Secure Socket Layer

version 3.0

http://www.erlang.org

Typeset in LaTeX from SGML source using the DOCBUILDER 3.3.2 Document System.

Contents

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 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 connections on that same address and port. This is typically done indefinitely many times.

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

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

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

1.2 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 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* (*CA*s) only. They issue certificates both for other CAs and ordinary users (which are not CAs).

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

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

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

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

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

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 `ssl` interface module.

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

2.1 The ssl Module

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

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

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

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

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

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

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

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

0 Do not verify the peer,

1 Verify peer,

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

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

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

2.2 A Client-Server Example

Here is a simple client server example.

```
%%% 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:port(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:accept(LSock),
    io:fwrite("Accept: accepted.~n"),
    {ok, Cert} = ssl:peercert(ASock, CertOpts),
    io:fwrite("Accept: peer cert:~n~p~n", [Cert]),
    io:fwrite("Accept: sending \"hello\".~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~p~n", [Cert]),
    {ok, Data} = ssl:recv(CSock, 0),
    io:fwrite("Connect: got data: ~p~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 Site[1].

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,
```

[1] URL: `http://asn1.elibel.tm.fr/`

```
     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 `signature` field of `Certificate` contains the value of the signature that the issuer computed by using the prescribed algorithm.

The `issuer<c>` and `<c>subject` fields can contain many different types av 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

```
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}),
        value   ATTRIBUTE-TYPE-AND-VALUE-CLASS.&Type
                ({SupportedAttributeTypeAndValues}{@type}) }
```

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

3.2.3 TBSCertificate extensions

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

```
Extension  ::=  SEQUENCE  {
     extnID      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 welldefined 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.

authorityKeyIdentifier	Used by to identify a certificate signed that has multiple signing keys.
subjectKeyIdentifier	Used to identify certificates that contain a public key. Must appear i CA certificates.
keyUsage	Defines the purpose of the certificate. Can be one or several of `digitalSignature`, `nonRepudiation`, `keyEncipherment`, `dataEncipherment`, `keyAgreement`, `keyCertSign`, `cRLSign`, `encipherOnly`, `decipherOnly`.
privateKeyUsagePeriod	Allows certificate issuer to provide a private key usage period to be short than the certificate usage period.
certificatePolicies	Contains one or more policy information terms indicating the policies under which the certificate has been issued.
policyMappings	Used i CA certificates.
subjectAltName	Allows additional identities to be bound the the subject.
issuerAltName	Allows additional identities to be bound the the issuer.
subjectDirectoryAttributes	Conveys identity attributes of the subject.
basicConstraints	Tells if the certificate holder is a CA or not.
nameConstraints	Used in CA certificates.

continued ...

... continued

policyConstraints	Used in CA certificates.
extKeyUsage	Indicates for which purposed the public key may be used.
cRLDistributionPoints	Indicates how CRL (Certificate Revokation List) information is obtained.
inhibitAnyPolicy	Used i CA certificates.
freshestCRL	For CRLs.
authorityInfoAccess	How to access CA information of the isssuer of the certificate.
subjectInfoAccess	How to access CA information of the subject of the certificate.

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:

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

Table 4.1: openssl subcommands to use

4.1.2 Creating the Erlang root CA

The Erlang root CA is created with the command

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

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

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

and the ask the Erlang CA to sign it:

