User Guide to `luaossl`, Comprehensive OpenSSL Module for Lua

William Ahern

October 26, 2017

Contents

1 Dependencies

1.1 Operating Systems

`luaossl` targets modern POSIX-conformant systems. A Windows port is feasible and patches welcome. Note however that the module employs the POSIX thread API, POSIX dlopen, and the non-POSIX dladdr interface to protect OpenSSL in threaded environments.

1.2 Libraries

1.2.1 Lua 5.1, 5.2, 5.3

`luaossl` targets Lua 5.1 and above.

1.2.2 OpenSSL

`luaossl` targets modern OpenSSL versions as installed on OS X, Linux, Solaris, OpenBSD, and similar platforms.

1.2.3 pthreads

Because it's not possible to detect threading use at runtime, or to *safely* and dynamically enable locking, this protection is builtin by default. At present the module only understands the POSIX threading API.

Linking Note that on some systems, such as NetBSD and FreeBSD, the loading application must be linked against pthreads (using -lpthread or -pthread). It is not enough for the `luaossl` module to pull in the dependency at load time. In particular, if using the stock Lua interpreter, it must have been linked against pthreads at build time. Add the appropriate linker flag to MYLIBS in lua-5.2.x/src/Makefile.

1.2.4 libdl

In multithreaded environments the module will initialize OpenSSL mutexes if they've not already been initialized. If the mutexes are initialized then the module must pin itself in memory to prevent unloading by the Lua garbage collector. The module first uses the non-standard but widely supported dladdr routine to derive the module's load path, and then increments the reference count to the module using dlopen. This is the safest and most portable method that I'm aware of.

1.3 GNU Make

The Makefile requires GNU Make, usually installed as gmake on platforms other than Linux or OS X. The actual `Makefile` proxies to `GNUmakefile`. As long as `gmake` is installed on non-GNU systems you can invoke your system's `make`.

2 Installation

All the C modules are built into a single core C library. The core routines are then wrapped and extended through Lua modules. Because there several extant versions of Lua often used in parallel on the same system, there are individual targets to build and install for each supported Lua version. The targets `all` and `install` will attempt to build and install both Lua 5.1 and 5.2 modules.

Note that building and installation and can accomplished in a single step by simply invoking one of the install targets with all the necessary variables defined.

2.1 Building

There is no separate `./configure` step. System introspection occurs during compile-time. However, the "`configure`" make target can be used to cache the build environment so one needn't continually use a long command-line invocation.

All the common GNU-style compiler variables are supported, including `CC`, `CPPFLAGS`, `CFLAGS`, `LDFLAGS`, and `SOFLAGS`. Note that you can specify the path to Lua 5.1, Lua 5.2, and Lua 5.3 include headers at the same time in CPPFLAGS; the build system will work things out to ensure the correct headers are loaded when compiling each version of the module.

2.1.1 Targets

`all`
: Build modules for Lua 5.1 and 5.2.

`all5.1`
: Build Lua 5.1 module.

`all5.2`
: Build Lua 5.2 module.

`all5.3`
: Build Lua 5.3 module.

2.2 Installing

All the common GNU-style installation path variables are supported, including `prefix`, `bindir`, `libdir`, `datadir`, `includedir`, and `DESTDIR`. These additional path variables are also allowed:

`lua51path`
: Install path for Lua 5.1 modules, e.g. `$(prefix)/share/lua/5.1`

`lua51cpath`
: Install path for Lua 5.1 C modules, e.g. `$(prefix)/lib/lua/5.1`

`lua52path`
: Install path for Lua 5.2 modules, e.g. `$(prefix)/share/lua/5.2`

`lua52cpath`
: Install path for Lua 5.2 C modules, e.g. `$(prefix)/lib/lua/5.2`

`lua53path`
: Install path for Lua 5.3 modules, e.g. `$(prefix)/share/lua/5.3`

`lua53cpath`
: Install path for Lua 5.3 C modules, e.g. `$(prefix)/lib/lua/5.3`

2.2.1 Targets

`install`
: Install modules for Lua 5.1 and 5.2.

`install5.1`
: Install Lua 5.1 module.

`install5.2`
: Install Lua 5.2 module.

`install5.3`
: Install Lua 5.3 module.

3 Usage

3.1 Modules

3.1.1 `openssl`

Binds various global interfaces, including version and build information.

`openssl[]`

Table of various compile-time constant values. See also `openssl.version`.

name	description
`SSLEAY_VERSION_NUMBER`	OpenSSL version as an integer.
`LIBRESSL_VERSION_NUMBER`	LibreSSL version as an integer.
`NO_[feature]`	If defined, signals that this installation of OpenSSL was compiled without [feature].

`openssl.version(`*info*`)`

Returns information about the run-time version of OpenSSL, as opposed to the compile-time version used to build the Lua module. If *info* is not specified, returns the version number as an integer. Otherwise, *info* may be one of the following constants.

name	description
`SSLEAY_VERSION`	OpenSSL version description as a string.
`SSLEAY_CFLAGS`	Description of compiler flags used to build OpenSSL as a string.
`SSLEAY_BUILT_ON`	Compilation date description as a string.
`SSLEAY_PLATFORM`	Platform description as a string.
`SSLEAY_DIR`	OpenSSL installation directory description as a string.

3.1.2 `openssl.bignum`

`openssl.bignum` binds OpenSSL's libcrypto bignum library. It supports all the standard arithmetic operations. Regular number operands in a mixed arithmetic expression are upgraded as-if `bignum.new` was used explicitly. The `__tostring` metamethod generates a decimal encoded representation.

`bignum.new(`*number*`)`

Wraps the sign and integral part of *number* as a bignum object, discarding any fractional part.

`bignum.interpose(`*name*`, `*function*`)`

Add or interpose a bignum class method. Returns the previous method, if any.

