SWI-Prolog SSL Interface

Markus Triska, Jan van der Steen, Matt Lilley and Jan Wielemaker
E-mail: jan@swi-prolog.org

October 10, 2017

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

Raw TCP/IP networking is dangerous for two reasons:

1. It is hard to tell whether the party you think you are talking to is indeed the right one.
2. 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.

Transport Layer Security (TLS) and its predecessor Secure Socket Layer (SSL), which are both often collectively called SSL, solve both problems. SSL uses:

- certificates to establish the *identity* of the peer
- *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.

The `crypto` library provides additional predicates related to cryptography and authentication, secure hashes and elliptic curves.

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 5 for some remarks regarding this implementation. This The Linux Documentation Project page provides some additional background and tips for managing certificates and keys.

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

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

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_accept/3`	`tcp_connect/3`
`tcp_open_socket/3`	`stream_pair/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 context 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.

host(*+HostName*)
: For the client, the host to which it connects. This option *should* be specified when *Role* is `client`. Otherwise, certificate verification may fail when negotiating a secure connection.

certificate_file(*+FileName*)
: Specify where the certificate file can be found. This can be the same as the `key_file(+FileName)` option. A server *must* have at least one certificate before clients can connect. A client *must* have a certificate only 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 necessary to also provide a matching *private key* via the `key_file/1` option. To configure multiple certificates, use the option `certificate_key_pairs/1` instead. Alternatively, use `ssl_add_certificate_key/4` to add certificates and keys to an existing context.

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(:Goal)` option.

certificate_key_pairs(*+Pairs*)
: Alternative method for specifying certificates and keys. The argument is a list of *pairs* of the form Certificate-Key, where each component is a string or an atom that holds, respectively, the PEM-encoded certificate and key. To each certificate, further certificates of the chain can be appended. Multiple types of certificates can be present at the same time to enable different ciphers. Using multiple certificate types with completely independent certificate chains requires OpenSSL 1.0.2 or greater.

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(*:Goal*)
: In case a password is required to access the private key the supplied predicate will be called to fetch it. The hook is called as `call(Goal, +SSL, -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(:Goal).

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

```
call(Goal, +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.

cipher_list(+*Atom*)
Specify a cipher preference list (one or more cipher strings separated by colons, commas or spaces).

ecdh_curve(+*Atom*)
Specify a curve for ECDHE ciphers. If this option is not specified, the OpenSSL default parameters are used. With OpenSSL prior to 1.1.0, `prime256v1` is used by default.

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`.

close_notify(+*Boolean*)
If `true` (default is `false`), the server sends TLS `close_notify` when closing the connection. In addition, this mitigates *truncation attacks* for both client and server role: If EOF is encountered without having received a TLS shutdown, an exception is raised. Well-designed protocols are self-terminating, and this attack is therefore very rarely a concern.

min_protocol_version(+*Atom*)
Set the *minimum* protocol version that can be negotiated. *Atom* is one of `sslv3`, `tlsv1`, `tlsv1_1` and `tlsv1_2`. This option is available with OpenSSL 1.1.0 and later, and should be used instead of `disable_ssl_methods/1`.

max_protocol_version(+*Atom*)
Set the *maximum* protocol version that can be negotiated. *Atom* is one of `sslv3`, `tlsv1`, `tlsv1_1` and `tlsv1_2`. This option is available with OpenSSL 1.1.0 and later, and should be used instead of `disable_ssl_methods/1`.

disable_ssl_methods(+*List*)
A list of methods to disable. Unsupported methods will be ignored. Methods include `sslv2`, `sslv3`, `sslv23`, `tlsv1`, `tlsv1_1` and `tlsv1_2`. This option is deprecated starting with OpenSSL 1.1.0. Use `min_protocol_version/1` and `max_protocol_version/1` instead.

ssl_method(+*Method*)
Specify the explicit *Method* to use when negotiating. For allowed values, see the list for `disable_ssl_methods` above. Using this option is discouraged. When using OpenSSL 1.1.0 or later, this option is ignored, and a version-flexible method is used to negotiate the connection. Using version-specific methods is deprecated in recent OpenSSL versions, and this option will become obsolete and ignored in the future.

sni_hook(*:Goal*)
This option provides Server Name Indication (SNI) for *SSL* servers. This means that depending on the host to which a client connects, different options (certificates etc.) can be used for the server. This TLS extension allows you to host different domains using the same IP address and physical machine. When a TLS connection is negotiated with a client that has provided a host name via SNI, the hook is called as follows:

