

# OpenSSL: Secure Communication

Version 8.0

February 4, 2021

```
(require openssl)      package: base
```

The `openssl` library provides glue for the OpenSSL library with the Racket port system. It provides functions nearly identically to the standard TCP subsystem in Racket, plus a generic `ports->ssl-ports` interface.

To use this library, you will need OpenSSL installed on your machine, but on many platforms the necessary libraries are included with the OS or with the Racket distribution. In particular:

- For Windows, `openssl` depends on "libeay32.dll" and "ssleay32.dll", which are included in the Racket distribution for Windows.
- For Mac OS, `openssl` depends on "libssl.dylib" and "libcrypto.dylib". Although those libraries are provided by Mac OS 10.2 and later, their use is deprecated, so the Racket distribution for Mac OS includes newer versions.
- For Unix, `openssl` depends on "libssl.so" and "libcrypto.so", which must be installed in a standard library location or in a directory listed by `LD_LIBRARY_PATH`. These libraries are included in many OS distributions.

```
| ssl-available? : boolean?
```

A boolean value that reports whether the system OpenSSL library was successfully loaded. Calling `ssl-connect`, etc. when this value is `#f` (library not loaded) will raise an exception.

```
| ssl-load-fail-reason : (or/c #f string?)
```

Either `#f` (when `ssl-available?` is `#t`) or an error string (when `ssl-available?` is `#f`).

# 1 TCP-like Client Procedures

Use `ssl-connect` or `ssl-connect/enable-break` to create an SSL connection over TCP. To create a secure connection, supply the result of `ssl-secure-client-context` or create a client context with `ssl-make-client-context` and configure it using the functions described in §4 “Context Procedures”.

```
(ssl-connect hostname
             port-no
             [client-protocol]) → input-port? output-port?
hostname : string?
port-no  : (integer-in 1 65535)
client-protocol : (or/c ssl-client-context?
                    'secure
                    'auto
                    'sslv2-or-v3 'sslv2 'sslv3 'tls 'tls11 'tls12)
                    = 'auto
```

Connect to the host given by `hostname`, on the port given by `port-no`. This connection will be encrypted using SSL. The return values are as for `tcp-connect`: an input port and an output port.

The default `'auto` protocol is **insecure**. Use `'secure` for a secure connection. See `ssl-secure-client-context` for details.

The optional `client-protocol` argument determines which encryption protocol is used, whether the server’s certificate is checked, etc. The argument can be either a client context created by `ssl-make-client-context` a symbol specifying the protocol to use; see `ssl-make-client-context` for further details, including the meanings of the protocol symbols.

Closing the resulting output port does not send a shutdown message to the server. See also `ports->ssl-ports`.

If hostname verification is enabled (see `ssl-set-verify-hostname!`), the peer’s certificate is checked against `hostname`.

Changed in version 6.3.0.12 of package `base`: Added `'secure` for `client-protocol`.

```
(ssl-connect/enable-break hostname
                        port-no
                        [client-protocol])
→ input-port? output-port?
hostname : string?
port-no  : (integer-in 1 65535)
```

```

client-protocol : (or/c ssl-client-context?
                  'secure 'auto
                  'ssl2-or-v3 'ssl2 'ssl3 'tls 'tls11 'tls12)
                  = 'auto

```

Like `ssl-connect`, but breaking is enabled while trying to connect.

```
(ssl-secure-client-context) → ssl-client-context?
```

Returns a client context that verifies certificates using the default verification sources from (`ssl-default-verify-sources`), verifies hostnames, and avoids using weak ciphers. The result is essentially equivalent to the following:

```

(let ([ctx (ssl-make-client-context 'auto)])
  ; Load default verification sources (root certificates)
  (ssl-load-default-verify-sources! ctx)
  ; Require certificate verification
  (ssl-set-verify! ctx #t)
  ; Require hostname verification
  (ssl-set-verify-hostname! ctx #t)
  ; No weak cipher suites
  (ssl-set-ciphers! ctx "DEFAULT:!aNULL:!eNULL:!LOW:!EXPORT:!SSLv2")
  ; Seal context so further changes cannot weaken it
  (ssl-seal-context! ctx)
  ctx)

```

The context is cached, so different calls to `ssl-secure-client-context` return the same context unless (`ssl-default-verify-sources`) has changed.