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

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.

```
%% 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_self_signed_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"]),
```

```
    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, "
" | Acc]
             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]),
```

```
    file:make_dir(CARoot),
    create_dirs(CARoot, ["certs", "crl", "newcerts", "private"]),
    create_rnd(Root, filename:join([CAName, "private"])),
    create_files(CARoot, [{"serial", "01
"},
                          {"index.txt", ""},
                          {"ca.cnf", Cnf}]).

create_req(Root, OpenSSLCmd, CnfFile, KeyFile, ReqFile) ->
    Cmd = [OpenSSLCmd, " req"
           " -new"
           " -config ", CnfFile,
           " -keyout ", KeyFile,
           " -out ", ReqFile],
    Env = [{"ROOTDIR", Root}],
    cmd(Cmd, Env).

sign_req(Root, OpenSSLCmd, CA, CertType, ReqFile, CertFile) ->
    CACnfFile = filename:join([Root, CA, "ca.cnf"]),
    Cmd = [OpenSSLCmd, " ca"
           " -batch"
           " -notext"
           " -config ", CACnfFile,
           " -extensions ", CertType,
           " -in ", ReqFile,
           " -out ", CertFile],
    Env = [{"ROOTDIR", Root}],
    cmd(Cmd, Env).

%%
%%  Misc
%%

create_dirs(Root, Dirs) ->
    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).
```

```
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: ~w~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
```

```
"
     "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
```

```
"
     "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 crypted connection. One can also make the SSL package verify the certificates of other nodes to get additional security. Cookies are however always used as they can be used to differentiate between two different Erlang networks.

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

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

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

5.2 Building boot scripts including the SSL application

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

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

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

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

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

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

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

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

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

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

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

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

5.3 Specifying distribution module for net_kernel

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

Extending the command line from above gives us the following:

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

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

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

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

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

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

5.4 Specifying security options and other SSL options

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

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

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

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

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

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

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

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
```

```
 *    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 rouines from the library
*    being used are not cryptographic related :-).
* 4. If you include any Windows specific code (or a derivative thereof) from
*    the apps directory (application code) you must include an acknowledgement:
*    "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
*
* THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
* ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
* FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
* OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.
*
* The licence and distribution terms for any publically available version or
* derivative of this code cannot be changed.  i.e. this code cannot simply be
* copied and put under another distribution licence
* [including the GNU Public Licence.]
*/
```

SSL Reference Manual

Short Summaries

- Application ssl [page 35] – The SSL Application
- Erlang Module ssl [page 37] – Interface Functions for Secure Socket Layer
- Erlang Module ssl_pkix [page 45] – Decoding of PKIX certificates with representation in Erlang.

ssl

No functions are exported.

ssl

The following functions are exported:

- `accept(ListenSocket) -> {ok, Socket} | {error, Reason}`
  [page 39] Accept an incoming connection request.
- `accept(ListenSocket, Timeout) -> {ok, Socket} | {error, Reason}`
  [page 39] Accept an incoming connection request.
- `ciphers() -> {ok, string()} | {error, enotstarted}`
  [page 39] Get supported ciphers.
- `close(Socket) -> ok | {error, Reason}`
  [page 39] Close a socket returned by `accept/1/2`, `connect/3/4`, or `listen/2`.
- `connect(Address, Port, Options) -> {ok, Socket} | {error, Reason}`
  [page 39] Connect to `Port` at `Address`.
- `connect(Address, Port, Options, Timeout) -> {ok, Socket} | {error, Reason}`
  [page 39] Connect to `Port` at `Address`.
- `connection_info(Socket) -> {ok, {Protocol, Cipher}} | {error, Reason}`
  [page 40] Get current protocol version and cipher.
- `controlling_process(Socket, NewOwner) -> ok | {error, Reason}`
  [page 40] Assign a new controlling process to the socket.
- `format_error(ErrorCode) -> string()`
  [page 40] Return an error string.
- `getopts(Socket, OptionsTags) -> {ok, Options} | {error, Reason}`
  [page 40] Get options set for socket

