Python OpenSSL Manual

Release 0.6

Martin Sjögren

martin@strakt.com

13th August 2004

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 **2**

2 Building and Installing **2**
2.1 Building the Module on a Unix System . . . 2
2.2 Building the Module on a Windows System . . . 3

3 `OpenSSL` — Python interface to OpenSSL **3**
3.1 `crypto` — Generic cryptographic module . . . 3
X509 objects . . . 5
X509Name objects . . . 6
X509Req objects . . . 6
X509Store objects . . . 6
PKey objects . . . 7
PKCS7 objects . . . 7
PKCS12 objects . . . 7
X509Extension objects . . . 7
NetscapeSPKI objects . . . 7
3.2 `rand` — An interface to the OpenSSL pseudo random number generator . . . 8
3.3 `SSL` — An interface to the SSL-specific parts of OpenSSL . . . 8
Context objects . . . 10
Connection objects . . . 11

4 Internals **13**
4.1 Exceptions . . . 13
4.2 Callbacks . . . 13
4.3 Acessing Socket Methods . . . 14

1 Introduction

The reason this module exists at all is that the SSL support in the socket module in the Python 2.1 distribution (which is what we used, of course I cannot speak for later versions) is severely limited.

When asking about SSL on the comp.lang.python newsgroup (or on python-list@python.org) people usually pointed you to the M2Crypto package. The M2Crypto.SSL module does implement a lot of OpenSSL's functionality but unfortunately its error handling system does not seem to be finished, especially for non-blocking I/O. I think that much of the reason for this is that M2Crypto[1] is developed using SWIG[2]. This makes it awkward to create functions that e.g. can return both an integer and NULL since (as far as I know) you basically write C functions and SWIG makes wrapper functions that parses the Python argument list and calls your C function, and finally transforms your return value to a Python object.

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://www.post1.com/home/ngps/m2/
[2] See http://swig.sourceforge.net/

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`
: A Python type object representing the X509 object type.

`X509()`
: Factory function that creates an X509 object.

`X509NameType`
: A Python type object representing the X509Name object type.

`X509Name(`*x509name*`)`
: Factory function that creates a copy of *x509name*.

`X509ReqType`
: A Python type object representing the X509Req object type.

`X509Req()`
: Factory function that creates an X509Req object.

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

`PKeyType`
: A Python type object representing the PKey object type.

`PKey()`
: Factory function that creates a PKey object.

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

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

`X509ExtensionType`
: A Python type object representing the X509Extension object type.

`X509Extension(`*typename, critical, value*`)`
: Factory function that creates a X509Extension object.

`NetscapeSPKIType`
: A Python type object representing the NetscapeSPKI object type.

`NetscapeSPKI(`[*enc*]`)`
: Factory function that creates a NetscapeSPKI object. 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.

X509 objects

X509 objects have the following methods:

`get_issuer()`
Return a *borrowed reference to a X509Name object representing the issuer of the certificate. When the corresponding X509 or X509Req object is destroyed, this object will be invalid!*

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

`get_serial_number()`
Return the certificate serial number.

`get_subject()`
Return a *borrowed reference to a X509Name object representing the subject of the certificate. When the corresponding X509 or X509Req object is destroyed, this object will be invalid!*

`get_version()`
Return the certificate version.

`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.

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

X509Name objects

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 `stateOrProvinceNameů`

`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 a *borrowed reference to a X509Name object representing the subject of the certificate. When the corresponding X509 or X509Req object is destroyed, this object will be invalid!*

`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*.

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:

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

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

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

X509Extension objects

X509Extension objects currently only have one method:

`get_critical()`
: Return the critical field 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*.

`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.

`egd(`*path*`[`*, bytes*`])`
: Query the Entropy Gathering Daemon[3] on socket *path* for *bytes* bytes of random data and and uses `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.

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`

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

`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`
: A Python type object representing the Context object type.

`Context(`*method*`)`
: Factory function that creates a new Context object given an SSL method. The method should be `SSLv2_METHOD`, `SSLv3_METHOD`, `SSLv23_METHOD` or `TLSv1_METHOD`.

`ConnectionType`
: A Python type object representing the Connection object type.

`Connection(`*context, socket*`)`
: Factory fucnction that creates a new Connection object given an SSL context and a socket [4] object.

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.

exception `WantWriteError`
: See `WantReadError`.

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,

[4] 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!

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:

`check_privatekey()`
Check if the private key (loaded with `use_privatekey[_file]`) matches the certificate (loaded with `use_certificate[_file]`). Returns true if they match, false 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_mode`.

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

`load_verify_locations(`*pemfile*`)`
Specify where CA certificates for verification purposes are located. These are trusted certificates. Note that the certificates have to be in PEM format.

`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 should take a boolean argument *repeat* and an arbitrary argument *data* and return the passphrase entered by the user. If *repeat* is true then *callback* should ask for the passphrase twice and make sure that the two entries are equal. The *data* argument is the *userdata* variable passed to the `set_passwd_cb` method. 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.

`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_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*$\left[\right.$, *buflen*$\left.\right]$`)`
: Call the `getsockopt` method of the underlying socket.

`pending()`
: Retrieve the number of bytes that can be safely read from the SSL 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*.

`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.

`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.

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

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

`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 thread lock to do a potentially

blocking operation, and the operation calls a callback. Then we must take the thread lock back[5].

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 also partially solved by app_data. Since we're associating our wrapper objects with the "real" objects, we can easily access data from the `Connection` object. The solution then is to simply include a `PyThreadState` variable in the `Connection` declaration, and write macros similar to `Py_BEGIN_ALLOW_THREADS` and `Py_END_ALLOW_THREADS` that allows specifying of the `PyThreadState` variable to use. Now we can simply "begin allow threads" before a potentially blocking operation, and "end allow threads" before calling a callback.

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?

[5] I'm not sure why this is necessary, but otherwise I get a segmentation violation on `PyEval_CallObject`