```
call(Goal, +SSL0, +HostName, -SSL)
```

Given the current context SSL0, and the host name of the client request, the predicate computes *SSL* which is used as the context for negotiating the connection. The first solution is used. If the predicate fails, the default options are used, which are those of the encompassing `ssl_context/3` call. In that case, if no default certificate and key are specified, the client connection is rejected.

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_add_certificate_key(+*SSL0*, +*Certificate*, +*Key*, -*SSL*)
Add an additional certificate/key pair to *SSL0*, yielding *SSL*. *Certificate* and *Key* are either strings or atoms that hold the PEM-encoded certificate plus certificate chain and private key, respectively. Using strings is preferred for security reasons.

This predicate allows dual-stack RSA and ECDSA servers (for example), and is an alternative for using the `certificate_key_pairs/1` option. As of OpenSSL 1.0.2, multiple certificate types with completely independent certificate chains are supported. If a certificate of the same type is added repeatedly to a context, the result is undefined. Currently, up to 12 additional certificates of different types are admissible.

ssl_set_options(+*SSL0, -SSL, +Options***)**

SSL is the same as *SSL0*, except for the options specified in *Options*. The following options are supported: `close_notify/1`, `close_parent/1`, `host/1`, `peer_cert/1`, `ecdh_curve/1`, `min_protocol_version/1`, `max_protocol_version/1`, `disable_ssl_methods/1`, `sni_hook/1`, `cert_verify_hook/1`. See `ssl_context/3` for more information about these options. This predicate allows you to tweak existing *SSL* contexts, which can be useful in hooks when creating servers with the HTTP infrastructure.

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.

After a successful handshake and finishing the communication the user must close *SSLRead* and *SSLWrite*, for example using `call_cleanup(close(SSLWrite), close(SSLRead))`. If the *SSL context* (created with `ssl_context/3` has the option `close_parent(true)` (default `false`), closing *SSLRead* and *SSLWrite* also closes the original *PlainRead* and *PlainWrite* streams. Otherwise these must be closed explicitly by the user.

> **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_peer_certificate_chain(+*Stream, -Certificates***)** *[det]*

Certificates is the certificate chain provided by the peer, represented as a list of certificates.

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.

cipher(*Cipher***)**

The negotiated cipher for this connection.

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`, `signature_algorithm/1`, `version/1`, `notbefore/1`, `notafter/1`, `serial/1`, `subject/1` and `key/1`. `subject/1` and `issuer_name/1` are both lists of `=/2` terms representing the name. With OpenSSL 1.0.2 and greater, `to_be_signed/1` is also available, yielding the hexadecimal representation of the TBS (to-be-signed) portion of the certificate.

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 or EC key. DH and DSA keys are not supported, and *PrivateKey* will be bound to an atom (dh_key or dsa_key) if you try and load such a key. Otherwise *PrivateKey* will be unified with `private_key(KeyTerm)` where KeyTerm is an `rsa/8` term representing an RSA key, or `ec/3` for EC keys.

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 or EC key. DH and DSA keys are not supported, and *PublicKey* will be bound to an atom (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, or `ec/3` for EC keys.

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)
            ])
```

ssl_secure_ciphers(-*Ciphers:atom***)** *[det]*

Secure ciphers must guarantee forward secrecy, and must mitigate all known critical attacks. As of 2017, using the following ciphers allows you to obtain grade A on `https://www.ssllabs.com`. For A+, you must also enable HTTP Strict Transport Security (HSTS) by sending a suitable header field in replies.

Note that obsolete ciphers **must** be disabled to reliably prevent protocol downgrade attacks.

The *Ciphers* list is read from the setting `ssl:secure_ciphers` and can be controlled using `set_setting/2` and other predicates from `library(settings)`.

> **BEWARE**: This list must be changed when attacks on these ciphers become known! Keep an eye on this setting and adapt it as necessary in the future.

3 library(crypto): Cryptography and authentication library

author Markus Triska
author Matt Lilley

3.1 Introduction

This library provides bindings to functionality of OpenSSL that is related to cryptography and authentication, not necessarily involving connections, sockets or streams.

3.2 Design principle: Secure default algorithms

A basic design principle of this library is that its *default algorithms are cryptographically secure* at the time of this writing. We will *change* the default algorithms if an attack on them becomes known, and replace them by new defaults that are deemed appropriate at that time.

This may mean, for example, that where `sha256` is currently the default algorithm, `blake2s256` or some other algorithm may become the default in the future.

To preserve interoperability and compatibility and at the same time allow us to transparently update default algorithms of this library, the following conventions are used:

1. If an explicit algorithm is specified as an option, then that algorithm is used.

2. If *no* algorithm is specified, then a cryptographically secure algorithm is used.

3. If an option that normally specifies an algorithm is present, and a *logical variable* appears instead of a concrete algorithm, then that variable is unified with the secure default value.

This allows application programmers to inspect *which* algorithm was actually used, and store it for later reference.

For example:

```
?- crypto_data_hash(test, Hash, [algorithm(A)]).
Hash = '9f86d081884c7d659a2feaa0c55ad015a3bf4f1b2b0b822cd15d6c15b0f00a08',
A = sha256.
```

This shows that at the time of this writing, `sha256` was deemed sufficiently secure, and was used as default algorithm for hashing.

You therefore must not rely on *which* concrete algorithm is being used by default. However, you can rely on the fact that the default algorithms are secure. In other words, if they are *not* secure, then this is a mistake in this library, and we ask you to please report such a situation as an urgent security issue.

3.3 Representing binary data

In the context of this library, *bytes* can be represented as lists of integers between 0 and 255. Such lists can be converted to and from *hexadecimal notation* with the following bidirectional relation:

hex_bytes(*?Hex, ?List*) *[det]*

Relation between a hexadecimal sequence and a list of bytes. *Hex* is an atom, string, list of characters or list of codes in hexadecimal encoding. This is the format that is used by `crypto_data_hash/3` and related predicates to represent *hashes*. Bytes is a list of *integers* between 0 and 255 that represent the sequence as a list of bytes. At least one of the arguments must be instantiated. When converting *List to Hex*, an *atom* is used to represent the sequence of hexadecimal digits.

Example:

```
?- hex_bytes('501ACE', Bs).
Bs = [80, 26, 206].
```

See also `base64_encoded/3` for Base64 encoding, which is often used to transfer or embed binary data in applications.

3.4 Cryptographically secure random numbers

Almost all cryptographic applications require the availability of numbers that are sufficiently unpredictable. Examples are the creation of keys, nonces and salts. With this library, you can generate cryptographically strong pseudo-random numbers for such use cases:

crypto_n_random_bytes(*+N, -Bytes*) *[det]*

Bytes is unified with a list of *N* cryptographically secure pseudo-random bytes. Each byte is an integer between 0 and 255. If the internal pseudo-random number generator (PRNG) has not been seeded with enough entropy to ensure an unpredictable byte sequence, an exception is thrown.

