Python OpenSSL Manual

Release 0.10

Jean-Paul Calderone

July 22, 2015

exarkun@twistedmatrix.com

Abstract

This module is a rather thin wrapper around (a subset of) the OpenSSL library. With thin wrapper I mean that a lot of the object methods do nothing more than calling a corresponding function in the OpenSSL library.

Contents

1 Introduction

The reason pyOpenSSL was created is that the SSL support in the socket module in Python 2.1 (the contemporary version of Python when the pyOpenSSL project was begun) was severely limited. Other OpenSSL wrappers for Python at the time were also limited, though in different ways. Unfortunately, Python's standard library SSL support has remained weak, although other packages (such as M2Crypto[1]) have made great advances and now equal or exceed pyOpenSSL's functionality.

The reason pyOpenSSL continues to be maintained is that there is a significant user community around it, as well as a large amount of software which depends on it. It is a great benefit to many people for pyOpenSSL to continue to exist and advance.

2 Building and Installing

These instructions can also be found in the file `INSTALL`.

I have tested this on Debian Linux systems (woody and sid), Solaris 2.6 and 2.7. Others have successfully compiled it on Windows and NT.

2.1 Building the Module on a Unix System

pyOpenSSL uses distutils, so there really shouldn't be any problems. To build the library:

```
python setup.py build
```

If your OpenSSL header files aren't in `/usr/include`, you may need to supply the `-I` flag to let the setup script know where to look. The same goes for the libraries of course, use the `-L` flag. Note that `build` won't accept these flags, so you have to run first `build_ext` and then `build`! Example:

```
python setup.py build_ext -I/usr/local/ssl/include -L/usr/local/ssl/lib
python setup.py build
```

Now you should have a directory called `OpenSSL` that contains e.g. `SSL.so` and `__init__.py` somewhere in the build dicrectory, so just:

```
python setup.py install
```

If you, for some arcane reason, don't want the module to appear in the `site-packages` directory, use the `--prefix` option.

You can, of course, do

```
python setup.py --help
```

to find out more about how to use the script.

[1] See http://chandlerproject.org/Projects/MeTooCrypto

2.2 Building the Module on a Windows System

Big thanks to Itamar Shtull-Trauring and Oleg Orlov for their help with Windows build instructions. Same as for Unix systems, we have to separate the `build_ext` and the `build`.

Building the library:

```
setup.py build_ext -I ...\openssl\inc32 -L ...\openssl\out32dll
setup.py build
```

Where `...\openssl` is of course the location of your OpenSSL installation.

Installation is the same as for Unix systems:

```
setup.py install
```

And similarily, you can do

```
setup.py --help
```

to get more information.

3 OpenSSL — Python interface to OpenSSL

This package provides a high-level interface to the functions in the OpenSSL library. The following modules are defined:

`crypto`
: Generic cryptographic module. Note that if anything is incomplete, this module is!

`rand`
: An interface to the OpenSSL pseudo random number generator.

`SSL`
: An interface to the SSL-specific parts of OpenSSL.

3.1 crypto — Generic cryptographic module

`X509Type`
: See `X509`.

class `X509()`
: A class representing X.509 certificates.

`X509NameType`
: See `X509Name`.

class `X509Name(`*x509name*`)`
: A class representing X.509 Distinguished Names.

This constructor creates a copy of *x509name* which should be an instance of `X509Name`.

`X509ReqType`

See `X509Req`.

class `X509Req()`
A class representing X.509 certificate requests.

`X509StoreType`
A Python type object representing the X509Store object type.

`PKeyType`
See `PKey`.

class `PKey()`
A class representing DSA or RSA keys.

`PKCS7Type`
A Python type object representing the PKCS7 object type.

`PKCS12Type`
A Python type object representing the PKCS12 object type.

`X509ExtensionType`
See `X509Extension`.

class `X509Extension(`*typename, critical, value*[*, subject*][*, issuer*]`)`
A class representing an X.509 v3 certificate extensions. See http://openssl.org/docs/apps/x509v3_config.html#STANDARD_EXTENSIONS for *typename* strings and their options. Optional parameters *subject* and *issuer* must be X509 objects.

