Secure Socket Layer

version 3.10
## Contents

1 SSL User's Guide  
   1.1 SSL Connections ....................................... 1  
   1.2 Certificates .......................................... 2  
   1.3 Encryption Algorithms .................................... 2  
      1.3.1 Symmetric Key Algorithms ............................. 2  
      1.3.2 Public Key Algorithms ................................ 2  
      1.3.3 Digital Signature Algorithms ......................... 3  
      1.3.4 Message Digests Algorithms .......................... 3  
   1.4 SSL Handshake ........................................ 3  
   1.5 Authentication ........................................ 4  
      1.5.1 Trusted Certificates ......................... 4  

2 Using the SSL application  
   2.1 The ssl Module ........................................ 7  
   2.2 A Client-Server Example ................................... 8  

3 PKIX Certificates  
   3.1 Introduction to Certificates .............................. 11  
   3.2 PKIX Certificates ...................................... 11  
      3.2.1 Certificate and TBSCertificate ....................... 11  
      3.2.2 TBSCertificate issuer and subject .................. 12  
      3.2.3 TBSCertificate extensions ......................... 13  

4 Creating Certificates  
   4.1 The openssl Command .................................... 15  
      4.1.1 The openssl configuration file ...................... 16  
      4.1.2 Creating the Erlang root CA ....................... 16  
      4.1.3 Creating the OTP CA ............................... 16  
   4.2 An Example ............................................ 17
Chapter 1

SSL User’s Guide

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 SSL and TLS [1].

Outline:

- Two types of connections - connection: handshake, data transfer, and shutdown - SSL/TLS protocol - server must have certificate - what 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 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 server listens for connections on a specific address and port. This is done once. The server then accepts each connection 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.
Chapter 1: SSL User's Guide

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.

1.3 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 Applied Cryptography [2].

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.

1.3.1 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.

1.3.2 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.
1.3.3 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 "Applied Cryptography" [2] for further details).

1.3.4 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.

1.4 SSL Handshake

The main purpose of the handshake performed before 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 "SSL and TLS" [1]).

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.

1.5 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 (CAs) 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, $C_1, C_2, \ldots, C_n$, say, where $C_1$ is a top CA certificate, and where $C_n$ is an ordinary user certificate, and where the holder of $C_1$ is the issuer of $C_2$, the holder of $C_2$ is the issuer of $C_3$, ..., and the holder of $C_{n-1}$ is the issuer of $C_n$, the ordinary user certificate. The holders of $C_2, C_3, \ldots, C_{n-1}$ are then intermediate CAs.

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

1.5.1 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.
Chapter 2

Using the SSL application

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

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

2.1 The \texttt{ssl} Module

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

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

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

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

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

The \texttt{keyfile} 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 server certificate is needed in the handshake preceding the establishment of a connection.

Similarly, the \texttt{keyfile} option points to a local file containing the private key of the holder of the endpoint. If the \texttt{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 \texttt{certfile} option (a certificate and a private key can thus coexist in the same file).

The \texttt{verify} option specifies how the peer should be verified:

\begin{itemize}
  \item \texttt{0} Do not verify the peer,
  \item \texttt{1} Verify peer,
  \item \texttt{2} Verify peer, fail the verification if the peer has no certificate.
\end{itemize}
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).

2.2 A Client-Server Example

Here is a simple client server example.

```erlang
%%% Purpose: Example of SSL client and server using example certificates.
-module(client_server).
-export([start/0, start/1, init_connect/1]).

start() ->
    start([ssl, subject]).

start(CertOpts) ->
    %% Start ssl application
    application:start(ssl),

    %% Always seed
    ssl:seed("ellynatefttidppohjeh"),

    %% Let the current process be the server that listens and accepts
    %% Listen
    {ok, LSock} = ssl:listen(0, mk_opts(listen)),
    {ok, {_, LPort}} = ssl:sockname(LSock),
    io:fwrite("Listen: port = ~w.~n", [LPort]),
```
%% Spawn the client process that connects to the server
spawn(?MODULE, init_connect, [{LPort, CertOpts}]),

%%% Accept
{ok, ASock} = ssl:transport_accept(LSock),
ok = ssl:ssl_accept(LSock),
io:fwrite("Accept: accepted.\n"),
{ok, Cert} = ssl:peercert(ASock, CertOpts),
io:fwrite("Accept: peer cert: \n\n\n", [Cert]),
io:fwrite("Accept: sending \n\n\n"),
ssl:send(ASock, "hello"),
{error, closed} = ssl:recv(ASock, 0),
io:fwrite("Accept: detected closed.\n"),
ssl:close(ASock),
io:fwrite("Listen: closing and terminating.\n"),
ssl:close(LSock),
application:stop(ssl).