- `listen(Port, Options) -> {ok, ListenSocket} | {error, Reason}`
  [page 40] Set up a socket to listen on a port on the local host.
- `peercert(Socket) ->`
  [page 41] Return the peer certificate.
- `peercert(Socket, Opts) -> {ok, Cert} | {ok, Subject} | {error, Reason}`
  [page 41] Return the peer certificate.
- `peername(Socket) -> {ok, {Address, Port}} | {error, Reason}`
  [page 41] Return peer address and port.
- `pid(Socket) -> pid()`
  [page 41] Return the pid of the socket process.
- `recv(Socket, Length) -> {ok, Data} | {error, Reason}`
  [page 42] Receive data on socket.
- `recv(Socket, Length, Timeout) -> {ok, Data} | {error, Reason}`
  [page 42] Receive data on socket.
- `seed(Data) -> ok | {error, Reason}`
  [page 42] Seed the ssl random generator.
- `send(Socket, Data) -> ok | {error, Reason}`
  [page 42] Write data to a socket.
- `setopts(Socket, Options) -> ok | {error, Reason}`
  [page 42] Set socket options.
- `sockname(Socket) -> {ok, {Address, Port}} | {error, Reason}`
  [page 42] Return the local address and port.
- `version() -> {ok, {SSLVsn, CompVsn, LibVsn}}`
  [page 43] Return the version of SSL.

ssl_pkix

The following functions are exported:

- `decode_cert(Bin) -> {ok, Cert} | {error, Reason}`
  [page 45] Decode a PKIX certificate.
- `decode_cert(Bin, Opts) -> {ok, Cert} | {error, Reason}`
  [page 45] Decode a PKIX certificate.
- `decode_cert_file(File) -> {ok, Cert} | {error, Reason}`
  [page 45] Decode a PKIX certificate file.
- `decode_cert_file(File, Opts) -> {ok, Cert} | {error, Reason}`
  [page 45] Decode a PKIX certificate file.

ssl

Application

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

Warning

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

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

Environment

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

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

```
erl ...  -ssl protcol_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 indentifier of the external port program of the SSL application. Default is `false`, i.e. no log file is produced.

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

`port_program = string() | false <optional>` Name of port program. The default is `ssl_esock`.

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

`proxylsport = integer() | false <optional>` Define the port number of the listen port of the SSL port program. Almost never is this option needed.

`proxylsbacklog = integer() | false <optional>` Set the listen queue size of the listen port of the SSL port program. The default is 128.

OpenSSL libraries

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

Source releases of OpenSSL can be downloaded from the OpenSSL[1] project home page, or mirror sites listed there.

The same URL also contains links to some compiled binaries and libraries of OpenSSL (see the `Related/Binaries` menu) of which the Shining Light Productions Win32 and OpenSSL[2] 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)

[1] URL: `http://www.openssl.org`
[2] URL: `http://www.shininglightpro.com/search.php?searchname=Win32+OpenSSL`

ssl

Erlang Module

This module contains interface functions to the Secure Socket Layer.

General

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 availabe ciphers.

The type sslsocket() is opaque to the user.

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

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

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

- {ssl_error, Socket, Reason}

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

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

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

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

Exports

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

Types:

- ListenSocket = Socket = sslsocket()
- Timeout = integer()

Accepts an incoming connection request on a listen socket. ListenSocket must be a socket returned from listen/2.

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.

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

Returns a string constisting 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 accept/1/2, connect/3/4, or listen/2

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

Types:

- Address = address()

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

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

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

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

Types:

- Socket = sslsocket()
- Protocol = protocol()
- Cipher = string()

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

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

Types:

- Socket = sslsocket()
- NewOwner = pid()

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

`format_error(ErrorCode) -> string()`

Types:

- ErrorCode = term()

Returns a diagnostic string describing an error.

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

Types:

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

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

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

Types:

- Port = integer()
- Options = [listen_option()]

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

Sets up a socket to listen on port `Port` at the local host. If `Port` is zero, `listen/2` picks an available port number (use `port/1` to retreive 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 `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()
- Length = integer() >= 0
- Timeout = integer()
- Data = bytes() | binary()

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

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

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

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

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

Types:

- Data = iolist() | binary()

Seeds the ssl random generator.

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

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

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

Types:

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

Writes `Data` to `Socket`.

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

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

Types:

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

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

```
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`.

