pyOpenSSL Documentation

Release 0.15.1

Jean-Paul Calderone

November 07, 2015

Contents

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:

CHAPTER 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) 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.

CHAPTER 2

`OpenSSL` — Python interface to OpenSSL

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

2.1 `crypto` — Generic cryptographic module

`OpenSSL.crypto.`**`X509Type`**
See *`X509`*.

class `OpenSSL.crypto.`**`X509`**
A class representing X.509 certificates.

`OpenSSL.crypto.`**`X509NameType`**
See *`X509Name`*.

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

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

`OpenSSL.crypto.`**`X509ReqType`**
See *`X509Req`*.

class `OpenSSL.crypto.`**`X509Req`**
A class representing X.509 certificate requests.

`OpenSSL.crypto.`**`X509StoreType`**
See `X509Store`

`OpenSSL.crypto.`**`X509StoreContext`**
A class representing the X.509 store context.

`OpenSSL.crypto.`**`PKeyType`**
See *`PKey`*.

class `OpenSSL.crypto.`**`PKey`**
A class representing DSA or RSA keys.

`OpenSSL.crypto.`**`PKCS7Type`**
A Python type object representing the PKCS7 object type.

`OpenSSL.crypto.`**`PKCS12Type`**
A Python type object representing the PKCS12 object type.

`OpenSSL.crypto.`**`X509ExtensionType`**
: See `X509Extension`.

class `OpenSSL.crypto.`**`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_EXTI for *typename* strings and their options. Optional parameters *subject* and *issuer* must be X509 objects.

`OpenSSL.crypto.`**`NetscapeSPKIType`**
: See `NetscapeSPKI`.

class `OpenSSL.crypto.`**`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.

class `OpenSSL.crypto.`**`CRL`**
: A class representing Certifcate Revocation List objects.

class `OpenSSL.crypto.`**`Revoked`**
: A class representing Revocation objects of CRL.

`OpenSSL.crypto.`**`FILETYPE_PEM`**
`OpenSSL.crypto.`**`FILETYPE_ASN1`**
: File type constants.

`OpenSSL.crypto.`**`TYPE_RSA`**
`OpenSSL.crypto.`**`TYPE_DSA`**
: Key type constants.

exception `OpenSSL.crypto.`**`Error`**
: Generic exception used in the `crypto` module.

`OpenSSL.crypto.`**`get_elliptic_curves`**()
: Return a set of objects representing the elliptic curves supported in the OpenSSL build in use.

  The curve objects have a `unicode name` attribute by which they identify themselves.

  The curve objects are useful as values for the argument accepted by `Context.set_tmp_ecdh()` to specify which elliptical curve should be used for ECDHE key exchange.

`OpenSSL.crypto.`**`get_elliptic_curve`**()
: Return a single curve object selected by name.

  See `get_elliptic_curves()` for information about curve objects.

  If the named curve is not supported then `ValueError` is raised.

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

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

`OpenSSL.crypto.`**`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.

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

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

`OpenSSL.crypto.`**`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.

`OpenSSL.crypto.`**`load_crl`**`(`*type*, *buffer*`)`
Load Certificate Revocation List (CRL) data from a string *buffer*. *buffer* encoded with the type *type*. The type *type* must either *`FILETYPE_PEM`* or *`FILETYPE_ASN1`*).

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

`OpenSSL.crypto.`**`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()`.

`OpenSSL.crypto.`**`sign`**`(`*key*, *data*, *digest*`)`
Sign a data string using the given key and message digest.

key is a *`PKey`* instance. *data* is a `str` instance. *digest* is a `str` naming a supported message digest type, for example `sha1`.

New in version 0.11.

`OpenSSL.crypto.`**`verify`**`(`*certificate*, *signature*, *data*, *digest*`)`
Verify the signature for a data string.

certificate is a *`X509`* instance corresponding to the private key which generated the signature. *signature* is a *str* instance giving the signature itself. *data* is a *str* instance giving the data to which the signature applies. *digest* is a *str* instance naming the message digest type of the signature, for example `sha1`.

New in version 0.11.

2.1.1 X509 objects

X509 objects have the following methods:

`X509.`**`get_issuer`**`()`
Return an X509Name object representing the issuer of the certificate.

