NAME

IO::Socket::SSL - SSL sockets with IO::Socket interface

SYNOPSIS

```
use strict;
use IO::Socket::SSL;

# simple client
my $cl = IO::Socket::SSL->new('www.google.com:443');
print $cl "GET / HTTP/1.0\r\n\r\n";
print <$cl>;

# simple server
my $srv = IO::Socket::SSL->new(
LocalAddr => '0.0.0.0:1234',
Listen => 10,
SSL_cert_file => 'server-cert.pem',
SSL_key_file => 'server-key.pem',
);
$srv->accept;
```

DESCRIPTION

IO::Socket::SSL makes using SSL/TLS much easier by wrapping the necessary functionality into the familiar IO::Socket interface and providing secure defaults whenever possible. This way, existing applications can be made SSL-aware without much effort, at least if you do blocking I/O and don't use select or poll.

But, under the hood, SSL is a complex beast. So there are lots of methods to make it do what you need if the default behavior is not adequate. Because it is easy to inadvertently introduce critical security bugs or just hard to debug problems, I would recommend studying the following documentation carefully.

The documentation consists of the following parts:

- "Essential Information About SSL/TLS"
- "Basic SSL Client"
- "Basic SSL Server"
- "Common Usage Errors"
- "Common Problems with SSL"
- "Using Non-Blocking Sockets"
- "Advanced Usage"
- "Integration Into Own Modules"
- "Description Of Methods"

Additional documentation can be found in

- IO::Socket::SSL::Intercept - Doing Man-In-The-Middle with SSL
- IO::Socket::SSL::Utils - Useful functions for certificates etc

Essential Information About SSL/TLS

SSL (Secure Socket Layer) or its successor TLS (Transport Layer Security) are protocols to facilitate end-to-end security. These protocols are used when accessing web sites (https), delivering or retrieving email, and in lots of other use cases. In the following documentation we will refer to both SSL and TLS as simply 'SSL'.

SSL enables end-to-end security by providing two essential functions:

Encryption
: This part encrypts the data for transit between the communicating parties, so that nobody in between can read them. It also provides tamper resistance so that nobody in between can manipulate the data.

Identification
: This part makes sure that you talk to the right peer. If the identification is done incorrectly it is easy to mount man-in-the-middle attacks, e.g. if Alice wants to talk to Bob it would be possible for Mallory to put itself in the middle, so that Alice talks to Mallory and Mallory to Bob. All the data would still be encrypted, but not end-to-end between Alice and Bob, but only between Alice and Mallory and then between Mallory and Bob. Thus Mallory would be able to read and modify all traffic between Alice and Bob.

Identification is the part which is the hardest to understand and the easiest to get wrong.

With SSL, the Identification is usually done with **certificates** inside a **PKI** (Public Key Infrastructure). These Certificates are comparable to an identity card, which contains information about the owner of the card. The card then is somehow **signed** by the **issuer** of the card, the **CA** (Certificate Agency).

To verify the identity of the peer the following must be done inside SSL:

- Get the certificate from the peer. If the peer does not present a certificate we cannot verify it.
- Check if we trust the certificate, e.g. make sure it's not a forgery.

  We believe that a certificate is not a fake if we either know the certificate already or if we **trust** the issuer (the CA) and can verify the issuers signature on the certificate. In reality there is often a hierarchy of certificate agencies and we only directly trust the root of this hierarchy. In this case the peer not only sends his own certificate, but also all **intermediate certificates**. Verification will be done by building a **trust path** from the trusted root up to the peers certificate and checking in each step if the we can verify the issuer's signature.

  This step often causes problems because the client does not know the necessary trusted root certificates. These are usually stored in a system dependent CA store, but often the browsers have their own CA store.
- Check if the certificate is still valid. Each certificate has a lifetime and should not be used after that time because it might be compromised or the underlying cryptography got broken in the mean time.
- Check if the subject of the certificate matches the peer. This is like comparing the picture on the identity card against the person representing the identity card.

  When connecting to a server this is usually done by comparing the hostname used for connecting against the names represented in the certificate. A certificate might contain multiple names or wildcards, so that it can be used for multiple hosts (e.g. *.example.com and *.example.org).

  Although nobody sane would accept an identity card where the picture does not match the person we see, it is a common implementation error with SSL to omit this check or get it wrong.
- Check if the certificate was revoked by the issuer. This might be the case if the certificate was compromised somehow and now somebody else might use it to claim the wrong identity. Such revocations happened a lot after the heartbleed attack.

  For SSL there are two ways to verify a revocation, CRL and OCSP. With CRLs (Certificate Revocation List) the CA provides a list of serial numbers for revoked certificates. The client somehow has to download the list (which can be huge) and keep it up to date. With OCSP (Online Certificate Status Protocol) the client can check a single certificate directly by asking the issuer.

Revocation is the hardest part of the verification and none of today's browsers get it fully correct. But, they are still better than most other implementations which don't implement revocation checks or leave the hard parts to the developer.

When accessing a web site with SSL or delivering mail in a secure way the identity is usually only checked one way, e.g. the client wants to make sure it talks to the right server, but the server usually does not care which client it talks to. But, sometimes the server wants to identify the client too and will request a certificate from the client which the server must verify in a similar way.

Basic SSL Client

A basic SSL client is simple:

```
my $client = IO::Socket::SSL->new('www.example.com:443')
or die "error=$!, ssl_error=$SSL_ERROR";
```

This will take the OpenSSL default CA store as the store for the trusted CA. This usually works on UNIX systems. If there are no certificates in the store it will try use Mozilla::CA which provides the default CAs of Firefox.

In the default settings, IO::Socket::SSL will use a safer cipher set and SSL version, do a proper hostname check against the certificate, and use SNI (server name indication) to send the hostname inside the SSL handshake. This is necessary to work with servers which have different certificates behind the same IP address. It will also check the revocation of the certificate with OCSP, but currently only if the server provides OCSP stapling (for deeper checks see `ocsp_resolver` method).

Lots of options can be used to change ciphers, SSL version, location of CA and much more. See documentation of methods for details.

With protocols like SMTP it is necessary to upgrade an existing socket to SSL. This can be done like this:

```
my $client = IO::Socket::INET->new('mx.example.com:25') or die $!;
# .. SMTP dialogs ... send STARTTLS and read reply ...
# SSL upgrade
IO::Socket::SSL->start_SSL($client,
# explicitly set hostname we should use for SNI
SSL_hostname => 'mx.example.com'
) or die $SSL_ERROR;
```

A more complete example for a simple HTTP client:

```
my $client = IO::Socket::SSL->new(
# where to connect
PeerHost => "www.example.com",
PeerPort => "https",

# certificate verification - VERIFY_PEER is default
SSL_verify_mode => SSL_VERIFY_PEER,

# location of CA store
# need only be given if default store should not be used
SSL_ca_path => '/etc/ssl/certs', # typical CA path on Linux
SSL_ca_file => '/etc/ssl/cert.pem', # typical CA file on BSD

# or just use default path on system:
IO::Socket::SSL::default_ca(), # either explicitly
# or implicitly by not giving SSL_ca_*
```

```
# easy hostname verification
# It will use PeerHost as default name a verification
# scheme as default, which is safe enough for most purposes.
SSL_verifycn_name => 'foo.bar',
SSL_verifycn_scheme => 'http',

# SNI support - defaults to PeerHost
SSL_hostname => 'foo.bar',

) or die "failed connect or ssl handshake: $!,$SSL_ERROR";

# send and receive over SSL connection
print $client "GET / HTTP/1.0\r\n\r\n";
print <$client>;
```