One way to relate such a list of bytes to an *integer* is to use CLP(FD) constraints as follows:

```
:- use_module(library(clpfd)).

bytes_integer(Bs, N) :-
        foldl(pow, Bs, 0-0, N-_).

pow(B, N0-I0, N-I) :-
        B in 0..255,
        N #= N0 + B*256^I0,
        I #= I0 + 1.
```

With this definition, you can generate a random 256-bit integer *from* a list of 32 random *bytes*:

```
?- crypto_n_random_bytes(32, Bs),
   bytes_integer(Bs, I).
Bs = [98, 9, 35, 100, 126, 174, 48, 176, 246|...],
I = 109798276762338328820827...(53 digits omitted).
```

The above relation also works in the other direction, letting you translate an integer *to* a list of bytes. In addition, you can use `hex_bytes/2` to convert bytes to *tokens* that can be easily exchanged in your applications. This also works if you have compiled SWI-Prolog without support for large integers.

3.5 Hashes

A **hash**, also called **digest**, is a way to verify the integrity of data. In typical cases, a hash is significantly shorter than the data itself, and already miniscule changes in the data lead to different hashes.

The hash functionality of this library subsumes and extends that of `library(sha)`, `library(hash_stream)` and `library(md5)` by providing a unified interface to all available digest algorithms.

The underlying OpenSSL library (`libcrypto`) is dynamically loaded if *either* `library(crypto)` or `library(ssl)` are loaded. Therefore, if your application uses `library(ssl)`, you can use `library(crypto)` for hashing without increasing the memory footprint of your application. In other cases, the specialised hashing libraries are more lightweight but less general alternatives to `library(crypto)`.

3.5.1 Hashes of data and files

The most important predicates to compute hashes are:

crypto_data_hash(*+Data, -Hash, +Options*) *[det]*

Hash is the hash of *Data*. The conversion is controlled by *Options*:

algorithm(*+Algorithm*)

One of `md5` (*insecure*), `sha1` (*insecure*), `ripemd160`, `sha224`, `sha256`, `sha384`, `sha512`, `sha3_224`, `sha3_256`, `sha3_384`, `sha3_512`, `blake2s256` or `blake2b512`. The BLAKE digest algorithms require OpenSSL 1.1.0 or greater, and the SHA-3 algorithms require OpenSSL 1.1.1 or greater. The default is a cryptographically secure algorithm. If you specify a variable, then that variable is unified with the algorithm that was used.

encoding(*+Encoding*)

If *Data* is a sequence of character *codes*, this must be translated into a sequence of *bytes*, because that is what the hashing requires. The default encoding is `utf8`. The other meaningful value is `octet`, claiming that *Data* contains raw bytes.

hmac(*+Key*)

If this option is specified, a *hash-based message authentication code* (HMAC) is computed, using the specified *Key* which is either an atom, string or list of *bytes*. Any of the available digest algorithms can be used with this option. The cryptographic strength of the HMAC depends on that of the chosen algorithm and also on the key. This option requires OpenSSL 1.1.0 or greater.

Arguments

Data	is either an atom, string or code-list
Hash	is an atom that represents the hash in hexadecimal encoding.

See also

- `hex_bytes/2` for conversion between hexadecimal encoding and lists of bytes.
- `crypto_password_hash/2` for the important use case of passwords.

crypto_file_hash(*+File, -Hash, +Options*) *[det]*

True if *Hash* is the hash of the content of *File*. For *Options*, see `crypto_data_hash/3`.

3.5.2 Hashes of passwords

For the important case of deriving hashes from *passwords*, the following specialised predicates are provided:

crypto_password_hash(*+Password, ?Hash*) *[semidet]*

If *Hash* is instantiated, the predicate succeeds *iff* the hash matches the given password. Otherwise, the call is equivalent to `crypto_password_hash(Password, Hash, [])` and computes a password-based hash using the default options.

crypto_password_hash(*+Password, -Hash, +Options*) *[det]*

Derive *Hash* based on *Password*. This predicate is similar to `crypto_data_hash/3` in that it derives a hash from given data. However, it is tailored for the specific use case of *passwords*. One essential distinction is that for this use case, the derivation of a hash should be *as slow as possible* to counteract brute-force attacks over possible passwords.

Another important distinction is that equal passwords must yield, with very high probability, *different* hashes. For this reason, cryptographically strong random numbers are automatically added to the password before a hash is derived.

Hash is unified with an atom that contains the computed hash and all parameters that were used, except for the password. Instead of storing passwords, store these hashes. Later, you can verify the validity of a password with `crypto_password_hash/2`, comparing the then entered password to the stored hash. If you need to export this atom, you should treat it as opaque ASCII data with up to 255 bytes of length. The maximal length may increase in the future.

Admissible options are:

algorithm(*+Algorithm*)

The algorithm to use. Currently, the only available algorithm is `pbkdf2-sha512`, which is therefore also the default.

cost(*+C*)

C is an integer, denoting the binary logarithm of the number of *iterations* used for the derivation of the hash. This means that the number of iterations is set to 2^*C*. Currently, the default is 17, and thus more than one hundred *thousand* iterations. You should set this option as high as your server and users can tolerate. The default is subject to change and will likely increase in the future or adapt to new algorithms.

salt(*+Salt*)

Use the given list of bytes as salt. By default, cryptographically secure random numbers are generated for this purpose. The default is intended to be secure, and constitutes the typical use case of this predicate.