`X509.`**`get_pubkey`**`()`
Return a *`PKey`* object representing the public key of the certificate.

`X509.`**`get_serial_number`**`()`
Return the certificate serial number.

`X509.`**`get_signature_algorithm`**`()`
Return the signature algorithm used in the certificate. If the algorithm is undefined, raise `ValueError`.

New in version 0.13.

`X509.`**`get_subject`**`()`
Return an *`X509Name`* object representing the subject of the certificate.

`X509.`**`get_version`**`()`
Return the certificate version.

X509.**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.

X509.**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.

X509.**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
```

X509.**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
```

X509.**gmtime_adj_notBefore**(*time*)
Adjust the timestamp (in GMT) when the certificate starts being valid.

X509.**gmtime_adj_notAfter**(*time*)
Adjust the timestamp (in GMT) when the certificate stops being valid.

X509.**has_expired**()
Checks the certificate's time stamp against current time. Returns true if the certificate has expired and false otherwise.

X509.**set_issuer**(*issuer*)
Set the issuer of the certificate to *issuer*.

X509.**set_pubkey**(*pkey*)
Set the public key of the certificate to *pkey*.

X509.**set_serial_number**(*serialno*)
Set the serial number of the certificate to *serialno*.

X509.**set_subject**(*subject*)
Set the subject of the certificate to *subject*.

X509.**set_version**(*version*)
Set the certificate version to *version*.

X509.**sign**(*pkey*, *digest*)
Sign the certificate, using the key *pkey* and the message digest algorithm identified by the string *digest*.

X509.**subject_name_hash**()
Return the hash of the certificate subject.

X509.**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"`.

X509.**add_extensions**(*extensions*)
Add the extensions in the sequence *extensions* to the certificate.

X509.**get_extension_count**()
Return the number of extensions on this certificate.

New in version 0.12.

X509.**get_extension**(*index*)
Retrieve the extension on this certificate at the given index.

Extensions on a certificate are kept in order. The index parameter selects which extension will be returned. The returned object will be an `X509Extension` instance.

New in version 0.12.

2.1.2 X509Name objects

X509Name objects have the following methods:

X509Name.**hash**()
Return an integer giving the first four bytes of the MD5 digest of the DER representation of the name.

X509Name.**der**()
Return a string giving the DER representation of the name.

X509Name.**get_components**()
Return a list of two-tuples of strings giving the components of the name.

X509Name objects have the following members:

X509Name.**countryName**
The country of the entity. `C` may be used as an alias for `countryName`.

X509Name.**stateOrProvinceName**
The state or province of the entity. `ST` may be used as an alias for `stateOrProvinceName`.

X509Name.**localityName**
The locality of the entity. `L` may be used as an alias for `localityName`.

X509Name.**organizationName**
The organization name of the entity. `O` may be used as an alias for `organizationName`.

X509Name.**organizationalUnitName**
The organizational unit of the entity. `OU` may be used as an alias for `organizationalUnitName`.

X509Name.**commonName**
The common name of the entity. `CN` may be used as an alias for `commonName`.

X509Name.**emailAddress**
The e-mail address of the entity.

2.1.3 X509Req objects

X509Req objects have the following methods:

`X509Req.`**`get_pubkey`**`()`
Return a `PKey` object representing the public key of the certificate request.

`X509Req.`**`get_subject`**`()`
Return an `X509Name` object representing the subject of the certificate.

`X509Req.`**`set_pubkey`**`(`*pkey*`)`
Set the public key of the certificate request to *pkey*.

`X509Req.`**`sign`**`(`*pkey*, *digest*`)`
Sign the certificate request, using the key *pkey* and the message digest algorithm identified by the string *digest*.

`X509Req.`**`verify`**`(`*pkey*`)`
Verify a certificate request using the public key *pkey*.

`X509Req.`**`set_version`**`(`*version*`)`
Set the version (RFC 2459, 4.1.2.1) of the certificate request to *version*.

`X509Req.`**`get_version`**`()`
Get the version (RFC 2459, 4.1.2.1) of the certificate request.

`X509Req.`**`get_extensions`**`()`
Get extensions to the request.

New in version 0.15.