And to do revocation checks with OCSP (only available with OpenSSL 1.0.0 or higher and Net::SSLeay 1.59 or higher):

```
# default will try OCSP stapling and check only leaf certificate
my $client = IO::Socket::SSL->new($dst);

# better yet: require checking of full chain
my $client = IO::Socket::SSL->new(
PeerAddr => $dst,
SSL_ocsp_mode => SSL_OCSP_FULL_CHAIN,
);

# even better: make OCSP errors fatal
# (this will probably fail with lots of sites because of bad OCSP setups)
# also use common OCSP response cache
my $ocsp_cache = IO::Socket::SSL::OCSP_Cache->new;
my $client = IO::Socket::SSL->new(
PeerAddr => $dst,
SSL_ocsp_mode => SSL_OCSP_FULL_CHAIN|SSL_OCSP_FAIL_HARD,
SSL_ocsp_cache => $ocsp_cache,
);

# disable OCSP stapling in case server has problems with it
my $client = IO::Socket::SSL->new(
PeerAddr => $dst,
SSL_ocsp_mode => SSL_OCSP_NO_STAPLE,
);

# check any certificates which are not yet checked by OCSP stapling or
# where we have already cached results. For your own resolving combine
# $ocsp->requests with $ocsp->add_response(uri,response).
my $ocsp = $client->ocsp_resolver();
my $errors = $ocsp->resolve_blocking();
if ($errors) {
warn "OCSP verification failed: $errors";
close($client);
}
```

Basic SSL Server

A basic SSL server looks similar to other IO::Socket servers, only that it also contains settings for certificate and key:

```
# simple server
my $server = IO::Socket::SSL->new(
# where to listen
LocalAddr => '127.0.0.1',
LocalPort => 8080,
Listen => 10,

# which certificate to offer
# with SNI support there can be different certificates per hostname
SSL_cert_file => 'cert.pem',
SSL_key_file => 'key.pem',
) or die "failed to listen: $!";

# accept client
my $client = $server->accept or die
"failed to accept or ssl handshake: $!,$SSL_ERROR";
```

This will automatically use a secure set of ciphers and SSL version and also supports Forward Secrecy with (Elliptic-Curve) Diffie-Hellmann Key Exchange.

If you are doing a forking or threading server, we recommend that you do the SSL handshake inside the new process/thread so that the master is free for new connections. We recommend this because a client with improper or slow SSL handshake could make the server block in the handshake which would be bad to do on the listening socket:

```
# inet server
my $server = IO::Socket::INET->new(
# where to listen
LocalAddr => '127.0.0.1',
LocalPort => 8080,
Listen => 10,
);

# accept client
my $client = $server->accept or die;

# SSL upgrade client (in new process/thread)
IO::Socket::SSL->start_SSL($client,
SSL_server => 1,
SSL_cert_file => 'cert.pem',
SSL_key_file => 'key.pem',
) or die "failed to ssl handshake: $SSL_ERROR";
```

Like with normal sockets, neither forking nor threading servers scale well. It is recommended to use non-blocking sockets instead, see "Using Non-Blocking Sockets"

Common Usage Errors

This is a list of typical errors seen with the use of IO::Socket::SSL:

- Disabling verification with `SSL_verify_mode`.

  As described in "Essential Information About SSL/TLS", a proper identification of the peer is essential and failing to verify makes Man-In-The-Middle attacks possible.

  Nevertheless, lots of scripts and even public modules or applications disable verification, because it is probably the easiest way to make the thing work and usually nobody notices any security problems anyway.

If the verification does not succeed with the default settings, one can do the following:

- Make sure the needed CAs are in the store, maybe use `SSL_ca_file` or `SSL_ca_path` to specify a different CA store.
- If the validation fails because the certificate is self-signed and that's what you expect, you can use the `SSL_fingerprint` option to accept specific certificates by their certificate fingerprint.
- If the validation failed because the hostname does not match and you cannot access the host with the name given in the certificate, you can use `SSL_verifycn_name` to specify they hostname you expect in the certificate.

A common error pattern is also to disable verification if they found no CA store (different modules look at different ''default'' places). Because IO::Socket::SSL is now able to provide a usable CA store on most platforms (UNIX, Mac OSX and Windows) it is better to use the defaults provided by IO::Socket::SSL. If necessary these can be checked with the `default_ca` method.

- Polling of SSL sockets (e.g. select, poll and other event loops).

  If you sysread one byte on a normal socket it will result in a syscall to read one byte. Thus, if more than one byte is available on the socket it will be kept in the network stack of your OS and the next select or poll call will return the socket as readable. But, with SSL you don't deliver single bytes. Multiple data bytes are packaged and encrypted together in an SSL frame. Decryption can only be done on the whole frame, so a sysread for one byte actually reads the complete SSL frame from the socket, decrypts it and returns the first decrypted byte. Further sysreads will return more bytes from the same frame until all bytes are returned and the next SSL frame will be read from the socket.

  Thus, in order to decide if you can read more data (e.g. if sysread will block) you must check if there are still data in the current SSL frame by calling `pending` and if there are no data pending you might check the underlying socket with select or poll. Another way might be if you try to sysread at least 16kByte all the time. 16kByte is the maximum size of an SSL frame and because sysread returns data from only a single SSL frame you can guarantee that there are no pending data.

  See also ''Using Non-Blocking Sockets''.

- Set 'SSL_version' or 'SSL_cipher_list' to a ''better'' value.

  IO::Socket::SSL tries to set these values to reasonable, secure values which are compatible with the rest of the world. But, there are some scripts or modules out there which tried to be smart and get more secure or compatible settings. Unfortunatly, they did this years ago and never updated these values, so they are still forced to do only 'TLSv1' (instead of also using TLSv12 or TLSv11). Or they set 'HIGH' as the cipher list and thought they were secure, but did not notice that 'HIGH' includes anonymous ciphers, e.g. without identification of the peer.

  So it is recommended to leave the settings at the secure defaults which IO::Socket::SSL sets and which get updated from time to time to better fit the real world.

- Make SSL settings inacessible by the user, together with bad builtin settings.

  Some modules use IO::Socket::SSL, but don't make the SSL settings available to the user. This is often combined with bad builtin settings or defaults (like switching verification off).

  Thus the user needs to hack around these restrictions by using `set_args_filter_hack` or similar.

- Use of constants as strings.

  Constants like `SSL_VERIFY_PEER` or `SSL_WANT_READ` should be used as constants and not be put inside quotes, because they represent numerical values.

Common Problems with SSL

SSL is a complex protocol with multiple implementations and each of these has their own quirks. While most of these implementations work together, it often gets problematic with older versions, minimal versions in load balancers, or plain wrong setups.

Unfortunaty these problems are hard to debug. Helpful for debugging are a knowledge of SSL internals, wireshark and the use of the debug settings of IO::Socket::SSL and Net::SSLeay, which can both be set with `$IO::Socket::SSL::DEBUG`. The following debugs levels are defined, but used not in any consistent way:

- 0 - No debugging (default).
- 1 - Print out errors from IO::Socket::SSL and ciphers from Net::SSLeay.
- 2 - Print also information about call flow from IO::Socket::SSL and progress information from Net::SSLeay.
- 3 - Print also some data dumps from IO::Socket::SSL and from Net::SSLeay.

Also, `analyze-ssl.pl` from the ssl-tools repository at <https://github.com/noxxi/p5-ssl-tools> might be a helpful tool when debugging SSL problems, as do the `openssl` command line tool and a check with a different SSL implementation (e.g. a web browser).

The following problems are not uncommon:

- Bad server setup: missing intermediate certificates.

  It is a regular problem that administrators fail to include all necessary certificates into their server setup, e.g. everything needed to build the trust chain from the trusted root. If they check the setup with the browser everything looks ok, because browsers work around these problems by caching any intermediate certificates and apply them to new connections if certificates are missing.

  But, fresh browser profiles which have never seen these intermediates cannot fill in the missing certificates and fail to verify; the same is true with IO::Socket::SSL.

- Old versions of servers or load balancers which do not understand specific TLS versions or croak on specific data.

  From time to time one encounters an SSL peer, which just closes the connection inside the SSL handshake. This can usually be worked around by downgrading the SSL version, e.g. by setting `SSL_version`. Modern Browsers usually deal with such servers by automatically downgrading the SSL version and repeat the connection attempt until they succeed.

  Worse servers do not close the underlying TCP connection but instead just drop the relevant packet. This is harder to detect because it looks like a stalled connection. But downgrading the SSL version often works here too.

