SWI-Prolog SSL Interface

Jan van der Steen, Matt Lilley and Jan Wielemaker
Diff Automatisering v.o.f

Jan Wielemaker
SWI, University of Amsterdam
The Netherlands
E-mail: jan@swi-prolog.org

August 25, 2015

Abstract

The SWI-Prolog SSL (Secure Socket Layer) library implements a pair of *filtered streams* that realises an SSL encrypted connection on top of a pair of Prolog *wire* streams, typically a network socket. SSL provides public key based encryption and digitally signed identity information of the *peer*. The SSL library is well integrated with SWI-Prolog's HTTP library for both implementing HTTPS servers and communicating with HTTPS servers. It is also used by the smtp pack for accessing secure mail agents. Plain SSL can be used to realise secure connections between e.g., Prolog agents.

Contents

1 Introduction **3**
1.1 library(ssl): Secure Socket Layer (SSL) library 3
1.2 SSL Security 8
1.3 CRLs and Revocation 9
1.3.1 Disabling certificate checking 10
1.3.2 Establishing a safe connection 10

2 Example code **11**
2.1 Accessing an HTTPS server 11
2.2 Creating an HTTPS server 11
2.3 HTTPS behind a proxy 12

3 Acknowledgments **13**

1 Introduction

Raw TCP/IP networking is dangerous for two reasons. It is hard to tell whether the body you think you are talking to is indeed the right one and anyone with access to a subnet through which your data flows can 'tap' the wire and listen for sensitive information such as passwords, credit card numbers, etc. Secure Socket Layer (SSL) deals with both problems. It uses certificates to establish the identity of the peer and encryption to make it useless to tap into the wire. SSL allows agents to talk in private and create secure web services.

The SWI-Prolog `ssl` library provides an API to turn a pair of arbitrary Prolog *wire* streams into SSL powered encrypted streams. Note that secure protocols such as secure HTTP simply run the plain protocol over (SSL) encrypted streams.

Cryptography is a difficult topic. If you just want to download documents from an HTTPS server without worrying much about security, `http_open/3` will do the job for you. As soon as you have higher security demands we strongly recommend you to read enough background material to understand what you are doing. See section 1.2 for some remarks regarding this implementation. This The Linux Documentation Project page provides some additional background and tips for managing certificates and keys.

1.1 library(ssl): Secure Socket Layer (SSL) library

See also `library(socket)`, `library(http/http_open)`

An SSL server and client can be built with the (abstracted) predicate calls from the table below. The `tcp_` predicates are provided by `library(socket)`. The predicate `ssl_context/3` defines properties of the SSL connection, while `ssl_negotiate/5` establishes the SSL connection based on the wire streams created by the TCP predicates and the context.

The SSL Server	**The SSL Client**
`ssl_context/3`	`ssl_context/3`
`tcp_socket/1`	`tcp_socket/1`
`tcp_accept/3`	`tcp_connect/2`
`tcp_open_socket/3`	`tcp_open_socket/3`
`ssl_negotiate/5`	`ssl_negotiate/5`

The library is abstracted to communication over streams, and is not reliant on those streams being directly attached to sockets. The `tcp_` calls here are simply the most common way to use the library. Other two-way communication channels such as (named), pipes can just as easily be used.

ssl_context(+*Role, -SSL, :Options*) *[det]*

Create an *SSL* context. The defines several properties of the *SSL* connection such as involved keys, preferred encryption and passwords. After establishing a context, an *SSL* connection can be negotiated using `ssl_negotiate/5`, turning two arbitrary plain Prolog streams into encrypted streams. This predicate processes the options below.

certificate_file(+*FileName*)

Specify where the certificate file can be found. This can be the same as the `key_file(+FileName)` option. A certificate file is obligatory for a server and

may be provided for a client if the server demands the client to identify itself with a client certificate using the `peer_cert(true)` option. If a certificate is provided, it is always necessary to provide a matching \jargon{private key} using the `key_file(+FileName)` option.