Currently, PBKDF2 with SHA-512 is used as the hash derivation function, using 128 bits of salt. All default parameters, including the algorithm, are subject to change, and other algorithms will also become available in the future. Since computed hashes store all parameters that were used during their derivation, such changes will not affect the operation of existing deployments. Note though that new hashes will then be computed with the new default parameters.

See also `crypto_data_hkdf/4` for generating keys from *Hash*.

3.5.3 HMAC-based key derivation function (HKDF)

The following predicate implements the *Hashed Message Authentication Code (HMAC)-based key derivation function*, abbreviated as HKDF. It supports a wide range of applications and requirements by concentrating possibly dispersed entropy of the input keying material and then expanding it to the desired length. The number and lengths of the output keys depend on the specific cryptographic algorithms for which the keys are needed.

crypto_data_hkdf(+*Data, +Length, -Bytes, +Options***)** *[det]*

Concentrate possibly dispersed entropy of *Data* and then expand it to the desired length. *Bytes* is unified with a list of *bytes* of length *Length*, and is suitable as input keying material and initialization vectors to the symmetric encryption predicates.

Admissible options are:

algorithm(+*Algorithm***)**
: A hashing algorithm as specified to `crypto_data_hash/3`. The default is a cryptographically secure algorithm. If you specify a variable, then it is unified with the algorithm that was used.

info(+*Info***)**
: Optional context and application specific information, specified as an atom, string or list of *bytes*. The default is the zero length atom ''.

salt(+*List***)**
: Optionally, a list of *bytes* that are used as salt. The default is all zeroes.

encoding(+*Atom***)**
: Either `utf8` (default) or `octet`, denoting the representation of *Data* as in `crypto_data_hash/3`.

The `info/1` option can be used to generate multiple keys from a single master key, using for example values such as `key` and `iv`, or the name of a file that is to be encrypted.

This predicate requires OpenSSL 1.1.0 or greater.

See also `crypto_n_random_bytes/2` to obtain a suitable salt.

3.5.4 Hashing incrementally

The following predicates are provided for building hashes *incrementally*. This works by first creating a **context** with `crypto_context_new/2`, then using this context with `crypto_data_context/3` to incrementally obtain further contexts, and finally extract the resulting hash with `crypto_context_hash/2`.

crypto_context_new(*-Context, +Options*) *[det]*

Context is unified with the empty context, taking into account *Options*. The context can be used in `crypto_data_context/3`. For *Options*, see `crypto_data_hash/3`.

Arguments	
Context	is an opaque pure Prolog term that is subject to garbage collection.

crypto_data_context(*+Data, +Context0, -Context*) *[det]*

Context0 is an existing computation context, and *Context* is the new context after hashing *Data* in addition to the previously hashed data. *Context0* may be produced by a prior invocation of either `crypto_context_new/2` or `crypto_data_context/3` itself.

This predicate allows a hash to be computed in chunks, which may be important while working with Metalink (RFC 5854), BitTorrent or similar technologies, or simply with big files.

crypto_context_hash(*+Context, -Hash*)

Obtain the hash code of *Context*. *Hash* is an atom representing the hash code that is associated with the current state of the computation context *Context*.

The following hashing predicates work over *streams*:

crypto_open_hash_stream(*+OrgStream, -HashStream, +Options*) *[det]*

Open a filter stream on *OrgStream* that maintains a hash. The hash can be retrieved at any time using `crypto_stream_hash/2`. Available *Options* in addition to those of `crypto_data_hash/3` are:

close_parent(*+Bool*)

If `true` (default), closing the filter stream also closes the original (parent) stream.

crypto_stream_hash(*+HashStream, -Hash*) *[det]*

Unify *Hash* with a hash for the bytes sent to or read from *HashStream*. Note that the hash is computed on the stream buffers. If the stream is an output stream, it is first flushed and the Digest represents the hash at the current location. If the stream is an input stream the Digest represents the hash of the processed input including the already buffered data.

3.6 Digital signatures

A digital **signature** is a relation between a key and data that only someone who knows the key can compute.

Signing uses a *private* key, and *verifying* a signature uses the corresponding *public* key of the signing entity. This library supports both RSA and ECDSA signatures. You can use `load_private_key/3` and `load_public_key/2` to load keys from files and streams.

In typical cases, we use this mechanism to sign the *hash* of data. See hashing (section 3.5). For this reason, the following predicates work on the *hexadecimal* representation of hashes that is also used by `crypto_data_hash/3` and related predicates.

Signatures are also represented in hexadecimal notation, and you can use `hex_bytes/2` to convert them to and from lists of bytes (integers).

3.6.1 ECDSA

ecdsa_sign(+*Key*, +*Data*, -*Signature*, +*Options***)**

Create an ECDSA signature for *Data* with EC private key *Key*. Among the most common cases is signing a hash that was created with `crypto_data_hash/3` or other predicates of this library. For this reason, the default encoding (`hex`) assumes that *Data* is an atom, string, character list or code list representing the data in hexadecimal notation. See `rsa_sign/4` for an example.

Options:

encoding(+*Encoding***)**
Encoding to use for *Data*. Default is `hex`. Alternatives are `octet`, `utf8` and `text`.

ecdsa_verify(+*Key*, +*Data*, +*Signature*, +*Options***)** *[semidet]*

True iff *Signature* can be verified as the ECDSA signature for *Data*, using the EC public key *Key*.

Options:

encoding(+*Encoding***)**
Encoding to use for *Data*. Default is `hex`. Alternatives are `octet`, `utf8` and `text`.