  A cause of such problems are often load balancers or security devices, which have hardware acceleration and only a minimal (and less robust) SSL stack. They can often be detected because they support much fewer ciphers than other implementations.

- Bad or old OpenSSL versions.

  IO::Socket::SSL uses OpenSSL with the help of the Net::SSLeay library. It is recommend to have a recent version of this library, because it has more features and usually fewer known bugs.

- Validation of client certificates fail.

  Make sure that the purpose of the certificate allows use as ssl client (check with `openssl` `x509 -purpose`, that the necessary root certificate is in the path specified by `SSL_ca*` (or the default path) and that any intermediate certificates needed to build the trust chain are sent by the client.

Using Non-Blocking Sockets

If you have a non-blocking socket, the expected behavior on read, write, accept or connect is to set `$!` to EWOULDBLOCK if the operation can not be completed immediately. Note that EWOULDBLOCK is the same as EAGAIN on UNIX systems, but is different on Windows.

With SSL, handshakes might occur at any time, even within an established connection. In these cases it is necessary to finish the handshake before you can read or write data. This might result in situations where you want to read but must first finish the write of a handshake or where you want to write but must first finish a read. In these cases `$!` is set to EGAIN like expected, and additionally `$SSL_ERROR` is set to either SSL_WANT_READ or SSL_WANT_WRITE. Thus if you get EWOULDBLOCK on a SSL socket you must check `$SSL_ERROR` for SSL_WANT_* and adapt your event mask accordingly.

Using readline on non-blocking sockets does not make much sense and I would advise against using it. And, while the behavior is not documented for other IO::Socket classes, it will try to emulate the behavior seen there, e.g. to return the received data instead of blocking, even if the line is not complete. If an unrecoverable error occurs it will return nothing, even if it already received some data.

Also, I would advise against using `accept` with a non-blocking SSL object because it might block and this is not what most would expect. The reason for this is that `accept` on a non-blocking TCP socket (e.g. IO::Socket::IP, IO::Socket::INET..) results in a new TCP socket which does not inherit the non-blocking behavior of the master socket. And thus, the initial SSL handshake on the new socket inside `IO::Socket::SSL::accept` will be done in a blocking way. To work around this you are safer by doing a TCP accept and later upgrade the TCP socket in a non-blocking way with `start_SSL` and `accept_SSL`.

```
my $cl = IO::Socket::SSL->new($dst);
$cl->blocking(0);
my $sel = IO::Select->new($cl);
while (1) {
# with SSL a call for reading n bytes does not result in reading of n
# bytes from the socket, but instead it must read at least one full SSL
# frame. If the socket has no new bytes, but there are unprocessed data
# from the SSL frame can_read will block!

# wait for data on socket
$sel->can_read();

# new data on socket or eof
READ:
# this does not read only 1 byte from socket, but reads the complete SSL
# frame and then just returns one byte. On subsequent calls it than
# returns more byte of the same SSL frame until it needs to read the
# next frame.
my $n = sysread( $cl,my $buf,1);
if ( ! defined $n ) {
die $! if not ${EWOULDBLOCK};
next if $SSL_ERROR == SSL_WANT_READ;
if ( $SSL_ERROR == SSL_WANT_WRITE ) {
# need to write data on renegotiation
$sel->can_write;
next;
```

```
}
die "something went wrong: $SSL_ERROR";
} elsif ( ! $n ) {
last; # eof
} else {
# read next bytes
# we might have still data within the current SSL frame
# thus first process these data instead of waiting on the underlying
# socket object
goto READ if $self->pending; # goto sysread
next; # goto $sel->can_read
}
}
```

Advanced Usage

SNI Support

Newer extensions to SSL can distinguish between multiple hostnames on the same IP address using Server Name Indication (SNI).

Support for SNI on the client side was added somewhere in the OpenSSL 0.9.8 series, but with 1.0 a bug was fixed when the server could not decide about its hostname. Therefore client side SNI is only supported with OpenSSL 1.0 or higher in IO::Socket::SSL. With a supported version, SNI is used automatically on the client side, if it can determine the hostname from `PeerAddr` or `PeerHost` (which are synonyms in the underlying IO::Socket:: classes and thus should never be set both or at least not to different values). On unsupported OpenSSL versions it will silently not use SNI. The hostname can also be given explicitly given with `SSL_hostname`, but in this case it will throw in error, if SNI is not supported. To check for support you might call `IO::Socket::SSL->can_client_sni()`.

On the server side, earlier versions of OpenSSL are supported, but only together with Net::SSLeay version >= 1.50. To check for support you might call `IO::Socket::SSL->can_server_sni()`. If server side SNI is supported, you might specify different certificates per host with `SSL_cert*` and `SSL_key*`, and check the requested name using `get_servername`.

Talk Plain and SSL With The Same Socket

It is often required to first exchange some plain data and then upgrade the socket to SSL after some kind of STARTTLS command. Protocols like FTPS even need a way to downgrade the socket again back to plain.

The common way to do this would be to create a normal socket and use `start_SSL` to upgrade and stop_SSL to downgrade:

```
my $sock = IO::Socket::INET->new(...) or die $!;
... exchange plain data on $sock until starttls command ...
IO::Socket::SSL->start_SSL($sock,%sslargs) or die $SSL_ERROR;
... now $sock is a IO::Socket::SSL object ...
... exchange data with SSL on $sock until stoptls command ...
$sock->stop_SSL or die $SSL_ERROR;
... now $sock is again a IO::Socket::INET object ...
```

But, lots of modules just derive directly from IO::Socket::INET. While this base class can be replaced with IO::Socket::SSL, these modules cannot easily support different base classes for SSL and plain data and switch between these classes on a starttls command.

To help in this case, IO::Socket::SSL can be reduced to a plain socket on startup, and connect_SSL/accept_SSL/start_SSL can be used to enable SSL and `stop_SSL` to talk plain again:

```
my $sock = IO::Socket::SSL->new(
PeerAddr => ...
SSL_startHandshake => 0,
%sslargs
) or die $!;
... exchange plain data on $sock until starttls command ...
$sock->connect_SSL or die $SSL_ERROR;
... now $sock is a IO::Socket::SSL object ...
... exchange data with SSL on $sock until stoptls command ...
$sock->stop_SSL or die $SSL_ERROR;
... $sock is still a IO::Socket::SSL object ...
... but data exchanged again in plain ...
```

Integration Into Own Modules

IO::Socket::SSL behaves similarly to other IO::Socket modules and thus could be integrated in the same way, but you have to take special care when using non-blocking I/O (like for handling timeouts) or using select or poll. Please study the documentation on how to deal with these differences.

Also, it is recommended to not set or touch most of the `SSL_*` options, so that they keep their secure defaults. It is also recommended to let the user override these SSL specific settings without the need of global settings or hacks like `set_args_filter_hack`.

The notable exception is `SSL_verifycn_scheme`. This should be set to the hostname verification scheme required by the module or protocol.

Description Of Methods

IO::Socket::SSL inherits from another IO::Socket module. The choice of the super class depends on the installed modules:

- If IO::Socket::IP with at least version 0.20 is installed it will use this module as super class, transparently providing IPv6 and IPv4 support.
- If IO::Socket::INET6 is installed it will use this module as super class, transparently providing IPv6 and IPv4 support.
- Otherwise it will fall back to IO::Socket::INET, which is a perl core module. With IO::Socket::INET you only get IPv4 support.

Please be aware that with the IPv6 capable super classes, it will look first for the IPv6 address of a given hostname. If the resolver provides an IPv6 address, but the host cannot be reached by IPv6, there will be no automatic fallback to IPv4. To avoid these problems you can either force IPv4 by specifying and AF_INET as `Domain` of the socket or globally enforce IPv4 by loading IO::Socket::SSL with the option 'inet4'.

IO::Socket::SSL will provide all of the methods of its super class, but sometimes it will override them to match the behavior expected from SSL or to provide additional arguments.

The new or changed methods are described below, but please also read the section about SSL specific error handling.