3.1.3 `openssl.pkey`

`openssl.pkey` binds OpenSSL's libcrypto public-private key library. The `__tostring` metamethod generates a PEM encoded representation of the public key—excluding the private key.

`pkey.new(`*string*`[, `*format*`])`

Initializes a new pkey object from the PEM- or DER-encoded key in *string*. *format* defaults to "*", which means to automatically test the input encoding. If *format* is explicitly "PEM" or "DER", then only that decoding format is used.

On failure throws an error.

`pkey.new{ ... }`

Generates a new pkey object according to the specified parameters.

field	type:default	description
.type	string:RSA	public key algorithm—"RSA", "DSA", "EC", "DH", or an internal OpenSSL identifier of a subclass of one of those basic types
.bits	number:1024	private key size
.exp	number:65537	RSA exponent
.generator	number:2	Diffie-Hellman generator
.dhparam	string	PEM encoded string with precomputed DH parameters
.curve	string:prime192v1	for elliptic curve keys, the OpenSSL string identifier of the curve

The DH parameters "dhparam" will be generated on the fly, "bits" wide. This is a slow process, and especially for larger sizes, you would precompute those; for example: "openssl dhparam -2 -out dh-2048.pem -outform PEM 2048". Using the field "dhparam" overrides the "bits" field.

`pkey.interpose(`*name*`, `*function*`)`

Add or interpose a pkey class method. Returns the previous method, if any.

`pkey:type()`

Returns the OpenSSL string identifier for the type of key.

`pkey:setPublicKey(`*string*`[, `*format*`])`

Set the public key component to that described by the PEM- or DER-encoded public key in *string*. *format* is as described in `openssl.pkey.new`—"PEM", "DER", or "*" (default).

`pkey:setPrivateKey(`*string*`[, `*format*`])`

Set the private key component to that described by the PEM encoded private key in *string*. *format* is as described in `openssl.pkey.new`.

`pkey:sign(`*digest*`)`

Sign data which has been consumed by the specified `openssl.digest` *digest*. Digests and keys are not all interchangeable.

Returns the signature as an opaque binary string[1] on success, and throws an error otherwise.

`pkey:verify(`*signature*`, `*digest*`)`

Verify the string *signature* as signing the document consumed by `openssl.digest` *digest*. See the :sign method for constraints on the format and type of the parameters.

Returns true on success, false for properly formatted but invalid signatures, and throws an error otherwise. Because the structure of the signature is opaque and not susceptible to sanity checking before passing to OpenSSL, an application should always be prepared for an error to be thrown when verifying untrusted signatures. OpenSSL, of course, should be able to handle all malformed inputs. But the module does not attempt to differentiate local system errors from errors triggered by malformed signatures because the set of such errors may change in the future.

`pkey:toPEM(`*which*`[, `*which*`])`

Returns the PEM encoded string representation(s) of the specified key component. *which* must be one of "public", "PublicKey", "private", or "PrivateKey". For the two argument form, returns two values.

3.1.4 `openssl.x509.name`

Binds the X.509 distinguished name OpenSSL ASN.1 object, used for representing certificate subject and issuer names.

`name.new()`

Returns an empty name object.

`name.interpose(`*name*`, `*function*`)`

Add or interpose a name class method. Returns the previous method, if any.

`name:add(`*type*`, `*value*`)`

Add a distinguished name component. *type* is the OpenSSL string identifier of the component type—short, long, or OID forms. *value* is the string value of the component. DN components are free-form, and are encoded raw.

[1] Elliptic curve signatures are two X.509 DER-encoded numbers, for example, while RSA signatures are encrypted DER structures.

`name:all()`

Returns a table array of the distinguished name components. Each element is a table with four fields:

field	description
.sn	short name identifier, if available
.ln	long name identifier, if available
.id	OID identifier
.blob	raw string value of the component

`name:__pairs()`

Returns a key-value iterator over the distinguished name components. The key is either the short, long, or OID identifier, with preference for the former.

3.1.5 `openssl.x509.altname`

Binds the X.509 alternative names (a.k.a "general names") OpenSSL ASN.1 object, used for representing certificate subject and issuer alternative names.

`altname.new()`

Returns an empty altname object.

`altname.interpose(`*name*`, `*function*`)`

Add or interpose an altname class method. Returns the previous method, if any.

`altname:add(`*type*`, `*value*`)`

Add an alternative name. *type* must specify one of the five basic types identified by "RFC822Name", "RFC822", "email", "UniformResourceIdentifier", "URI", "DNSName", "DNS", "IPAddress", "IP", or "DirName".

For all types except "DirName", *value* is a string acceptable to OpenSSL's sanity checks. For an IP address, *value* must be parseable by the system's `inet_pton` routine, as IP addresses are stored as raw 4- or 16-byte octets. "DirName" takes an `openssl.x509.name` object.

`name:__pairs()`

Returns a key-value iterator over the alternative names. The key is one of "email", "URI", "DNS", "IP", or "DirName". The value is the string representation of the name.

3.1.6 `openssl.x509.extension`

Binds the X.509 extension OpenSSL object.

`extension.new(`*name*`,` *value* `[,` *data*`])`

Returns a new X.509 extension. If *value* is the string "DER" or "critical,DER", then *data* is an ASN.1-encoded octet string. Otherwise, *name* and *value* are plain text strings in OpenSSL's arbitrary extension format; and if specified, *data* is either an OpenSSL configuration string defining any referenced identifiers in *value*, or a table with members:

field	type:default	description
.db	string:*nil*	OpenSSL configuration string
.issuer	`openssl.x509`:*nil*	issuer certificate
.subject	`openssl.x509`:*nil*	subject certificate
.request	`openssl.x509.csr`:*nil*	certificate signing request
.crl	`openssl.x509.crl`:*nil*	certificate revocation list
.flags	integer:0	a bitwise combination of flags

`extension.interpose(`*name*`,` *function*`)`

Add or interpose an extension class method. Returns the previous method, if any.