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

Types:

- SSLVsn = CompVsn = LibVsn = string()()

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

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

ERRORS

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

`closed` Connection closed for the operation in question.

`ebadsocket` Connection not found (internal error).

`ebadstate` Connection not in connect state (internal error).

`ebrokertype` Wrong broker type (internal error).

`ecacertfile` Own CA certificate file is invalid.

`ecertfile` Own certificate file is invalid.

`echaintoolong` The chain of certificates provided by peer is too long.

`ecipher` Own list of specified ciphers is invalid.

`ekeyfile` Own private key file is invalid.

`ekeymismatch` Own private key does not match own certificate.

`enoissuercert` Cannot find certificate of issuer of certificate provided by peer.

`enoservercert` Attempt to do accept without having set own certificate.

`enotlistener` Attempt to accept on a non-listening socket.

`enoproxysocket` No proxy socket found (internal error).

`enooptions` The list of options is empty.

`enotstarted` The SSL application has not been started.

`eoptions` Invalid list of options.

`epeercert` Certificate provided by peer is in error.

`epeercertexpired` Certificate provided by peer has expired.

`epeercertinvalid` Certificate provided by peer is invalid.

`eselfsignedcert` Certificate provided by peer is self signed.

`esslaccept` Server SSL handshake procedure between client and server failed.

`esslconnect` Client SSL handshake procedure between client and server failed.

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

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

`ex509lookup` Protocol wants X.509 lookup, which is not supported in this version of the SSL application.

{`badcall, Call`} Call not recognized for current mode (active or passive) and state of socket.

{`badcast, Cast`} Call not recognized for current mode (active or passive) and state of socket.

{`badinfo, Info`} Call not recognized for current mode (active or passive) and state of socket.

SEE ALSO

gen_tcp(3), inet(3)

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`.

Chapter 8

SSL Release Notes

This document describes the changes made to the SSL application.

8.1 SSL 3.0.11

8.1.1 Fixed Bugs and Malfunctions

- The state of a connection in active mode could be in a restrictive state, so that an internal tcp_closed message was incorrectly considered illegal, resulting in a premature termination of the connection process.
  *** INCOMPATIBILITY with no. ***
  Own Id: OTP-5972 Aux Id: seq10188

8.2 SSL 3.0.10

8.2.1 Fixed Bugs and Malfunctions

- Erlang distribution over SSL was broken. Corrected. (Thanks to Fredrik Thulin.)
  Own Id: OTP-5863

8.3 SSL 3.0.9

8.3.1 Fixed Bugs and Malfunctions

- The port program for the ssl application could waste huge amounts of CPU time if a write could not be completed directly and was put in the write queue. (Only on platforms where poll() is used, such as Solaris and Linux.)
  Own Id: OTP-5784

8.4 SSL 3.0.8

8.4.1 Fixed Bugs and Malfunctions

- A process reading only a portion of a sufficiently large amount of data from an accepted socket, and then quering the ssl library (e.g. ssl:getpeername()), would cause a global deadlock in the esock port program.
  Own Id: OTP-5702
- A spelling error in the module `ssl_pkix` caused the call to `ssl:peercert/2` to fail when the option `subject` was used.
  Own Id: OTP-5708
- Because fopen() on Solaris 8 can't handle file descriptor numbers above 255, reading of certificate files would fail if all file descriptors below 256 were in use (typically, if many connections were open). This problem has been worked around.
  The ssl application's port program used to use select(), which meant that it could not handle more than FD_SETSIZE file descriptors (usually 1024). To eliminate that limitation, poll() is now used on all platforms that support it.
  Solaris/Sparc, 64-bit emulator: The SO_REUSEADDR option was not set for listen sockets, which essentially made the ssl application unusuable. Corrected.
  The default listen queue size for ssl port program was changed to 128 (from 5).
  Own Id: OTP-5755 Aux Id: seq10068

8.5 Ssl 3.0.7

8.5.1 Fixed Bugs and Malfunctions

- The R/W buffer length i esock.c was too small. It has been increased from 4k to 32k.
  Own Id: OTP-5620