3.6.2 RSA

rsa_sign(+*Key*, +*Data*, -*Signature*, +*Options***)** *[det]*

Create an RSA signature for *Data* with private key *Key*. *Options*:

type(+*Type***)**
SHA algorithm used to compute the digest. Values are `sha1`, `sha224`, `sha256`, `sha384` or `sha512`. The default is a cryptographically secure algorithm. If you specify a variable, then it is unified with the algorithm that was used.

encoding(+*Encoding***)**
Encoding to use for *Data*. Default is `hex`. Alternatives are `octet`, `utf8` and `text`.

This predicate can be used to compute a `sha256WithRSAEncryption` signature as follows:

```
sha256_with_rsa(PemKeyFile, Password, Data, Signature) :-
    Algorithm = sha256,
    read_key(PemKeyFile, Password, Key),
    crypto_data_hash(Data, Hash, [algorithm(Algorithm),
                                  encoding(octet)]),
    rsa_sign(Key, Hash, Signature, [type(Algorithm)]).

read_key(File, Password, Key) :-
    setup_call_cleanup(
        open(File, read, In, [type(binary)]),
        load_private_key(In, Password, Key),
        close(In)).
```

Note that a hash that is computed by `crypto_data_hash/3` can be directly used in `rsa_sign/4` as well as `ecdsa_sign/4`.

rsa_verify(+*Key, +Data, +Signature, +Options***)** *[semidet]*

Verify an RSA signature for *Data* with public key *Key*.

Options:

type(+*Type***)**
SHA algorithm used to compute the digest. Values are `sha1`, `sha224`, `sha256`, `sha384` or `sha512`. The default is the same as for `rsa_sign/4`. This option must match the algorithm that was used for signing. When operating with different parties, the used algorithm must be communicated over an authenticated channel.

encoding(+*Encoding***)**
Encoding to use for *Data*. Default is `hex`. Alternatives are `octet`, `utf8` and `text`.

3.7 Asymmetric encryption and decryption

The following predicates provide *asymmetric* RSA encryption and decryption. This means that the key that is used for *encryption* is different from the one used to *decrypt* the data:

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

Options:

encoding(+*Encoding***)**
Encoding to use for Data. Default is `utf8`. Alternatives are `utf8` and `octet`.

padding(+*PaddingScheme***)**
Padding scheme to use. Default is `pkcs1`. Alternatives are `pkcs1_oaep`, `sslv23` and `none`. Note that `none` should only be used if you implement cryptographically sound padding modes in your application code as encrypting unpadded data with RSA is insecure

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.

3.8 Symmetric encryption and decryption

The following predicates provide *symmetric* encryption and decryption. This means that the *same* key is used in both cases.

crypto_data_encrypt(+*PlainText, +Algorithm, +Key, +IV, -CipherText, +Options*)

Encrypt the given *PlainText*, using the symmetric algorithm *Algorithm*, key *Key*, and initialization vector (or nonce) *IV*, to give *CipherText*.

PlainText must be a string, atom or list of codes or characters, and *CipherText* is created as a string. *Key* and *IV* are typically lists of *bytes*, though atoms and strings are also permitted. *Algorithm* must be an algorithm which your copy of OpenSSL knows about.

Keys and IVs can be chosen at random (using for example `crypto_n_random_bytes/2`) or derived from input keying material (IKM) using for example `crypto_data_hkdf/4`. This input is often a shared secret, such as a negotiated point on an elliptic curve, or the hash that was computed from a password via `crypto_password_hash/3` with a freshly generated and specified *salt*.

Reusing the same combination of *Key* and *IV* typically leaks at least *some* information about the plaintext. For example, identical plaintexts will then correspond to identical ciphertexts. For some algorithms, reusing an *IV* with the same *Key* has disastrous results and can cause the loss of all properties that are otherwise guaranteed. Especially in such cases, an *IV* is also called a *nonce* (number used once). If an *IV* is not needed for your algorithm (such as `'aes-128-ecb'`) then any value can be provided as it will be ignored by the underlying implementation. Note that such algorithms do not provide *semantic security* and are thus insecure. You should use stronger algorithms instead.

It is safe to store and transfer the used initialization vector (or nonce) in plain text, but the key *must be kept secret*.

Commonly used algorithms include:

`'chacha20-poly1305'` A powerful and efficient *authenticated* encryption scheme, providing secrecy and at the same time reliable protection against undetected *modifications* of the encrypted data. This is a very good choice for virtually all use cases. It is a *stream cipher* and can encrypt data of any length up to 256 GB. Further, the encrypted data has exactly the same length as the original, and no padding is used. It requires OpenSSL 1.1.0 or greater. See below for an example.

`'aes-128-gcm'` Also an authenticated encryption scheme. It uses a 128-bit (i.e., 16 bytes) key and a 96-bit (i.e., 12 bytes) nonce. It requires OpenSSL 1.1.0 or greater.

`'aes-128-cbc'` A *block cipher* that provides secrecy, but does not protect against unintended modifications of the cipher text. This algorithm uses 128-bit (16 bytes) keys and initialization vectors. It works with all supported versions of OpenSSL. If possible, consider using an authenticated encryption scheme instead.

Options:

encoding(+*Encoding*)

Encoding to use for *PlainText*. Default is `utf8`. Alternatives are `utf8` and `octet`.

padding(+*PaddingScheme*)

For block ciphers, the padding scheme to use. Default is `block`. You can disable padding by supplying `none` here. If padding is disabled for block ciphers, then the length of the ciphertext must be a multiple of the block size.

tag(*-List*)
For authenticated encryption schemes, *List* is unified with a list of *bytes* holding the tag. This tag must be provided for decryption. Authenticated encryption requires OpenSSL 1.1.0 or greater.

tag_length(*+Length*)
For authenticated encryption schemes, the desired length of the tag, specified as the number of bytes. The default is 16. Smaller numbers are not recommended.