2.1.4 X509Store objects

The X509Store object has currently just one method:

`X509Store.`**`add_cert`**`(`*cert*`)`
Add the certificate *cert* to the certificate store.

2.1.5 X509StoreContextError objects

The X509StoreContextError is an exception raised from *X509StoreContext.verify_certificate* in circumstances where a certificate cannot be verified in a provided context.

The certificate for which the verification error was detected is given by the `certificate` attribute of the exception instance as a `X509` instance.

Details about the verification error are given in the exception's `args` attribute.

2.1.6 X509StoreContext objects

The X509StoreContext object is used for verifying a certificate against a set of trusted certificates.

`X509StoreContext.`**`verify_certificate`**`()`
Verify a certificate in the context of this initialized *X509StoreContext*. On error, raises *X509StoreContextError*, otherwise does nothing.

New in version 0.15.

2.1.7 PKey objects

The PKey object has the following methods:

`PKey.`**`bits`**`()`
Return the number of bits of the key.

`PKey.`**`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*.

`PKey.`**`type`**`()`
Return the type of the key.

`PKey.`**`check`**`()`
Check the consistency of this key, returning True if it is consistent and raising an exception otherwise. This is only valid for RSA keys. See the OpenSSL RSA_check_key man page for further limitations.

2.1.8 PKCS7 objects

PKCS7 objects have the following methods:

`PKCS7.`**`type_is_signed`**`()`
FIXME

`PKCS7.`**`type_is_enveloped`**`()`
FIXME

`PKCS7.`**`type_is_signedAndEnveloped`**`()`
FIXME

`PKCS7.`**`type_is_data`**`()`
FIXME

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

2.1.9 PKCS12 objects

PKCS12 objects have the following methods:

`PKCS12.`**`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()`.

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

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

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

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

`PKCS12.`**`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.

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

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

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

2.1.10 X509Extension objects

X509Extension objects have several methods:

`X509Extension.`**`get_critical`**`()`
Return the critical field of the extension object.

`X509Extension.`**`get_short_name`**`()`
Retrieve the short descriptive name for this extension.

The result is a byte string like `basicConstraints`.

New in version 0.12.

`X509Extension.`**`get_data`**`()`
Retrieve the data for this extension.

The result is the ASN.1 encoded form of the extension data as a byte string.

New in version 0.12.

2.1.11 NetscapeSPKI objects

NetscapeSPKI objects have the following methods:

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

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

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

`NetscapeSPKI.`**`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"`.

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

2.1.12 CRL objects

CRL objects have the following methods:

`CRL.`**`add_revoked`**`(`*revoked*`)`
Add a Revoked object to the CRL, by value not reference.

`CRL.`**`export`**`(`*cert, key[, type=FILETYPE_PEM][, days=100][, digest=b'md5']*`)`
Use *cert* and *key* to sign the CRL and return the CRL as a string. *days* is the number of days before the next CRL is due. *digest* is the algorithm that will be used to sign CRL.

`CRL.`**`get_revoked`**`()`
Return a tuple of Revoked objects, by value not reference.

2.1.13 Revoked objects

Revoked objects have the following methods:

`Revoked.`**`all_reasons`**`()`
Return a list of all supported reasons.

`Revoked.`**`get_reason`**`()`
Return the revocation reason as a str. Can be None, which differs from "Unspecified".

`Revoked.`**`get_rev_date`**`()`
Return the revocation date as a str. The string is formatted as an ASN1 GENERALIZEDTIME.

`Revoked.`**`get_serial`**`()`
Return a str containing a hex number of the serial of the revoked certificate.

`Revoked.`**`set_reason`**`(`*reason*`)`
Set the revocation reason. *reason* must be None or a string, but the values are limited. Spaces and case are ignored. See `all_reasons()`.

`Revoked.`**`set_rev_date`**`(`*date*`)`
Set the revocation date. The string is formatted as an ASN1 GENERALIZEDTIME.

`Revoked.`**`set_serial`**`(`*serial*`)`
serial is a string containing a hex number of the serial of the revoked certificate.

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

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

`OpenSSL.rand.`**`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**.

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

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

`OpenSSL.rand.`**`cleanup`**`()`
Erase the memory used by the PRNG.