Note that (`ssl-secure-client-context`) returns a sealed context, so it is not possible to add a private key and certificate chain to it. If client credentials are required, use `ssl-make-client-context` instead.

```

(ssl-make-client-context
 [protocol
  #:private-key private-key
  #:certificate-chain certificate-chain])
→ ssl-client-context?
protocol : (or/c 'secure 'auto
                'ssl2-or-v3 'ssl2 'ssl3 'tls 'tls11 'tls12)
          = 'auto
private-key : (or/c (list/c 'pem path-string?) = #f
                   (list/c 'der path-string?)
                   #f)
certificate-chain : (or/c path-string? #f) = #f

```

Creates a context to be supplied to `ssl-connect`. The context is **insecure** unless `'secure` is supplied or additional steps are taken; see `ssl-secure-client-context` for details.

The client context identifies a communication protocol (as selected by `protocol`), and also holds certificate information (i.e., the client's identity, its trusted certificate authorities, etc.). See the section §4 “Context Procedures” below for more information on certificates.

The `protocol` should be one of the following:

- `'secure` : Equivalent to (`ssl-secure-client-context`).
- `'auto` : Automatically negotiates the protocol version from those that this library considers sufficiently secure—currently TLS versions 1.0 and higher, but subject to change.
- `'tls12` : Only TLS protocol version 1.2.

Note that later TLS versions are supported, but there is no corresponding `protocol` symbol; using `'secure` is best and forward-compatible. The following `protocol` symbols are deprecated but still supported:

- `'sslv2-or-v3` : Alias for `'auto`. Note that despite the name, neither SSL 2.0 nor 3.0 are considered sufficiently secure, so this `protocol` no longer allows either of them.
- `'sslv2` : SSL protocol version 2.0. **Insecure.** Note that SSL 2.0 support has been removed from many platforms.
- `'sslv3` : SSL protocol version 3.0. **Insecure.**
- `'tls` : Only TLS protocol version 1.0.
- `'tls11` : Only TLS protocol version 1.1.

Not all protocol versions are supported by all servers. The `'secure` and `'auto` options offer broad compatibility at a reasonable level of security. Note that the security of connections depends on more than the protocol version; see `ssl-secure-client-context` for details. See also `supported-client-protocols` and `supported-server-protocols`.

If `private-key` and `certificate-chain` are provided, they are loaded into the context using `ssl-load-private-key!` and `ssl-load-certificate-chain!`, respectively. Client credentials are rarely used with HTTPS, but they are occasionally used in other kind of servers.

Changed in version 6.1 of package `base`: Added `'tls11` and `'tls12`.

Changed in version 6.1.1.3: Default to new `'auto` and disabled SSL 2.0 and 3.0 by default.

Changed in version 6.3.0.12: Added `'secure`.

Changed in version 7.3.0.10: Added `#:private-key` and `#:certificate-chain` arguments.

```
(supported-client-protocols)
→ (listof (or/c 'secure 'auto
                'ssl2-or-v3 'ssl2 'ssl3 'tls 'tls11 'tls12))
```

Returns a list of symbols representing protocols that are supported for clients on the current platform.

Changed in version 6.3.0.12 of package `base`: Added `'secure`.

```
(ssl-client-context? v) → boolean?
v : any/c
```

Returns `#t` if `v` is a value produced by `ssl-make-client-context`, `#f` otherwise.

Added in version 6.0.1.3 of package `base`.

```
(ssl-max-client-protocol)
→ (or/c 'ssl2 'ssl3 'tls 'tls11 'tls12 #f)
```

Returns the most recent SSL/TLS protocol version supported by the current platform for client connections.