%%% Client connect
init_connect({LPort, CertOpts}) ->
  {ok, Host} = inet:gethostname(),
  {ok, CSock} = ssl:connect(Host, LPort, mk_opts(connect)),
io:fwrite("Connect: connected.\n"),
  {ok, Cert} = ssl:peercert(CSock, CertOpts),
io:fwrite("Connect: peer cert: \n\n\n", [Cert]),
  {ok, Data} = ssl:recv(CSock, 0),
io:fwrite("Connect: got data: \n\n\n", [Data]),
io:fwrite("Connect: closing and terminating.\n"),
ssl:close(CSock).

mk_opts(listen) ->
  mk_opts("server");
mk_opts(connect) ->
  mk_opts("client");
mk_opts(Role) ->
  Dir = filename:join([code:lib_dir(ssl), "examples", "certs", "etc"],
  [{active, false},
   {verify, 2},
   {depth, 2},
   {cacertfile, filename:join([Dir, Role, "cacerts.pem"])}],
  [{certfile, filename:join([Dir, Role, "cert.pem"])}],
  [{keyfile, filename:join([Dir, Role, "key.pem"])}].
Chapter 3

PKIX Certificates

3.1 Introduction to Certificates

Certificates were originally defined by ITU (CCITT) and the latest definitions are described in ITU-T X.509 [3], but those definitions are (as always) not working.

Working certificate definitions for the Internet Community are found in the the PKIX RFCs RFC 3279 [4] and RFC 3280 [5]. The parsing of certificates in the Erlang/OTP SSL application is based on those RFCs.

Certificates are defined in terms of ASN.1 (ITU-T X.680 [6]). For an introduction to ASN.1 see ASN.1 Information Site1.

3.2 PKIX Certificates

Here we base the PKIX certificate definitions in RFCs RFC 3279 [4] and RFC 3280 [5]. We however present the definitions according to SSL-PKIX.asn1 module, which is an amelioration of the PKIX1Explicit88.asn1, PKIX1Implicit88.asn1, and PKIX1Algorithms88.asn1 modules. You find all these modules in the pkix subdirectory of SSL.

The Erlang terms that are returned by the functions ssl:peercert/1/2, ssl_pkix:decode_cert/1/2, and ssl_pkix:decode_cert_file/1/2 when the option ssl is used in those functions, correspond the ASN.1 structures described in the sequel.

3.2.1 Certificate and TBSCertificate

Certificate ::= SEQUENCE { 
tbsCertificate TBSCertificate, 
signatureAlgorithm SignatureAlgorithm, 
signature BIT STRING }

TBSCertificate ::= SEQUENCE { 
version [0] Version DEFAULT v1, 
serialNumber CertificateSerialNumber, 
signature SignatureAlgorithm, 
issuer Name,

1URL: http://asn1.elibel.tm.fr/
Chapter 3: PKIX Certificates

```plaintext
validity       Validity, 
subject       Name, 
subjectPublicKeyInfo SubjectPublicKeyInfo, 
issuerUniqueID [1] IMPLICIT UniqueIdentifier OPTIONAL, 
    -- If present, version MUST be v2 or v3 
subjectUniqueID [2] IMPLICIT UniqueIdentifier OPTIONAL, 
    -- If present, version MUST be v2 or v3 
extensions [3] Extensions OPTIONAL 
    -- If present, version MUST be v3 -- }

Version ::= INTEGER { v1(0), v2(1), v3(2) }
CertificateSerialNumber ::= INTEGER
Validity ::= SEQUENCE {
    notBefore   Time, 
    notAfter    Time }
Time ::= CHOICE {
    utcTime    UTCTime, 
    generalTime GeneralizedTime }
```

The meaning of the fields `version`, `serialNumber`, and `validity` are quite obvious given the type definitions above, so we do not go further into their details.

The `signatureAlgorithm` field of `Certificate` and the `signature` field of `TBSCertificate` contain the name and parameters of the algorithm used for signing the certificate. The values of these two fields must be equal.

The `issuer` and `subject` fields can contain many different types of data, and is therefore considered in a separate section. The same holds for the `extensions` field. The `issuerUniqueID` and the `subjectUniqueID` fields are not considered further.

3.2.2 TBSCertificate issuer and subject

```plaintext
Name ::= CHOICE { -- only one possibility for now --
    rdnSequence   RDNSequence }
RDNSequence ::= SEQUENCE OF RelativeDistinguishedName
DistinguishedName ::= RDNSequence
RelativeDistinguishedName ::= SET SIZE (1 .. MAX) OF AttributeTypeAndValue
AttributeTypeAndValue ::= SEQUENCE {
    type ATTRIBUTE-TYPE-AND-VALUE-CLASS.&id
        ({{SupportedAttributeTypeAndValues}},
        ({{SupportedAttributeTypeAndValues}@type})
    value ATTRIBUTE-TYPE-AND-VALUE-CLASS.&Type
        ({{SupportedAttributeTypeAndValues}@type})
```