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

`OpenSSL.rand.`**`egd`**`(`*path*[, *bytes*]`)`
: Query the Entropy Gathering Daemon on socket *path* for *bytes* bytes of random data and uses `add()` to seed the PRNG. The default value of *bytes* is 255.

`OpenSSL.rand.`**`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.

`OpenSSL.rand.`**`screen`**`()`
: Add the current contents of the screen to the PRNG state.

  Availability: Windows.

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

`OpenSSL.rand.`**`status`**`()`
: Returns true if the PRNG has been seeded with enough data, and false otherwise.

`OpenSSL.rand.`**`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 `OpenSSL.rand.`**`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.

2.3 `SSL` — An interface to the SSL-specific parts of OpenSSL

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

`OpenSSL.SSL.`**`SSLv2_METHOD`**
`OpenSSL.SSL.`**`SSLv3_METHOD`**
`OpenSSL.SSL.`**`SSLv23_METHOD`**
`OpenSSL.SSL.`**`TLSv1_METHOD`**
`OpenSSL.SSL.`**`TLSv1_1_METHOD`**
`OpenSSL.SSL.`**`TLSv1_2_METHOD`**
: These constants represent the different SSL methods to use when creating a context object. If the underlying OpenSSL build is missing support for any of these protocols, constructing a `Context` using the corresponding `*_METHOD` will raise an exception.

`OpenSSL.SSL.`**`VERIFY_NONE`**
`OpenSSL.SSL.`**`VERIFY_PEER`**
`OpenSSL.SSL.`**`VERIFY_FAIL_IF_NO_PEER_CERT`**
: These constants represent the verification mode used by the Context object's `set_verify()` method.

`OpenSSL.SSL.`**`FILETYPE_PEM`**
`OpenSSL.SSL.`**`FILETYPE_ASN1`**
: File type constants used with the `use_certificate_file()` and `use_privatekey_file()` methods of Context objects.

`OpenSSL.SSL.`**`OP_SINGLE_DH_USE`**
: Constant used with `set_options()` of Context objects.

  When this option is used, a new key will always be created when using ephemeral Diffie-Hellman.

`OpenSSL.SSL.`**`OP_EPHEMERAL_RSA`**

Constant used with `set_options()` of Context objects.

When this option is used, ephemeral RSA keys will always be used when doing RSA operations.

`OpenSSL.SSL.`**`OP_NO_TICKET`**

Constant used with `set_options()` of Context objects.

When this option is used, the session ticket extension will not be used.

`OpenSSL.SSL.`**`OP_NO_COMPRESSION`**

Constant used with `set_options()` of Context objects.

When this option is used, compression will not be used.

`OpenSSL.SSL.`**`OP_NO_SSLv2`**
`OpenSSL.SSL.`**`OP_NO_SSLv3`**
`OpenSSL.SSL.`**`OP_NO_TLSv1`**
`OpenSSL.SSL.`**`OP_NO_TLSv1_1`**
`OpenSSL.SSL.`**`OP_NO_TLSv1_2`**

Constants used with `set_options()` of Context objects.

Each of these options disables one version of the SSL/TLS protocol. This is interesting if you're using e.g. *`SSLv23_METHOD`* to get an SSLv2-compatible handshake, but don't want to use SSLv2. If the underlying OpenSSL build is missing support for any of these protocols, the `OP_NO_*` constant may be undefined.

`OpenSSL.SSL.`**`SSLEAY_VERSION`**
`OpenSSL.SSL.`**`SSLEAY_CFLAGS`**
`OpenSSL.SSL.`**`SSLEAY_BUILT_ON`**
`OpenSSL.SSL.`**`SSLEAY_PLATFORM`**
`OpenSSL.SSL.`**`SSLEAY_DIR`**

Constants used with *`SSLeay_version()`* to specify what OpenSSL version information to retrieve. See the man page for the *`SSLeay_version()`* C API for details.