Added in version 6.1.1.3 of package `base`.

## 2 TCP-like Server Procedures

```
(ssl-listen port-no
           [queue-k
            reuse?
            hostname-or-#f
            server-protocol]) → ssl-listener?
port-no : listen-port-number?
queue-k : exact-nonnegative-integer? = 5
reuse?  : any/c = #f
hostname-or-#f : (or/c string? #f) = #f
server-protocol : (or/c ssl-server-context?
                  'secure 'auto
                  'sslv2-or-v3 'sslv2 'sslv3 'tls 'tls11 'tls12)
                  = 'auto
```

Like `tcp-listen`, but the result is an SSL listener. The extra optional `server-protocol` is as for `ssl-connect`, except that a context must be a server context instead of a client context, and `'secure` is simply an alias for `'auto`.

Call `ssl-load-certificate-chain!` and `ssl-load-private-key!` to avoid a *no shared cipher* error on accepting connections. The file `"test.pem"` in the `"openssl"` collection is a suitable argument for both calls when testing. Since `"test.pem"` is public, however, such a test configuration obviously provides no security.

An SSL listener is a synchronizable value (see `sync`). It is ready—with itself as its value—when the underlying TCP listener is ready. At that point, however, accepting a connection with `ssl-accept` may not complete immediately, because further communication is needed to establish the connection.

Changed in version 6.3.0.12 of package `base`: Added `'secure`.

```
(ssl-close listener) → void?
listener : ssl-listener?
(ssl-listener? v) → boolean?
v : any/c
```

Analogous to `tcp-close` and `tcp-listener?`.

```
(ssl-accept listener) → input-port? output-port?
listener : ssl-listener?
(ssl-accept/enable-break listener) → input-port? output-port?
listener : ssl-listener?
```

Analogous to `tcp-accept`.

Closing the resulting output port does not send a shutdown message to the client. See also [ports->ssl-ports](#).

See also [ssl-connect](#) about the limitations of reading and writing to an SSL connection (i.e., one direction at a time).

The [ssl-accept/enable-break](#) procedure is analogous to [tcp-accept/enable-break](#).

```
(ssl-abandon-port p) → void?  
p : ssl-port?
```

Analogous to [tcp-abandon-port](#).

```
(ssl-addresses p [port-numbers?])  
→ (or/c (values string? string?)  
        (values string? port-number? string? listen-port-number?))  
p : (or/c ssl-port? ssl-listener?)  
port-numbers? : any/c = #f
```

Analogous to [tcp-addresses](#).

```
(ssl-port? v) → boolean?  
v : any/c
```

Returns #t if *v* is an SSL port produced by [ssl-connect](#), [ssl-connect/enable-break](#), [ssl-accept](#), [ssl-accept/enable-break](#), or [ports->ssl-ports](#).

```
(ssl-make-server-context  
 [protocol  
  #:private-key private-key  
  #:certificate-chain certificate-chain])  
→ ssl-server-context?  
protocol : (or/c 'secure 'auto  
                'sslv2-or-v3 'sslv2 'sslv3 'tls 'tls11 'tls12)  
          = 'auto  
private-key : (or/c (list/c 'pem path-string?) = #f  
                   (list/c 'der path-string?)  
                   #f)  
certificate-chain : (or/c path-string? #f) = #f
```

Like [ssl-make-client-context](#), but creates a server context. For a server context, the 'secure protocol is the same as 'auto.

If *private-key* and *certificate-chain* are provided, they are loaded into the context using [ssl-load-private-key!](#) and [ssl-load-certificate-chain!](#), respectively.

Changed in version 6.3.0.12 of package `base`: Added `'secure`.