For example, with OpenSSL 1.1.0 and greater, we can use the ChaCha20 stream cipher with the Poly1305 authenticator. This cipher uses a 256-bit key and a 96-bit *nonce*, i.e., 32 and 12 *bytes*, respectively:

```
?- Algorithm = 'chacha20-poly1305',
   crypto_n_random_bytes(32, Key),
   crypto_n_random_bytes(12, IV),
   crypto_data_encrypt("this is some input", Algorithm,
               Key, IV, CipherText, [tag(Tag)]),
   crypto_data_decrypt(CipherText, Algorithm,
               Key, IV, RecoveredText, [tag(Tag)]).
Algorithm = 'chacha20-poly1305',
Key = [65, 147, 140, 197, 27, 60, 198, 50, 218|...],
IV = [253, 232, 174, 84, 168, 208, 218, 168, 228|...],
CipherText = <binary string>,
Tag = [248, 220, 46, 62, 255, 9, 178, 130, 250|...],
RecoveredText = "this is some input".
```

In this example, we use `crypto_n_random_bytes/2` to generate a key and nonce from cryptographically secure random numbers. For repeated applications, you must ensure that a nonce is only used *once* together with the same key. Note that for *authenticated* encryption schemes, the *tag* that was computed during encryption is necessary for decryption. It is safe to store and transfer the tag in plain text.

See also
- `crypto_data_decrypt/6`.
- `hex_bytes/2` for conversion between bytes and hex encoding.

crypto_data_decrypt(*+CipherText, +Algorithm, +Key, +IV, -PlainText, +Options*)
Decrypt the given *CipherText*, using the symmetric algorithm *Algorithm*, key *Key*, and initialization vector *IV*, to give *PlainText*. *CipherText* must be a string, atom or list of codes or characters, and *PlainText* is created as a string. *Key* and *IV* are typically lists of *bytes*, though atoms and strings are also permitted. *Algorithm* must be an algorithm which your copy of OpenSSL knows. See `crypto_data_encrypt/6` for an example.

encoding(*+Encoding*)
Encoding to use for *CipherText*. Default is `utf8`. Alternatives are `utf8` and `octet`.

padding(*+PaddingScheme*)
For block ciphers, the padding scheme to use. Default is `block`. You can disable padding by supplying `none` here.

tag(+*Tag*)
For authenticated encryption schemes, the tag must be specified as a list of bytes exactly as they were generated upon encryption. This option requires OpenSSL 1.1.0 or greater.

min_tag_length(+*Length*)
If the tag length is smaller than 16, this option must be used to permit such shorter tags. This is used as a safeguard against truncation attacks, where an attacker provides a short tag that is easier to guess.

3.9 Number theory

This library provides operations from number theory that frequently arise in cryptographic applications, complementing the existing built-ins and GMP bindings:

crypto_modular_inverse(+*X*, +*M*, -*Y*) *[det]*
Compute the modular multiplicative inverse of the integer *X*. *Y* is unified with an integer such that *X***Y* is congruent to 1 modulo *M*.

crypto_generate_prime(+*N*, -*P*, +*Options*) *[det]*
Generate a prime *P* with at least *N* bits. *Options* is a list of options. Currently, the only supported option is:

safe(*Boolean*)
If *Boolean* is `true` (default is `false`), then a *safe* prime is generated. This means that *P* is of the form 2*Q + 1 where Q is also prime.

crypto_is_prime(+*P*, +*Options*) *[semidet]*
True iff *P* passes a probabilistic primality test. *Options* is a list of options. Currently, the only supported option is:

iterations(*N*)
N is the number of iterations that are performed. If this option is not specified, a number of iterations is used such that the probability of a false positive is at most 2^(-80).

3.10 Elliptic curves

This library provides functionality for reasoning over *elliptic curves*. Elliptic curves are represented as opaque objects. You acquire a handle for an elliptic curve via `crypto_name_curve/2`.

A *point* on a curve is represented by the Prolog term `point(X, Y)`, where *X* and *Y* are integers that represent the point's affine coordinates.

The following predicates are provided for reasoning over elliptic curves:

crypto_name_curve(+*Name*, -*Curve*) *[det]*
Obtain a handle for a *named* elliptic curve. *Name* is an atom, and *Curve* is unified with an opaque object that represents the curve. Currently, only elliptic curves over prime fields are supported. Examples of such curves are `prime256v1` and `secp256k1`.

If you have OpenSSL installed, you can get a list of supported curves via:

```
$ openssl ecparam -list_curves
```

crypto_curve_order(+*Curve, -Order***)** *[det]*

Obtain the order of an elliptic curve. *Order* is an integer, denoting how many points on the curve can be reached by multiplying the curve's generator with a scalar.

crypto_curve_generator(+*Curve, -Point***)** *[det]*

Point is the *generator* of the elliptic curve *Curve*.

crypto_curve_scalar_mult(+*Curve*, +*N*, +*Point*, -*R***)** *[det]*

R is the result of *N* times *Point* on the elliptic curve *Curve*. *N* must be an integer, and *Point* must be a point on the curve.

3.11 Example: Establishing a shared secret

As one example that involves most predicates of this library, we explain a way to establish a *shared secret* over an insecure channel. We shall use *elliptic curves* for this purpose.