`OpenSSL.SSL.`**`SESS_CACHE_OFF`**
`OpenSSL.SSL.`**`SESS_CACHE_CLIENT`**
`OpenSSL.SSL.`**`SESS_CACHE_SERVER`**
`OpenSSL.SSL.`**`SESS_CACHE_BOTH`**
`OpenSSL.SSL.`**`SESS_CACHE_NO_AUTO_CLEAR`**
`OpenSSL.SSL.`**`SESS_CACHE_NO_INTERNAL_LOOKUP`**
`OpenSSL.SSL.`**`SESS_CACHE_NO_INTERNAL_STORE`**
`OpenSSL.SSL.`**`SESS_CACHE_NO_INTERNAL`**

Constants used with *`Context.set_session_cache_mode()`* to specify the behavior of the session cache and potential session reuse. See the man page for the `SSL_CTX_set_session_cache_mode()` C API for details.

New in version 0.14.

`OpenSSL.SSL.`**`OPENSSL_VERSION_NUMBER`**

An integer giving the version number of the OpenSSL library used to build this version of pyOpenSSL. See the man page for the *`SSLeay_version()`* C API for details.

`OpenSSL.SSL.`**`SSLeay_version`**`(`*type*`)`

Retrieve a string describing some aspect of the underlying OpenSSL version. The type passed in should be one of the `SSLEAY_*` constants defined in this module.

`OpenSSL.SSL.`**`ContextType`**

See *`Context`*.

class `OpenSSL.SSL.`**`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*, *TLSv1_METHOD*, *TLSv1_1_METHOD*, or *TLSv1_2_METHOD*.

class `OpenSSL.SSL.`**`Session`**

A class representing an SSL session. A session defines certain connection parameters which may be re-used to speed up the setup of subsequent connections.

New in version 0.14.

`OpenSSL.SSL.`**`ConnectionType`**

See *Connection*.

class `OpenSSL.SSL.`**`Connection`**(*context*, *socket*)

A class representing SSL connections.

context should be an instance of *Context* and *socket* should be a socket [1] 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 `OpenSSL.SSL.`**`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 `OpenSSL.SSL.`**`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 `OpenSSL.SSL.`**`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 `OpenSSL.SSL.`**`WantWriteError`**

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

exception `OpenSSL.SSL.`**`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 `OpenSSL.SSL.`**`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)*.

[1] 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!

2.3.1 Context objects

Context objects have the following methods:

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

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

`Context.`**`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.

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

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

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

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

`Context.`**`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.X509Name`'s.

New in version 0.10.

`Context.`**`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.

`Context.`**`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`.

`Context.`**`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.

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

`Context.`**`set_tmp_ecdh`**`(`*curve*`)`
Select a curve to use for ECDHE key exchange.

The valid values of *curve* are the objects returned by `OpenSSL.crypto.get_elliptic_curves()` or `OpenSSL.crypto.get_elliptic_curve()`.

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

`Context.`**`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)`)

`Context.`**`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.

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

`Context.`**`set_mode`**`(`*mode*`)`
Add SSL mode. Modes you have set before are not cleared! This method should be used with the `MODE_*` constants.

`Context.`**`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).

`Context.`**`set_session_cache_mode`**`(`*mode*`)`
Set the behavior of the session cache used by all connections using this Context. The previously set mode is returned. See `SESS_CACHE_*` for details about particular modes.

New in version 0.14.

`Context.`**`get_session_cache_mode`**`()`
Get the current session cache mode.

New in version 0.14.

`Context.`**`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.

`Context.`**`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)`*).

`Context.`**`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.

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

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

`Context.`**`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.

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

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

`Context.`**`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`.

`Context.`**`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`.

`Context.`**`set_tlsext_servername_callback`**(*callback*)
Specify a one-argument callable to use as the TLS extension server name callback. When a connection using the server name extension is made using this context, the callback will be invoked with the `Connection` instance.

New in version 0.13.

`Context.`**`set_npn_advertise_callback`**(*callback*)
Specify a callback function that will be called when offering Next Protocol Negotiation as a server.

callback should be the callback function. It will be invoked with one argument, the `Connection` instance. It should return a list of bytestrings representing the advertised protocols, like `[b'http/1.1', b'spdy/2']`.

New in version 0.15.

`Context.set_npn_select_callback(callback):`
Specify a callback function that will be called when a server offers Next Protocol Negotiation options.

callback should be the callback function. It will be invoked with two arguments: the `Connection`, and a list of offered protocols as bytestrings, e.g. `[b'http/1.1', b'spdy/2']`. It should return one of those bytestrings, the chosen protocol.

New in version 0.15.