Error Handling
: If an SSL specific error occurs, the global variable `$SSL_ERROR` will be set. If the error occurred on an existing SSL socket, the method `errstr` will give access to the latest socket specific error. Both `$SSL_ERROR` and the `errstr` method give a dualvar similar to `$!`, e.g. providing an error number in numeric context or an error description in string context.

new(...)
: Creates a new IO::Socket::SSL object. You may use all the friendly options that came bundled with the super class (e.g. IO::Socket::IP, IO::Socket::INET, If you don't specify any SSL related options it will do its best in using secure defaults, e.g. choosing good ciphers, enabling proper verification, etc.

SSL_server
Set this option to a true value if the socket should be used as a server. If this is not explicitly set it is assumed if the `Listen` parameter is given when creating the socket.

SSL_hostname
This can be given to specify the hostname used for SNI, which is needed if you have multiple SSL hostnames on the same IP address. If not given it will try to determine the hostname from `PeerAddr`, which will fail if only an IP was given or if this argument is used within `start_SSL`.

If you want to disable SNI, set this argument to ''.

Currently only supported for the client side and will be ignored for the server side.

See section "SNI Support" for details of SNI the support.

SSL_startHandshake
If this option is set to false (defaults to true) it will not start the SSL handshake yet. This has to be done later with `accept_SSL` or `connect_SSL`. Before the handshake is started read/write/etcc can be used to exchange plain data.

SSL_ca | SSL_ca_file | SSL_ca_path
Usually you want to verify that the peer certificate has been signed by a trusted certificate authority. In this case you should use this option to specify the file (`SSL_ca_file`) or directory (`SSL_ca_path`) containing the certificate(s) of the trusted certificate authorities. Also you can give X509* certificate handles (from Net::SSLeay or IO::Socket::SSL::Utils) as a list with `SSL_ca`. These will be added to the CA store before path and file and thus take precedence. If neither SSL_ca, nor SSL_ca_file or SSL_ca_path are set it will use `default_ca()` to determine the user-set or system defaults. If you really don't want to set a CA set SSL_ca_file or SSL_ca_path to `\undef` or SSL_ca to an empty list. (unfortunatly `''` is used by some modules using IO::Socket::SSL when CA is not exlicitly given).

SSL_client_ca | SSL_client_ca_file
If verify_mode is VERIFY_PEER on the server side these options can be used to set the list of acceptable CAs for the client. This way the client can select they required certificate from a list of certificates. The value for these options is similar to `SSL_ca` and `SSL_ca_file`.

SSL_fingerprint
Sometimes you have a self-signed certificate or a certificate issued by an unknown CA and you really want to accept it, but don't want to disable verification at all. In this case you can specify the fingerprint of the certificate as `'algo$hex_fingerprint'`. `algo` is a fingerprint algorithm supported by OpenSSL, e.g. 'sha1','sha256'... and `hex_fingerprint` is the hexadecimal representation of the binary fingerprint. To get the fingerprint of an established connection you can use `get_fingerprint`.

You can specify a list of fingerprints in case you have several acceptable certificates. If a fingerprint matches the topmost certificate no additional validations can make the verification fail.

SSL_cert_file | SSL_cert | SSL_key_file | SSL_key
If you create a server you usually need to specify a server certificate which should be verified by the client. Same is true for client certificates, which should be verified by the server. The certificate can be given as a file with SSL_cert_file or as an internal representation of a X509* object with SSL_cert. If given as a file it will automatically detect the format. Supported file formats are PEM, DER and PKCS#12, where PEM and PKCS#12 can contain the certicate and the chain to use, while DER can only contain a single certificate.

If given as a list of X509* please note, that the all the chain certificates (e.g. all except the first) will be "consumed" by openssl and will be freed if the SSL context gets destroyed - so you should never free them yourself. But the servers certificate (e.g. the first) will not be consumed by openssl and thus must be freed by the application.

For each certificate a key is need, which can either be given as a file with SSL_key_file or as an internal representation of a EVP_PKEY* object with SSL_key. If a key was already given within the PKCS#12 file specified by SSL_cert_file it will ignore any SSL_key or SSL_key_file. If no SSL_key or SSL_key_file was given it will try to use the PEM file given with SSL_cert_file again, maybe it contains the key too.

If your SSL server should be able to use different certificates on the same IP address, depending on the name given by SNI, you can use a hash reference instead of a file with <hostname =cert_file>>.

In case certs and keys are needed but not given it might fall back to builtin defaults, see "Defaults for Cert, Key and CA".

Examples:

```
SSL_cert_file => 'mycert.pem',
SSL_key_file => 'mykey.pem',

SSL_cert_file => {
"foo.example.org" => 'foo-cert.pem',
"bar.example.org" => 'bar-cert.pem',
# used when nothing matches or client does not support SNI
'' => 'default-cert.pem',
}
SSL_key_file => {
"foo.example.org" => 'foo-key.pem',
"bar.example.org" => 'bar-key.pem',
# used when nothing matches or client does not support SNI
'' => 'default-key.pem',
}
```

SSL_passwd_cb

If your private key is encrypted, you might not want the default password prompt from Net::SSLeay. This option takes a reference to a subroutine that should return the password required to decrypt your private key.

SSL_use_cert

If this is true, it forces IO::Socket::SSL to use a certificate and key, even if you are setting up an SSL client. If this is set to 0 (the default), then you will only need a certificate and key if you are setting up a server.

SSL_use_cert will implicitly be set if SSL_server is set. For convenience it is also set if it was not given but a cert was given for use (SSL_cert_file or similar).

SSL_version

Sets the version of the SSL protocol used to transmit data. 'SSLv23' uses a handshake compatible with SSL2.0, SSL3.0 and TLS1.x, while 'SSLv2', 'SSLv3', 'TLSv1', 'TLSv1_1' or 'TLSv1_2' restrict handshake and protocol to the specified version. All values are case-insensitive. Instead of 'TLSv1_1' and 'TLSv1_2' one can also use 'TLSv11' and 'TLSv12'. Support for 'TLSv1_1' and 'TLSv1_2' requires recent versions of Net::SSLeay and openssl.

Independent from the handshake format you can limit to set of accepted SSL versions by adding !version separated by ':'.

The default SSL_version is 'SSLv23:!SSLv3:!SSLv2' which means, that the handshake format is compatible to SSL2.0 and higher, but that the successful handshake is limited to TLS1.0 and higher, that is no SSL2.0 or SSL3.0 because both of these versions have serious security issues and should not be used anymore. You can also use !TLSv1_1 and !TLSv1_2 to disable TLS versions 1.1 and 1.2 while still allowing TLS version 1.0.

Setting the version instead to 'TLSv1' might break interaction with older clients, which need and SSL2.0 compatible handshake. On the other side some clients just close the connection when they receive a TLS version 1.1 request. In this case setting the version to 'SSLv23:!SSLv2:!SSLv3:!TLSv1_1:!TLSv1_2' might help.

SSL_cipher_list
: If this option is set the cipher list for the connection will be set to the given value, e.g. something like 'ALL:!LOW:!EXP:!aNULL'. Look into the OpenSSL documentation (<http://www.openssl.org/docs/apps/ciphers.html#CIPHER_STRINGS>) for more details.

Unless you fail to contact your peer because of no shared ciphers it is recommended to leave this option at the default setting. The default setting prefers ciphers with forward secrecy, disables anonymous authentication and disables known insecure ciphers like MD5, DES etc. This gives a grade A result at the tests of SSL Labs. To use the less secure OpenSSL builtin default (whatever this is) set SSL_cipher_list to ''.

SSL_honor_cipher_order
: If this option is true the cipher order the server specified is used instead of the order proposed by the client. This option defaults to true to make use of our secure cipher list setting.

SSL_dh_file
: If you want Diffie-Hellman key exchange you need to supply a suitable file here or use the SSL_dh parameter. See dhparam command in openssl for more information. To create a server which provides forward secrecy you need to either give the DH parameters or (better, because faster) the ECDH curve.

If neither `SSL_dh_file` not `SSL_dh` is set a builtin DH parameter with a length of 2048 bit is used to offer DH key exchange by default. If you don't want this (e.g. disable DH key exchange) explicitly set this or the `SSL_dh` parameter to undef.