key_file(+*FileName*)
Specify where the private key that matches the certificate can be found. If the key is encrypted with a password, this must be supplied using the `password(+Text)` or pem_password_hook(:PredicateName) option.

password(+*Text*)
Specify the password the private key is protected with (if any). If you do not want to store the password you can also specify an application defined handler to return the password (see next option). *Text* is either an atom or string. Using a string is preferred as strings are volatile and local resources.

pem_password_hook(*:PredicateName*)
In case a password is required to access the private key the supplied predicate will be called to fetch it. The predicate is called as `call(PredicateName, Password)` and typically unifies *Password* with a *string* containing the password.

require_crl(+*Boolean*)
If true (default is false), then all certificates will be considered invalid unless they can be verified as not being revoked. You can do this explicity by passing a list of CRL filenames via the `crl/1` option, or by doing it yourself in the cert_verify_hook. If you specify `require_crl(true)` and provide neither of these options, verification will necessarily fail

crl(+*ListOfFileNames*)
Provide a list of filenames of PEM-encoded CRLs that will be given to the context to attempt to establish that a chain of certificates is not revoked. You must also set `require_crl(true)` if you want CRLs to actually be checked by OpenSSL.

cacert_file(+*FileName*)
Specify a file containing certificate keys of *trusted* certificates. The peer is trusted if its certificate is signed (ultimately) by one of the provided certificates. Using the *FileName* `system(root_certificates)` uses a list of trusted root certificates as provided by the OS. See `system_root_certificates/1` for details.

Additional verification of the peer certificate as well as accepting certificates that are not trusted by the given set can be realised using the hook `cert_verify_hook(PredicateName)`.

cert_verify_hook(*:PredicateName*)
The predicate `ssl_negotiate/5` calls *PredicateName* as follows:

```
call(PredicateName, +SSL,
     +ProblemCertificate, +AllCertificates, +FirstCertificate,
     +Error)
```

In case the certificate was verified by one of the provided certifications from the `cacert_file` option, Error is unified with the atom `verified`. Otherwise it contains the error string passed from OpenSSL. Access will be granted iff the predicate

succeeds. See `load_certificate/2` for a description of the certificate terms. See `cert_accept_any/5` for a dummy implementation that accepts any certificate.

cert(+*Boolean***)**
Trigger the sending of our certificate specified by `certificate_file(FileName)`. Sending is automatic for the server role and implied if both a certificate and key are supplied for clients, making this option obsolete.

peer_cert(+*Boolean***)**
Trigger the request of our peer's certificate while establishing the *SSL* layer. This option is automatically turned on in a client *SSL* socket. It can be used in a server to ask the client to identify itself using an *SSL* certificate.

close_parent(+*Boolean***)**
If `true`, close the raw streams if the *SSL* streams are closed. Default is `false`.

disable_ssl_methods(+*List***)**
A list of methods to disable. Unsupported methods will be ignored. Methods include `sslv2`, `sslv2`, `sslv23`, `tlsv1`, `tlsv1_1` and `tlsv1_2`.

ssl_method(+*Method***)**
Specify the explicit *Method* to use when negotiating. For allowed values, see the list for `disable_ssl_methods` above.

Arguments

Role	is one of `server` or `client` and denotes whether the *SSL* instance will have a server or client role in the established connection.
SSL	is a SWI-Prolog *blob* of type `ssl_context`, i.e., the type-test for an *SSL* context is `blob(SSL, ssl_context)`.

ssl_negotiate(+*SSL, +PlainRead, +PlainWrite, -SSLRead, -SSLWrite***)** *[det]*
Once a connection is established and a read/write stream pair is available, (*PlainRead* and *PlainWrite*), this predicate can be called to negotiate an *SSL* session over the streams. If the negotiation is successful, *SSLRead* and *SSLWrite* are returned.