`NetscapeSPKIType`
See `NetscapeSPKI`.

class `NetscapeSPKI(`[*enc*]`)`
A class representing Netscape SPKI objects.

If the *enc* argument is present, it should be a base64-encoded string representing a NetscapeSPKI object, as returned by the `b64_encode` method.

`FILETYPE_PEM`
`FILETYPE_ASN1`
File type constants.

`TYPE_RSA`
`TYPE_DSA`
Key type constants.

exception `Error`
Generic exception used in the `crypto` module.

`dump_certificate(`*type, cert*`)`
Dump the certificate *cert* into a buffer string encoded with the type *type*.

`dump_certificate_request(`*type, req*`)`
Dump the certificate request *req* into a buffer string encoded with the type *type*.

`dump_privatekey(`*type, pkey*[*, cipher, passphrase*]`)`
Dump the private key *pkey* into a buffer string encoded with the type *type*, optionally (if *type* is `FILETYPE_PEM`) encrypting it using *cipher* and *passphrase*.

passphrase must be either a string or a callback for providing the pass phrase.

`load_certificate(`*type, buffer*`)`
Load a certificate (X509) from the string *buffer* encoded with the type *type*.

`load_certificate_request(`*type, buffer*`)`
Load a certificate request (X509Req) from the string *buffer* encoded with the type *type*.

`load_privatekey(`*type, buffer*[, *passphrase*]`)`
Load a private key (PKey) from the string *buffer* encoded with the type *type* (must be one of `FILETYPE_PEM` and `FILETYPE_ASN1`).

passphrase must be either a string or a callback for providing the pass phrase.

`load_pkcs7_data(`*type, buffer*`)`
Load pkcs7 data from the string *buffer* encoded with the type *type*.

`load_pkcs12(`*buffer*[, *passphrase*]`)`
Load pkcs12 data from the string *buffer*. If the pkcs12 structure is encrypted, a *passphrase* must be included. The MAC is always checked and thus required.

See also the man page for the C function `PKCS12_parse`.

X509 objects

X509 objects have the following methods:

`get_issuer()`
Return an X509Name object representing the issuer of the certificate.

`get_pubkey()`
Return a PKey object representing the public key of the certificate.

`get_serial_number()`
Return the certificate serial number.

`get_subject()`
Return an X509Name object representing the subject of the certificate.

`get_version()`
Return the certificate version.

`get_notBefore()`
Return a string giving the time before which the certificate is not valid. The string is formatted as an ASN1 GENERALIZEDTIME:

```
YYYYMMDDhhmmssZ
YYYYMMDDhhmmss+hhmm
YYYYMMDDhhmmss-hhmm
```

If no value exists for this field, `None` is returned.

`get_notAfter()`
Return a string giving the time after which the certificate is not valid. The string is formatted as an ASN1 GENERALIZEDTIME:

```
YYYYMMDDhhmmssZ
YYYYMMDDhhmmss+hhmm
YYYYMMDDhhmmss-hhmm
```

If no value exists for this field, `None` is returned.

`set_notBefore(`*when*`)`
Change the time before which the certificate is not valid. *when* is a string formatted as an ASN1 GENERALIZEDTIME:

```
YYYYMMDDhhmmssZ
YYYYMMDDhhmmss+hhmm
YYYYMMDDhhmmss-hhmm
```

`set_notAfter(`*when*`)`
Change the time after which the certificate is not valid. *when* is a string formatted as an ASN1 GENERALIZEDTIME:

```
YYYYMMDDhhmmssZ
YYYYMMDDhhmmss+hhmm
YYYYMMDDhhmmss-hhmm
```

`gmtime_adj_notBefore(`*time*`)`
Adjust the timestamp (in GMT) when the certificate starts being valid.

`gmtime_adj_notAfter(`*time*`)`
Adjust the timestamp (in GMT) when the certificate stops being valid.

`has_expired()`
Checks the certificate's time stamp against current time. Returns true if the certificate has expired and false otherwise.

`set_issuer(`*issuer*`)`
Set the issuer of the certificate to *issuer*.

`set_pubkey(`*pkey*`)`
Set the public key of the certificate to *pkey*.

`set_serial_number(`*serialno*`)`
Set the serial number of the certificate to *serialno*.