8.6 Ssl 3.0.6

8.6.1 Improvements and New Features

- A configuration option for choosing protocol versions has been added (`sslv2`, `sslv3`, and `tlsv1`).
  Own Id: OTP-5429 Aux Id: seq9755

8.7 Ssl 3.0.5

8.7.1 Fixed Bugs and Malfunctions

- Linked in drivers in the crypto, and asn1 applications are now compiled with the -D_THREAD_SAFE and -D_REENTRANT switches on unix when the emulator has thread support enabled.
  Linked in drivers on MacOSX are not compiled with the undocumented -lbundle1.o switch anymore. Thanks to Sean Hinde who sent us a patch.
  Linked in driver in crypto, and port programs in ssl, now compiles on OSF1.
  Minor makefile improvements in runtime_tools.
  Own Id: OTP-5346

8.8 Ssl 3.0.4

8.8.1 Fixed Bugs and Malfunctions

- `ssl:recv/3` with finite timeout value, closed the connection at timeout.
  Own Id: OTP-4882

8.9 Ssl 3.0.3

8.9.1 Fixed Bugs and Malfunctions

- When a file descriptor was marked for closing, and and end-of-file condition had already been detected, the file descriptor was never closed.
  Own Id: OTP-5093 Aux Id: seq8806
- When the number of open file descriptors reached FD_SETSIZE, the SSL port program entered a busy loop.
  Own Id: OTP-5094 Aux Id: seq8806

8.9.2 Improvements and New Features

- The SSL application now supports SSL sessions for servers, which typically speeds up HTTP requests from browsers.
  Own Id: OTP-5095

8.10 SSL 3.0.2

8.10.1 Fixed Bugs and Malfunctions

- The UTF8String type is now defined in asn1-1.4.4.2 and later. Therefore the definitions of UTF8String has been removed from the ASN.1 modules PKIX1Explicit88.asn1 and PKIXAttributeCertificate.asn1. The SSL application can now only be built using asn-1.4.4.2 or later.
  OwnId: OTP-4971.

8.10.2 Known Bugs and Problems

See SSL-3.0.

8.11 SSL 3.0.1

8.11.1 Fixed Bugs and Malfunctions

- An unexpected object identifier would crash `ssl:peercert`.
  OwnId: OTP-4771.

8.11.2 Known Bugs and Problems

See SSL-3.0.

8.12 SSL 3.0

8.12.1 Improvements and New Features

- The `cache_timout` option was silently ignored. It had to do with SSL sessions, where multiple connections can occur. Since the Erlang SSL applicaton does not support sessions the option is still ignored, and consequently the documentation about it has been removed.
  OwnId: OTP-3146
- The Erlang SSL application is now based on OpenSSL version 0.9.7a. OpenSSL 0.9.6 should also work.
  OwnId: OTP-4002
- When connecting it is now possible to bind to a local address and local port.
  OwnId: OTP-4675
- The `ssl_esock` port program is now part of the distribution and thus does not have to be created explicitely. It is dynamically linked to OpenSSL libraries in a “standard” location (typically `/usr/local/lib` on UNIX; in the path on Win32).
  OwnId: OTP-4676
- The new functions `ssl:peercert/1/2` provide information from the certificate of a peer of a connection.
  OwnId: OTP-4680
  Aux Id: seq7688
- The function `ssl:port/1` has been removed from the documentation, but not from the `ssl` interface module. The recommendation is to use `ssl:peername/1` instead, which provides both address and port of the peer.
  OwnId: OTP-4681
- New User's Guide documentation has been added.
  OwnId: OTP-4682
- The old `ssl_socket` interface has been removed and also the documentation of it.
  OwnId: OTP-4683
- The use of ephemeral RSA keys is now supported. It is a global configuration option (see the ssl(6) manual page).
  OwnId: OTP-4691.