Suppose Alice wants to establish an encrypted connection with Bob. To achieve this even over a channel that may be subject to eavesdrooping and man-in-the-middle attacks, Bob performs the following steps:

1. Choose an elliptic curve *C*, using `crypto_name_curve/2`.
2. Pick a random integer *k* such that *k* is greater than 0 and smaller than the order of *C*. This can be done using `crypto_curve_order/2` and `crypto_n_random_bytes/2`.
3. Use `crypto_curve_generator/2` to obtain the generator *G* of *C*, and use `crypto_curve_scalar_mult/4` to compute the scalar product *k**G*. We call this result *R*, denoting a point on the curve.
4. Sign *R* (using for example `rsa_sign/4` or `ecdsa_sign/4`) and send this to Alice.

This mechanism hinges on a way for Alice to establish the *authenticity* of the signed message (using predicates like `rsa_verify/4` and `ecdsa_verify/4`), for example by means of a public key that was previously exchanged or is signed by a trusted party in such a way that Alice can be sufficiently certain that it belongs to Bob. However, none of these steps require any encryption!

Alice in turn performs the following steps:

1. Create a random integer *j* such that *j* is greater than 0 and smaller than the order of C. Alice can also use `crypto_curve_order/2` and `crypto_n_random_bytes/2` for this.
2. Compute the scalar product *j**G*, where *G* is again the generator of *C* as obtained via `crypto_curve_generator/2`.
3. Further, compute the scalar product *j**R*, which is a point on the curve that we shall call Q. We can derive a *shared secret* from *Q*, using for example `crypto_data_hkdf/4`, and encrypt any message with it (using for example `crypto_data_encrypt/6`).
4. Send the point *j**G* and the encrypted message to Bob.

Bob receives *j*G* in plain text and can arrive at the same shared secret by performing the calculation *k*(j*G)*, which is - by associativity and commutativity of scalar multiplication - identical to the point *j*(k*G)*, which is again Q from which the shared secret can be derived, and the message can be decrypted with `crypto_data_decrypt/6`.

This method is known as Diffie-Hellman-Merkle key exchange over elliptic curves, abbreviated as ECDH. It provides forward secrecy (FS): Even if the private key that was used to establish the *authenticity* of Bob is later compromised, the encrypted messages cannot be decrypted with it.

A major attraction of using elliptic curves for this purpose is found in the comparatively small key size that suffices to make any attacks unrealistic as far as we currently know. In particular, given any point on the curve, we currently have no efficient way to determine by which scalar the generator was multiplied to obtain that point. The method described above relies on the hardness of this so-called *elliptic curve discrete logarithm problem* (ECDLP). On the other hand, some of the named curves have been suspected to be chosen in such a way that they could be prone to attacks that are not publicly known.

As an alternative to ECDH, you can use the original DH key exchange scheme, where the prime field GF(p) is used instead of an elliptic curve, and *exponentiation* of a suitable generator is used instead of scalar multiplication. You can use `crypto_generate_prime/3` to generate a sufficiently large prime for this purpose.

4 XML cryptographic libraries

The SSL package provides several libraries dealing with cryptographic operations of XML documents. These libraries depend on the `sgml` package. These libraries are part of this package because the `sgml` package has no external dependencies and will thus be available in any SWI-Prolog installation while configuring and building this `ssl` package is much more involved.

4.1 library(saml): SAML Authentication

See also https://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf

There are four primary integration points for applications to use this code: 1) You must declare at least one service provider (SP) 2) You must declare at least one identity provider (IdP) per SP 3) Finally, you can call `saml_authenticate(+SP, +IdP, +Callback, +Request)` to obtain assertions The asynchronous nature of the SAML process means that a callback must be used. Assuming that the IdP was able to provide at least some valid assertions about the user, after calling Callback with 2 extra arguments (a list of the assertion terms and the URL being request by the user), the user will be redirected back to their original URL. It is therefore up to the callback to ensure that this does not simply trigger another round of SAML negotiations - for example, by throwing `http_reply(forbidden(RequestURL))` if the assertions are not strong enough 4) Finally, your SP metadata will be available from the web server directly. This is required to configure the IdP. This will be available at './metadata.xml', relative to the LocationSpec provided when the SP was declared.

Configuring an SP: To declare an SP, use the declaration :

```
-saml_sp(+ServiceProvider: atom, +LocationSpec: term, +PrivateKeySpec: term, +Password: a
```

The ServiceProvider is the identifier of your service. Ideally, this should be a fully-qualified URI The LocationSpec is a location that the HTTP dispatch layer will understand for example '.' or `root('saml')`. The Private KeySpec is a 'file specifier' that resolves to a private key (see below for specifiers) The Password is a password used for reading the private key. If the key is not encrypted, any atom can be supplied as it will be ignored The CertificateSpec is a file specifier that resolves to a certificate holding the public key corresponding to PrivateKeySPec There are currently no implemented options (the list is ignored).

Configuring an IdP: To declare an IdP, use the declaration `:-saml_idp(+ServiceProvider: atom, +MetadataSpec: term)`. ServiceProvider is the identifier used when declaring your SP. You do not need to declare them in a particular order, but both must be present in the system before running `saml_authenticate/4`. MetadataSpec is a file specifier that resolves to the metadata for the IdP. Most IdPs will be able to provide this on request

File Specifiers: The following specifiers are supported for locating files:

- `file(Filename)`: The local file Filename
- `resource(Resource)`: The prolog resource Resource. See `resource/3`
- `url(URL)`: The file identified by the HTTP (or HTTPS if you have the HTTPS plugin loaded) URL

This library uses SAML to exchange messages with an Identity Provider to establish assertions about the current user's session. It operates only as the service end, not the identity provider end.