`set_subject(`*subject*`)`
Set the subject of the certificate to *subject*.

`set_version(`*version*`)`
Set the certificate version to *version*.

`sign(`*pkey, digest*`)`
Sign the certificate, using the key *pkey* and the message digest algorithm identified by the string *digest*.

`subject_name_hash()`
Return the hash of the certificate subject.

`digest(`*digest_name*`)`
Return a digest of the certificate, using the *digest_name* method. *digest_name* must be a string describing a digest algorithm supported by OpenSSL (by EVP_get_digestbyname, specifically). For example, `"md5"` or `"sha1"`.

`add_extensions(`*extensions*`)`
Add the extensions in the sequence *extensions* to the certificate.

X509Name objects

X509Name objects have the following methods:

`hash()`
Return an integer giving the first four bytes of the MD5 digest of the DER representation of the name.

`der()`

Return a string giving the DER representation of the name.

`get_components()`
Return a list of two-tuples of strings giving the components of the name.

X509Name objects have the following members:

`countryName`
The country of the entity. `C` may be used as an alias for `countryName`.

`stateOrProvinceName`
The state or province of the entity. `ST` may be used as an alias for `stateOrProvinceName`ů

`localityName`
The locality of the entity. `L` may be used as an alias for `localityName`.

`organizationName`
The organization name of the entity. `O` may be used as an alias for `organizationName`.

`organizationalUnitName`
The organizational unit of the entity. `OU` may be used as an alias for `organizationalUnitName`.

`commonName`
The common name of the entity. `CN` may be used as an alias for `commonName`.

`emailAddress`
The e-mail address of the entity.

X509Req objects

X509Req objects have the following methods:

`get_pubkey()`
Return a PKey object representing the public key of the certificate request.

`get_subject()`
Return an X509Name object representing the subject of the certificate.

`set_pubkey(`*pkey*`)`
Set the public key of the certificate request to *pkey*.

`sign(`*pkey, digest*`)`
Sign the certificate request, using the key *pkey* and the message digest algorithm identified by the string *digest*.

`verify(`*pkey*`)`
Verify a certificate request using the public key *pkey*.

`set_version(`*version*`)`
Set the version (RFC 2459, 4.1.2.1) of the certificate request to *version*.

`get_version()`
Get the version (RFC 2459, 4.1.2.1) of the certificate request.

X509Store objects

The X509Store object has currently just one method:

`add_cert(`*cert*`)`
Add the certificate *cert* to the certificate store.

PKey objects

The PKey object has the following methods:

`bits()`
: Return the number of bits of the key.

`generate_key(`*type, bits*`)`
: Generate a public/private key pair of the type *type* (one of `TYPE_RSA` and `TYPE_DSA`) with the size *bits*.

`type()`
: Return the type of the key.

PKCS7 objects

PKCS7 objects have the following methods:

`type_is_signed()`
: FIXME

`type_is_enveloped()`
: FIXME

`type_is_signedAndEnveloped()`
: FIXME

`type_is_data()`
: FIXME

`get_type_name()`
: Get the type name of the PKCS7.

PKCS12 objects

PKCS12 objects have the following methods:

`export(`[*passphrase=None*]`[`*, iter=2048*`]`[*, maciter=1*]`)`
: Returns a PKCS12 object as a string.

  The optional *passphrase* must be a string not a callback.

  See also the man page for the C function `PKCS12_create`.

`get_ca_certificates()`
: Return CA certificates within the PKCS12 object as a tuple. Returns `None` if no CA certificates are present.

`get_certificate()`
: Return certificate portion of the PKCS12 structure.

`get_friendlyname()`
: Return friendlyName portion of the PKCS12 structure.

`get_privatekey()`
: Return private key portion of the PKCS12 structure

`set_ca_certificates(`*cacerts*`)`
: Replace or set the CA certificates within the PKCS12 object with the sequence *cacerts*.

  Set *cacerts* to `None` to remove all CA certificates.

`set_certificate(`*cert*`)`
: Replace or set the certificate portion of the PKCS12 structure.

`set_friendlyname(`*name*`)`
Replace or set the friendlyName portion of the PKCS12 structure.