`extension:getID()`

Returns the ASN.1 OID as a plain text string.

`extension:getName()`

Returns a more human-readable name as a plain text string in the following order of preference: OpenSSL's short name, OpenSSL's long name, ASN.1 OID.

`extension:getShortName()`

Returns OpenSSL's short name as a plain text string if available.

`extension:getLongName()`

Returns OpenSSL's long name as a plain text string if available.

`extension:getData()`

Returns the extension value as an ASN.1-encoded octet string.

`extension:getCritical()`

Returns the extension critical flag as a boolean.

3.1.7 `openssl.x509`

Binds the X.509 certificate OpenSSL ASN.1 object.

`x509.new([`*string*`[, `*format*`]])`

Returns a new x509 object, optionally initialized to the PEM- or DER-encoded certificate specified by *string*. *format* is as described in `openssl.pkey.new`–"PEM", "DER", or "*" (default).

`x509.interpose(`*name*`, `*function*`)`

Add or interpose an x509 class method. Returns the previous method, if any.

`x509:getVersion()`

Returns the X.509 version of the certificate.

`x509:setVersion(`*number*`)`

Sets the X.509 version of the certificate.

`x509:getSerial()`

Returns the serial of the certificate as an `openssl.bignum`.

`x509:setSerial(`*number*`)`

Sets the serial of the certificate. *number* is a Lua or `openssl.bignum` number.

`x509:digest([`*type*`[, `*format*`]])`

Returns the cryptographic one-way message digest of the certificate. *type* is the OpenSSL string identifier of the hash type—e.g. "md5", "sha1" (default), "sha256", etc. *format* specifies the representation of the digest—"s" for an octet string, "x" for a hexadecimal string (default), and "n" for an `openssl.bignum` number.

`x509:getLifetime()`

Returns the certificate validity "Not Before" and "Not After" dates as two Unix timestamp numbers.

`x509:setLifetime([`*notbefore*`][, `*notafter*`])`

Sets the certificate validity dates. *notbefore* and *notafter* should be UNIX timestamps. A nil value leaves the particular date unchanged.

`x509:getIssuer()`

Returns the issuer distinguished name as an `x509.name` object.

`x509:setIssuer(`*name*`)`

Sets the issuer distinguished name.

`x509:getSubject()`

Returns the subject distinguished name as an `x509.name` object.

`x509:setSubject(`*name*`)`

Sets the subject distinguished name.

`x509:getIssuerAlt()`

Returns the issuer alternative names as an `x509.altname` object.

`x509:setIssuer(`*altname*`)`

Sets the issuer alternative names.

`x509:getSubjectAlt()`

Returns the subject alternative names as an `x509.name` object.

`x509:setSubjectAlt(`*name*`)`

Sets the subject alternative names.

`x509:getIssuerAltCritical()`

Returns the issuer alternative names critical flag as a boolean.

`x509:setIssuerAltCritical(`*boolean*`)`

Sets the issuer alternative names critical flag.

`x509:getSubjectAltCritical()`

Returns the subject alternative names critical flag as a boolean.

`x509:setSubjectAltCritical(`*boolean*`)`

Sets the subject alternative names critical flag.

`x509:getBasicConstraints([`*which*`[, `*which* `... ]])`

Returns the X.509 'basic constraint' flags. If specified, *which* should be one of "CA" or "pathLen", which returns the specified constraint—respectively, a boolean and a number. If no parameters are specified, returns a table with fields "CA" and "pathLen".

`x509:setBasicConstraints{ ... }`

Sets the basic constraint flag according to the defined field values for "CA" (boolean) and "pathLen" (number).

`x509:getBasicConstraintsCritical()`

Returns the basic constraints critical flag as a boolean.

`x509:setBasicConstraintsCritical(`*boolean*`)`

Sets the basic constraints critical flag.

`x509:addExtension(`*ext*`)`

Adds a copy of the `x509.extension` object to the certificate.

`x509:getExtension(`*key*`)`

Returns a copy of the `x509.extension` object identified by *key* where *key* is the plain text string of the OID, long name, or short name; or the integer index (1-based) of the extension. Returns nothing if no such extension was found by that name or at that index.

`x509:getExtensionCount()`

Returns the integer count of the number of extensions.

`x509:getOCSP()`

Returns the OCSP urls for the certificate.

`x509:isIssuedBy(`*issuer*`)`

Returns a boolean according to whether the specified issuer—an `openssl.x509.name` object—signed the instance certificate.

`x509:getPublicKey()`

Returns the public key component as an `openssl.pkey` object.

`x509:setPublicKey(`*key*`)`

Sets the public key component referenced by the `openssl.pkey` object *key*.

`x509:getPublicKeyDigest([`*type*`])`

Returns the digest of the public key as a binary string. *type* is an optional string describing the digest type, and defaults to "sha1".

`x509:getSignatureName()`

Returns the type of signature used to sign the certificate as a string. e.g. "RSA-SHA1"

`x509:sign(`*key* `[,` *type*`])`

Signs and updates the instance certificate using the `openssl.pkey` *key*. *type* is an optional string describing the digest type. See `pkey:sign`, regarding which types of digests are valid. If *type* is omitted than a default type is used—"sha1" for RSA keys, "dss1" for DSA keys, and "ecdsa-with-SHA1" for EC keys.

`x509:text()`

Returns a human-readable textual representation of the X.509 certificate.

`x509:__tostring`

Returns the PEM encoded representation of the instance certificate.

3.1.8 `openssl.x509.csr`

Binds the X.509 certificate signing request OpenSSL ASN.1 object.