Changed in version 7.3.0.10: Added `#:private-key` and `#:certificate-chain` arguments.

```
(ssl-server-context? v) → boolean?  
v : any/c
```

Returns `#t` if `v` is a value produced by `ssl-make-server-context`, `#f` otherwise.

```
(supported-server-protocols)  
→ (listof (or/c 'secure 'auto  
               'sslv2-or-v3 'sslv2 'sslv3 'tls 'tls11 'tls12))
```

Returns a list of symbols representing protocols that are supported for servers on the current platform.

Added in version 6.0.1.3 of package `base`.

Changed in version 6.3.0.12: Added `'secure`.

```
(ssl-max-server-protocol)  
→ (or/c 'sslv2 'sslv3 'tls 'tls11 'tls12 #f)
```

Returns the most recent SSL/TLS protocol version supported by the current platform for server connections.

Added in version 6.1.1.3 of package `base`.



### 3 SSL-wrapper Interface

```
(ports->ssl-ports input-port
                  output-port
                  [#:mode mode
                  #:context context
                  #:encrypt protocol
                  #:close-original? close-original?
                  #:shutdown-on-close? shutdown-on-close?
                  #:error/ssl error
                  #:hostname hostname])
→ input-port? output-port?
input-port : input-port?
output-port : output-port?
mode : (or/c 'connect 'accept) = 'accept
context : (or/c ssl-client-context? ssl-server-context?)
          = ((if (eq? mode 'accept)
                  ssl-make-server-context
                  ssl-make-client-context)
             protocol)
protocol : (or/c 'secure 'auto
                'sslv2-or-v3 'sslv2 'sslv3 'tls 'tls11 'tls12)
          = 'auto
close-original? : boolean? = #f
shutdown-on-close? : boolean? = #f
error : procedure? = error
hostname : (or/c string? #f) = #f
```

Returns two values—an input port and an output port—that implement the SSL protocol over the given input and output port. (The given ports should be connected to another process that runs the SSL protocol.)

The `mode` argument can be `'connect` or `'accept`. The mode determines how the SSL protocol is initialized over the ports, either as a client or as a server. As with `ssl-listen`, in `'accept` mode, supply a `context` that has been initialized with `ssl-load-certificate-chain!` and `ssl-load-private-key!` to avoid a *no shared cipher* error.

The `context` argument should be a client context for `'connect` mode or a server context for `'accept` mode. If it is not supplied, a context is created using the protocol specified by a `protocol` argument.

If the `protocol` argument is not supplied, it defaults to `'auto`. See `ssl-make-client-context` for further details (including all options and the meanings of the protocol symbols). This argument is ignored if a `context` argument is supplied.

If `close-original?` is true, then when both SSL ports are closed, the given input and

output ports are automatically closed.

If `shutdown-on-close?` is true, then when the output SSL port is closed, it sends a shutdown message to the other end of the SSL connection. When shutdown is enabled, closing the output port can fail if the given output port becomes unwritable (e.g., because the other end of the given port has been closed by another process).

The `error` argument is an error procedure to use for raising communication errors. The default is `error`, which raises `exn:fail`; in contrast, `ssl-accept` and `ssl-connect` use an error function that raises `exn:fail:network`.

See also `ssl-connect` about the limitations of reading and writing to an SSL connection (i.e., one direction at a time).

If hostname verification is enabled (see `ssl-set-verify-hostname!`), the peer's certificate is checked against `hostname`.

## 4 Context Procedures

```
(ssl-load-verify-source! context
                          src
                          [#:try? try?]) → void?
context : (or/c ssl-client-context? ssl-server-context?)
src : (or/c path-string?
      (list/c 'directory path-string?)
      (list/c 'win32-store string?)
      (list/c 'macosx-keychain path-string?))
try? : any/c = #f
```

Loads verification sources from *src* into *context*. Currently, only certificates are loaded; the certificates are used to verify the certificates of a connection peer. Call this procedure multiple times to load multiple sets of trusted certificates.