`set_privatekey(`*pkey*`)`
Replace or set private key portion of the PKCS12 structure

X509Extension objects

X509Extension objects have several methods:

`get_critical()`
Return the critical field of the extension object.

`get_short_name()`
Return the short type name of the extension object.

NetscapeSPKI objects

NetscapeSPKI objects have the following methods:

`b64_encode()`
Return a base64-encoded string representation of the object.

`get_pubkey()`
Return the public key of object.

`set_pubkey(`*key*`)`
Set the public key of the object to *key*.

`sign(`*key, digest_name*`)`
Sign the NetscapeSPKI object using the given *key* and *digest_name*. *digest_name* must be a string describing a digest algorithm supported by OpenSSL (by EVP_get_digestbyname, specifically). For example, `"md5"` or `"sha1"`.

`verify(`*key*`)`
Verify the NetscapeSPKI object using the given *key*.

3.2 `rand` — An interface to the OpenSSL pseudo random number generator

This module handles the OpenSSL pseudo random number generator (PRNG) and declares the following:

`add(`*string, entropy*`)`
Mix bytes from *string* into the PRNG state. The *entropy* argument is (the lower bound of) an estimate of how much randomness is contained in *string*, measured in bytes. For more information, see e.g. RFC 1750.

`bytes(`*num_bytes*`)`
Get some random bytes from the PRNG as a string.

This is a wrapper for the C function `RAND_bytes`.

`cleanup()`
Erase the memory used by the PRNG.

This is a wrapper for the C function `RAND_cleanup`.

`egd(`*path*[, *bytes*]`)`
Query the Entropy Gathering Daemon[2] on socket *path* for *bytes* bytes of random data and and uses

[2] See http://www.lothar.com/tech/crypto/

add to seed the PRNG. The default value of *bytes* is 255.

load_file(*path*[, *bytes*])
Read *bytes* bytes (or all of it, if *bytes* is negative) of data from the file *path* to seed the PRNG. The default value of *bytes* is -1.

screen()
Add the current contents of the screen to the PRNG state. Availability: Windows.

seed(*string*)
This is equivalent to calling add with *entropy* as the length of the string.

status()
Returns true if the PRNG has been seeded with enough data, and false otherwise.

write_file(*path*)
Write a number of random bytes (currently 1024) to the file *path*. This file can then be used with load_file to seed the PRNG again.

exception Error
If the current RAND method supports any errors, this is raised when needed. The default method does not raise this when the entropy pool is depleted.

Whenever this exception is raised directly, it has a list of error messages from the OpenSSL error queue, where each item is a tuple (*lib*, *function*, *reason*). Here *lib*, *function* and *reason* are all strings, describing where and what the problem is. See *err*(3) for more information.

3.3 SSL — An interface to the SSL-specific parts of OpenSSL

This module handles things specific to SSL. There are two objects defined: Context, Connection.

SSLv2_METHOD
SSLv3_METHOD
SSLv23_METHOD
TLSv1_METHOD
These constants represent the different SSL methods to use when creating a context object.

VERIFY_NONE
VERIFY_PEER
VERIFY_FAIL_IF_NO_PEER_CERT
These constants represent the verification mode used by the Context object's set_verify method.

FILETYPE_PEM
FILETYPE_ASN1
File type constants used with the use_certificate_file and use_privatekey_file methods of Context objects.

OP_SINGLE_DH_USE
OP_EPHEMERAL_RSA
OP_NO_SSLv2
OP_NO_SSLv3
OP_NO_TLSv1
Constants used with set_options of Context objects. OP_SINGLE_DH_USE means to always create a new key when using ephemeral Diffie-Hellman. OP_EPHEMERAL_RSA means to always use ephemeral RSA keys when doing RSA operations. OP_NO_SSLv2, OP_NO_SSLv3 and OP_NO_TLSv1 means to disable those specific protocols. This is interesting if you're using e.g. SSLv23_METHOD to get an SSLv2-compatible handshake, but don't want to use SSLv2.

ContextType
See Context.

class `Context(`*method*`)`

A class representing SSL contexts. Contexts define the parameters of one or more SSL connections.

method should be `SSLv2_METHOD`, `SSLv3_METHOD`, `SSLv23_METHOD` or `TLSv1_METHOD`.