`csr.new([`*x*509`|`*string*`[,` *format*`]])`

Returns a new request object, optionally initialized to the specified `openssl.x509` certificate *x*509 or the PEM- or DER-encoded certificate signing request *string*. *format* is as described in `openssl.pkey.new`—"PEM", "DER", or "*" (default).

`csr.interpose(`*name*`,` *function*`)`

Add or interpose a request class method. Returns the previous method, if any.

`csr:getVersion()`

Returns the X.509 version of the request.

`csr:setVersion(`*number*`)`

Sets the X.509 version of the request.

`csr:getSubject()`

Returns the subject distinguished name as an `x509.name` object.

`csr:setSubject(`*name*`)`

Sets the subject distinguished name. *name* should be an `x509.name` object.

`csr:getSubjectAlt()`

Returns the subject alternative name as an `x509.altname` object.

`csr:setSubjectAlt(`*name*`)`

Sets the subject alternative names. *name* should be an `x509.altname` object.

`csr:getPublicKey()`

Returns the public key component as an `openssl.pkey` object.

`csr:setPublicKey(`*key*`)`

Sets the public key component referenced by the `openssl.pkey` object *key*.

`csr:sign(`*key*`)`

Signs the instance request using the `openssl.pkey` *key*.

`csr:__tostring`

Returns the PEM encoded representation of the instance request.

3.1.9 `openssl.x509.crl`

Binds the X.509 certificate revocation list OpenSSL ASN.1 object.

`crl.new([`*string*`[, `*format*`]])`

Returns a new CRL object, optionally initialized to the specified PEM- or DER-encoded CRL *string*. *format* is as described in `openssl.pkey.new`—"PEM", "DER", or "*" (default).

`crl.interpose(`*name*`, `*function*`)`

Add or interpose a request class method. Returns the previous method, if any.

`crl:getVersion()`

Returns the CRL version.

`crl:setVersion(`*number*`)`

Sets the CRL version.

`crl:getLastUpdate()`

Returns the Last Update time of the CRL as a Unix timestamp, or *nil* if not set.

`crl:setLastUpdate(`*time*`)`

Sets the Last Update time of the CRL. *time* should be a Unix timestamp.

`crl:getNextUpdate()`

Returns the Next Update time of the CRL as a Unix timestamp, or *nil* if not set.

`crl:setNextUpdate(`*time*`)`

Sets the Next Update time of the CRL. *time* should be a Unix timestamp.

`crl:getIssuer()`

Returns the issuer distinguished name as an `x509.name` object.

`crl:setIssuer(`*name*`)`

Sets the issuer distinguished name. *name* should be an `x509.name` object.

`crl:add(`*serial* `[,` *time*`])`

Add the certificate identified by *serial* to the revocation list. *serial* should be a `openssl.bignum` object, as returned by `x509:getSerial`. *time* is the revocation date as a Unix timestamp. If unspecified *time* defaults to the current time.

`crl:addExtension(`*ext*`)`

Adds a copy of the `x509.extension` object to the revocation list.

`crl:getExtension(`*key*`)`

Returns a copy of the `x509.extension` object identified by *key* where *key* is the plain text string of the OID, long name, or short name; or the integer index (1-based) of the extension. Returns nothing if no such extension was found by that name or at that index.

`crl:getExtensionCount()`

Returns the integer count of the number of extensions.

`crl:sign(`*key*`)`

Signs the instance CRL using the `openssl.pkey` *key*.

`crl:verify(`*publickey*`)`

Verifies the instance CRL using a public key.

`crl:text()`

Returns a human-readable textual representation of the instance CRL.

`crl:__tostring`

Returns the PEM encoded representation of the instance CRL.

3.1.10 `openssl.x509.chain`

Binds the "STACK_OF(X509)" OpenSSL object, principally used in the OpenSSL library for representing a validation chain.

`chain.new()`

Returns a new chain object.

`chain.interpose(`*name*`, `*function*`)`

Add or interpose a chain class method. Returns the previous method, if any.

`chain:add(`*crt*`)`

Append the X.509 certificate *crt*.

`chain:__ipairs()`

Returns an iterator over the stored certificates.

3.1.11 `openssl.x509.store`

Binds the X.509 certificate "X509_STORE" OpenSSL object, principally used for loading and storing trusted certificates, paths to trusted certificates, and verification policy.

`store.new()`

Returns a new store object.

`store.interpose(`*name*`, `*function*`)`

Add or interpose a store class method. Returns the previous method, if any.

`store:add(`*crt*`|`*filepath*`|`*dirpath*`)`

Add the X.509 certificate *crt* to the store, load the certificates from the file *filepath*, or set the OpenSSL 'hashdir' certificate path *dirpath*.

`store:verify(`*crt*`[, `*chain*`])`

Returns two values. The first is a boolean value for whether the specified certificate *crt* was verified. If true, the second value is a `openssl.x509.chain` object validation chain. If false, the second value is a string describing why verification failed. The optional parameter *chain* is an `openssl.x509.chain` object of untrusted certificates linking the certificate *crt* to one of the trusted certificates in the instance store.

3.1.12 `openssl.pkcs12`

Binds the PKCS #12 container OpenSSL object.

`pkcs12.new{ ... }`

Returns a new PKCS12 object initialized according to the named parameters—*password*, *key*, *certs*.

FIXME.

`pkcs12.interpose(`*name*`, `*function*`)`

Add or interpose a store class method. Returns the previous method, if any.

`pkcs12:__tostring()`

Returns a PKCS #12 binary encoded string.

`pkcs12.parse(`*bag*`[, `*passphrase*`])`

Parses a PKCS#12 bag, presented as a binary string *bag*. The second parameter *passphrase* is the passphrase required to decrypt the PKCS#12 bag. The function returns three items; namely the key, certificate and the CA chain, as their respective objects. If an item is absent, it will be substituted with nil.