8.12.2 Fixed Bugs and Malfunctions

- The option `cacertfile` is now in effect, and can therefore no longer be set with the OS environment variable SSL_CERT_FILE (which did set the same value for all connections).
  OwnId: OTP-3146
- There was a syncronization error at closing of an SSL connnection.
  OwnId: OTP-4435
  Aux Id: seq7534
- C macros in `debuglog.c` were not ANSI C compliant.
  OwnId: OTP-4674

- The `binary` option was not properly handled.
  OwnId: OTP-4678
- The `ssl:format_error/1` did not consider `inet` error codes, nor did it have a catch all for unknown error codes.
  OwnId: OTP-4679

8.12.3 Known Bugs and Problems

- Change of controlling process in not OTP compliant.
  OwnId; OTP-4712
- There is still no way to restrict the cipher sizes.
  OwnId: OTP-4712
- The `keep_alive` and `reuse_addr` options will be added in a future release.
  OwnId: OTP-4677
- There is currently no way to restrict the SSL/TLS protocol versions to use. In a future release this will be supported as a configuration option, and as an option for each connection as well.
  OwnId: OTP-4711.

8.13 SSL 2.3.6

8.13.1 Fixed Bugs and Malfunctions

- There was a synchronization error at closing, which could result in that an SSL socket was removed prematurely, resulting in that a user process referring to it received an unexpected exit.
  OwnId: OTP-4435
  Aux Id: seq7600

8.13.2 Known Bugs and Problems

See SSL 2.2 .

8.14 SSL 2.3.5

8.14.1 Fixed Bugs and Malfunctions

- Setting of the option `nodelay' caused the SSL port program to dump core.
  OwnId: OTP-4380
  Aux Id: -
- Setting of the option '{active, once}' in `setopts` was wrong, causing a correct socket message to be regarded as erroneous.
  OwnId: OTP-4380
  Aux Id: -
- A self-signed peer certificate was always rejected with the error `eselfsignedcert', irrespective of the `depth' value.
  OwnId: OTP-4374
  Aux Id: seq7417

8.14.2 Known Bugs and Problems

See SSL 2.2 .

8.15 SSL 2.3.4

8.15.1 Improvements and New Features

- All TCP options allowed in gen_tcp, are now also allowed in SSL, except the option {`reuseaddr`, `Boolean`}. A new function `getopts` has been added to the SSL interface module `ssl`.
  OwnId: OTP-4305, OTP-4159

8.16 SSL 2.3.3

8.16.1 Fixed Bugs and Malfunctions

- The roles of the SSLeay and OpenSSL packages has been clarified in the ssl(6) application manual page. Also the URLs from which to download SSLeay has been updated.
  OwnId: OTP-4002
  Aux Id: seq5269
- A call to `ssl:listen(Port, Options)` with `Options = []` resulted in the cryptic {`error`, `ebadf`} return value. The return value has been changed to {`error, enooptions`}, and the behaviour has been documented in the `listen/2` function.
  OwnId: OTP-4016
  Aux Id: seq7006
- Use of the option {`nodelay, boolean()`} crashed the `ssl_server`.
  OwnId: OTP-4070
  Aux Id:
- A bug caused the Erlang distribution over ssl to fail. This bug has now been fixed.
  OwnId: OTP-4072
  Aux Id:
- On Windows when the SSL port program encountered an error code not anticipated it crashed.
  OwnId: OTP-4132
  Aux Id:

8.17 SSL 2.3.2

8.17.1 Fixed Bugs and Malfunctions

- The `ssl:accept/1-2` function sometimes returned {`error, {What, Where}`} instead of {`error, What`}, where `What` is an atom.
  OwnId: OTP-3775
  Aux Id: seq4991

8.18 SSL 2.3.1

8.18.1 Fixed Bugs and Malfunctions

- Sometimes the SSL portprogram would loop in an accept loop, without terminating even when the SSL application was stopped..
  OwnId: OTP-3691

8.19 SSL 2.3

Functions have been added to SSL to experimentally support Erlang distribution.

