SWI-Prolog SSL Interface

Jan van der Steen
Diff Automatisering v.o.f

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

March 4, 2012

Abstract

This document describes the SWI-Prolog SSL library, a set of predicates which provides secure sockets to Prolog applications, for example to run a secure HTTPS server, or access websites using the `https` protocol. It can also be used to provide authentication and secure data exchange between Prolog processes over the network.

Contents

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, creditcard 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 very similar to `socket` for raw TCP/IP connections that provides SSL server and client sockets.

2 About SSL

The SWI-Prolog SSL interface is built on top of the OpenSSL library. This library is commonly provided as a standard package in many Linux distributions. The MS-Windows version is built using a binary distribution available from `http://www.slproweb.com/products/Win32OpenSSL.html`.

A good introduction on key- and certificate handling for OpenSSL can be found at `http://www.tldp.org/HOWTO/SSL-Certificates-HOWTO/`

3 Overview of the Prolog API

An SSL server and client can be built with the following (abstracted) predicate calls:

SSL server	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_negotiatate/5`	`ssl_negotiate/5`
...	...
`ssl_exit/1`	`ssl_exit/1`

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. In UNIX, pipes could just as easily be used, for example.

What follows is a description of each of these functions and the arguments they accept.

ssl_context(+*Role, -SSL, +Options*)

Role with legal values `server` or `client` denotes whether the SSL instance will have a server or client role in the established connection. With *Options* various properties of the SSL session can be defined, some of which required, some optional. An overview is given below. The handle of the connection is returned in *SSL*.

Below is an overview of the *Options* argument. Some options are only required by the client (C), some are required by the server (marked S), some by both server as client (marked CS).

host(+*HostName*)

[C] The host to connect to by the client or identified by the server. Both IP addresses and hostnames can be supplied here. This option is required for the client and optionally for the server.

port(+*Integer*)

[CS] The port to connect or listen to. This option is required since no default port can sensibly be defined for an abstract layer. The webserver *https* protocol uses port 443.

certificate_file(+*FileName*)

[S] Specify where the certificate file can be found. This can be the same as the key file (see next option).

key_file(+*FileName*)

[S] Specify where the private key can be found. This can be the same as the certificate file.

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).

pem_password_hook(*:PredicateName*)

In case a password is required to access the private key the supplied function will be called to fetch it. The function has the following prototype: `function`(+*SSL, -Password*)

cacert_file(+*FileName*)

Specify a file containing certificate keys which will thus automatically be verified as trusted. You can also install an application defined handler to verify certificates (see next option).

cert_verify_hook(*:PredicateName*)

In case a certificate cannot be verified or has some properties which makes it invalid (invalid validity date for example) the supplied function will be called to ask its opinion about the certificate. The predicate is called as follows: `function`(+*SSL, +Problem-Certificate, +AllCertificates, +FirstCertificate +Error*). Access will be granted iff the predicate succeeds. See load_certificate for a description of the certificate terms

cert(+*Boolean*)

Trigger the sending of our certificate as specified using the option `certificate_file` described earlier. For a server this option is automatically turned on.

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.

close_parent(+*Boolean*)

If `true`, close the raw streams if the SSL streams are closed.

ssl_negotiate(+*SSL, +PlainRead, +PlainWrite, -SSLRead, -SSLWrite*)

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.

ssl_exit(+*SSL*)
Clean up all resources related to the SSLinstance.

load_certificate(+*Stream, -Certificate*)
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.

load_crl(+*Stream, -CRL*)
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)

4 Backward compatibility

There are some predicates included to provide an API similar to the one exposed by a previous version of the library.

ssl_init(-*SSL, +Role, +Options*)
Analogous to `ssl_context/3`.

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

ssl_open/3(+*SSL, -Read, -Write*)
(Client) Connect to the host and port specified by the SSL object, negotiate an SSL connection and return Read and Write streams if successful

ssl_open/4(+*SSL, +Socket -Read, -Write*)
(Server) Given the *Socket* returned from

ssl_accept/3(*n*)
egotiate the connection on the accepted socket and return Read and Write streams if successful.

5 Using SSL to provide and access HTTPS

This packages installs the library `http/http_ssl_plugin.pl` alongside the http package. This library is a plugin for `http/thread_httpd.pl` and `http/http_open.pl` that enables these libraries to serve and access HTTPS services. Note that HTTPS is simply HTTP over an SSL socket.

5.1 Accessing an HTTPS server

Accessing an =—https://—= server can be achieved using the code skeleton below. See `ssl_context/3` for the `cert_verify_hook`(*:Hook*) option.

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

cert_verify(_SSL, _ProblemCert, _AllCerts, _FirstCert, _Error) :-
        format(user_error, 'Accepting certificate~n', []).

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

5.2 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_init/3`. Here is an example that uses the demo certificates distributed with the SSL package.

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

https_server(Port, Options) :-
        http_server(reply,
                    [ port(Port),
                      timeout(60),
                      ssl([ host('localhost'),
                            cacert_file('etc/demoCA/cacert.pem'),
                            certificate_file('etc/server/server-cert.pem'),
                            key_file('etc/server/server-key.pem'),
                            password('apenoot1')
                          ])
                    | Options
                    ]).
```

6 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.

7 Installation

The OpenSSL libraries are *not* part of the SWI-Prolog distribution and on systems using packagers with dependency checking, dependency on OpenSSL is deliberatly avoided. This implies that OpenSSL must be installed seperatly before using SSL with a binary distribution of SWI-Prolog. Most modern Linux distributions have an SSL package. An installer for MS-Windows is available from `http://www.slproweb.com/products/Win32OpenSSL.html` The SWI-Prolog SSL interface is currently built using OpenSSL 0.97b.

When installing from the source, the package configuration automatically builds the ssl library if a suitable OpenSSL implementation is found. On Windows systems, OpenSSL must be installed prior to building SWI-Prolog and `rules.mk` must be edited to reflect the position of the header and libraries if they are not in the standard search path.

8 Multithreading

OpenSSL is not intrinsically threadsafe, but can be made so by providing some callbacks for managing locking. These callbacks are installed when the ssl4pl library is loaded, and will overwrite any existing callbacks.

When the ssl4pl library is unloaded, the original callbacks will be restored.

9 Acknowledgments

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

References

Index