3.1.13 `openssl.ssl.context`

Binds the "SSL_CTX" OpenSSL object, used as a configuration prototype for SSL connection instances. See `socket.starttls`.

`context[]`

A table mapping OpenSSL named constants. The available constants are documented with the relevant method.

`context.new([`*protocol*`][, `*server*`])`

Returns a new context object. *protocol* is an optional string identifier selecting the OpenSSL constructor, defaulting to "TLS". If *server* is true, then SSL connections instantiated using this context will be placed into server mode, otherwise they behave as clients.

protocol identifiers

name	description
TLS	Supports TLS 1.0 *and above*. Internally uses `SSLv23_method` and disables SSLv2 and SSLv3 using `SSL_OP_NO_SSLv2` and `SSL_OP_NO_SSLv3`.
SSL	Supports SSL 3.0 *and above*. Internally uses `SSLv23_method` and disables SSLv2 using `SSL_OP_NO_SSLv2`.

SSLv23	A catchall for all versions of SSL/TLS supported by OpenSSL. Individual versions can be disabled using `context:setOptions`. Internally uses `SSLv23_method`.
TLSv1_2	Supports *only* TLS 1.2. Internally uses `TLSv1_2_method`.
TLSv1_1	Supports *only* TLS 1.1. Internally uses `TLSv1_1_method`.
TLSv1	Supports *only* TLS 1.0. Internally uses `TLSv1_method`.
SSLv3	Supports *only* SSL 3.0. Internally uses `SSLv3_method`.
SSLv2	Supports *only* SSL 2.0. Internally uses `SSLv2_method`.
DTLS	Supports DTLS 1.0 *and above*. Internally uses `DTLS_method`.
DTLSv1	Supports *only* DTLS 1.0. Internally uses `DTLSv1_method`.
DTLSv1_2	Supports *only* DTLS 1.2. Internally uses `DTLSv1_2_method`.

`context.interpose(`*name*`, `*function*`)`

Add or interpose a context class method. Returns the previous method, if any.

`context:setOptions(`*flags*`)`

Adds the option flags to the context instance. *flags* is a bit-wise set of option flags to be ORd with the current set. The resultant option flags of the context instance will be the union of the old and new flags.[2]

name	description
`OP_MICROSOFT_SESS_ID_BUG`	When talking SSLv2, if session-id reuse is performed, the session-id passed back in the server-finished message is different from the one decided upon.
`OP_NETSCAPE_CHALLENGE_BUG`	Workaround for Netscape-Commerce/1.12 servers.
`OP_LEGACY_SERVER_CONNECT`	...
`OP_NETSCAPE_REUSE_CIPHER_CHANGE_BUG`	As of OpenSSL 0.9.8q and 1.0.0c, this option has no effect.
`OP_MICROSOFT_BIG_SSLV3_BUFFER`	...
`OP_SSLEAY_080_CLIENT_DH_BUG`	...
`OP_TLS_D5_BUG`	...
`OP_TLS_BLOCK_PADDING_BUG`	...
`OP_DONT_INSERT_EMPTY_FRAGMENTS`	Disables a countermeasure against a SSL 3.0/TLS 1.0 protocol vulnerability affecting CBC ciphers, which cannot be handled by some broken SSL implementations. This option has no effect for connections using other ciphers.

[2] This idiosyncratic union behavior is how the OpenSSL routine works.

OP_NO_QUERY_MTU	...
OP_COOKIE_EXCHANGE	...
OP_NO_TICKET	Disable RFC4507bis ticket stateless session resumption.
OP_CISCO_ANYCONNECT	...
OP_NO_SESSION_RESUMPTION_ON_RENEGOTIATION	When performing renegotiation as a server, always start a new session (i.e., session resumption requests are only accepted in the initial handshake). This option is not needed for clients.
OP_NO_COMPRESSION	...
OP_ALLOW_UNSAFE_LEGACY_RENEGOTIATION	...
OP_SINGLE_ECDH_USE	Always create a new key when using temporary/ephemeral ECDH parameters.
OP_SINGLE_DH_USE	Always create a new key when using temporary/ephemeral DH parameters.
OP_EPHEMERAL_RSA	Always use ephemeral (temporary) RSA key when doing RSA operations.
OP_CIPHER_SERVER_PREFERENCE	When choosing a cipher, use the server's preferences instead of the client preferences.
OP_TLS_ROLLBACK_BUG	Disable version rollback attack detection.
OP_NO_SSLv2	Do not use the SSLv2 protocol.
OP_NO_SSLv3	Do not use the SSLv3 protocol.
OP_NO_TLSv1	Do not use the TLSv1.0 protocol.
OP_NO_TLSv1_2	Do not use the TLSv1.2 protocol.
OP_NO_TLSv1_1	Do not use the TLSv1.1 protocol.
OP_NETSCAPE_CA_DN_BUG	...
OP_NETSCAPE_DEMO_CIPHER_CHANGE_BUG	...
OP_CRYPTOPRO_TLSEXT_BUG	...
OP_ALL	All of the bug workarounds.

`context:getOptions()`

Returns the option flags of the context instance as an integer.

`context:clearOptions()`

Clears the option flags of the context instance.

`context:setStore(`*store*`)`

Associate the `openssl.x509.store` object *store* with *context*. Replaces any existing store.

`context:getStore()`

Returns the `openssl.x509.store` object associated with *context*.

`context:setParam(`*params*`)`

Causes *context* to inherit the parameters from the `openssl.x509.verify_param` object *params*. Only parameters set in *params* will take effect (others will stay unchanged).

`context:getParam()`

Returns an `openssl.x509.verify_param` object containing a copy of *context*'s parameters.