SSL Application
3.2: PKIX Certificates

SupportedAttributeTypeAndValues

\[\text{ATTRIBUTE-TYPE-AND-VALUE-CLASS} := \{
\text{name} | \text{surname} | \text{givenName} | \text{initials} | \text{generationQualifier} |
\text{commonName} | \text{localityName} | \text{stateOrProvinceName} | \text{organizationName} |
\text{organizationalUnitName} | \text{title} | \text{dnQualifier} | \text{countryName} |
\text{serialNumber} | \text{pseudonym} | \text{domainComponent} | \text{emailAddress}\}\]

3.2.3 TBSCertificate extensions

The `extensions` field of a `TBSCertificate` is a sequence of type `Extension`, defined as follows:

\[\text{Extension} ::= \text{SEQUENCE} \{\text{extnID} \text{ OBJECT IDENTIFIER, critical BOOLEAN DEFAULT FALSE, extnValue ANY}\}\]

Each extension has a unique object identifier. An extension with a `critical` value set to `TRUE` must be recognised by the reader of a certificate, or else the certificate must be rejected.

Extensions are divided into two groups: standard extensions and internet certificate extensions. All extensions listed in the table that follows are standard extensions, except for `authorityInfoAccess` and `subjectInfoAccess`, which are internet extensions.

Depending on the object identifier the `extnValue` is parsed into an appropriate well-defined structure.

The following table shows the purpose of each extension, but does not specify the structure. To see the structure consult the `PKIX1Implicit88.asn1` module.

<table>
<thead>
<tr>
<th>Extension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>authorityKeyIdentifier</td>
<td>Used by to identify a certificate signed that has multiple signing keys.</td>
</tr>
<tr>
<td>subjectKeyIdentifier</td>
<td>Used to identify certificates that contain a public key. Must appear i CA certificates.</td>
</tr>
<tr>
<td>keyUsage</td>
<td>Defines the purpose of the certificate. Can be one or several of digitalSignature, nonRepudiation, keyEncipherment, dataEncipherment, keyAgreement, keyCertSign, cRLSign, encipherOnly, decipherOnly.</td>
</tr>
<tr>
<td>privateKeyUsagePeriod</td>
<td>Allows certificate issuer to provide a private key usage period to be short than the certificate usage period.</td>
</tr>
<tr>
<td>certificatePolicies</td>
<td>Contains one or more policy information terms indicating the policies under which the certificate has been issued.</td>
</tr>
<tr>
<td>policyMappings</td>
<td>Used i CA certificates.</td>
</tr>
<tr>
<td>subjectAltName</td>
<td>Allows additional identities to be bound the the subject.</td>
</tr>
<tr>
<td>issuerAltName</td>
<td>Allows additional identities to be bound the the issuer.</td>
</tr>
<tr>
<td>subjectDirectoryAttributes</td>
<td>Conveys identity attributes of the subject.</td>
</tr>
<tr>
<td>basicConstraints</td>
<td>Tells if the certificate holder is a CA or not.</td>
</tr>
<tr>
<td>nameConstraints</td>
<td>Used in CA certificates.</td>
</tr>
<tr>
<td>policyConstraints</td>
<td>Used in CA certificates.</td>
</tr>
<tr>
<td>extKeyUsage</td>
<td>Indicates for which purposed the public key may be used.</td>
</tr>
<tr>
<td>cRLDistributionPoints</td>
<td>Indicates how CRL (Certificate Revocation List) information is obtained.</td>
</tr>
<tr>
<td>inhibitAnyPolicy</td>
<td>Used i CA certificates.</td>
</tr>
</tbody>
</table>

continued ...
Chapter 3: PKIX Certificates

... continued

<table>
<thead>
<tr>
<th>Extension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>freshestCRL</td>
<td>For CRLs</td>
</tr>
<tr>
<td>authorityInfoAccess</td>
<td>How to access CA information of the issuer of the certificate.</td>
</tr>
<tr>
<td>subjectInfoAccess</td>
<td>How to access CA information of the subject of the certificate.</td>
</tr>
</tbody>
</table>

Table 3.1: PKIX Extensions
Chapter 4

Creating Certificates

Here we consider the creation of example certificates.

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.
4.1.1 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:

<table>
<thead>
<tr>
<th>subcommand</th>
<th>required/default section</th>
<th>override command line option</th>
<th>configuration file option</th>
</tr>
</thead>
<tbody>
<tr>
<td>req</td>
<td>[req]</td>
<td>-</td>
<td>-config FILE</td>
</tr>
<tr>
<td>ca</td>
<td>[ca]</td>
<td>-name section</td>
<td>-config FILE</td>
</tr>
</tbody>
</table>

Table 4.1: openssl subcommands to use

4.1.2 Creating the Erlang root CA

The Erlang root CA is created with the command

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

4.1.3 Creating the OTP CA

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

```bash
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:

```bash
openssl ca -batch -notext -config /some/path/req.cnf 
-extensions ca_cert -in /some/path/req.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.
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.

```erlang
%% The purpose of this module is to create example certificates for
%% testing.
%% Run it as:
%%
%% erl -noinput -run make_certs all "/path/to/openssl" -s erlang halt
%%
-module(make_certs).
-export([all/0, all/1]).

-record(dn, {commonName, organizationalUnitName = "Erlang OTP", organizationName = "Ericsson AB", localityName = "Stockholm", countryName = "SE", emailAddress = "peter@erix.ericsson.se"}).

all() ->
    all("
    openssl").

all([OpenSSLCmd]) ->
    Root = filename:dirname(filename:dirname((code:which(?MODULE)))).
    io:fwrite("Root : ~s\n", [Root]),
    NRoot = filename:join([Root, "etc"]),
    file:make_dir(NRoot),
    create_rnd(Root, "etc"), % For all requests
    rootCA(NRoot, OpenSSLCmd, "erlangCA"),
    intermediateCA(NRoot, OpenSSLCmd, "otpCA", "erlangCA"),
    endusers(NRoot, OpenSSLCmd, "otpCA", ["client", "server"],
    collect_certs(NRoot, ["erlangCA", "otpCA"], ["client", "server"]),
    remove_rnd(Root, "etc").

rootCA(Root, OpenSSLCmd, Name) ->
    create_ca_dir(Root, Name, ca_cnf(Name)),
    DN = #dn{commonName = Name},
    create_selfSigned_cert(Root, OpenSSLCmd, Name, req_cnf(DN)),
    ok.

intermediateCA(Root, OpenSSLCmd, CA, ParentCA) ->
    CA = "otpCA",
    create_ca_dir(Root, CA, ca_cnf(CA)),
    CARoot = filename:join([Root, CA]),
    DN = #dn{commonName = CA},
    CnfFile = filename:join([CARoot, "req.cnf"]),
```

SSL Application 17
file:write_file(CnfFile, req_cnf(DN)),
KeyFile = filename:join([CARoot, "private", "key.pem"]),
ReqFile = filename:join([CARoot, "req.pem"]),
create_req(Root, OpenSSLCmd, CnfFile, KeyFile, ReqFile),
CertFile = filename:join([CARoot, "cert.pem"]),
sign_req(Root, OpenSSLCmd, ParentCA, "ca_cert", ReqFile, CertFile).

endusers(Root, OpenSSLCmd, CA, Users) ->
  lists:foreach(fun(User) -> enduser(Root, OpenSSLCmd, CA, User) end, Users).

enduser(Root, OpenSSLCmd, CA, User) ->
  UsrRoot = filename:join([Root, User]),
  file:make_dir(UsrRoot),
  CnfFile = filename:join([UsrRoot, "req.cnf"]),
  DN = #dn{commonName = User},
  file:write_file(CnfFile, req_cnf(DN)),
  KeyFile = filename:join([UsrRoot, "key.pem"]),
  ReqFile = filename:join([UsrRoot, "req.pem"]),
  create_req(Root, OpenSSLCmd, CnfFile, KeyFile, ReqFile),
  CertFile = filename:join([UsrRoot, "cert.pem"]),
  sign_req(Root, OpenSSLCmd, CA, "user_cert", ReqFile, CertFile).

collect_certs(Root, CAs, Users) ->
  Bins = lists:foldr(
    fun(CA, Acc) ->
      File = filename:join([Root, CA, "cert.pem"]),
      {ok, Bin} = file:read_file(File),
      [Bin, " "]
    end, [], CAs),
  lists:foreach(
    fun(User) ->
      File = filename:join([Root, User, "cacerts.pem"]),
      file:write_file(File, Bins)
    end, Users).

create_self_signed_cert(Root, OpenSSLCmd, CAName, Cnf) ->
  CARoot = filename:join([Root, CAName]),
  CnfFile = filename:join([CARoot, "req.cnf"]),
  file:write_file(CnfFile, Cnf),
  KeyFile = filename:join([CARoot, "private", "key.pem"]),
  CertFile = filename:join([CARoot, "cert.pem"]),
  Cmd = [OpenSSLCmd, " req"
    " -new"
    " -x509"
    " -config ", CnfFile,
    " -keyout ", KeyFile,
    " -out ", CertFile],
  Env = [{"ROOTDIR", Root}],
  cmd(Cmd, Env).

create_ca_dir(Root, CAName, Cnf) ->
  CARoot = filename:join([Root, CAName]),
create_dirs(Root, Dir) ->
    lists:foreach(fun(Dir) ->
        file:make_dir(filename:join([Root, Dir])) end, Dirs).

create_files(Root, NameContents) ->
    lists:foreach(fun({Name, Contents}) ->
        file:write_file(filename:join([Root, Name]), Contents) end, NameContents).

create_rnd(Root, Dir) ->
    From = filename:join([Root, "rnd", "RAND"]),
    To = filename:join([Root, Dir, "RAND"]),
    file:copy(From, To).

remove_rnd(Root, Dir) ->
    File = filename:join([Root, Dir, "RAND"]),
    file:delete(File).
Chapter 4: Creating Certificates

cmd(Cmd, Env) ->
    FCmd = lists:flatten(Cmd),
    Port = open_port({spawn, FCmd}, [stream, eof, exit_status, 
                        {env, Env}],
    eval_cmd(Port).

eval_cmd(Port) ->
    receive
        {Port, {data, _}} ->
            eval_cmd(Port);
        {Port, eof} ->
            ok
    end,
    receive
        {Port, {exit_status, Status}} when Status /= 0 ->
            %% io:fwrite("exit status: \"-n\", [Status]),
            erlang:halt(Status)
        after 0 ->
            ok
    end.