`Context.`**`set_alpn_protos`**(*protos*)
Specify the protocols that the client is prepared to speak after the TLS connection has been negotiated using Application Layer Protocol Negotiation.

protos should be a list of protocols that the client is offering, each as a bytestring. For example, `[b'http/1.1', b'spdy/2']`.

`Context.`**`set_alpn_select_callback`**(*callback*)
Specify a callback function that will be called on the server when a client offers protocols using Application Layer Protocol Negotiation.

callback should be the callback function. It will be invoked with two arguments: the `Connection` and a list of offered protocols as bytestrings, e.g. `[b'http/1.1', b'spdy/2']`. It should return one of these bytestrings, the chosen protocol.

2.3.2 Session objects

Session objects have no methods.

2.3.3 Connection objects

Connection objects have the following methods:

`Connection.`**`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()`.

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

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

`Connection.`**`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.

`Connection.`**`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.

`Connection.`**`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()`.

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

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

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

`Connection.`**`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.

`Connection.`**`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.

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

`Connection.`**`set_context`**`(`*context*`)`
: Specify a replacement Context object for this Connection.

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

`Connection.`**`get_peer_cert_chain`**`()`
: Retrieve the tuple of the other side's certificate chain (if any)

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

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

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

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

`Connection.`**`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*.

`Connection.`**`recv_into`**`(`*buffer*[, *nbytes*[, *flags*]]`)`
: Receive data from the Connection and copy it directly into the provided buffer. The return value is the number of bytes read from the connection. The maximum amount of data to be received at once is specified by *nbytes*. *flags* is accepted for compatibility with `socket.recv_into` but its value is ignored.

`Connection.`**`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()`*).

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

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

`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.

`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.

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

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

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

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

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

`Connection.`**`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.

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

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

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

`Connection.`**`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.

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

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

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

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

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

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

`Connection.`**`set_tlsext_host_name`**`(`*name*`)`
Specify the byte string to send as the server name in the client hello message.

New in version 0.13.

`Connection.`**`get_servername`**`()`
Get the value of the server name received in the client hello message.

New in version 0.13.

`Connection.`**`get_session`**`()`
Get a `Session` instance representing the SSL session in use by the connection, or `None` if there is no session.

New in version 0.14.

`Connection.`**`set_session`**`(`*session*`)`
Set a new SSL session (using a `Session` instance) to be used by the connection.

New in version 0.14.

`Connection.`**`get_finished`**`()`
Obtain latest TLS Finished message that we sent, or `None` if handshake is not completed.

New in version 0.15.

`Connection.`**`get_peer_finished`**`()`
Obtain latest TLS Finished message that we expected from peer, or `None` if handshake is not completed.

New in version 0.15.

`Connection.`**`get_cipher_name`**`()`
Obtain the name of the currently used cipher.

New in version 0.15.

`Connection.`**`get_cipher_bits`**`()`

Obtain the number of secret bits of the currently used cipher.

New in version 0.15.

`Connection.`**`get_cipher_version`**`()`

Obtain the protocol name of the currently used cipher.

New in version 0.15.

`Connection.get_next_proto_negotiated():`

Get the protocol that was negotiated by Next Protocol Negotiation. Returns a bytestring of the protocol name. If no protocol has been negotiated yet, returns an empty string.

New in version 0.15.

`Connection.`**`set_alpn_protos`**`(`*protos*`)`

Specify the protocols that the client is prepared to speak after the TLS connection has been negotiated using Application Layer Protocol Negotiation.

protos should be a list of protocols that the client is offering, each as a bytestring. For example, `[b'http/1.1', b'spdy/2']`.

`Connection.`**`get_alpn_proto_negotiated`**`()`

Get the protocol that was negotiated by Application Layer Protocol Negotiation. Returns a bytestring of the protocol name. If no protocol has been negotiated yet, returns an empty string.

CHAPTER 3

Internals

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

3.1 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 *SSL — An interface to the SSL-specific parts of OpenSSL*.

3.2 Callbacks

Callbacks were more of a problem when pyOpenSSL was written in C. Having switched to being written in Python using cffi, callbacks are now straightforward. The problems that originally existed no longer do (if you are interested in the details you can find descriptions of those problems in the version control history for this document).

3.3 Accessing 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?

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index