considered valid by the path validation to allow access to each certificate in the path to the user application. Note that it will differentiate between the peer certificate and CA certificates by using `valid_peer` or `valid` as the second argument to the verify fun. See *the public\_key User's Guide* for definition of `#'OTPCertificate'{}` and `#'Extension'{}.`

If the verify callback fun returns `{fail, Reason}`, the verification process is immediately stopped and an alert is sent to the peer and the TLS/SSL handshake is terminated. If the verify callback fun returns `{valid, UserState}`, the verification process is continued. If the verify callback fun always returns `{valid, UserState}`, the TLS/SSL handshake will not be terminated with respect to verification failures and the connection will be established. If called with an extension unknown to the user application the return value `{unknown, UserState}` should be used.

The default `verify_fun` option in `verify_peer` mode:

```
{fun(_, {bad_cert, _} = Reason, _) ->
  {fail, Reason};
  (_, {extension, _}, UserState) ->
  {unknown, UserState};
  (_, valid, UserState) ->
  {valid, UserState};
  (_, valid_peer, UserState) ->
  {valid, UserState}
end, []}
```

The default `verify_fun` option in `verify_none` mode:

```
{fun(_, {bad_cert, _}, UserState) ->
  {valid, UserState};
  (_, {extension, _}, UserState) ->
  {unknown, UserState};
  (_, valid, UserState) ->
  {valid, UserState};
  (_, valid_peer, UserState) ->
  {valid, UserState}
end, []}
```

Possible path validation errors:

```
{bad_cert, cert_expired}, {bad_cert, invalid_issuer}, {bad_cert, invalid_signature}, {bad_cert, unknown_ca},
{bad_cert, name_not_permitted}, {bad_cert, missing_basic_constraint}, {bad_cert, invalid_key_usage}
```

## SSL OPTION DESCRIPTIONS - CLIENT SIDE

Options described here are client specific or has a slightly different meaning in the client than in the server.

`{verify, verify_type()}`

In `verify_none` mode the default behavior will be to allow all x509-path validation errors. See also the `verify_fun` option.



`{reuse_sessions, boolean()}`

Specifies if client should try to reuse sessions when possible.

## SSL OPTION DESCRIPTIONS - SERVER SIDE

Options described here are server specific or has a slightly different meaning in the server than in the client.

`{dh, der_bin()}`

The DER encoded Diffie Hellman parameters. If this option is supplied it will override the `dhfile` option.

`{dhfile, path()}`

Path to file containing PEM encoded Diffie Hellman parameters, for the server to use if a cipher suite using Diffie Hellman key exchange is negotiated. If not specified default parameters will be used.

`{verify, verify_type()}`

Servers only do the x509-path validation in `verify_peer` mode, 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, i.e. 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).

`{reuse_sessions, boolean()}`

Specifies if the server should agree to reuse sessions when the clients request to do so. See also the `reuse_session` option.

`{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 i.e. 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.

---

```
format_error(Reason) -> string()
```

Types:

```
Reason = term()
```

Presents the error returned by an ssl function as a printable string.

```
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.

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

Types:

```
Socket = sslsocket()
```

```
Cert = binary()
```

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`.

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

Types:

**`Socket = sslsocket()`**

Initiates a new handshake. A notable return value is `{error, renegotiation_rejected}` indicating that the peer refused to go through with the renegotiation but the connection is still active using the previously negotiated session.

**`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 i.e. performs the ssl server-side handshake.

### Warning:

Note that the listen socket should be in `{active, false}` mode before telling the client that the server is ready to upgrade and calling this function, otherwise the upgrade may or may not succeed depending on timing.

**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)*

## old\_ssl

Erlang module

This module contains interface functions to the Secure Socket Layer.

### General

This manual page describes functions that are defined in the `ssl` module and represents the old `ssl` implementation that coexists with the new one until it has been totally phased out.

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

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()]0**

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].

**peer\_cert(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.

epeer-cert

Certificate provided by peer is in error.

epeer-cert-expired

Certificate provided by peer has expired.

epeer-cert-invalid

Certificate provided by peer is invalid.

eself-signed-cert

Certificate provided by peer is self signed.

essl-accept

Server SSL handshake procedure between client and server failed.

essl-connect

Client SSL handshake procedure between client and server failed.

essl-error-ssl

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

ewant-connect

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

ex509-lookup

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)

## ssl\_session\_cache\_api

---

Erlang module

### Common Data Types

The following data types are used in the functions below:

`cache_ref()` = `opaque()`

`key()` = {`partialkey()`, `session_id()`}

`partialkey()` = `opaque()`

`session_id()` = `binary()`

`session()` = `opaque()`

### Exports

**`delete(Cache, Key) -> _`**

Types:

**`Cache = cache_ref()`**

**`Key = key()`**

Deletes a cache entry. Will only be called from the cache handling process.

**`foldl(Fun, Acc0, Cache) -> Acc`**

Types:

Calls `Fun(Elem, AccIn)` on successive elements of the cache, starting with `AccIn == Acc0`. `Fun/2` must return a new accumulator which is passed to the next call. The function returns the final value of the accumulator. `Acc0` is returned if the cache is empty.

**`init() -> opaque()`**

Types:

Performs possible initializations of the cache and returns a reference to it that will be used as parameter to the other api functions. Will be called by the cache handling processes init function, hence putting the same requirements on it as a normal process init function.

**`lookup(Cache, Key) -> Entry`**

Types:

**`Cache = cache_ref()`**

**`Key = key()`**

**`Entry = session() | undefined`**

Looks up a cache entry. Should be callable from any process.

**`select_session(Cache, PartialKey) -> [session()]`**

Types:

**`Cache = cache_ref()`**

**PartialKey = partialkey()**

**Session = session()**

Selects sessions that could be reused. Should be callable from any process.

**terminate(Cache) -> \_**

Types:

**Cache = term() - as returned by init/0**

Takes care of possible cleanup that is needed when the cache handling process terminates.

**update(Cache, Key, Session) -> \_**

Types:

**Cache = cache\_ref()**

**Key = key()**

**Session = session()**

Caches a new session or updates a already cached one. Will only be called from the cache handling process.