%%
%% Contents of configuration files
%%

req_cnf(DN) ->
    ["# Purpose: Configuration for requests (end users and CAs)."
     ",
     "
     "ROOTDIR = $ENV::ROOTDIR
     ",
     "
     "[req]
     ",
     "input_password = secret
     ",
     "output_password = secret
     ",
     "default_bits = 1024
     ",
     "RANDFILE = $ROOTDIR/RAND
     ",
     "encrypt_key = no
     ",
     "default_md = sha1
     ",
     "#string_mask = pkix
     ",
     "x509_extensions = ca_ext
     ",
     "prompt = no
     ",
     "
     ]
4.2: An Example

```plaintext
"distinguished_name= name

"[name]

"commonName = " , DN#dn.commonName , "
"organizationalUnitName = " , DN#dn.organizationalUnitName , "
"organizationName = " , DN#dn.organizationName , "
"localityName = " , DN#dn.localityName , "
"countryName = " , DN#dn.countryName , "
"emailAddress = " , DN#dn.emailAddress , "

"[ca_ext]

"basicConstraints = critical, CA:true
"keyUsage = cRLSign, keyCertSign
"subjectKeyIdentifier = hash
"subjectAltName = email:copy
"

ca_cnf(CA) ->
["# Purpose: Configuration for CAs.

"ROOTDIR = $ENV::ROOTDIR
"default_ca = ca

"[ca]

dir = $ROOTDIR/" , CA , "
certs = $dir/certs
```
"crl_dir = $dir/crl
"database = $dir/index.txt
"new_certs_dir = $dir/newcerts
"certificate = $dir/cert.pem
"serial = $dir/serial
"crl = $dir/crl.pem
"private_key = $dir/private/key.pem
"RANDFILE = $dir/private/RAND
"

"x509_extensions = user_cert
"default_days = 3600
"default_md = sha1
"preserve = no
"policy = policy_match
"

"[policy_match]
"commonName = supplied
"organizationalUnitName = optional
"organizationName = match
"countryName = match
"localityName = match
"emailAddress = supplied
"

"[user_cert]
"basicConstraints = CA:false
"keyUsage = nonRepudiation, digitalSignature, keyEncipherment
4.2: An Example

```
"subjectKeyIdentifier = hash
"authorityKeyIdentifier = keyid,issuer:always
"subjectAltName = email:copy
"issuerAltName = issuer:copy
"
"
"[ca_cert]
"basicConstraints = critical,CA:true
"keyUsage = cRLSign, keyCertSign
"subjectKeyIdentifier = hash
"authorityKeyIdentifier = keyid:always,issuer:always
"subjectAltName = email:copy
"issuerAltName = issuer:copy
"].
```
Chapter 5

Using SSL for Erlang Distribution

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

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 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.
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 boot script, 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.
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 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.

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_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.

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.
Chapter 6

Licenses

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

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
 */
Chapter 6: Licenses

* 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).
*
*
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 routines 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.] */
SSL Reference Manual

Short Summaries

- Application ssl [page 37] - The SSL Application
- Erlang Module new_ssl [page 39] - Interface Functions for Secure Socket Layer
- Erlang Module ssl [page 47] - Interface Functions for Secure Socket Layer
- Erlang Module ssl_pkix [page 55] - Decoding of PKIX certificates with representation in Erlang.

ssl

No functions are exported.

new_ssl

The following functions are exported:

- cipher_suites() -> [page 42] Returns a list of supported cipher suites
- cipher_suites(Type) -> ciphers() [page 42] Returns a list of supported cipher suites
- connect(Socket, SslOptions) -> [page 42] Upgrades a gen_tcp, or equivalent, connected socket to a ssl socket.
- connect(Socket, SslOptions, Timeout) -> {ok, SslSocket} | {error, Reason} [page 42] Upgrades a gen_tcp, or equivalent, connected socket to a ssl socket.
- connect(Host, Port, Options) -> [page 42] Opens an ssl connection to Host, Port.
- connect(Host, Port, Options, Timeout) -> {ok, SslSocket} | {error, Reason} [page 43] Opens an ssl connection to Host, Port.
- close(SslSocket) -> ok | {error, Reason} [page 43] Close a ssl connection
- controlling_process(SslSocket, NewOwner) -> ok | {error, Reason} [page 43] Assigns a new controlling process to the ssl-socket.
- connection_info(SslSocket) -> {ok, [ProtocolVersion, CipherSuite]} | {error, Reason} [page 43] Returns the negotiated protocol version and cipher suite.
• `getopts(Socket)` ->
  [page 43] Get the value of the specified options.