The following kinds of verification sources are supported:

- If *src* is a path or string, it is treated as a PEM file containing root certificates. The file is loaded immediately.
- If *src* is `(list 'directory dir)`, then *dir* should contain PEM files with hashed symbolic links (see the `openssl c_rehash` utility). The directory contents are not loaded immediately; rather, they are searched only when a certificate needs verification.
- If *src* is `(list 'win32-store store)`, then the certificates from the store named *store* are loaded immediately. Only supported on Windows.
- If *src* is `(list 'macosx-keychain path)`, then the certificates from the keychain stored at *path* are loaded immediately. Only supported on Mac OS.

If *try?* is `#f` and loading *src* fails (for example, because the file or directory does not exist), then an exception is raised. If *try?* is a true value, then a load failure is ignored.

You can use the file `"test.pem"` of the `"openssl"` collection for testing purposes. Since `"test.pem"` is public, such a test configuration obviously provides no security.

```
(ssl-default-verify-sources)
→ (let ([source/c (or/c path-string?
                       (list/c 'directory path-string?)
                       (list/c 'win32-store string?)
                       (list/c 'macosx-keychain path-string?))])
      (listof source/c))
(ssl-default-verify-sources srcs) → void?
```

```

srcs : (let ([source/c (or/c path-string?
                            (list/c 'directory path-string?)
                            (list/c 'win32-store string?)
                            (list/c 'macosx-keychain path-string?))])
         (listof source/c))

```

Holds a list of verification sources, used by `ssl-load-default-verify-sources!`. The default sources depend on the platform:

- On Linux, the default sources are determined by the `SSL_CERT_FILE` and `SSL_CERT_DIR` environment variables, if the variables are set, or the system-wide default locations otherwise.
- On Mac OS, the default sources consist of the system keychain for root certificates: `'(macosx-keychain "/System/Library/Keychains/SystemRootCertificates.keychain")`.
- On Windows, the default sources consist of the system certificate store for root certificates: `'(win32-store "ROOT")`.

```

(ssl-load-default-verify-sources! context) → void?
context : (or/c ssl-client-context? ssl-server-context?)

```

Loads the default verification sources, as determined by `(ssl-default-verify-sources)`, into `context`. Load failures are ignored, since some default sources may refer to nonexistent paths.

```

(ssl-load-verify-root-certificates! context-or-listener
                                     pathname)
→ void?
context-or-listener : (or/c ssl-client-context? ssl-server-context?
                             ssl-listener?)
pathname : path-string?

```

Deprecated; like `ssl-load-verify-source!`, but only supports loading certificate files in PEM format.

```

(ssl-set-ciphers! context cipher-spec) → void?
context : (or/c ssl-client-context? ssl-server-context?)
cipher-spec : string?

```

Specifies the cipher suites that can be used in connections created with `context`. The meaning of `cipher-spec` is the same as for the `openssl ciphers` command.

```

(ssl-seal-context! context) → void?
context : (or/c ssl-client-context? ssl-server-context?)

```

Seals *context*, preventing further modifications. After a context is sealed, passing it to functions such as `ssl-set-verify!` and `ssl-load-verify-root-certificates!` results in an error.

```
(ssl-load-certificate-chain! context-or-listener
                             pathname) → void?
context-or-listener : (or/c ssl-client-context? ssl-server-context?
                             ssl-listener?)
pathname : path-string?
```

Loads a PEM-format certification chain file for connections to made with the given server context (created by `ssl-make-server-context`) or listener (created by `ssl-listen`). A certificate chain can also be loaded into a client context (created by `ssl-make-client-context`) when connecting to a server requiring client credentials, but that situation is uncommon.

This chain is used to identify the client or server when it connects or accepts connections. Loading a chain overwrites the old chain. Also call `ssl-load-private-key!` to load the certificate's corresponding key.