Errors `ssl_error(Code, LibName, FuncName, Reason)` is raised if the negotiation fails. The streams *PlainRead* and *PlainWrite* are **not** closed, but an unknown amount of data may have been read and written.

ssl_peer_certificate(+*Stream, -Certificate***)** *[semidet]*
True if the peer certificate is provided (this is always the case for a client connection) and *Certificate* unifies with the peer certificate. The example below uses this to obtain the *Common Name* of the peer after establishing an https client connection:

```
http_open(HTTPS_url, In, []),
ssl_peer_certificate(In, Cert),
memberchk(subject(Subject), Cert),
memberchk('CN' = CommonName), Subject)
```

ssl_session(+*Stream, -Session***)** *[det]*
Retrieves (debugging) properties from the SSL context associated with *Stream*. If *Stream* is not

an SSL stream, the predicate raises a domain error. *Session* is a list of properties, containing the members described below. Except for *Version*, all information are byte arrays that are represented as Prolog strings holding characters in the range 0..255.

ssl_version(*Version***)**
The negotiated version of the session as an integer.

session_key(*Key***)**
The key material used in SSLv2 connections (if present).

master_key(*Key***)**
The key material comprising the master secret. This is generated from the server_random, client_random and pre-master key.

client_random(*Random***)**
The random data selected by the client during handshaking.

server_random(*Random***)**
The random data selected by the server during handshaking.

session_id(*SessionId***)**
The SSLv3 session ID. Note that if ECDHE is being used (which is the default for newer versions of OpenSSL), this data will not actually be sent to the server.

load_certificate(+*Stream, -Certificate***)** *[det]*
Loads a certificate from a PEM- or DER-encoded stream, returning a term which will unify with the same certificate if presented in cert_verify_hook. A certificate is a list containing the following terms: `issuer_name/1`, `hash/1`, `signature/1`, `version/1`, `notbefore/1`, `notafter/1`, `serial/1`, `subject/1` and `key/1`. `subject/1` and issuer_name are both lists of `=/2` terms representing the name.

Note that the OpenSSL `CA.pl` utility creates certificates that have a human readable textual representation in front of the PEM representation. You can use the following to skip to the certificate if you know it is a PEM certificate:

```
skip_to_pem_cert(In) :-
      repeat,
      (   peek_char(In, '-')
      ->  !
      ;   skip(In, 0'\n),
          at_end_of_stream(In), !
      ).
```

load_crl(+*Stream, -CRL***)** *[det]*
Loads a *CRL* from a PEM- or DER-encoded stream, returning a term containing terms `hash/1`, `signature/1`, `issuer_name/1` and `revocations/1`, which is a list of `revoked/2` terms. Each `revoked/2` term is of the form `revoked(+Serial, DateOfRevocation)`

system_root_certificates(-*List***)** *[det]*
List is a list of trusted root certificates as provided by the OS. This is the list used by

`ssl_context/3` when using the option `system(root_certificates)`. The list is obtained using an OS specific process. The current implementation is as follows:

- On Windows, CertOpenSystemStore() is used to import the `"ROOT"` certificates from the OS.
- On MacOSX, the trusted keys are loaded from the *SystemRootCertificates* key chain. The Apple API for this requires the SSL interface to be compiled with an XCode compiler, i.e., **not** with native gcc.
- Otherwise, certificates are loaded from a file defined by the Prolog flag `system_cacert_filename`. The initial value of this flag is operating system dependent. For security reasons, the flag can only be set prior to using the SSL library. For example:

```
:- use_module(library(ssl)).
:- set_prolog_flag(system_cacert_filename,
                   '/home/jan/ssl/ca-bundle.crt').
```

load_private_key(+*Stream, +Password, -PrivateKey*) *[det]*

Load a private key *PrivateKey* from the given stream *Stream*, using *Password* to decrypt the key if it is encrypted. Note that the password is currently only supported for PEM files. DER-encoded keys which are password protected will not load. The key must be an RSA key. EC, DH and DSA keys are not supported, and *PrivateKey* will be bound to an atom (ec_key, dh_key or dsa_key) if you try and load such a key. Otherwise *PrivateKey* will be unified with `privtate_key(KeyTerm)` where KeyTerm is a `rsa/8` term representing an RSA key.