SSL_dh
: Like SSL_dh_file, but instead of giving a file you use a preloaded or generated DH*.

SSL_ecdh_curve
: If you want Elliptic Curve Diffie-Hellmann key exchange you need to supply the OID or NID of a suitable curve (like 'prime256v1') here. To create a server which provides forward secrecy you need to either give the DH parameters or (better, because faster) the ECDH curve.

This parameter defaults to 'prime256v1' (builtin of OpenSSL) to offer ECDH key exchange by default. If you don't want this explicitly set it to undef.

You can check if ECDH support is available by calling `IO::Socket::SSL->can_ecdh`.

SSL_verify_mode
: This option sets the verification mode for the peer certificate. You may combine SSL_VERIFY_PEER (verify_peer), SSL_VERIFY_FAIL_IF_NO_PEER_CERT (fail verification if no peer certificate exists; ignored for clients), SSL_VERIFY_CLIENT_ONCE (verify client once; ignored for clients). See OpenSSL man page for SSL_CTX_set_verify for more information.

The default is SSL_VERIFY_NONE for server (e.g. no check for client certificate) and SSL_VERIFY_PEER for client (check server certificate).

SSL_verify_callback
: If you want to verify certificates yourself, you can pass a sub reference along with this parameter to do so. When the callback is called, it will be passed:

1. a true/false value that indicates what OpenSSL thinks of the certificate,
2. a C-style memory address of the certificate store,
3. a string containing the certificate's issuer attributes and owner attributes, and
4. a string containing any errors encountered (0 if no errors).
5. a C-style memory address of the peer's own certificate (convertible to PEM form with *Net::SSLeay::PEM_get_string_X509()*).
6. The depth of the certificate in the chain. Depth 0 is the leaf certificate.

The function should return 1 or 0, depending on whether it thinks the certificate is valid or invalid. The default is to let OpenSSL do all of the busy work.

The callback will be called for each element in the certificate chain.

See the OpenSSL documentation for SSL_CTX_set_verify for more information.

SSL_verifycn_scheme

The scheme is used to correctly verify the identity inside the certificate by using the hostname of the peer. See the information about the verification schemes in **verify_hostname**.

If you don't specify a scheme it will use 'default', but only complain loudly if the name verification fails instead of letting the whole certificate verification fail. THIS WILL CHANGE, e.g. it will let the certificate verification fail in the future if the hostname does not match the certificate !!!! To override the name used in verification use **SSL_verifycn_name**.

The scheme 'default' is a superset of the usual schemes, which will accept the hostname in common name and subjectAltName and allow wildcards everywhere. While using this scheme is way more secure than no name verification at all you better should use the scheme specific to your application protocol, e.g. 'http', 'ftp'...

If you are really sure, that you don't want to verify the identity using the hostname you can use 'none' as a scheme. In this case you'd better have alternative forms of verification, like a certificate fingerprint or do a manual verification later by calling **verify_hostname** yourself.

SSL_verifycn_publicsuffix

This option is used to specify the behavior when checking wildcards certificates for public suffixes, e.g. no wildcard certificates for *.com or *.co.uk should be accepted, while *.example.com or *.example.co.uk is ok.

If not specified it will simply use the builtin default of IO::Socket::SSL::PublicSuffix, you can create another object with from_string or from_file of this module.

To disable verification of public suffix set this option to `''`.

SSL_verifycn_name

Set the name which is used in verification of hostname. If SSL_verifycn_scheme is set and no SSL_verifycn_name is given it will try to use SSL_hostname or PeerHost and PeerAddr settings and fail if no name can be determined. If SSL_verifycn_scheme is not set it will use a default scheme and warn if it cannot determine a hostname, but it will not fail.

Using PeerHost or PeerAddr works only if you create the connection directly with `IO::Socket::SSL->new`, if an IO::Socket::INET object is upgraded with **start_SSL** the name has to be given in **SSL_verifycn_name** or **SSL_hostname**.

SSL_check_crl

If you want to verify that the peer certificate has not been revoked by the signing authority, set this value to true. OpenSSL will search for the CRL in your SSL_ca_path, or use the file specified by SSL_crl_file. See the Net::SSLeay documentation for more details. Note that this functionality appears to be broken with OpenSSL < v0.9.7b, so its use with lower versions will result in an error.

SSL_crl_file

If you want to specify the CRL file to be used, set this value to the pathname to be used. This must be used in addition to setting SSL_check_crl.

SSL_ocsp_mode

Defines how certificate revocation is done using OCSP (Online Status Revocation Protocol). The default is to send a request for OCSP stapling to the server and if the server sends an OCSP response back the result will be used.

Any other OCSP checking needs to be done manually with `ocsp_resolver`.

The following flags can be combined with `|`:

SSL_OCSP_NO_STAPLE

Don't ask for OCSP stapling. This is the default if SSL_verify_mode is VERIFY_NONE.

SSL_OCSP_TRY_STAPLE

Try OCSP stapling, but don't complain if it gets no stapled response back. This is the default if SSL_verify_mode is VERIFY_PEER (the default).

SSL_OCSP_MUST_STAPLE

Consider it a hard error, if the server does not send a stapled OCSP response back. Most servers currently send no stapled OCSP response back.

SSL_OCSP_FAIL_HARD

Fail hard on response errors, default is to fail soft like the browsers do. Soft errors mean, that the OCSP response is not usable, e.g. no response, error response, no valid signature etc. Certificate revocations inside a verified response are considered hard errors in any case.

Soft errors inside a stapled response are never considered hard, e.g. it is expected that in this case an OCSP request will be send to the responsible OCSP responder.

SSL_OCSP_FULL_CHAIN

This will set up the `ocsp_resolver` so that all certificates from the peer chain will be checked, otherwise only the leaf certificate will be checked against revocation.

SSL_ocsp_staple_callback

If this callback is defined, it will be called with the SSL object and the OCSP response handle obtained from the peer, e.g. `<$cb-`($ssl,$resp)>>. If the peer did not provide a stapled OCSP response the function will be called with `$resp=undef`. Because the OCSP response handle is no longer valid after leaving this function it should not by copied or freed. If access to the response is necessary after leaving this function it can be serialized with `Net::SSLeay::i2d_OCSP_RESPONSE`

If no such callback is provided, it will use the default one, which verifies the response and uses it to check if the certificate(s) of the connection got revoked.

SSL_ocsp_cache

With this option a cache can be given for caching OCSP responses, which could be shared between different SSL contextes. If not given a cache specific to the SSL context only will be used.

You can either create a new cache with `<IO::Socket::SSL::OCSP_Cache-`new([size]) >> or implement your own cache, which needs to have methods `put($key,\%entry)` and `get($key)-`%entry> where entry is the hash representation of the OCSP response with fields like `nextUpdate`. The default implementation of the cache will consider responses valid as long as `nextUpdate` is less then the current time.

SSL_reuse_ctx

If you have already set the above options for a previous instance of IO::Socket::SSL, then you can reuse the SSL context of that instance by passing it as the value for the SSL_reuse_ctx parameter. You may also create a new instance of the IO::Socket::SSL::SSL_Context class, using any context options that you desire without specifying connection options, and pass that here instead.

If you use this option, all other context-related options that you pass in the same call to *new()* will be ignored unless the context supplied was invalid. Note that, contrary to versions of IO::Socket::SSL below v0.90, a global SSL context will not be implicitly used unless you use the *set_default_context()* function.

SSL_create_ctx_callback

With this callback you can make individual settings to the context after it got created and the default setup was done. The callback will be called with the CTX object from Net::SSLeay as the single argument.

Example for limiting the server session cache size:

```
SSL_create_ctx_callback => sub {
my $ctx = shift;
Net::SSLeay::CTX_sess_set_cache_size($ctx,128);
}
```

SSL_session_cache_size

If you make repeated connections to the same host/port and the SSL renegotiation time is an issue, you can turn on client-side session caching with this option by specifying a positive cache size. For successive connections, pass the SSL_reuse_ctx option to the *new()* calls (or use *set_default_context()*) to make use of the cached sessions. The session cache size refers to the number of unique host/port pairs that can be stored at one time; the oldest sessions in the cache will be removed if new ones are added.