`context:setVerify([`*mode*`][, `*depth*`])`

Sets the verification mode flags and maximum validation chain depth.

name	description
VERIFY_NONE	disable client peer certificate verification
VERIFY_PEER	enable client peer certificate verification
VERIFY_FAIL_IF_NO_PEER_CERT	require a peer certificate
VERIFY_CLIENT_ONCE	do not request peer certificates after initial handshake

See the NOTES section in the OpenSSL documentation for `SSL_CTX_set_verify_mode`.

`context:getVerify()`

Returns two values: the bitwise verification mode flags, and the maximum validation depth.

`context:setCertificate(`*crt*`)`

Sets the X.509 certificate `openssl.x509` object *crt* to send during SSL connection instance handshakes.

`context:setPrivateKey(`*key*`)`

Sets the private key `openssl.pkey` object *key* for use during SSL connection instance handshakes.

`context:setCipherList(`*string* `[, ...])`

Sets the allowed public key and private key algorithm(s). The string format is documented in the OpenSSL ciphers(1) utility documentation.

`context:setCurvesList(`*string* `[, ...])`

Sets the supported curves. The string format is a list of colon separated curve names similar to `ctx:setCipherList(...)`. A list of supported curves can be found by running `openssl ecparam -list_curves`.

Only supported since OpenSSL 1.0.2.

`context:setEphemeralKey(`*key*`)`

Sets `openssl.pkey` object *key* as the ephemeral key during key exchanges which use that particular key type. Typically *key* will be either a Diffie-Hellman or Elliptic Curve key.

In order to configure an SSL server to support an ephemeral key exchange cipher suite (i.e. DHE- and ECDHE-*), the application must explicitly set the ephemeral keys. Simply enabling the cipher suite is not sufficient. The application can statically generate Diffie-Hellman public key parameters, and many servers ship with such a key compiled into the software. Elliptic curve keys are necessarily static, and instantiated by curve name*[3].

In addition, to attain Perfect Forward Secrecy the options `OP_SINGLE_DH_USE` and `OP_SINGLE_ECDH_USE` must be set so that OpenSSL discards and regenerates the secret keying parameters for each key exchange.

`context:setAlpnProtos(`*table*`)`

Sets the advertised ALPN protocols. *table* is an array of protocol string identifiers.

Only supported since OpenSSL 1.0.2.

`context:setAlpnSelect(`*cb*`)`

Sets the callback used to select an ALPN protocol. *cb* should be a function that takes two arguments: an `openssl.ssl` object and a table containing a sequence of ALPN protocol strings; it should return the ALPN protocol string it selected or *nil* to select none of them.

Only supported since OpenSSL 1.0.2.

`context:setTLSextStatusType(`*type*`)`

Sets the default TLS extension status for SSL objects derived from this context. See `ssl:setTLSextStatusType`

Only supported since OpenSSL 1.1.0.

`context:getTLSextStatusType()`

Gets the default TLS extension status for SSL objects derived from this context as a string. See `ssl:getTLSextStatusType`

Only supported since OpenSSL 1.1.0.

3.1.14 `openssl.ssl`

Binds the "SSL" OpenSSL object, which represents an SSL connection instance. See `cqueues.socket:checktls`.

[3] OpenSSL ¡ 1.0.2 only supports a single curve, according to Wikipedia the most widely supported curve is prime256v1, so to enable ECDHE-*, applications can simply do `ctx:setEphemeralKey(pkey.new{ type = ''EC'', curve = ''prime256v1'' })`. To achieve Perfect Forward Secrecy for ECDHE-*, applications must also do `ctx:setOptions(context.OP_SINGLE_ECDH_USE)`. The `ctx` object must then be used to configure each SSL session, such as by passing it to `cqueues.socket:starttls()`.

`ssl[]`

A table mapping OpenSSL named constants. Includes all constants provided by `openssl.ssl.context`. Additional constants are documented with the relevant method.

`ssl.interpose(`*name*`, `*function*`)`

Add or interpose an ssl class method. Returns the previous method, if any.

`ssl:setContext(`*context*`)`

Replaces the `openssl.ssl.context` used by *ssl* with *context*.

`ssl:setOptions(`*flags*`)`

Adds the option flags of the SSL connection instance. See `openssl.ssl.context:setOptions`.

`ssl:getOptions()`

Returns the option flags of the SSL connection instance. See `openssl.ssl.context:getOptions`.

`ssl:clearOptions()`

Clears the option flags of the SSL connection instance. See `openssl.ssl.context:clearOptions`.

`ssl:setVerify([`*mode*`][, `*depth*`])`

Sets the verification mode flags and maximum validation chain depth. See `openssl.ssl.context:setVerify`.

`ssl:getVerify()`

Returns two values: the bitwise verification mode flags, and the maximum validation depth. See `openssl.ssl.context:getVerify`.

`ssl:getVerifyResult()`

Returns two values: the integer verification result code and the string representation of that code.

`ssl:setCertificate(`*crt*`)`

Sets the X.509 certificate `openssl.x509` object *crt* to send during SSL connection instance handshakes. See `openssl.ssl.context:setCertificate`.

`ssl:setPrivateKey(`*key*`)`

Sets the private key `openssl.pkey` object *key* for use during SSL connection instance handshakes. See `openssl.ssl.context:setPrivateKey`.

`ssl:getPeerCertificate()`

Returns the X.509 peer certificate as an `openssl.x509` object. If no peer certificate is available, returns *nil*.

`ssl:getPeerChain()`

Similar to :getPeerCertifiate, but returns the entire chain sent by the peer as an `openssl.x509.chain` object.

`ssl:getCipherInfo()`

Returns a table of information on the current cipher.

field	description
.name	cipher name returned by `SSL_CIPHER_get_name`
.bits	number of secret bits returned by `SSL_CIPHER_get_bits`
.version	SSL/TLS version string returned by `SSL_CIPHER_get_version`
.description	key:value cipher description returned by `SSL_CIPHER_description`

`ssl:setHostName(`*host*`)`

Sets the Server Name Indication (SNI) host name. Using the SNI TLS extension, clients tells the server which domain they're contacting so the server can select the proper certificate and key. This permits SSL virtual hosting. This routine is only relevant for clients.