8.20 SSL 2.2.1

The 2.2.1 version of SSL provides code replacement in runtime by upgrading from, or downgrading to, versions 2.1 and 2.2.

8.21 SSL 2.2

8.21.1 Improvements and New Features

- The restriction that only the creator of an SSL socket can read from and write to the socket has been lifted.
  OwnId: OTP-3301
- The option {`packet, cdr`} for SSL sockets has been added, which means that SSL sockets also supports CDR encoded packets.
  OwnId: OTP-3302

8.21.2 Known Bugs and Problems

- Setting of a CA certificate file with the `cacertfile` option (in calls to `ssl:accept/1/2` or `ssl:connect/3/4`) does not work due to weaknesses in the SSLeay package.
  A work-around is to set the OS environment variable `SSL_CERT_FILE` before SSL is started. However, then the CA certificate file will be global for all connections.
  OwnId: OTP-3146
- When changing controlling process of an SSL socket, a temporary process is started, which is not gen_server compliant.
  OwnId: OTP-3146
- Although there is a `cache` timeout option, it is silently ignored.
  OwnId: OTP-3146
- There is currently no way to restrict the cipher sizes.
  OwnId: OTP-3146

8.22 SSL 2.1

8.22.1 Improvements and New Features

- The set of possible error reasons has been extended to contain diagnostics on erronous certificates and failures to verify certificates.
  OwnId: OTP-3145
- The maximum number of simultaneous SSL connections on Windows has been increased from 31 to 127.
  OwnId: OTP-3145

8.22.2 Fixed Bugs and Malfunctions

- A dead-lock occuring when write queues are not empty has been removed.
  OwnId: OTP-3145
- Error reasons have been unified and changed.
  (** POTENTIAL INCOMPATIBILITY **)
  OwnId: OTP-3145
- On Windows a check of the existence of the environment variable `ERLSRV_SERVICE_NAME` has been added. If that variable is defined, the port program of the SSL application will not terminated when a user logs off.
  OwnId: OTP-3145
- An error in the setting of the `nodelay` option has been corrected.
  OwnId: OTP-3145
- The confounded notions of verify mode and verify depth has been corrected. The option `verifydepth` has been removed, and the two separate options `verify` and `depth` has been added.
  (** POTENTIAL INCOMPATIBILITY **)
  OwnId: OTP-3145

8.22.3 Known Bugs and Problems

- Setting of a CA certificate file with the `cacertfile` option (in calls to `ssl:accept/1/2` or `ssl:connect/3/4`) does not work due to weaknesses in the SSLeay package.
  A work-around is to set the OS environment variable `SSL_CERT_FILE` before SSL is started. However, then the CA certificate file will be global for all connections.
  OwnId: OTP-3146
- When changing controlling process of an SSL socket, a temporary process is started, which is not gen_server compliant.
  OwnId: OTP-3146
- Although there is a `cache` timeout option, it is silently ignored.
  OwnId: OTP-3146
- There is currently no way to restrict the cipher sizes.
  OwnId: OTP-3146

8.23 SSL 2.0

A complete new version of SSL with separate I/O channels for all connections with non-blocking I/O multiplexing.

List of Tables

Bibliography

[1] Eric Rescorla: SSL and TLS - Designing and Building Secure Systems, Addison-Wesley, 2001, ISBN 0-201-61598-3.

[2] Bruce Schneier: Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition, John Wiley & Sons, 1995, ISBN 0471117099.

[3] ITU-T Recommendation X.509 (1997) | ISO/IEC 9594-8:1997 : http://www.itu.int/ITU-T/asn1/database/itu-t/x/x509/1997/index.html.

[4] Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile.

[5] Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile.

[6] ITU-T Recommendation X.680 (1994) | ISO/IEC 8824-1: 1995, Abstract Syntax Notation One (ASN.1): Specification of Basic Notation.

Index of Modules and Functions

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