OpenSSL

Version 5.0.2

November 6, 2010

```
(require openssl)
```

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

- for Windows, the Racket distribution for Windows includes the necessary DLLs.
- for Mac OS X, version 10.2 and later provides the necessary OpenSSL libraries.
- for Unix, `"libssl.so"` and `"libcrypto.so"` are likely to be installed on your machine, already.

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

A boolean value which says 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 false/c string?)
```

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

1 TCP-like Client 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? symbol?)
                  = 'sslv2-or-v3
```

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 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`, or one of the following symbols: `'sslv2-or-v3` (the default), `'sslv2`, `'sslv3`, or `'tls`; 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`.

```
(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? symbol?)
                  = 'sslv2-or-v3
```

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

```
(ssl-make-client-context [protocol]) → ssl-client-context?
  protocol : symbol? = 'sslv2-or-v3
```

Creates a context to be supplied to `ssl-connect`. The 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* must be one of the following:

- `'sslv2-or-v3` : SSL protocol versions 2 or 3, as appropriate (this is the default)
- `'sslv2` : SSL protocol version 2
- `'sslv3` : SSL protocol version 3
- `'tls` : the TLS protocol version 1

By default, the context returned by `ssl-make-client-context` does not request verification of a server's certificate. Use `ssl-set-verify!` to enable such verification.

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

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

2 TCP-like Server Procedures

```
(ssl-listen  port-no
             queue-k
            [reuse?
             hostname-or-#f
             server-protocol]) → ssl-listener?
  port-no : (integer-in 1 65535)
  queue-k : exact-nonnegative-integer?
  reuse? : any/c = #f
  hostname-or-#f : (or/c string? false/c) = #f
  server-protocol :  (or/c ssl-server-context? symbol?)
                  = 'sslv2-or-v3
```

Like `tcp-listen`, but the result is an SSL listener (which is a synchronizable value; see `sync`). The extra optional *`server-protocol`* is as for `ssl-connect`, except that a context must be a server context instead of a client context.

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.

```
(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 in) → void?
  in : (and/c ssl-port? output-port?)
```

Analogous to `tcp-abandon-port`.

```
(ssl-addresses p [port-numbers?]) → void?
  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` of *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) → ssl-server-context?
  protocol : symbol?
```

Like `ssl-make-client-context`, but creates a server context.

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

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

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])
 → input-port? output-port?
  input-port : input-port?
  output-port : output-port?
  mode : symbol? = 'accept
  context :  (or/c ssl-client-context? ssl-server-context?)
          = ((if (eq? mode 'accept)
                 ssl-make-server-context
                 ssl-make-client-context)
             protocol)
  protocol : symbol? = 'sslv2-or-v3
  close-original? : boolean? = #f
  shutdown-on-close? : boolean? = #f
  error : procedure? = error
```

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 `'sslv2-or-v3`. 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).

4 Context Procedures

```
(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 context (created by `ssl-make-client-context` or `ssl-make-server-context`) or listener (created by `ssl-listen`).

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.

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-set-verify! context-or-listener
                 verify?)             → void?
  context-or-listener : (or/c ssl-client-context? ssl-server-context?
                               ssl-listener?)
  verify? : boolean?
```

Enables or disables verification of a connection peer's certificates. By default, verification is disabled.

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

```
(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?
```

Loads a PEM-format file containing trusted certificates that are used to verify the certificates of a connection peer. Call this procedure multiple times to load multiple sets of trusted certificates.

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"`.

5 SHA-1 Hashing

```
(require openssl/sha1)
```

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.

```
(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.

6 Implementation Notes

For Windows, `openssl` relies on `"libeay32.dll"` and `"ssleay32.dll"`, where the DLLs are located in the same place as `"libmzsch⟨vers⟩.dll"` (where ⟨*vers*⟩ is either `xxxxxxx` or a mangling of Racket's version number). The DLLs are distributed as part of Racket.

For Unix variants, `openssl` relies on `"libcryto.so"` and `"libssl.so"`, which must be installed in a standard library location, or in a directory listed by `LD_LIBRARY_PATH`.

For Mac OS X, `openssl` relies on `"libssl.dylib"` and `"libcryto.dylib"`, which are part of the OS distribution for Mac OS X 10.2 and later.