This option does not effect the session cache a server has for it's clients, e.g. it does not affect SSL objects with SSL_server set.

SSL_session_cache

Specifies session cache object which should be used instead of creating a new. Overrules SSL_session_cache_size. This option is useful if you want to reuse the cache, but not the rest of the context.

A session cache object can be created using `IO::Socket::SSL::Session_Cache->new( cachesize )`.

Use *set_default_session_cache()* to set a global cache object.

SSL_session_key

Specifies a key to use for lookups and inserts into client-side session cache. Per default ip:port of destination will be used, but sometimes you want to share the same session over multiple ports on the same server (like with FTPS).

SSL_session_id_context

This gives an id for the servers session cache. It's necessary if you want clients to connect with a client certificate. If not given but SSL_verify_mode specifies the need for client certificate a context unique id will be picked.

SSL_error_trap

When using the *accept()* or *connect()* methods, it may be the case that the actual socket connection works but the SSL negotiation fails, as in the case of an HTTP client connecting to an HTTPS server. Passing a subroutine ref attached to this parameter allows you to gain control of the orphaned socket instead of having it be closed forcibly. The subroutine, if called, will be passed two parameters: a reference to the socket on which the SSL negotiation failed and the full text of the error message.

SSL_npn_protocols

If used on the server side it specifies list of protocols advertised by SSL server as an array ref, e.g. ['spdy/2','http1.1']. On the client side it specifies the protocols offered by the client for NPN as an array ref. See also method `next_proto_negotiated`.

Next Protocol Negotiation (NPN) is available with Net::SSLeay 1.46+ and openssl-1.0.1+. To check support you might call `IO::Socket::SSL->can_npn()`. If you use this option with an unsupported Net::SSLeay/OpenSSL it will throw an error.

SSL_alpn_protocols

If used on the server side it specifies list of protocols supported by the SSL server as an array ref, e.g. ['http/2.0', 'spdy/3.1','http/1.1']. On the client side it specifies the protocols advertised by the client for ALPN as an array ref. See also method `alpn_selected`.

Application-Layer Protocol Negotiation (ALPN) is available with Net::SSLeay 1.56+ and openssl-1.0.2+. More details about the extension are in RFC7301. To check support you might call `IO::Socket::SSL-`*can_alpn()* >. If you use this option with an unsupported Net::SSLeay/OpenSSL it will throw an error.

Note that some client implementations may encounter problems if both NPN and ALPN are specified. Since ALPN is intended as a replacement for NPN, try providing ALPN protocols then fall back to NPN if that fails.

accept

This behaves similar to the accept function of the underlying socket class, but additionally does the initial SSL handshake. But because the underlying socket class does return a blocking file handle even when accept is called on a non-blocking socket, the SSL handshake on the new file object will be done in a blocking way. Please see the section about non-blocking I/O for details. If you don't like this behavior you should do accept on the TCP socket and then upgrade it with `start_SSL` later.

connect(...)

This behaves similar to the connnect function but also does an SSL handshake. Because you cannot give SSL specific arguments to this function, you should better either use `new` to create a connect SSL socket or `start_SSL` to upgrade an established TCP socket to SSL.

close(...)

There are a number of nasty traps that lie in wait if you are not careful about using *close()*. The first of these will bite you if you have been using *shutdown()* on your sockets. Since the SSL protocol mandates that a SSL "close notify" message be sent before the socket is closed, a *shutdown()* that closes the socket's write channel will cause the *close()* call to hang. For a similar reason, if you try to close a copy of a socket (as in a forking server) you will affect the original socket as well. To get around these problems, call close with an object-oriented syntax (e.g. `$socket`->close(SSL_no_shutdown => 1)) and one or more of the following parameters:

SSL_no_shutdown

If set to a true value, this option will make *close()* not use the *SSL_shutdown()* call on the socket in question so that the close operation can complete without problems if you have used *shutdown()* or are working on a copy of a socket.

Not using a real ssl shutdown on a socket will make session caching unusable.

SSL_fast_shutdown

If set to true only a unidirectional shutdown will be done, e.g. only the close_notify (see *SSL_shutdown(3)*) will be sent. Otherwise a bidirectional shutdown will be done where it waits for the close_notify of the peer too.

Because a unidirectional shutdown is enough to keep session cache working it defaults to fast shutdown inside close.

SSL_ctx_free
: If you want to make sure that the SSL context of the socket is destroyed when you close it, set this option to a true value.

sysread(BUF, LEN, [OFFSET])

This function behaves from the outside the same as **sysread** in other IO::Socket objects, e.g. it returns at most LEN bytes of data. But in reality it reads not only LEN bytes from the underlying socket, but at a single SSL frame. It then returns up to LEN bytes it decrypted from this SSL frame. If the frame contained more data than requested it will return only LEN data, buffer the rest and return it on further read calls. This means, that it might be possible to read data, even if the underlying socket is not readable, so using poll or select might not be sufficient.

sysread will only return data from a single SSL frame, e.g. either the pending data from the already buffered frame or it will read a frame from the underlying socket and return the decrypted data. It will not return data spanning several SSL frames in a single call.

Also, calls to sysread might fail, because it must first finish an SSL handshake.

To understand these behaviors is essential, if you write applications which use event loops and/or non-blocking sockets. Please read the specific sections in this documentation.

syswrite(BUF, [LEN, [OFFSET]])

This functions behaves from the outside the same as **syswrite** in other IO::Socket objects, e.g. it will write at most LEN bytes to the socket, but there is no guarantee, that all LEN bytes are written. It will return the number of bytes written. syswrite will write all the data within a single SSL frame, which means, that no more than 16.384 bytes, which is the maximum size of an SSL frame, can be written at once.

For non-blocking sockets SSL specific behavior applies. Pease read the specific section in this documentation.

peek(BUF, LEN, [OFFSET])

This function has exactly the same syntax as **sysread**, and performs nearly the same task but will not advance the read position so that successive calls to *peek()* with the same arguments will return the same results. This function requires OpenSSL 0.9.6a or later to work.

pending()

This function gives you the number of bytes available without reading from the underlying socket object. This function is essential if you work with event loops, please see the section about polling SSL sockets.

get_fingerprint([algo])

This methods returns the fingerprint of the peer certificate in the form `algo$digest_hex`, where `algo` is the used algorithm, default 'sha256'.

get_fingerprint_bin([algo])

This methods returns the binary fingerprint of the peer certificate by using the algorithm `algo`, default 'sha256'.

get_cipher()

Returns the string form of the cipher that the IO::Socket::SSL object is using.

get_sslversion()

Returns the string representation of the SSL version of an established connection.

get_sslversion_int()

Returns the integer representation of the SSL version of an established connection.

dump_peer_certificate()
Returns a parsable string with select fields from the peer SSL certificate. This method directly returns the result of the *dump_peer_certificate()* method of Net::SSLeay.

peer_certificate($field;[$refresh])
If a peer certificate exists, this function can retrieve values from it. If no field is given the internal representation of certificate from Net::SSLeay is returned. If refresh is true it will not used a cached version, but check again in case the certificate of the connection has changed due to renegotiation.

The following fields can be queried:

authority (alias issuer)
The certificate authority which signed the certificate.

owner (alias subject)
The owner of the certificate.

commonName (alias cn) - only for
Net::SSLeay version >=1.30 8 The common name, usually the server name for SSL certificates.

subjectAltNames - only for
Net::SSLeay version >=1.33 8 Alternative names for the subject, usually different names for the same server, like example.org, example.com, *.example.com.

It returns a list of (typ,value) with typ GEN_DNS, GEN_IPADD etc (these constants are exported from IO::Socket::SSL). See Net::SSLeay::X509_get_subjectAltNames.

peer_certificates
This returns all the certificates send by the peer, e.g. first the peers own certificate and then the rest of the chain. You might use **CERT_asHash** from IO::Socket::SSL::Utils to inspect each of the certificates.