You can use the file "test.pem" of the "openssl" collection for testing purposes. Since "test.pem" is public, such a test configuration obviously provides no security.

```
(ssl-load-private-key! context-or-listener
                       pathname
                       [rsa?
                        asn1?]) → void?
context-or-listener : (or/c ssl-client-context? ssl-server-context?
                             ssl-listener?)
pathname : path-string?
rsa? : boolean? = #t
asn1? : boolean? = #f
```

Loads the first private key from *pathname* for the given context or listener. The key goes with the certificate that identifies the client or server. Like `ssl-load-certificate-chain!`, this procedure is usually used with server contexts or listeners, seldom with client contexts.

If *rsa?* is `#t` (the default), the first RSA key is read (i.e., non-RSA keys are skipped). If *asn1?* is `#t`, the file is parsed as ASN1 format instead of PEM.

You can use the file "test.pem" of the "openssl" collection for testing purposes. Since "test.pem" is public, such a test configuration obviously provides no security.

```
(ssl-load-suggested-certificate-authorities!
 context-or-listener
 pathname)
```

```

→ void?
context-or-listener : (or/c ssl-client-context? ssl-server-context?
                        ssl-listener?)
pathname : path-string?

```

Loads a PEM-format file containing certificates that are used by a server. The certificate list is sent to a client when the server requests a certificate as an indication of which certificates the server trusts.

Loading the suggested certificates does not imply trust, however; any certificate presented by the client will be checked using the trusted roots loaded by `ssl-load-verify-root-certificates!`.

You can use the file "test.pem" of the "openssl" collection for testing purposes where the peer identifies itself using "test.pem".

```

(ssl-server-context-enable-dhe! context
 [dh-param]) → void?
context : ssl-server-context?
dh-param : (or/c path-string? bytes?) = ssl-dh4096-param-bytes
(ssl-server-context-enable-ecdh! context
 [curve-name]) → void?
context : ssl-server-context?
curve-name : symbol? = 'secp521r1

```

Enables cipher suites that provide perfect forward secrecy via ephemeral Diffie-Hellman (DHE) or ephemeral elliptic-curve Diffie-Hellman (ECDHE) key exchange, respectively.

For DHE, the `dh-param` must be a path to a ".pem" file containing DH parameters or the content of such a file as a byte string.

For ECDHE, the `curve-name` must be one of the following symbols naming a standard elliptic curve: 'sect163k1, 'sect163r1, 'sect163r2, 'sect193r1, 'sect193r2, 'sect233k1, 'sect233r1, 'sect239k1, 'sect283k1, 'sect283r1, 'sect409k1, 'sect409r1, 'sect571k1, 'sect571r1, 'secp160k1, 'secp160r1, 'secp160r2, 'secp192k1, 'secp224k1, 'secp224r1, 'secp256k1, 'secp384r1, 'secp521r1, 'prime192v, 'prime256v.

Changed in version 7.7.0.4 of package base: Allow a byte string as the `dh-param` argument to `ssl-server-context-enable-dhe!`.

```

ssl-dh4096-param-bytes : bytes?

```

Byte string describing 4096-bit Diffie-Hellman parameters in ".pem" format.

Changed in version 7.7.0.4 of package base: Added as a replacement for `ssl-dh4096-param-path`.

```
(ssl-set-server-name-identification-callback! context
                                           callback) → void?
context : ssl-server-context?
callback : (string? . -> . (or/c ssl-server-context? #f))
```

Provides an SSL server context with a procedure it can use for switching to alternative contexts on a per-connection basis. The procedure is given the hostname the client was attempting to connect to, to use as the basis for its decision.

The client sends this information via the TLS Server Name Identification extension, which was created to allow virtual hosting for secure servers.