`ConnectionType`

See `Connection`.

class `Connection(`*context, socket*`)`

A class representing SSL connections.

context should be an instance of `Context` and *socket* should be a socket [3] object. *socket* may be *None*; in this case, the Connection is created with a memory BIO: see the `bio_read`, `bio_write`, and `bio_shutdown` methods.

exception `Error`

This exception is used as a base class for the other SSL-related exceptions, but may also be raised directly.

Whenever this exception is raised directly, it has a list of error messages from the OpenSSL error queue, where each item is a tuple `(`*lib*`, `*function*`, `*reason*`)`. Here *lib*, *function* and *reason* are all strings, describing where and what the problem is. See *err*(3) for more information.

exception `ZeroReturnError`

This exception matches the error return code `SSL_ERROR_ZERO_RETURN`, and is raised when the SSL Connection has been closed. In SSL 3.0 and TLS 1.0, this only occurs if a closure alert has occurred in the protocol, i.e. the connection has been closed cleanly. Note that this does not necessarily mean that the transport layer (e.g. a socket) has been closed.

It may seem a little strange that this is an exception, but it does match an `SSL_ERROR` code, and is very convenient.

exception `WantReadError`

The operation did not complete; the same I/O method should be called again later, with the same arguments. Any I/O method can lead to this since new handshakes can occur at any time.

The wanted read is for *dirty* data sent over the network, not the *clean* data inside the tunnel. For a socket based SSL connection, *read* means data coming at us over the network. Until that read succeeds, the attempted `OpenSSL.SSL.Connection.recv`, `OpenSSL.SSL.Connection.send`, or `OpenSSL.SSL.Connection.do_handshake` is prevented or incomplete. You probably want to `select()` on the socket before trying again.

exception `WantWriteError`

See `WantReadError`. The socket send buffer may be too full to write more data.

exception `WantX509LookupError`

The operation did not complete because an application callback has asked to be called again. The I/O method should be called again later, with the same arguments. Note: This won't occur in this version, as there are no such callbacks in this version.

exception `SysCallError`

The `SysCallError` occurs when there's an I/O error and OpenSSL's error queue does not contain any information. This can mean two things: An error in the transport protocol, or an end of file that violates the protocol. The parameter to the exception is always a pair `(`*errnum*`, `*errstr*`)`.

Context objects

Context objects have the following methods:

[3] Actually, all that is required is an object that *behaves* like a socket, you could even use files, even though it'd be tricky to get the handshakes right!

`check_privatekey()`
: Check if the private key (loaded with `use_privatekey`[`_file`]) matches the certificate (loaded with `use_certificate`[`_file`]). Returns `None` if they match, raises `Error` otherwise.

`get_app_data()`
: Retrieve application data as set by `set_app_data`.

`get_cert_store()`
: Retrieve the certificate store (a X509Store object) that the context uses. This can be used to add "trusted" certificates without using the. `load_verify_locations()` method.

`get_timeout()`
: Retrieve session timeout, as set by `set_timeout`. The default is 300 seconds.

`get_verify_depth()`
: Retrieve the Context object's verify depth, as set by `set_verify_depth`.

`get_verify_mode()`
: Retrieve the Context object's verify mode, as set by `set_verify`.

`load_client_ca(`*pemfile*`)`
: Read a file with PEM-formatted certificates that will be sent to the client when requesting a client certificate.

`set_client_ca_list(`*certificate_ authorities*`)`
: Replace the current list of preferred certificate signers that would be sent to the client when requesting a client certificate with the *certificate_ authorities* sequence of `OpenSSL.crypto.X509Names`.

  New in version 0.10.

`add_client_ca(`*certificate_ authority*`)`
: Extract a `OpenSSL.crypto.X509Name` from the *certificate_ authority* `OpenSSL.crypto.X509` certificate and add it to the list of preferred certificate signers sent to the client when requesting a client certificate.

  New in version 0.10.

`load_verify_locations(`*pemfile, capath*`)`
: Specify where CA certificates for verification purposes are located. These are trusted certificates. Note that the certificates have to be in PEM format. If capath is passed, it must be a directory prepared using the `c_rehash` tool included with OpenSSL. Either, but not both, of *pemfile* or *capath* may be `None`.