`ssl:getHostName()`

Returns the Server Name Indication (SNI) host name sent by the client. If no host name was sent, returns *nil*. This routine is only relevant for servers.

`ssl:getVersion([`*format*`])`

Returns the SSL/TLS version of the negotiated SSL connection. By default returns a 16-bit integer where the top 8 bits are the major version number and the bottom 8 bits the minor version number. For example, SSL 3.0 is 0x0300 and TLS 1.1 is 0x0302. SSL 2.0 is 0x0002.

If *format* is "." returns a floating point number. 0x0300 becomes 3.0, and 0x0302 becomes 3.2. If the minor version is ≥ 10 an error is thrown.[4]

The following OpenSSL named constants can be used.

[4] This condition shouldn't be possible.

name	description
SSL2_VERSION	16-bit SSLv2 identifier (0x0002).
SSL3_VERSION	16-bit SSLv3 identifier (0x0300).
TLS1_VERSION	16-bit TLSv1.0 identifier (0x0301).
TLS1_1_VERSION	16-bit TLSv1.1 identifier (0x0302).
TLS1_2_VERSION	16-bit TLSv1.2 identifier (0x0303).

`ssl:getClientVersion([`*format*`])`

Returns the SSL/TLS version supported by the client, which should be greater than or equal to the negotiated version. See `ssl:getVersion`.

`ssl:setCurvesList(`*string* `[, ...])`

Sets the supported curves for this SSL connection instance. See `openssl.ssl.context:setCurvesList`.

Only supported since OpenSSL 1.0.2.

`ssl:getAlpnSelected()`

Returns the negotiated ALPN protocol as a string.

Only supported since OpenSSL 1.0.2.

`ssl:setAlpnProtos(`*table*`)`

Sets the advertised ALPN protocols. *table* is an array of protocol string identifiers.

Only supported since OpenSSL 1.0.2.

`ssl:setTLSextStatusType(`*type*`)`

Sets the TLS extension status.

Only the *type* "ocsp" is currently supported, this is used by a client to request that a server sends a stapled OCSP response as part of the TLS handshake.

See also: `context:setTLSextStatusType()`

`ssl:getTLSextStatusType()`

Gets the TLS extension status. As set by `ssl:setTLSextStatusType` or `context:setTLSextStatusType`.

Only the type "ocsp" is currently known.

Only supported since OpenSSL 1.1.0.

`ssl:setTLSextStatusOCSPResp(`*or*`)`

Sets an `openssl.ocsp.response`. Used by a server to staple an OCSP response into a TLS handshake.

`ssl:getTLSextStatusOCSPResp()`

Returns the `openssl.ocsp.response` associated with the ssl object (or *nil* if one has not been set).

3.1.15 `openssl.digest`

Binds the "EVP_MD_CTX" OpenSSL object, which represents a cryptographic message digest (i.e. hashing) algorithm instance.

`digest.interpose(`*name*`, `*function*`)`

Add or interpose a digest class method. Returns the previous method, if any.

`digest.new([`*type*`])`

Return a new digest instance using the specified algorithm *type*. *type* is a string suitable for passing to the OpenSSL routine EVP_get_digestbyname, and defaults to "sha1".

`digest:update([`*string* `[, ...]])`

Update the digest with the specified string(s). Returns the digest object.

`digest:final([`*string* `[, ...]])`

Update the digest with the specified string(s). Returns the final message digest as a binary string.

3.1.16 `openssl.hmac`

Binds the "HMAC_CTX" OpenSSL object, which represents a cryptographic HMAC algorithm instance.

`hmac.interpose(`*name*`, `*function*`)`

Add or interpose an HMAC class method. Returns the previous method, if any.

`hmac.new(`*key* `[, `*type*`])`

Return a new HMAC instance using the specified *key* and *type*. *key* is the secret used for HMAC authentication. *type* is a string suitable for passing to the OpenSSL routine EVP_get_digestbyname, and defaults to "sha1".

`hmac:update([`*string* `[, ...]])`

Update the HMAC with the specified string(s). Returns the HMAC object.

`hmac:final([`*string* `[, ...]])`

Update the HMAC with the specified string(s). Returns the final HMAC checksum as a binary string.

3.1.17 `openssl.cipher`

Binds the "EVP_CIPHER_CTX" OpenSSL object, which represents a cryptographic cipher instance.

`cipher.interpose(`*name*`, `*function*`)`

Add or interpose a cipher class method. Returns the previous method, if any.

`cipher.new(`*type*`)`

Return a new, uninitialized cipher instance. *type* is a string suitable for passing to the OpenSSL routine EVP_get_cipherbyname, typically of a form similar to "AES-128-CBC".

The cipher is uninitialized because some algorithms support or require additional *ad hoc* parameters before key initialization. The API still allows one-shot encryption like "cipher.new(type):encrypt(key, iv):final(plaintext)".

`cipher:encrypt(`*key* `[, `*iv*`] [, `*padding*`])`

Initialize the cipher in encryption mode. *key* and *iv* are binary strings with lengths equal to that required by the cipher instance as configured. In other words, key stretching and other transformations must be done explicitly. If the mode does not take an IV or equivalent, such as in ECB mode, then it may be nil. *padding* is a boolean which controls whether PKCS padding is applied, and defaults to true. Returns the cipher instance.

`cipher:decrypt(`*key* `[, `*iv*`] [, `*padding*`])`

Initialize the cipher in decryption mode. *key*, *iv*, and *padding* are as described in `:encrypt`. Returns the cipher instance.

`cipher:update([`*string* `[, ...]])`

Update the cipher instance with the specified string(s). Returns a string on success, or nil and an error message on failure. The returned string may be empty if no blocks can be flushed.