load_public_key(+*Stream, -PublicKey*) *[det]*

Load a public key *PublicKey* from the given stream *Stream*. Supports loading both DER- and PEM-encoded keys. The key must be an RSA key. EC, DH and DSA keys are not supported, and *PublicKey* will be bound to an atom (one of ec_key, dh_key or dsa_key) if you try and load such a key. Otherwise *PublicKey* will be unified with `public_key(KeyTerm)` where KeyTerm is an `rsa/8` term representing an RSA key.

rsa_private_decrypt(+*PrivateKey, +CipherText, -PlainText*) *[det]*
rsa_private_encrypt(+*PrivateKey, +PlainText, -CipherText*) *[det]*
rsa_public_decrypt(+*PublicKey, +CipherText, -PlainText*) *[det]*
rsa_public_encrypt(+*PublicKey, +PlainText, -CipherText*) *[det]*
rsa_private_decrypt(+*PrivateKey, +CipherText, -PlainText, +Enc*) *[det]*
rsa_private_encrypt(+*PrivateKey, +PlainText, -CipherText, +Enc*) *[det]*
rsa_public_decrypt(+*PublicKey, +CipherText, -PlainText, +Enc*) *[det]*
rsa_public_encrypt(+*PublicKey, +PlainText, -CipherText, +Enc*) *[det]*

RSA Public key encryption and decryption primitives. A string can be safely communicated by first encrypting it and have the peer decrypt it with the matching key and predicate. The length of the string is limited by the key length. Text is encoded using encoding *Enc*, which is one of `octet`, `text` or `utf8` (default).

Errors `ssl_error(Code, LibName, FuncName, Reason)` is raised if there is an error, e.g., if the text is too long for the key.

See also `load_private_key/3`, `load_public_key/2` can be use to load keys from a file. The predicate `load_certificate/2` can be used to obtain the public key from a certificate.

ssl_init(*-SSL, +Role, +Options*) *[det]*

Create an *SSL* context. Similar to `ssl_context/3`.

deprecated New code should use `ssl_context/3` and `ssl_negotiate/5` to realise an *SSL* connection.

ssl_accept(*+SSL, -Socket, -Peer*) *[det]*

(Server) Blocks until a connection is made to the host on the port specified by the *SSL* object. *Socket* and *Peer* are then returned.

deprecated New code should use `tcp_accept/3` and `ssl_negotiate/5`.

ssl_open(*+SSL, -Read, -Write*) *[det]*

(Client) Connect to the host and port specified by the *SSL* object, negotiate an *SSL* connection and return *Read* and *Write* streams if successful. It calls `ssl_open/4` with the socket associated to the *SSL* instance. See `ssl_open/4` for error handling.

deprecated New code should use `ssl_negotiate/5`.

ssl_open(*+SSL, +Socket, -Read, -Write*) *[det]*

Given the *Socket* returned from `ssl_accept/3`, negotiate the connection on the accepted socket and return *Read* and *Write* streams if successful. If `ssl_negotiate/5` raises an exception, the *Socket* is closed and the exception is re-thrown.

deprecated New code should use `ssl_negotiate/5`.

ssl_exit(*+SSL*)

Free an *SSL* context. *SSL* contexts are reclaimed by the Prolog (atom) garbage collector. Calling `ssl_exit/1` is needed if the deprecated `ssl_init/3` interface is used rather than the `ssl_context/3` based interface to reclaim the associated socket.

cert_accept_any(*+SSL, +ProblemCertificate, +AllCertificates, +FirstCertificate, +Error*) *[det]*

Implementation for the hook 'cert_verify_hook(:Hook)' that accepts *any* certificate. This is intended for `http_open/3` if no certificate verification is desired as illustrated below.

```
http_open('https:/...', In,
          [ cert_verify_hook(cert_accept_any)
          ])
```

1.2 SSL Security

Using SSL (in this particular case based on the OpenSSL implementation) to connect to SSL services (e.g., an `https://` address) easily gives a false sense of security. This section explains some of the pitfalls.[1]. As stated in the introduction, SSL aims at solving two issues: tapping information from the wire by means of encryption and make sure that you are talking to the right address.