`set_default_verify_paths()`
: Specify that the platform provided CA certificates are to be used for verification purposes. This method may not work properly on OS X.

`load_tmp_dh(`*dhfile*`)`
: Load parameters for Ephemeral Diffie-Hellman from *dhfile*.

`set_app_data(`*data*`)`
: Associate *data* with this Context object. *data* can be retrieved later using the `get_app_data` method.

`set_cipher_list(`*ciphers*`)`
: Set the list of ciphers to be used in this context. See the OpenSSL manual for more information (e.g. ciphers(1))

`set_info_callback(`*callback*`)`
: Set the information callback to *callback*. This function will be called from time to time during SSL handshakes. *callback* should take three arguments: a Connection object and two integers. The first integer specifies where in the SSL handshake the function was called, and the other the return code from a (possibly failed) internal function call.

`set_options(`*options*`)`

Add SSL options. Options you have set before are not cleared! This method should be used with the `OP_*` constants.

`set_passwd_cb(`*callback*[, *userdata*]`)`
Set the passphrase callback to *callback*. This function will be called when a private key with a passphrase is loaded. *callback* must accept three positional arguments. First, an integer giving the maximum length of the passphrase it may return. If the returned passphrase is longer than this, it will be truncated. Second, a boolean value which will be true if the user should be prompted for the passphrase twice and the callback should verify that the two values supplied are equal. Third, the value given as the *userdata* parameter to `set_passwd_cb`. If an error occurs, *callback* should return a false value (e.g. an empty string).

`set_session_id(`*name*`)`
Set the context *name* within which a session can be reused for this Context object. This is needed when doing session resumption, because there is no way for a stored session to know which Context object it is associated with. *name* may be any binary data.

`set_timeout(`*timeout*`)`
Set the timeout for newly created sessions for this Context object to *timeout*. *timeout* must be given in (whole) seconds. The default value is 300 seconds. See the OpenSSL manual for more information (e.g. SSL_CTX_set_timeout(3)).

`set_verify(`*mode, callback*`)`
Set the verification flags for this Context object to *mode* and specify that *callback* should be used for verification callbacks. *mode* should be one of `VERIFY_NONE` and `VERIFY_PEER`. If `VERIFY_PEER` is used, *mode* can be OR:ed with `VERIFY_FAIL_IF_NO_PEER_CERT` and `VERIFY_CLIENT_ONCE` to further control the behaviour. *callback* should take five arguments: A Connection object, an X509 object, and three integer variables, which are in turn potential error number, error depth and return code. *callback* should return true if verification passes and false otherwise.

`set_verify_depth(`*depth*`)`
Set the maximum depth for the certificate chain verification that shall be allowed for this Context object.

`use_certificate(`*cert*`)`
Use the certificate *cert* which has to be a X509 object.

`add_extra_chain_cert(`*cert*`)`
Adds the certificate *cert*, which has to be a X509 object, to the certificate chain presented together with the certificate.

`use_certificate_chain_file(`*file*`)`
Load a certificate chain from *file* which must be PEM encoded.

`use_privatekey(`*pkey*`)`
Use the private key *pkey* which has to be a PKey object.

`use_certificate_file(`*file*[, *format*]`)`
Load the first certificate found in *file*. The certificate must be in the format specified by *format*, which is either `FILETYPE_PEM` or `FILETYPE_ASN1`. The default is `FILETYPE_PEM`.

`use_privatekey_file(`*file*[, *format*]`)`
Load the first private key found in *file*. The private key must be in the format specified by *format*, which is either `FILETYPE_PEM` or `FILETYPE_ASN1`. The default is `FILETYPE_PEM`.

Connection objects

Connection objects have the following methods:

`accept()`
: Call the `accept` method of the underlying socket and set up SSL on the returned socket, using the Context object supplied to this Connection object at creation. Returns a pair `(`*conn*`, `*address*`)`. where *conn* is the new Connection object created, and *address* is as returned by the socket's `accept`.

`bind(`*address*`)`
: Call the `bind` method of the underlying socket.

`close()`
: Call the `close` method of the underlying socket. Note: If you want correct SSL closure, you need to call the `shutdown` method first.