`cipher:final([`*string* `[, ...]])`

Update the cipher with the specified string(s). Returns the final output string on success, or nil and an error message on failure. The returned string may be empty if all blocks have already been flushed in prior `:update` calls.

3.1.18 `openssl.ocsp.response`

Binds OpenSSL's `OCSP_RESPONSE` object.

`response:getBasic()`

Returns a `openssl.ocsp.basic` representation of the object contained within the OCSP response.

`response:tostring()`

Returns a human readable description of the OCSP response as a string.

`response:toPEM()`

Returns the OCSP response as a PEM encoded string.

3.1.19 `openssl.ocsp.basic`

Binds OpenSSL's `OCSP_BASICRESP` object.

`basic:verify([`*certs* `[,` *store*`[,` *flags*`]]])`

Verifies that the OCSP response is signed by a certificate in the `openssl.x509.chain` *certs* or a trusted certificate in `openssl.x509.store` *store*.

3.1.20 `openssl.rand`

Binds OpenSSL's random number interfaces.

OpenSSL will automatically attempt to seed itself from the system. The only time this could theoretically fail is if /dev/urandom (or similar) were not visible or could not be opened. This might happen if within a chroot jail, or if a file descriptor limit were reached.

`rand.bytes(`*count*`)`

Returns *count* cryptographically-strong bytes as a single string. Throws an error if OpenSSL could not complete the request—e.g. because the CSPRNG could not be seeded.

`rand.ready()`

Returns a boolean describing whether the CSPRNG has been properly seeded.

In the default CSPRNG engine this routine will also attempt to seed the system if not already. Because seeding only needs to happen once per process to ensure a successful RAND_bytes invocation[5], it may be prudent to assert on rand:ready() at application startup.

`rand.uniform([`n`])`

Returns a cryptographically strong uniform random integer in the interval $[0, n-1]$. If n is omitted, the interval is $[0, 2^{64}-1]$.

The routine operates internally on 64-bit unsigned integers.[6] Because neither Lua 5.1 nor 5.2 support 64-bit integers, it's probably best to generate numbers that fit the integral range of your

[5] At least this appeared to be the case when examining the source code of OpenSSL 1.0.1. See md_rand.c near line 407—"Once we've had enough initial seeding we don't bother to adjust the entropy count, though, because we're not ambitious to provide *information-theoretic* randomness."

[6] Actually, `unsigned long long`.

Lua implementation. Lua 5.3 supports a new arithmetic type for 64-bit signed integers in two's-complement representation. This new arithmetic type will be used for argument and return values when available.

3.1.21 `openssl.des`

Binds OpenSSL's DES interfaces. These bindings are incomplete. No modern protocol would ever need to use these directly. However, legacy protocols like Windows LAN Manager authentication require some of these low-level interfaces.

`des.string_to_key(`*password*`)`

Converts an arbitrary length string, *password*, to a DES key using DES_string_to_key. Returns an 8-byte string.

Note that OpenSSL's DES_string_to_key is not compatible with Windows LAN Manager hashing scheme. Use `des.set_odd_parity` instead. See examples/lm.hash.

`des.set_odd_parity(`*key*`)`

Applies DES_set_odd_parity to the string *key*. Only the first 8 bytes of *key* are processed. Returns an 8-byte string.

4 Examples

These examples and others are made available under examples/ in the source tree.

4.1 Self-Signed Certificate

```
--
-- Example self-signed X.509 certificate generation.
--
-- Skips intermediate CSR object, which is just an antiquated way for
-- specifying subject DN and public key to CAs. See API documentation for
-- CSR generation.
--
local pkey = require"openssl.pkey"
local x509 = require"openssl.x509"
local name = require"openssl.x509.name"
local altname = require"openssl.x509.altname"

-- generate our public/private key pair
local key = pkey.new{ type = "EC", curve = "prime192v1" }

-- our Subject and Issuer DN (self-signed, so same)
local dn = name.new()
dn:add("C", "US")
dn:add("ST", "California")
dn:add("L", "San␣Francisco")
dn:add("O", "Acme,␣Inc")
dn:add("CN", "acme.inc")

-- our Alternative Names
local alt = altname.new()
alt:add("DNS", "acme.inc")
alt:add("DNS", "*.acme.inc")

-- build our certificate
local crt = x509.new()

crt:setVersion(3)
crt:setSerial(42)

crt:setSubject(dn)
crt:setIssuer(crt:getSubject())
crt:setSubjectAlt(alt)

local issued, expires = crt:getLifetime()
crt:setLifetime(issued, expires + 60) -- good for 60 seconds

crt:setBasicConstraints{ CA = true, pathLen = 2 }
```

```
43 crt:setBasicConstraintsCritical(true)

45 crt:setPublicKey(key)
   crt:sign(key)
47
   -- pretty-print using openssl command-line utility.
49 io.popen("openssl␣x509␣-text␣-noout", "w"):write(tostring(crt))
```

4.2 Signature Generation & Verification

```
--
-- Example public-key signature verification.
--
local pkey = require"openssl.pkey"
local digest = require"openssl.digest"

-- generate a public/private key pair
local key = pkey.new{ type = "EC", curve = "prime192v1" }

-- digest our message using an appropriate digest
local data = digest.new "sha1"
data:update(... or "hello␣world")

-- generate a signature for our data
local sig = key:sign(data)

-- to prove verification works, instantiate a new object holding just
-- the public key
local pub = pkey.new(key:toPEM"public")

-- a utility routine to output our signature
local function tohex(b)
    local x = ""
    for i = 1, #b do
        x = x .. string.format("%.2x", string.byte(b, i))
    end
    return x
end

print("okay", pub:verify(sig, data))
print("type", pub:type())
print("sig", tohex(sig))
```