[1] We do not claim to be complete, just to start warning you if security is important to you. Please make sure you understand (Open)SSL before relying on it.

Encryption is generally well arranged as long as you ensure that the underlying SSL library has all known security patches installed and you use an encryption that is not known to be weak. The Windows version of SWI-Prolog ships with its own binary of the OpenSSL library. Ensure this is up-to-date. Most other systems ship with the OpenSSL library and SWI-Prolog uses the system version. This applies for the binaries we distribute for MacOSX and Linux, as well as official Linux packages. Check the origin and version of the OpenSSL libraries if SWI-Prolog was compiled from source. The OpenSSL library version as reported by SSLeay_version() is available in the Prolog flag `ssl_library_version` as illustrated below on Ubuntu 14.04.

```
?- [library(ssl)].
?- current_prolog_flag(ssl_library_version, X).
X = 'OpenSSL 1.0.1f 6 Jan 2014'.
```

Whether you are talking to the right address is a complicated issue. The core of the validation is that the server provides a *certificate* that identifies the server. This certificate is digitally *signed* by another certificate, and ultimately by a *root certificate*. (There may be additional links in this chain as well, or there may just be one certificate signed by itself) Verifying the peer implies:

1. Verifying the chain or digital signatures until a trusted root certificate is found, taking care that the chain does not contain any invalid certificates, such as certificates which have expired, are not yet valid, have altered or forged signatures, are valid for the purposes of SSL (and in the case of an issuer, issuing child certificates)
2. Verifying that the signer of a certificate did not *revoke* the signed certificate.
3. Verifying that the host we connected to is indeed the host claimed in the certificate.

The default https client plugin (`http/http_ssl_plugin`) registers the system trusted root certificate with OpenSSL. This is achieved using the option `cacert_file`(*system(root_certificates)*) of `ssl_context/3`. The verification is left to OpenSSL. To the best of our knowledge, the current (1.0) version of OpenSSL **only** implements step (1) of the verification process outlined above. This implies that an attacker that can control DNS mapping (host name to IP) or routing (IP to physical machine) can fake to be a secure host as long as they manage to obtain a certificate that is signed from a recognised authority. Version 1.0.2 supports hostname checking, and will not validate a certificate chain if the leaf certificate does not match the hostname. 'Match' here is not a simple string comparison; certificates are allowed (subject to many rules) to have wildcards in their SubjectAltName field. Care must also be taken to ensure that the name we are checking against does not contain embedded NULLs. If SWI-Prolog is compiled against a version of OpenSSL that does NOT have hostname checking (ie 1.0.0 or earlier), it will attempt to do the validation itself. This is not guaranteed to be perfect, and it only supports a small subset of allowed wildcards. If security is important, use OpenSSL 1.0.2 or higher.

After validation, the predicate `ssl_peer_certificate/2` can be used to obtain the peer certificate and inspect its properties.

1.3 CRLs and Revocation

Certificates must sometimes be revoked. Unfortunately this means that the elegant chain-of-trust model breaks down, since the information you need to determine whether a certificate is trustworthy

no longer depends on just the certificate and whether the issuer is trustworthy, but now on a third piece of data - whether the certificate has been revoked. These are managed in two ways in OpenSSL: CRLs and OCSP. SWI-Prolog supports CRLs only. (Typically OCSP responders are configured in such a way as to just consult CRLs anyway. This gives the illusion of up-to-the-minute revocation information because OCSP is an interactive, online, real-time protocol. However the information provided can still be several *weeks* out of date!)

To do CRL checking, pass require_crl(true) as an option to the `ssl_context/3` (or `http_open/3`) option list. If you do this, a certificate will not be validated unless it can be *checked* for on a revocation list. There are two options for this:

First, you can pass a list of filenames in as the option `crl/1`. If the CRL corresponds to an issuer in the chain, and the issued certificate is not on the CRL, then it is assumed to not be revoked. Note that this does NOT prove the certificate is actually trustworthy - the CRL you pass may be out of date! This is quite awkward to get right, since you do not necessarily know in advance what the chain of certificates the other party will present are, so you cannot reasonably be expected to know which CRLs to pass in.