4.2 library(xmlenc): XML encryption library

See also
- https://www.w3.org/TR/xmlenc-core1/
- https://en.wikipedia.org/wiki/Security_Assertion_Markup_Language

This library is a partial implementation of the XML encryption standard. It implements the *decryption* part, which is needed by SAML clients.

decrypt_xml(+*DOMIn*, -*DOMOut*, :*KeyCallback*, +*Options*) *[det]*

	Arguments
KeyCallback	may be called as follows: • `call(KeyCallback, name, KeyName, Key)` • `call(KeyCallback, public_key, public_key(RSA), Key)` • `call(KeyCallback, certificate, Certificate, Key)`

load_certificate_from_base64_string(+*String*, -*Certificate*) *[det]*

Loads a certificate from a string, adding newlines and header where appropriate so that OpenSSL 1.0.1+ will be able to parse it

4.3 library(xmldsig): XML Digital signature

See also
- http://www.di-mgt.com.au/xmldsig.html
- https://www.bmt-online.org/geekisms/RSA_verify
- http://stackoverflow.com/questions/5576777/whats-the-difference-between-nid-sha-and-nid-sha1-in-openssl

This library deals with *XMLDSIG*, RSA signed XML documents.

xmld_signed_DOM(+*DOM, -SignedDOM, +Options***)** *[det]*

Translate an XML *DOM* structure in a signed version. *Options*:

key_file(+*File***)**
: *File* holding the private key needed to sign

key_password(+*Password***)**
: String holding the password to op the private key.

The *SignedDOM* must be emitted using `xml_write/3` or `xml_write_canonical/3`. If `xml_write/3` is used, the option `layout(false)` is needed to avoid changing the layout of the `SignedInfo` element and the signed *DOM*, which will cause the signature to be invalid.

xmld_verify_signature(+*DOM, +SignatureDOM, -Certificate, +Options***)** *[det]*

Confirm that an `ds:Signature` element contains a valid signature. *Certificate* is bound to the certificate that appears in the element if the signature is valid. It is up to the caller to determine if the certificate is trusted or not.

Note: The *DOM* and *SignatureDOM* must have been obtained using the `load_structure/3` option `keep_prefix(true)` otherwise it is impossible to generate an identical document for checking the signature. See also `xml_write_canonical/3`.

5 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.

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 and MacOS binaries of SWI-Prolog ships with its own binary of the OpenSSL library. Ensure this is up-to-date. On systems that ship with the OpenSSL library SWI-Prolog uses the system version. This applies notably for all Linux packages. Check the origin and version of the OpenSSL libraries and verify there are no more recent security patches regularly if security is important to you. 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'.
```

[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.

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.

6 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.

6.0.1 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!

6.0.2 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` 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 demands 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 6 for more information about revocations.

7 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.

7.1 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 actually be omitted because the plugin is dynamically loaded by `http_open/3` if the `https` scheme is detected. See section 5 for more information about security aspects.

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

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

7.2 Creating an HTTPS server

The SWI-Prolog infrastructure provides two main ways to launch an HTTPS server:

- Using `library(http/thread_httpd)`, the server is started in HTTPS mode by adding an option `ssl/1` to `http_server/2`. The argument of `ssl/1` is an option list that is passed as the third argument to `ssl_context/3`.
- Using `library(http/http_unix_daemon)`, an HTTPS server is started by using the command line argument `--https`.

Two items are typically specified as, respectively, options or additional command line arguments:

- **server certificate**. This identifies the server and acts as a *public key* for the encryption.
- **private key** of the server, which must be kept secret. The key *may* be protected by a password. If this is the case, the server must provide the password by means of the `password` option, the `pem_password_hook` callback or, in case of the Unix daemon, via the `--pwfile` or `--password` command line options.

Here is an example that uses the self-signed demo certificates distributed with the SSL package. As is typical for publicly accessible HTTPS servers, this version does *not* require a certificate from the client:

```
:- 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
                    ]).
```

There are two *hooks* that let you extend HTTPS servers with custom definitions:

- `http:ssl_server_create_hook(+SSL0, -SSL, +Options)`: This extensible predicate is called exactly *once*, after creating an HTTPS server with Options. If this predicate succeeds, `SSL` is the context that is used for negotiating all new connections. Otherwise, `SSL0` is used, which is the context that was created with the given options.

- `http:ssl_server_open_client_hook(+SSL0, -SSL, +Options)`: This predicate is called before *each* connection that the server negotiates with a client. If this predicate succeeds, `SSL` is the context that is used for the new connection. Otherwise, `SSL0` is used, which is the context that was created when launching the server.

Important use cases of these hooks are running dual-stack RSA/ECDSA servers, updating certificates while the server keeps running, and tweaking SSL parameters for connections. Use `ssl_set_options/3` to create and configure copies of existing contexts in these hooks.

The example file `https.pl` also provides a server that *does* require the client to show its certificate. This provides an additional level of security, often used to allow a selected set of clients to perform sensitive tasks.

Note that a single Prolog program 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.

7.3 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.

Virtual hosts for HTTPS are available via *Server Name Indication* (SNI). This is a TLS extension that allows servers to host different domains from the same IP address. See the `sni_hook/1` option of `ssl_context/3` for more information.

8 Acknowledgments

The development of the SWI-Prolog SSL interface has been sponsored by Scientific Software and Systems Limited. The current version contains contributions from many people. Besides the mentioned authors, Markus Triska has submitted several patches, and improved and documented the integration of this package with the HTTP infrastructure.

References

Index