`connect(`*address*`)`
: Call the `connect` method of the underlying socket and set up SSL on the socket, using the Context object supplied to this Connection object at creation.

`connect_ex(`*address*`)`
: Call the `connect_ex` method of the underlying socket and set up SSL on the socket, using the Context object supplied to this Connection object at creation. Note that if the `connect_ex` method of the socket doesn't return 0, SSL won't be initialized.

`do_handshake()`
: Perform an SSL handshake (usually called after `renegotiate` or one of `set_accept_state` or `set_accept_state`). This can raise the same exceptions as `send` and `recv`.

`fileno()`
: Retrieve the file descriptor number for the underlying socket.

`listen(`*backlog*`)`
: Call the `listen` method of the underlying socket.

`get_app_data()`
: Retrieve application data as set by `set_app_data`.

`get_cipher_list()`
: Retrieve the list of ciphers used by the Connection object. WARNING: This API has changed. It used to take an optional parameter and just return a string, but not it returns the entire list in one go.

`get_client_ca_list()`
: Retrieve the list of preferred client certificate issuers sent by the server as `OpenSSL.crypto.X509Name` objects.

  If this is a client `Connection`, the list will be empty until the connection with the server is established.

  If this is a server `Connection`, return the list of certificate authorities that will be sent or has been sent to the client, as controlled by this `Connection`'s `Context`.

  New in version 0.10.

`get_context()`
: Retrieve the Context object associated with this Connection.

`get_peer_certificate()`
: Retrieve the other side's certificate (if any)

`getpeername()`
: Call the `getpeername` method of the underlying socket.

`getsockname()`
: Call the `getsockname` method of the underlying socket.

`getsockopt(`*level, optname*[*, buflen*]`)`
: Call the `getsockopt` method of the underlying socket.

`pending()`
: Retrieve the number of bytes that can be safely read from the SSL buffer (*not* the underlying transport buffer).

`recv(`*bufsize*`)`
: Receive data from the Connection. The return value is a string representing the data received. The maximum amount of data to be received at once, is specified by *bufsize*.

`bio_write(`*bytes*`)`
: If the Connection was created with a memory BIO, this method can be used to add bytes to the read end of that memory BIO. The Connection can then read the bytes (for example, in response to a call to `recv`).

`renegotiate()`
: Renegotiate the SSL session. Call this if you wish to change cipher suites or anything like that.

`send(`*string*`)`
: Send the *string* data to the Connection.

`bio_read(`*bufsize*`)`
: If the Connection was created with a memory BIO, this method can be used to read bytes from the write end of that memory BIO. Many Connection methods will add bytes which must be read in this manner or the buffer will eventually fill up and the Connection will be able to take no further actions.

`sendall(`*string*`)`
: Send all of the *string* data to the Connection. This calls `send` repeatedly until all data is sent. If an error occurs, it's impossible to tell how much data has been sent.

`set_accept_state()`
: Set the connection to work in server mode. The handshake will be handled automatically by read/write.

`set_app_data(`*data*`)`
: Associate *data* with this Connection object. *data* can be retrieved later using the `get_app_data` method.

`set_connect_state()`
: Set the connection to work in client mode. The handshake will be handled automatically by read/write.

`setblocking(`*flag*`)`
: Call the `setblocking` method of the underlying socket.

`setsockopt(`*level, optname, value*`)`
: Call the `setsockopt` method of the underlying socket.

`shutdown()`
: Send the shutdown message to the Connection. Returns true if the shutdown message exchange is completed and false otherwise (in which case you call `recv()` or `send()` when the connection becomes readable/writeable.

`get_shutdown()`
: Get the shutdown state of the Connection. Returns a bitvector of either or both of *SENT_SHUTDOWN* and *RECEIVED_SHUTDOWN*.

`set_shutdown(`*state*`)`
: Set the shutdown state of the Connection. *state* is a bitvector of either or both of *SENT_SHUTDOWN* and *RECEIVED_SHUTDOWN*.

`sock_shutdown(`*how*`)`
: Call the `shutdown` method of the underlying socket.

`bio_shutdown()`
: If the Connection was created with a memory BIO, this method can be used to indicate that "end of

file" has been reached on the read end of that memory BIO.