This function depends on a version of Net::SSLeay >= 1.58 .

get_servername
This gives the name requested by the client if Server Name Indication (SNI) was used.

verify_hostname($hostname,$scheme,$publicsuffix)
This verifies the given hostname against the peer certificate using the given scheme. Hostname is usually what you specify within the PeerAddr. See the `SSL_verifycn_publicsuffix` parameter for an explanation of suffix checking and for the possible values.

Verification of hostname against a certificate is different between various applications and RFCs. Some scheme allow wildcards for hostnames, some only in subjectAltNames, and even their different wildcard schemes are possible. RFC 6125 provides a good overview.

To ease the verification the following schemes are predefined (both protocol name and rfcXXXX name can be used):

rfc2818, xmpp (rfc3920), ftp (rfc4217)
Extended wildcards in subjectAltNames and common name are possible, e.g. *.example.org or even www*.example.org. The common name will be only checked if no DNS names are given in subjectAltNames.

http (alias www)
While name checking is defined in rfc2818 the current browsers usually accept also an IP address (w/o wildcards) within the common name as long as no subjectAltNames are defined. Thus this is rfc2818 extended with this feature.

smtp (rfc2595), imap, pop3, acap (rfc4642), netconf (rfc5538), syslog (rfc5425), snmp (rfc5953)
: Simple wildcards in subjectAltNames are possible, e.g. *.example.org matches www.example.org but not lala.www.example.org. If nothing from subjectAltNames match it checks against the common name, where wildcards are also allowed to match the full leftmost label.

ldap (rfc4513)
: Simple wildcards are allowed in subjectAltNames, but not in common name. Common name will be checked even if subjectAltNames exist.

sip (rfc5922)
: No wildcards are allowed and common name is checked even if subjectAltNames exist.

gist (rfc5971)
: Simple wildcards are allowed in subjectAltNames and common name, but common name will only be checked if their are no DNS names in subjectAltNames.

default
: This is a superset of all the rules and is automatically used if no scheme is given but a hostname (instead of IP) is known. Extended wildcards are allowed in subjectAltNames and common name and common name is checked always.

none
: No verification will be done. Actually is does not make any sense to call verify_hostname in this case.

The scheme can be given either by specifying the name for one of the above predefined schemes, or by using a hash which can have the following keys and values:

check_cn: 0|'always'|'when_only'
: Determines if the common name gets checked. If 'always' it will always be checked (like in ldap), if 'when_only' it will only be checked if no names are given in subjectAltNames (like in http), for any other values the common name will not be checked.

wildcards_in_alt: 0|'full_label'|'anywhere'
: Determines if and where wildcards in subjectAltNames are possible. If 'full_label' only cases like *.example.org will be possible (like in ldap), for 'anywhere' www*.example.org is possible too (like http), dangerous things like but www.*.org or even '*' will not be allowed. For compatibility with older versions 'leftmost' can be given instead of 'full_label'.

wildcards_in_cn: 0|'full_label'|'anywhere'
: Similar to wildcards_in_alt, but checks the common name. There is no predefined scheme which allows wildcards in common names.

ip_in_cn: 0|1|4|6
: Determines if an IP address is allowed in the common name (no wildcards are allowed). If set to 4 or 6 it only allows IPv4 or IPv6 addresses, any other true value allows both.

callback: &coderef
: If you give a subroutine for verification it will be called with the arguments ($hostname,$commonName,@subjectAltNames), where hostname is the name given for verification, commonName is the result from peer_certificate('cn') and subjectAltNames is the result from peer_certificate('subjectAltNames').

  All other arguments for the verification scheme will be ignored in this case.

next_proto_negotiated()

This method returns the name of negotiated protocol - e.g. 'http/1.1'. It works for both client and server side of SSL connection.

NPN support is available with Net::SSLeay 1.46+ and openssl-1.0.1+. To check support you might call `IO::Socket::SSL->can_npn()`.

alpn_selected()

Returns the protocol negotiated via ALPN as a string, e.g. 'http/1.1', 'http/2.0' or 'spdy/3.1'.

ALPN support is available with Net::SSLeay 1.56+ and openssl-1.0.2+. To check support, use `IO::Socket::SSL->can_alpn()`.

errstr()

Returns the last error (in string form) that occurred. If you do not have a real object to perform this method on, call *IO::Socket::SSL::errstr()* instead.

For read and write errors on non-blocking sockets, this method may include the string `SSL wants a read first!`or `SSL wants a write first!` meaning that the other side is expecting to read from or write to the socket and wants to be satisfied before you get to do anything. But with version 0.98 you are better comparing the global exported variable `$SSL_ERROR` against the exported symbols SSL_WANT_READ and SSL_WANT_WRITE.

opened()

This returns false if the socket could not be opened, 1 if the socket could be opened and the SSL handshake was successful done and -1 if the underlying IO::Handle is open, but the SSL handshake failed.

IO::Socket::SSL->start_SSL($socket, ...)

This will convert a glob reference or a socket that you provide to an IO::Socket::SSL object. You may also pass parameters to specify context or connection options as with a call to *new()*. If you are using this function on an *accept()*ed socket, you must set the parameter "SSL_server" to 1, i.e. IO::Socket::SSL->start_SSL($socket, SSL_server => 1). If you have a class that inherits from IO::Socket::SSL and you want the `$socket` to be blessed into your own class instead, use MyClass->start_SSL($socket) to achieve the desired effect.

Note that if *start_SSL()* fails in SSL negotiation, `$socket` will remain blessed in its original class. For non-blocking sockets you better just upgrade the socket to IO::Socket::SSL and call accept_SSL or connect_SSL and the upgraded object. To just upgrade the socket set **SSL_startHandshake** explicitly to 0. If you call start_SSL w/o this parameter it will revert to blocking behavior for accept_SSL and connect_SSL.

If given the parameter "Timeout" it will stop if after the timeout no SSL connection was established. This parameter is only used for blocking sockets, if it is not given the default Timeout from the underlying IO::Socket will be used.

stop_SSL(...)

This is the opposite of *start_SSL()*, *connect_SSL()* and *accept_SSL()*, e.g. it will shutdown the SSL connection and return to the class before *start_SSL()*. It gets the same arguments as *close()*, in fact *close()* calls *stop_SSL()* (but without downgrading the class).

Will return true if it succeeded and undef if failed. This might be the case for non-blocking sockets. In this case $! is set to EWOULDBLOCK and the ssl error to SSL_WANT_READ or SSL_WANT_WRITE. In this case the call should be retried again with the same arguments once the socket is ready.

For calling from `stop_SSL` `SSL_fast_shutdown` default to false, e.g. it waits for the close_notify of the peer. This is necesarry in case you want to downgrade the socket and continue to use it as a plain socket.

After stop_SSL the socket can again be used to exchange plain data.

connect_SSL, **accept_SSL**

These functions should be used to do the relevant handshake, if the socket got created with `new` or upgraded with `start_SSL` and `SSL_startHandshake` was set to false. They will return undef until the handshake succeeded or an error got thrown. As long as the function returns undef and $! is set to EWOULDBLOCK one could retry the call after the socket got readable (SSL_WANT_READ) or writeable (SSL_WANT_WRITE).

ocsp_resolver

This will create an OCSP resolver object, which can be used to create OCSP requests for the certificates of the SSL connection. Which certificates are verified depends on the setting of `SSL_ocsp_mode`: by default only the leaf certificate will be checked, but with SSL_OCSP_FULL_CHAIN all chain certificates will be checked.

Because to create an OCSP request the certificate and its issuer certificate need to be known it is not possible to check certificates when the trust chain is incomplete or if the certificate is self-signed.

The OCSP resolver gets created by calling `$ssl`-ocsp_resolver> and provides the following methods:

hard_error

This returns the hard error when checking the OCSP response. Hard errors are certificate revocations. With the `SSL_ocsp_mode` of SSL_OCSP_FAIL_HARD any soft error (e.g. failures to get signed information about the certificates) will be considered a hard error too.

The OCSP resolving will stop on the first hard error.