• `getopts(Socket, OptionNames)` -> `ok, [socketoption()]` | `error, Reason`
  [page 43] Get the value of the specified options.

• `listen(Port, Options)` -> `ok, ListenSocket` | `error, Reason`
  [page 43] Creates a ssl listen socket.

• `peercert(Socket)` ->
  [page 44] Return the peer certificate.

• `peercert(Socket, Opts)` -> `ok, Cert` | `error, Reason`
  [page 44] Return the peer certificate.

• `peername(Socket)` -> `ok, [Address, Port]` | `error, Reason`
  [page 44] Return peer address and port.

• `recv(Socket, Length)` ->
  [page 44] Receive data on a socket.

• `recv(Socket, Length, Timeout)` -> `ok, Data` | `error, Reason`
  [page 44] Receive data on a socket.

• `send(Socket, Data)` -> `ok` | `error, Reason`
  [page 44] Write data to a socket.

• `setopts(Socket, Options)` -> `ok` | `error, Reason`
  [page 45] Set socket options.

• `shutdown(Socket, How)` -> `ok` | `error, Reason`
  [page 45] Immediately close a socket.

• `ssl_accept(ListenSocket)` ->
  [page 45] Perform server-side SSL handshake.

• `ssl_accept(ListenSocket, Timeout)` -> `ok` | `error, Reason`
  [page 45] Perform server-side SSL handshake.

• `ssl_accept(ListenSocket, SslOptions)` ->
  [page 45] Perform server-side SSL handshake.

• `ssl_accept(ListenSocket, SslOptions, Timeout)` -> `ok, Socket` | `error, Reason`
  [page 45] Perform server-side SSL handshake.

• `sockname(Socket)` -> `ok, [Address, Port]` | `error, Reason`
  [page 45] Return the local address and port.

• `start()` ->
  [page 46] Starts the Ssl application.

• `start(Type)` -> `ok` | `error, Reason`
  [page 46] Starts the Ssl application.

• `stop()` -> `ok`
  [page 46] Stops the Ssl application.

• `transport_accept(Socket)` ->
  [page 46] Accept an incoming connection and prepare for ssl_accept.

• `transport_accept(Socket, Timeout)` -> `ok, NewSocket` | `error, Reason`
  [page 46] Accept an incoming connection and prepare for ssl_accept.

• `versions()` -> `[[SslAppVer, SupportedSslVer, AvailableSslVsn]]`
  [page 46] Returns version information relevant for the ssl application.
ssl

The following functions are exported:

- `ciphers()` -> {ok, string()} | {error, enotstarted} [page 49] Get supported ciphers.
- `close(Socket)` -> ok | {error, Reason} [page 49] Close a socket returned by transport:accept/[1,2], connect/3/4, or listen/2.
- `connect(Address, Port, Options)` -> {ok, Socket} | {error, Reason} [page 49] Connect to Port at Address.
- `connect(Address, Port, Options, Timeout)` -> {ok, Socket} | {error, Reason} [page 49] Connect to Port at Address.
- `connection_info(Socket)` -> {ok, {Protocol, Cipher}} | {error, Reason} [page 50] Get current protocol version and cipher.
- `controlling_process(Socket, NewOwner)` -> ok | {error, Reason} [page 50] Assign a new controlling process to the socket.
- `format_error(ErrorCode)` -> string() [page 50] Return an error string.
- `getopts(Socket, OptionsTags)` -> {ok, Options} | {error, Reason} [page 50] Get options set for socket.
- `listen(Port, Options)` -> {ok, ListenSocket} | {error, Reason} [page 50] Set up a socket to listen on a port on the local host.
- `peercert(Socket)` -> [page 51] Return the peer certificate.
- `peercert(Socket, Opts)` -> {ok, Cert} | {ok, Subject} | {error, Reason} [page 51] Return the peer certificate.
- `peername(Socket)` -> {ok, {Address, Port}} | {error, Reason} [page 51] Return peer address and port.
- `pid(Socket)` -> pid() [page 51] Return the pid of the socket process.
- `recv(Socket, Length)` -> {ok, Data} | {error, Reason} [page 51] Receive data on socket.
- `recv(Socket, Length, Timeout)` -> {ok, Data} | {error, Reason} [page 51] Receive data on socket.
- `seed(Data)` -> ok | {error, Reason} [page 52] Seed the ssl random generator.
- `send(Socket, Data)` -> ok | {error, Reason} [page 52] Write data to a socket.
- `setopts(Socket, Options)` -> ok | {error, Reason} [page 52] Set socket options.
- `ssl_accept(Socket)` -> ok | {error, Reason} [page 52] Perform server-side SSL handshake and key exchange
- `ssl_accept(Socket, Timeout)` -> ok | {error, Reason} [page 52] Perform server-side SSL handshake and key exchange
SSL Reference Manual

- `sockname(Socket) -> {ok, {Address, Port}} | {error, Reason}` [page 53] Return the local address and port.
- `transport_accept(Socket) -> {ok, NewSocket} | {error, Reason}` [page 53] Accept an incoming connection and prepare for `ssl_accept`
- `transport_accept(Socket, Timeout) -> {ok, NewSocket} | {error, Reason}` [page 53] Accept an incoming connection and prepare for `ssl_accept`
- `version() -> {ok, {SSLVsn, CompVsn, LibVsn}}` [page 53] Return the version of SSL.