Secondly, you can handle the CRL checking in the cert_verify_hook when the Error is bound to unknown_crl. At this point you can obtain the issuer certificate (also given in the hook), find the CRL distribution point on it (the `crl/1` argument), try downloading the CRL (the URL can have literally any protocol, most commonly HTTP and LDAP, but theoretically anything else, too, including the possibility that the certificate has no CRL distribution point given, and you are expected to obtain the CRL by email, fax, or telegraph. Therefore how to actually obtain a CRL is out of scope of this document), load it using `load_crl/2`, then check to see whether the certificate currently under scrutiny appears in the list of revocations. It is up to the application to determine what to do if the CRL cannot be obtained - either because the protocol to obtain it is not supported or because the place you are obtaining it from is not responding. Just because the CRL server is not responding does not mean that your certificate is safe, of course - it has been suggested that an ideal way to extend the life of a stolen certificate key would be to force a denial of service of the CRL server.

1.3.1 Disabling certificate checking

In some cases clients are not really interested in host validation of the peer and whether or not the certificate can be trusted. In these cases the client can pass `cert_verify_hook`(*cert_accept_any*), calling `cert_accept_any/5` which accepts any certificate. Note that this will accept literally ANY certificate presented - including ones which have expired, have been revoked, and have forged signatures. This is probably not a good idea!

1.3.2 Establishing a safe connection

Applications that exchange sensitive data with e.g., a backend server typically need to ensure they have a secure connection to their peer. To do this, first obtain a non-secure connection to the peer (eg via a TCP socket connection). Then create an SSL context via `ssl_context/3`. For the client initiating the connection, the role is 'client', and you should pass options `host/1`, `port/1` and `cacert_file/1` at the very least. If you expect the peer to have a certificate which would be accepted by your host system, you can pass cacert_file(system(root_certificates)), otherwise you will need a copy of the CA certificate which was used to sign the peer's certificate. Alternatively, you can pass `cert_verify_hook/1` to write your own custom validation for the peer's certificate. Depending on the requirements, you may also have to provide your /own/ certificate if the peer de-

mands mutual authentication. This is done via the `certificate_file/1`, `key_file/1` and either `password/1` or `pem_password_hook/1`.

Once you have the SSL context and the non-secure stream, you can call `ssl_negotiate/5` to obtain a secure stream. `ssl_negotiate/5` will raise an exception if there were any certificate errors that could not be resolved.

The peer behaves in a symmetric fashion: First, a non-secure connection is obtained, and a context is created using `ssl_context/3` with the role set to server. In the server case, you must provide `certificate_file/1` and `key_file/1`, and then either `password/1` or `pem_password_hook/1`. If you require the other party to present a certificate as well, then peer_cert(true) should be provided. If the peer does not present a certificate, or the certificate cannot be validated as trusted, the connection will be rejected.

By default, revocation is not checked. To enable certificate revocation checking, pass require_crl(true) when creating the SSL context. See section 1.3 for more information about revocations.

2 Example code

Examples of a simple server and client (`server.pl` and `client.pl` as well as a simple HTTPS server (`https.pl`) can be found in the example directory which is located in `doc/packages/examples/ssl` relative to the SWI-Prolog installation directory. The `etc` directory contains example certificate files as well as a `README` on the creation of certificates using OpenSSL tools.