The method will return undef as long as no hard errors occured and still requests to be resolved. If all requests got resolved and no hard errors occured the method will return `''`.

soft_error

This returns the soft error(s) which occured when asking the OCSP responders.

requests

This will return a hash consisting of `(url,request)`-tuples, e.g. which contain the OCSP request string and the URL where it should be sent too. The usual way to send such a request is as HTTP POST request with an content-type of `application/ocsp-request` or as a GET request with the base64 and url-encoded request is added to the path of the URL.

After you've handled all these requests and added the response with `add_response` you should better call this method again to make sure, that no more requests are outstanding. IO::Socket::SSL will combine multiple OCSP requests for the same server inside a single request, but some server don't give an response to all these requests, so that one has to ask again with the remaining requests.

add_response($uri,$response)

This method takes the HTTP body of the response which got received when sending the OCSP request to `$uri`. If no response was received or an error occured one should either retry or consider `$response` as empty which will trigger a soft error.

The method returns the current value of `hard_error`, e.g. a defined value when no more requests need to be done.

resolve_blocking(%args)

This combines `requests` and `add_response` which HTTP::Tiny to do all necessary requests in a blocking way. `%args` will be given to HTTP::Tiny so that you can put proxy settings etc here. HTTP::Tiny will be called with `verify_SSL` of false, because the OCSP responses have their own signatures so no extra SSL verification is needed.

If you don't want to use blocking requests you need to roll your own user agent with `requests` and `add_response`.

IO::Socket::SSL->new_from_fd($fd, [mode], `%sslargs`)

This will convert a socket identified via a file descriptor into an SSL socket. Note that the argument list does not include a ''MODE'' argument; if you supply one, it will be thoughtfully ignored (for compatibility with IO::Socket::INET). Instead, a mode of '+<' is assumed, and the file descriptor passed must be able to handle such I/O because the initial SSL handshake requires bidirectional communication.

Internally the given `$fd` will be upgraded to a socket object using the `new_from_fd` method of the super class (IO::Socket::INET or similar) and then `start_SSL` will be called using the given `%sslargs`. If`$fd` is already an IO::Socket object you should better call `start_SSL` directly.

IO::Socket::SSL::default_ca([path|dir| SSL_ca_file =..., SSL_ca_path => ...])>

Determines or sets the default CA path. If existing path or dir or a hash is given it will set the default CA path to this value and never try to detect it automatically. If`undef` is giv en it will forget any stored defaults and continue with detection of system defaults. If no arguments are given it will start detection of system defaults, unless it has already stored user-set or previously detected values.

The detection of system defaults works similar to OpenSSL, e.g. it will check the directory specified in environment variable SSL_CERT_DIR or the path OPENSSLDIR/certs (SSLCERTS: on VMS) and the file specified in environment variable SSL_CERT_FILE or the path OPENSSLDIR/cert.pem (SSLCERTS:cert.pem on VMS). Contrary to OpenSSL it will check if the SSL_ca_path contains PEM files with the hash as file name and if the SSL_ca_file looks like PEM. If no usable system default can be found it will try to load and use Mozilla::CA and if not available give up detection. The result of the detection will be saved to speed up future calls.

The function returns the saved default CA as hash with SSL_ca_file and SSL_ca_path.

IO::Socket::SSL::set_default_context(...)

You may use this to make IO::Socket::SSL automatically re-use a given context (unless specifically overridden in a call to *new()*). It accepts one argument, which should be either an IO::Socket::SSL object or an IO::Socket::SSL::SSL_Context object. See the SSL_reuse_ctx option of *new()* for more details. Note that this sets the default context globally, so use with caution (esp. in mod_perl scripts).

IO::Socket::SSL::set_default_session_cache(...)

You may use this to make IO::Socket::SSL automatically re-use a given session cache (unless specifically overridden in a call to *new()*). It accepts one argument, which should be an IO::Socket::SSL::Session_Cache object or similar (e.g something which implements get_session and add_session like IO::Socket::SSL::Session_Cache does). See the SSL_session_cache option of *new()* for more details. Note that this sets the default cache globally, so use with caution.

IO::Socket::SSL::set_defaults(%args)

With this function one can set defaults for all SSL_* parameter used for creation of the context, like the SSL_verify* parameter. Any SSL_* parameter can be given or the following short versions:

mode - SSL_verify_mode
callback - SSL_verify_callback
scheme - SSL_verifycn_scheme
name - SSL_verifycn_name

IO::Socket::SSL::set_client_defaults(%args)

Similar to `set_defaults`, but only sets the defaults for client mode.

IO::Socket::SSL::set_server_defaults(%args)

Similar to `set_defaults`, but only sets the defaults for server mode.

IO::Socket::SSL::set_args_filter_hack(&code|'use_defaults')

Sometimes one has to use code which uses unwanted or invalid arguments for SSL, typically disabling SSL verification or setting wrong ciphers or SSL versions. With this hack it is possible to override these settings and restore sanity. Example:

```
IO::Socket::SSL::set_args_filter_hack( sub {
my ($is_server,$args) = @_;
if ( ! $is_server ) {
# client settings - enable verification with default CA
# and fallback hostname verification etc
delete @{$args}{qw(
SSL_verify_mode
SSL_ca_file
SSL_ca_path
SSL_verifycn_scheme
SSL_version
)};
# and add some fingerprints for known certs which are signed by
# unknown CAs or are self-signed
$args->{SSL_fingerprint} = ...
}
});
```

With the short setting `set_args_filter_hack('use_defaults')` it will prefer the default settings in all cases. These default settings can be modified with `set_defaults`, `set_client_defaults` and `set_server_defaults`.

The following methods are unsupported (not to mention futile!) and IO::Socket::SSL will emit a large *CROAK()* if you are silly enough to use them:

truncate
stat
ungetc
setbuf
setvbuf
fdopen
send/recv

Note that *send()* and *recv()* cannot be reliably trapped by a tied filehandle (such as that used by IO::Socket::SSL) and so may send unencrypted data over the socket. Object-oriented calls to these functions will fail, telling you to use the print/printf/syswrite and read/sysread families instead.

DEPRECATIONS

The following functions are deprecated and are only retained for compatibility:

context_init()
use the SSL_reuse_ctx option if you want to re-use a context

socketToSSL() and *socket_to_SSL()*
use IO::Socket::SSL->*start_SSL()* instead

kill_socket()
use *close()* instead

get_peer_certificate()
use the *peer_certificate()* function instead. Used to return X509_Certificate with methods subject_name and issuer_name. Now simply returns `$self` which has these methods (although deprecated).

issuer_name()
use peer_certificate('issuer') instead

subject_name()
use peer_certificate('subject') instead

EXAMPLES

See the 'example' directory, the tests in 't' and also the tools in 'util'.

BUGS

If you use IO::Socket::SSL together with threads you should load it (e.g. use or require) inside the main thread before creating any other threads which use it. This way it is much faster because it will be initialized only once. Also there are reports that it might crash the other way.

Creating an IO::Socket::SSL object in one thread and closing it in another thread will not work.

IO::Socket::SSL does not work together with Storable::fd_retrieve/fd_store. See BUGS file for more information and how to work around the problem.

Non-blocking and timeouts (which are based on non-blocking) are not supported on Win32, because the underlying IO::Socket::INET does not support non-blocking on this platform.

If you have a server and it looks like you have a memory leak you might check the size of your session cache. Default for Net::SSLeay seems to be 20480, see the example for SSL_create_ctx_callback for how to limit it.

SEE ALSO

IO::Socket::INET, IO::Socket::INET6, IO::Socket::IP, Net::SSLeay.

THANKS

Many thanks to all who added patches or reported bugs or helped IO::Socket::SSL another way. Please keep reporting bugs and help with patches, even if they just fix the documentation.

Special thanks to the team of Net::SSLeay for the good cooperation.

AUTHORS

Steffen Ullrich, <sullr at cpan.org> is the current maintainer.

Peter Behroozi, <behrooz at fas.harvard.edu> (Note the lack of an "i" at the end of "behrooz")

Marko Asplund, <marko.asplund at kronodoc.fi>, was the original author of IO::Socket::SSL.

Patches incorporated from various people, see file Changes.

COPYRIGHT