**ssl_pkix**

The following functions are exported:

- `decode_cert(Bin) -> {ok, Cert} | {error, Reason}` [page 55] Decode a PKIX certificate.
- `decode_cert(Bin, Opts) -> {ok, Cert} | {error, Reason}` [page 55] Decode a PKIX certificate.
- `decode_cert_file(File) -> {ok, Cert} | {error, Reason}` [page 55] Decode a PKIX certificate file.
- `decode_cert_file(File, Opts) -> {ok, Cert} | {error, Reason}` [page 55] Decode a PKIX certificate file.
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 advised 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 OpenSSL¹ 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 OpenSSL² 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)

¹URL: http://www.openssl.org
²URL: http://www.shininglightpro.com/search.php?searchname=Win32+OpenSSL
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. The alfa version has a few limitations that will be removed before the ssl-4.0 release. Main differences and limitations in the alfa 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()] | string() (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 | '3desede_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 when the verify options is verify_none 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

E.i 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, invalidIssuer}, {bad_cert, invalidSignature}, {bad_cert, nameNotPermitted}, {bad_cert, unknownCa}, {bad_cert, certExpired}, {bad_cert, invalidIssuer}, {bad_cert, invalidSignature}, {bad_cert, nameNotPermitted}, {bad_cert, certRevoked} (not implemented yet), {bad_cert, unknownCriticalExtension} 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 enteris 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.
The function ciphers Suites() 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.

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

Types:
- Socket = socket()
- Ssl Options = [{ssl option()}
- Timeout = integer() | infinity
- SslSocket = ssl socket()
- Reason = term()

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

c 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()
peercert(Socket) ->
peercert(Socket, Opts) -> [ok, Cert] | [error, Reason]

Types:
- Socket = sslsocket()
- Opts = [] | [otp] | [plain]
- Cert = term()
- Subject = term()

peercert(Cert) is equivalent to peercert(Cert, []).
The form of the returned certificate depends on the options.
If the options list is empty the certificate is returned as a DER encoded binary.
The option otp or plain implies that the certificate will be returned as a parsed ASN.1
structure in the form of an Erlang term. For detail see the public_key application.
Currently only plain is officially supported see the public_key users guide.

peername(Socket) ->
peername(Socket, Length) -> [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.

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

**Types:**
- `Socket = sslsocket()`
- `Options = [socketoption]()`

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

```erlang
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.

```erlang
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.

```erlang
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.

```erlang
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.

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

Types:
- Type = permanent | transient | temporary

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

```plaintext
stop() -> ok
```

Stops the Ssl application. [application(3)]

```plaintext
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.

```plaintext
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) ]
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 [page 39]
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.
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}

cconnect(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 or 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, OptionTags) -> \{ok, Options\} | \{error, Reason\}

Types:
- Socket = sslsocket()
- OptionTags = [optiontag()]()

Returns the options the tags of which are OptionTags 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) ->**

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

Types:
- `Socket = sslsocket()`
- `Opts = [pkix | ssl | subject]()`
- `Cert = term()`
- `Subject = term()`

`peercert(Cert)` is equivalent to `peercert(Cert, [])`.

The form of the returned certificate depends on the options.
If the options list is empty the certificate is returned as a DER encoded binary.

The options `pkix` and `ssl` implies that the certificate is returned as a parsed ASN.1 structure in the form of an Erlang term.

The `ssl` option gives a more elaborate return structure, with more explicit information. In particular object identifiers are replaced by atoms.

The options `pkix`, and `ssl` are mutually exclusive.

The option `subject` implies that only the subject’s distinguished name part of the peer certificate is returned. It can only be used together with the option `pkix` or the option `ssl`.

**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()`
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\}.

\[
\text{seed(Data)} \rightarrow \text{ok} | \{\text{error, Reason}\}
\]

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.

\[
\text{send(Socket, Data)} \rightarrow \text{ok} | \{\text{error, Reason}\}
\]

Types:
- Socket = sslsocket()
- Data = iolist() | binary()

Writes Data to Socket.

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

\[
\text{setopts(Socket, Options)} \rightarrow \text{ok} | \{\text{error, Reason}\}
\]

Types:
- Socket = sslsocket()
- Options = [socketoption]()

Sets options according to Options for the socket Socket.

\[
\text{ssl_accept(Socket)} \rightarrow \text{ok} | \{\text{error, Reason}\}
\]

\[
\text{ssl_accept(Socket, Timeout)} \rightarrow \text{ok} | \{\text{error, Reason}\}
\]

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.

```plaintext
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`.