The suggested use it to prepare the appropriate server contexts, define a single callback which can dispatch between them, and then apply it to all the contexts before sealing them. A minimal example:

```
(define ctx-a (ssl-make-server-context 'tls))
(define ctx-b (ssl-make-server-context 'tls))
...
(ssl-load-certificate-chain! ctx-a "cert-a.pem")
(ssl-load-certificate-chain! ctx-b "cert-b.pem")
...
(ssl-load-private-key! ctx-a "key-a.pem")
(ssl-load-private-key! ctx-b "key-b.pem")
...
(define (callback hostname)
  (cond [(equal? hostname "a") ctx-a]
        [(equal? hostname "b") ctx-b]
        ...
        [else #f]))
(ssl-set-server-name-identification-callback! ctx-a callback)
(ssl-set-server-name-identification-callback! ctx-b callback)
...
(ssl-seal-context! ctx-a)
(ssl-seal-context! ctx-b)
...
(ssl-listen 443 5 #t #f ctx-a)
```

If the callback returns `#f`, the connection attempt will continue, using the original server context.

## 5 Peer Verification

```
(ssl-set-verify! clp on?) → void?  
  clp : (or/c ssl-client-context? ssl-server-context?  
         ssl-listener? ssl-port?)  
  on? : any/c
```

Requires certificate verification on the peer SSL connection when `on?` is `#t`. If `clp` is an SSL port, then the connection is immediately renegotiated, and an exception is raised immediately if certificate verification fails. If `clp` is a context or listener, certification verification happens on each subsequent connection using the context or listener.

Enabling verification also requires, at a minimum, designating trusted certificate authorities with `ssl-load-verify-source!`.

Verifying the certificate is not sufficient to prevent attacks by active adversaries, such as man-in-the-middle attacks. See also `ssl-set-verify-hostname!`.

```
(ssl-try-verify! clp on?) → void?  
  clp : (or/c ssl-client-context? ssl-server-context?  
         ssl-listener? ssl-port?)  
  on? : any/c
```

Like `ssl-set-verify!`, but when peer certificate verification fails, then connection continues to work. Use `ssl-peer-verified?` to determine whether verification succeeded.

```
(ssl-peer-verified? p) → boolean?  
  p : ssl-port?
```

Returns `#t` if the peer of SSL port `p` has presented a valid and verified certificate, `#f` otherwise.

```
(ssl-set-verify-hostname! ctx on?) → void?  
  ctx : (or/c ssl-client-context? ssl-server-context?)  
  on? : any/c
```

Requires hostname verification of SSL peers of connections made using `ctx` when `on?` is `#t`. When hostname verification is enabled, the hostname associated with a connection (see `ssl-connect` or `ports->ssl-ports`) is checked against the hostnames listed in the peer's certificate. If the peer certificate does not contain an entry matching the hostname, or if the peer does not present a certificate, the connection is rejected and an exception is raised.

Hostname verification does not imply certificate verification. To verify the certificate itself, also call `ssl-set-verify!`.



```
(ssl-peer-certificate-hostnames p) → (listof string?)  
p : ssl-port?
```

Returns the list of hostnames for which the certificate of *p*'s peer is valid according to RFC 2818. If the peer has not presented a certificate, '()' is returned.

The result list may contain both hostnames such as "www.racket-lang.org" and host-name patterns such as "\*.racket-lang.org".

```
(ssl-peer-check-hostname p hostname) → boolean?  
p : ssl-port?  
hostname : string?
```

Returns #t if the peer certificate of *p* is valid for *hostname* according to RFC 2818.

```
(ssl-peer-subject-name p) → (or/c bytes? #f)  
p : ssl-port?
```

If `ssl-peer-verified?` would return #t for *p*, the result is a byte string for the subject field of the certificate presented by the SSL port's peer, otherwise the result is #f.

Use `ssl-peer-check-hostname` or `ssl-peer-certificate-hostnames` instead to check the validity of an SSL connection.

```
(ssl-peer-issuer-name p) → (or/c bytes? #f)  
p : ssl-port?
```

If `ssl-peer-verified?` would return #t for *p*, the result is a byte string for the issuer field of the certificate presented by the SSL port's peer, otherwise the result is #f.

```
(ssl-channel-binding p type) → bytes?  
p : ssl-port?  
type : (or/c 'tls-unique 'tls-server-end-point)
```

Returns channel binding information for the TLS connection of *p*. An authentication protocol run over TLS can incorporate information identifying the TLS connection ('tls-unique) or server certificate ('tls-server-end-point) into the authentication process, thus preventing the authentication steps from being replayed on another channel. Channel binding is described in general in RFC 5056; channel binding for TLS is described in RFC 5929.