`state_string()`
: Retrieve a verbose string detailing the state of the Connection.

`client_random()`
: Retrieve the random value used with the client hello message.

`server_random()`
: Retrieve the random value used with the server hello message.

`master_key()`
: Retrieve the value of the master key for this session.

`want_read()`
: Checks if more data has to be read from the transport layer to complete an operation.

`want_write()`
: Checks if there is data to write to the transport layer to complete an operation.

4 Internals

We ran into three main problems developing this: Exceptions, callbacks and accessing socket methods. This is what this chapter is about.

4.1 Exceptions

We realized early that most of the exceptions would be raised by the I/O functions of OpenSSL, so it felt natural to mimic OpenSSL's error code system, translating them into Python exceptions. This naturally gives us the exceptions `SSL.ZeroReturnError`, `SSL.WantReadError`, `SSL.WantWriteError`, `SSL.WantX509LookupError` and `SSL.SysCallError`.

For more information about this, see section 3.3.

4.2 Callbacks

There are a number of problems with callbacks. First of all, OpenSSL is written as a C library, it's not meant to have Python callbacks, so a way around that is needed. Another problem is thread support. A lot of the OpenSSL I/O functions can block if the socket is in blocking mode, and then you want other Python threads to be able to do other things. The real trouble is if you've released the global CPython interpreter lock to do a potentially blocking operation, and the operation calls a callback. Then we must take the GIL back, since calling Python APIs without holding it is not allowed.

There are two solutions to the first problem, both of which are necessary. The first solution to use is if the C callback allows "userdata" to be passed to it (an arbitrary pointer normally). This is great! We can set our Python function object as the real userdata and emulate userdata for the Python function in another way. The other solution can be used if an object with an "app_data" system always is passed to the callback. For example, the SSL object in OpenSSL has app_data functions and in e.g. the verification callbacks, you can retrieve the related SSL object. What we do is to set our wrapper `Connection` object as app_data for the SSL object, and we can easily find the Python callback.

The other problem is solved using thread local variables. Whenever the GIL is released before calling into an OpenSSL API, the PyThreadState pointer returned by `PyEval_SaveState` is stored in a global thread local variable (using Python's own TLS API, `PyThread_set_key_value`). When it is necessary to re-acquire the GIL, either after the OpenSSL API returns or in a C callback invoked by that OpenSSL API, the value of the thread local variable is retrieved (`PyThread_get_key_value`) and used to re-acquire the GIL. This allows

Python threads to execute while OpenSSL APIs are running and allows use of any particular pyOpenSSL object from any Python thread, since there is no per-thread state associated with any of these objects and since OpenSSL is threadsafe (as long as properly initialized, as pyOpenSSL initializes it).

4.3 Acessing Socket Methods

We quickly saw the benefit of wrapping socket methods in the `SSL.Connection` class, for an easy transition into using SSL. The problem here is that the `socket` module lacks a C API, and all the methods are declared static. One approach would be to have `OpenSSL` as a submodule to the `socket` module, placing all the code in 'socketmodule.c', but this is obviously not a good solution, since you might not want to import tonnes of extra stuff you're not going to use when importing the `socket` module. The other approach is to somehow get a pointer to the method to be called, either the C function, or a callable Python object. This is not really a good solution either, since there's a lot of lookups involved.

The way it works is that you have to supply a "`socket`-like" transport object to the `SSL.Connection`. The only requirement of this object is that it has a `fileno()` method that returns a file descriptor that's valid at the C level (i.e. you can use the system calls read and write). If you want to use the `connect()` or `accept()` methods of the `SSL.Connection` object, the transport object has to supply such methods too. Apart from them, any method lookups in the `SSL.Connection` object that fail are passed on to the underlying transport object.

Future changes might be to allow Python-level transport objects, that instead of having `fileno()` methods, have `read()` and `write()` methods, so more advanced features of Python can be used. This would probably entail some sort of OpenSSL "BIOs", but converting Python strings back and forth is expensive, so this shouldn't be used unless necessary. Other nice things would be to be able to pass in different transport objects for reading and writing, but then the `fileno()` method of `SSL.Connection` becomes virtually useless. Also, should the method resolution be used on the read-transport or the write-transport?