2.1 Accessing an HTTPS server

Accessing an `https://` server can be achieved using the code skeleton below. The line `:- use_module(library(http/http_ssl_plugin)).` can be omitted if the development environment is present because the plugin is dynamically loaded by `http_open/3` of the `https` scheme is detected. See section 1.2 for more information about security aspects.

```
:- use_module(library(http/http_open)).
:- use_module(library(http/http_ssl_plugin)).

    ...,
    http_open(HTTPS_url, In, []),
    ...
```

2.2 Creating an HTTPS server

The HTTP server is started in HTTPS mode by adding an option `ssl` to `http_server/2`. The argument of the `ssl` option is an option list passed to `ssl_context/3`. Note that a server requires two items:

1. The *server certificate* identifies the server and acts as a *public key* for the encryption.
2. The *server key* provides the matching *private key* and must be kept secret. It key *may* be protected by a password. If this is the case, the server must provide the password by means of the `password` option or the `pem_password_hook` callback.

Below is an example that uses the self-signed demo certificates distributed with the SSL package. This version does not require a certificate from the client, which is the normal case for publically accessible HTTPS servers. The example file `https.pl` also provides a server that does require the client to show its certificate. This version provides an additional level of security, often used to allow a selected set of clients to perform sensitive tasks.

```
:- use_module(library(http/thread_httpd)).
:- use_module(library(http/http_ssl_plugin)).

https_server(Port, Options) :-
        http_server(reply,
                    [ port(Port),
                      ssl([ certificate_file('etc/server/server-cert.pem'),
                            key_file('etc/server/server-key.pem'),
                            password("apenoot1")
                          ])
                    | Options
                    ]).
```

Note that a single Prolog server can call `http_server/2` with different parameters to provide services at several security levels as described below. These servers can either use their own dispatching or commonly use `http_dispatch/1` and check the `port` property of the request to verify they are called with the desired security level. If a service is approached at a too low level of security, the handler can deny access or use HTTP redirect to send the client to to appropriate interface.

- A plain HTTP server at port 80. This can either be used for non-sensitive information or for *redirecting* to a more secure service.
- An HTTPS server at port 443 for sensitive services to the general public.
- An HTTPS server that demands for a client key on a selected port for administrative tasks or sensitive machine-to-machine communication.

2.3 HTTPS behind a proxy

The above expects Prolog to be accessible directly from the internet. This is becoming more popular now that services are often deployed using *virtualization*. If the Prolog services are placed behind a reverse proxy, HTTPS implementation is the task of the proxy server (e.g., Apache or Nginx). The communication from the proxy server to the Prolog server can use either plain HTTP or HTTPS. As plain HTTP is easier to setup and faster, this is typically preferred if the network between the proxy server and Prolog server can be trusted.

Note that the proxy server *must* decrypt the HTTPS traffic because it must decide on the destination based on the encrypted HTTP header. *Port forwarding* provides another option to make a server running on a machine that is not directly connected to the internet visible. It is not needed to decrypt the traffic using port forwarding, but it is also not possible to realise *virtual hosts* or *path-based* proxy rules.

3 Acknowledgments

The development of the SWI-Prolog SSL interface has been sponsored by Scientific Software and Systems Limited.

References

Index

cacert_file/1, 10
cert_accept_any/5, 8, 10
cert_verify_hook/1, 10
certificate_file/1, 11
crl/1, 10

host/1, 10
http/http_ssl_plugin *library*, 9
http_dispatch/1, 12
http_open/3, 3, 10, 11
http_server/2, 11, 12

key_file/1, 11

load_certificate/2, 6
load_crl/2, 6, 10
load_private_key/3, 7
load_public_key/2, 7

password/1, 11
pem_password_hook/1, 11
port/1, 10

rsa_private_decrypt/3, 7
rsa_private_decrypt/4, 7
rsa_private_encrypt/3, 7
rsa_private_encrypt/4, 7
rsa_public_decrypt/3, 7
rsa_public_decrypt/4, 7
rsa_public_encrypt/3, 7
rsa_public_encrypt/4, 7

ssl *library*, 3
ssl_accept/3, 8
ssl_context/3, 3, 9–11
ssl_exit/1, 8
ssl_init/3, 8
ssl_negotiate/5, 5, 11
ssl_open/3, 8
ssl_open/4, 8
ssl_peer_certificate/2, 5, 9
ssl_session/2, 5
system_root_certificates/1, 6