If the channel binding cannot be retrieved (for example, if the connection is closed), an exception is raised.

Added in version 7.7.0.9 of package `base`.

## 6 SHA-1 Hashing

```
(require openssl/sha1)      package: base
```

The `openssl/sha1` library provides a Racket wrapper for the OpenSSL library's SHA-1 hashing functions. If the OpenSSL library cannot be opened, this library logs a warning and falls back to the implementation in `file/sha1`.

```
(sha1 in) → string?  
in : input-port?
```

Returns a 40-character string that represents the SHA-1 hash (in hexadecimal notation) of the content from `in`, consuming all of the input from `in` until an end-of-file.

The `sha1` function composes `bytes->hex-string` with `sha1-bytes`.

```
(sha1-bytes in) → bytes?  
in : input-port?
```

Returns a 20-byte byte string that represents the SHA-1 hash of the content from `in`, consuming all of the input from `in` until an end-of-file.

The `sha1-bytes` function from `racket/base` computes the same result and is only slightly slower.

```
(bytes->hex-string bstr) → string?  
bstr : bytes?
```

Converts the given byte string to a string representation, where each byte in `bstr` is converted to its two-digit hexadecimal representation in the resulting string.

```
(hex-string->bytes str) → bytes?  
str : string?
```

The inverse of `bytes->hex-string`.

## 7 MD5 Hashing

```
(require openssl/md5)      package: base
```

The `openssl/md5` library provides a Racket wrapper for the OpenSSL library's MD5 hashing functions. If the OpenSSL library cannot be opened, this library logs a warning and falls back to the implementation in `file/md5`.

Added in version 6.0.0.3 of package `base`.

```
(md5 in) → string?  
in : input-port?
```

Returns a 32-character string that represents the MD5 hash (in hexadecimal notation) of the content from `in`, consuming all of the input from `in` until an end-of-file.

The `md5` function composes `bytes->hex-string` with `md5-bytes`.

```
(md5-bytes in) → bytes?  
in : input-port?
```

Returns a 16-byte byte string that represents the MD5 hash of the content from `in`, consuming all of the input from `in` until an end-of-file.

## 8 The "libcrypto" Shared Library

```
(require openssl/libcrypto)      package: base
```

The `openssl/libcrypto` library provides a foreign-library value for the "libcrypto" shared library.

```
| libcrypto : (or/c #f ffi-lib?)
```

Returns a foreign-library value for "libcrypto", or `#f` if the library could not be found or loaded. The load attempt uses the versions specified by `openssl-lib-versions`.

```
| libcrypto-load-fail-reason : (or/c #f string?)
```

Either `#f` when `libcrypto` is non-`#f`, or a string when `libcrypto` is `#f`. In the latter case, the string provides an error message for the attempt to load "libcrypto".

```
| openssl-lib-versions : (listof string?)
```

A list of versions that are tried for loading "libcrypto". The list of version strings is suitable as a second argument to `ffi-lib`.

## 9 The "libssl" Shared Library

```
(require openssl/libssl)      package: base
```

The `openssl/libssl` library provides a foreign-library value for the "libssl" shared library.

```
| libssl : (or/c #f ffi-lib?)
```

Returns a foreign-library value for "libssl", or `#f` if the library could not be found or loaded. The load attempt uses the versions specified by `openssl-lib-versions`.

```
| libssl-load-fail-reason : (or/c #f string?)
```

Either `#f` when `libssl` is non-`#f`, or a string when `libssl` is `#f`. In the latter case, the string provides an error message for the attempt to load "libssl".