```plaintext
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, reason}` is returned.

```plaintext
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.
echain too long 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.
enproxysocket No proxy socket found (internal error).
enoptions The list of options is empty.
enotstarted The SSL application has not been started.
options Invalid list of options.
eperrcert Certificate provided by peer is in error.
eperrcertexpired Certificate provided by peer has expired.
eperrcertinvalid 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.

SEE ALSO

gen_tcp(3), inet(3)
ssl_pkix

Erlang Module

This module provides decoding of PKIX certificates either provided as files or as binaries.

Exports

decode_cert(Bin) -> {ok, Cert} | {error, Reason}
decode_cert(Bin, Opts) -> {ok, Cert} | {error, Reason}

Types:
  • Bin = binary()
  • Opts = [pkix | ssl | subject]
  • Cert = term()
decode_cert(Bin) is equivalent to decode_cert(Bin, []).

The form of the returned certificate depends on the options.

If the options list is empty the certificate is returned as a DER encoded binary, i.e. {ok, Bin} is returned, where Bin is the provided input.

The options pkix and ssl imply that the certificate is returned as a parsed ASN.1 structure in the form of an Erlang term.

The ssl option gives a more elaborate return structure, with more explicit information. In particular object identifiers are replaced by atoms.

The options pkix, and ssl are mutually exclusive.

The option subject implies that only the subject's distinguished name part of the certificate is returned. It can only be used together with the option pkix or the option ssl.

decode_cert_file(File) -> {ok, Cert} | {error, Reason}
decode_cert_file(File, Opts) -> {ok, Cert} | {error, Reason}

Types:
  • File = path()
  • Opts = [pem | pkix | ssl | subject]
  • Cert = term()
decode_cert_file(File) is equivalent to decode_cert_file(File, []).  
The form of the returned certificate depends on the options.  
If the options list is empty the certificate is returned as a DER encoded binary, i.e. the contents of the input File is returned as a binary.  
The options pkix and ssl implies that the certificate is returned as a parsed ASN.1 structure in the form of an Erlang term.  
The ssl option gives a more elaborate return structure, with more explicit information. In particular object identifiers are replaced by atoms.  
The options pkix, and ssl are mutually exclusive.  
The option subject implies that only the subject's distinguished name part of the peer certificate is returned. It can only be used together with the option pkix or the option ssl.
List of Tables

3.1 PKIX Extensions ........................................ 14
4.1 openssl subcommands to use .................................. 16
Bibliography

Index of Modules and Functions

Modules are typed in this way. Functions are typed in this way.

cipher_suites/0
  ssl, 42

cipher_suites/1
  ssl, 42

ciphers/0
  ssl, 49

close/1
  ssl, 43, 49

collcting_process/2
  ssl, 43, 50

decode_cert/1
  ssl_pkix, 55

decode_cert/2
  ssl_pkix, 55

decode_cert_file/1
  ssl_pkix, 55

decode_cert_file/2
  ssl_pkix, 55

format_error/1
  ssl, 50

getopts/1
  ssl, 43

getopts/2

listen/2
  ssl, 43, 50

peercert/1
  ssl, 44, 51

peercert/2
  ssl, 44, 51

peername/1
  ssl, 44, 51

pid/1
  ssl, 51

recv/2
  ssl, 44, 51

recv/3
  ssl, 44, 51

send/2
  ssl, 44, 52

setopts/2
  ssl, 45, 52

shutdown/2
  ssl, 45

sockname/1
  ssl, 45, 53

ssl

cipher_suites/0, 42

cipher_suites/1, 42

ciphers/0, 49

close/1, 43, 49

connect/2, 42

connect/3, 42, 49
Index of Modules and Functions

connect/4, 43, 49
connection_info/1, 43, 50
controlling_process/2, 43, 50
format_error/1, 50
getopts/1, 43
getopts/2, 43, 50
listen/2, 43, 50
peercert/1, 44, 51
peercert/2, 44, 51
peername/1, 44, 51
pid/1, 51
recv/2, 44, 51
recv/3, 44, 51
seed/1, 52
send/2, 44, 52
setopts/2, 45, 52
shutdown/2, 45
sockname/1, 45, 53
ssl_accept/1, 45, 52
ssl_accept/2, 45, 52
ssl_accept/3, 45
start/0, 46
start/1, 46
stop/0, 46
transport_accept/1, 46, 53
transport_accept/2, 46, 53
version/0, ss1
versions/0, ss1

ssl_accept/1
ss1, 45, 52
ssl_accept/2
ss1, 45, 52
ssl_accept/3
ss1, 45
ssl_pki

decode_cert/1, 55
decode_cert/2, 55
decode_cert_file/1, 55
decode_cert_file/2, 55

start/0
ss1, 46
start/1
ss1, 46
stop/0
ss1, 46

transport_accept/1
ss1, 46, 53
transport_accept/2
ss1, 46, 53