18.3. `ssl` — SSL wrapper for socket objects¶

New in version 2.6.

This module provides access to Transport Layer Security (often known as “Secure Sockets Layer”) encryption and peer authentication facilities for network sockets, both client-side and server-side. This module uses the OpenSSL library. It is available on all modern Unix systems, Windows, Mac OS X, and probably additional platforms, as long as OpenSSL is installed on that platform.

Note

Some behavior may be platform dependent, since calls are made to the operating system socket APIs. The installed version of OpenSSL may also cause variations in behavior.

This section documents the objects and functions in the `ssl` module; for more general information about TLS, SSL, and certificates, the reader is referred to the documents in the “See Also” section at the bottom.

This module provides a class, `ssl.SSLSocket`, which is derived from the `socket.socket` type, and provides a socket-like wrapper that also encrypts and decrypts the data going over the socket with SSL. It supports additional `read()` and `write()` methods, along with a method, `getpeercert()`, to retrieve the certificate of the other side of the connection, and a method, `cipher()`, to retrieve the cipher being used for the secure connection.

18.3.1. Functions, Constants, and Exceptions¶

exception `ssl.SSLError`¶

Raised to signal an error from the underlying SSL implementation. This signifies some problem in the higher-level encryption and authentication layer that’s superimposed on the underlying network connection. This error is a subtype of `socket.error`, which in turn is a subtype of `IOError`.

`ssl.wrap_socket`(*sock*, *keyfile=None*, *certfile=None*, *server_side=False*, *cert_reqs=CERT_NONE*, *ssl_version={see docs}*, *ca_certs=None*, *do_handshake_on_connect=True*, *suppress_ragged_eofs=True*)¶

Takes an instance `sock` of `socket.socket`, and returns an instance of `ssl.SSLSocket`, a subtype of `socket.socket`, which wraps the underlying socket in an SSL context. For client-side sockets, the context construction is lazy; if the underlying socket isn’t connected yet, the context construction will be performed after `connect()` is called on the socket. For server-side sockets, if the socket has no remote peer, it is assumed to be a listening socket, and the server-side SSL wrapping is automatically performed on client connections accepted via the `accept()` method. `wrap_socket()` may raise `SSLError`.

The `keyfile` and `certfile` parameters specify optional files which contain a certificate to be used to identify the local side of the connection. See the discussion of *Certificates* for more information on how the certificate is stored in the `certfile`.

Often the private key is stored in the same file as the certificate; in this case, only the `certfile` parameter need be passed. If the private key is stored in a separate file, both parameters must be used. If the private key is stored in the `certfile`, it should come before the first certificate in the certificate chain:

```
-----BEGIN RSA PRIVATE KEY-----
... (private key in base64 encoding) ...
-----END RSA PRIVATE KEY-----
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

The parameter `server_side` is a boolean which identifies whether server-side or client-side behavior is desired from this socket.

The parameter `cert_reqs` specifies whether a certificate is required from the other side of the connection, and whether it will be validated if provided. It must be one of the three values `CERT_NONE` (certificates ignored), `CERT_OPTIONAL` (not required, but validated if provided), or `CERT_REQUIRED` (required and validated). If the value of this parameter is not `CERT_NONE`, then the `ca_certs` parameter must point to a file of CA certificates.

The `ca_certs` file contains a set of concatenated “certification authority” certificates, which are used to validate certificates passed from the other end of the connection. See the discussion of *Certificates* for more information about how to arrange the certificates in this file.

The parameter `ssl_version` specifies which version of the SSL protocol to use. Typically, the server chooses a particular protocol version, and the client must adapt to the server’s choice. Most of the versions are not interoperable with the other versions. If not specified, for client-side operation, the default SSL version is SSLv3; for server-side operation, SSLv23. These version selections provide the most compatibility with other versions.

Here's a table showing which versions in a client (down the side) can connect to which versions in a server (along the top):

client / **server**	**SSLv2**	**SSLv3**	**SSLv23**	**TLSv1**
SSLv2	yes	no	yes*	no
SSLv3	yes	yes	yes	no
SSLv23	yes	no	yes	no
TLSv1	no	no	yes	yes

In some older versions of OpenSSL (for instance, 0.9.7l on OS X 10.4), an SSLv2 client could not connect to an SSLv23 server.

The parameter `do_handshake_on_connect` specifies whether to do the SSL handshake automatically after doing a `socket.connect()`, or whether the application program will call it explicitly, by invoking the `SSLSocket.do_handshake()` method. Calling `SSLSocket.do_handshake()` explicitly gives the program control over the blocking behavior of the socket I/O involved in the handshake.

The parameter `suppress_ragged_eofs` specifies how the `SSLSocket.read()` method should signal unexpected EOF from the other end of the connection. If specified as `True` (the default), it returns a normal EOF in response to unexpected EOF errors raised from the underlying socket; if `False`, it will raise the exceptions back to the caller.

`ssl.RAND_status`()¶
Returns True if the SSL pseudo-random number generator has been seeded with 'enough' randomness, and False otherwise. You can use `ssl.RAND_egd()` and `ssl.RAND_add()` to increase the randomness of the pseudo-random number generator.

`ssl.RAND_egd`(*path*)¶

If you are running an entropy-gathering daemon (EGD) somewhere, and `path` is the pathname of a socket connection open to it, this will read 256 bytes of randomness from the socket, and add it to the SSL pseudo-random number generator to increase the security of generated secret keys. This is typically only necessary on systems without better sources of randomness.

See http://egd.sourceforge.net/ or http://prngd.sourceforge.net/ for sources of entropy-gathering daemons.

`ssl.RAND_add`(*bytes*, *entropy*)¶
Mixes the given `bytes` into the SSL pseudo-random number generator. The parameter `entropy` (a float) is a lower bound on the entropy contained in string (so you can always use `0.0`). See **RFC 1750** for more information on sources of entropy.

`ssl.cert_time_to_seconds`(*timestring*)¶

Returns a floating-point value containing a normal seconds-after-the-epoch time value, given the time-string representing the "notBefore" or "notAfter" date from a certificate.

Here's an example:

```
>>> import ssl
>>> ssl.cert_time_to_seconds("May  9 00:00:00 2007 GMT")
1178694000.0
>>> import time
>>> time.ctime(ssl.cert_time_to_seconds("May  9 00:00:00 2007 GMT"))
'Wed May  9 00:00:00 2007'
>>>
```

`ssl.get_server_certificate`(*addr*, *ssl_version=PROTOCOL_SSLv3*, *ca_certs=None*)¶
Given the address `addr` of an SSL-protected server, as a (*hostname*, *port-number*) pair, fetches the server's certificate, and returns it as a PEM-encoded string. If `ssl_version` is specified, uses that version of the SSL protocol to attempt to connect to the server. If `ca_certs` is specified, it should be a file containing a list of root certificates, the same format as used for the same parameter in `wrap_socket()`. The call will attempt to validate the server certificate against that set of root certificates, and will fail if the validation attempt fails.

`ssl.DER_cert_to_PEM_cert`(*DER_cert_bytes*)¶
Given a certificate as a DER-encoded blob of bytes, returns a PEM-encoded string version of the same certificate.

`ssl.PEM_cert_to_DER_cert`(*PEM_cert_string*)¶
Given a certificate as an ASCII PEM string, returns a DER-encoded sequence of bytes for that same certificate.

`ssl.CERT_NONE`¶

Value to pass to the `cert_reqs` parameter to `sslobject()` when no certificates will be required or validated from the other side of the socket connection.

`ssl.CERT_OPTIONAL`¶

Value to pass to the `cert_reqs` parameter to `sslobject()` when no certificates will be required from the other side of the socket connection, but if they are provided, will be validated. Note that use of this setting requires a valid certificate validation file also be passed as a value of the `ca_certs` parameter.

`ssl.CERT_REQUIRED`¶

Value to pass to the `cert_reqs` parameter to `sslobject()` when certificates will be required from the other side of the socket connection. Note that use of this setting requires a valid certificate validation file also be passed as a value of the `ca_certs` parameter.

`ssl.PROTOCOL_SSLv2`¶

Selects SSL version 2 as the channel encryption protocol.

`ssl.PROTOCOL_SSLv23`¶

Selects SSL version 2 or 3 as the channel encryption protocol. This is a setting to use with servers for maximum compatibility with the other end of an SSL connection, but it may cause the specific ciphers chosen for the encryption to be of fairly low quality.

`ssl.PROTOCOL_SSLv3`¶

Selects SSL version 3 as the channel encryption protocol. For clients, this is the maximally compatible SSL variant.

`ssl.PROTOCOL_TLSv1`¶

Selects TLS version 1 as the channel encryption protocol. This is the most modern version, and probably the best choice for maximum protection, if both sides can speak it.

18.3.2. SSLSocket Objects¶

`SSLSocket.read`([*nbytes=1024*])¶

Reads up to `nbytes` bytes from the SSL-encrypted channel and returns them.

`SSLSocket.write`(*data*)¶

Writes the `data` to the other side of the connection, using the SSL channel to encrypt. Returns the number of bytes written.

`SSLSocket.getpeercert`(*binary_form=False*)¶

If there is no certificate for the peer on the other end of the connection, returns `None`.

If the parameter `binary_form` is `False`, and a certificate was received from the peer, this method returns a `dict` instance. If the certificate was not validated, the dict is empty. If the certificate was validated, it returns a dict with the keys `subject` (the principal for which the certificate was issued), and `notAfter` (the time after which the certificate should not be trusted). The certificate was already validated, so the `notBefore` and `issuer` fields are not returned. If a certificate contains an instance of the *Subject Alternative Name* extension (see **RFC 3280**), there will also be a `subjectAltName` key in the dictionary.

The “subject” field is a tuple containing the sequence of relative distinguished names (RDNs) given in the certificate’s data structure for the principal, and each RDN is a sequence of name-value pairs:

```
{'notAfter': 'Feb 16 16:54:50 2013 GMT',
'subject': ((('countryName', u'US'),),
            (('stateOrProvinceName', u'Delaware'),),
            (('localityName', u'Wilmington'),),
            (('organizationName', u'Python Software Foundation'),),
            (('organizationalUnitName', u'SSL'),),
            (('commonName', u'somemachine.python.org'),))}
```

If the `binary_form` parameter is `True`, and a certificate was provided, this method returns the DER-encoded form of the entire certificate as a sequence of bytes, or `None` if the peer did not provide a certificate. This return value is independent of validation; if validation was required (`CERT_OPTIONAL` or `CERT_REQUIRED`), it will have been validated, but if `CERT_NONE` was used to establish the connection, the certificate, if present, will not have been validated.

`SSLSocket.cipher`()¶

Returns a three-value tuple containing the name of the cipher being used, the version of the SSL protocol that defines its use, and the number of secret bits being used. If no connection has been established, returns `None`.

`SSLSocket.do_handshake`()¶

Perform a TLS/SSL handshake. If this is used with a non-blocking socket, it may raise `SSLError` with an `arg[0]` of `SSL_ERROR_WANT_READ` or `SSL_ERROR_WANT_WRITE`, in which case it must be called again until it completes successfully. For example, to simulate the behavior of a blocking socket, one might write:

```
while True:
   try:
       s.do_handshake()
       break
   except ssl.SSLError, err:
       if err.args[0] == ssl.SSL_ERROR_WANT_READ:
           select.select([s], [], [])
       elif err.args[0] == ssl.SSL_ERROR_WANT_WRITE:
           select.select([], [s], [])
       else:
           raise
```

`SSLSocket.unwrap`()¶

Performs the SSL shutdown handshake, which removes the TLS layer from the underlying socket, and returns the underlying socket object. This can be used to go from encrypted operation over a connection to unencrypted. The socket instance returned should always be used for further communication with the other side of the connection, rather than the original socket instance (which may not function properly after the unwrap).

18.3.3. Certificates¶

Certificates in general are part of a public-key / private-key system. In this system, each *principal*, (which may be a machine, or a person, or an organization) is assigned a unique two-part encryption key. One part of the key is public, and is called the *public key*; the other part is kept secret, and is called the *private key*.

The two parts are related, in that if you encrypt a message with one of the parts, you can decrypt it with the other part, and **only** with the other part.

A certificate contains information about two principals. It contains the name of a *subject*, and the subject's public key. It also contains a statement by a second principal, the *issuer*, that the subject is who he claims to be, and that this is indeed the subject's public key. The issuer's statement is signed with the issuer's private key, which only the issuer knows. However, anyone can verify the issuer's statement by finding the issuer's public key, decrypting the statement with it, and comparing it to the other information in the certificate. The certificate also contains information about the time period over which it is valid. This is expressed as two fields, called "notBefore" and "notAfter".

In the Python use of certificates, a client or server can use a certificate to prove who they are. The other side of a network connection can also be required to produce a certificate, and that certificate can be validated to the satisfaction of the client or server that requires such validation. The connection attempt can be set to raise an exception if the validation fails. Validation is done automatically, by the underlying OpenSSL framework; the application need not concern itself with its mechanics. But the application does usually need to provide sets of certificates to allow this process to take place.

Python uses files to contain certificates. They should be formatted as "PEM" (see **RFC 1422**), which is a base-64 encoded form wrapped with a header line and a footer line:

```
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

The Python files which contain certificates can contain a sequence of certificates, sometimes called a *certificate chain*. This chain should start with the specific certificate for the principal who "is" the client or server, and then the certificate for the issuer of that certificate, and then the certificate for the issuer of *that* certificate, and so on up the chain till you get to a certificate which is *self-signed*, that is, a certificate which has the same subject and issuer, sometimes called a *root certificate*. The certificates should just be concatenated together in the certificate file. For example, suppose we had a three certificate chain, from our server certificate to the certificate of the certification authority that signed our server certificate, to the root certificate of the agency which issued the certification authority's certificate:

```
-----BEGIN CERTIFICATE-----
... (certificate for your server)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the certificate for the CA)...
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
... (the root certificate for the CA's issuer)...
-----END CERTIFICATE-----
```

If you are going to require validation of the other side of the connection's certificate, you need to provide a "CA certs" file, filled with the certificate chains for each issuer you are willing to trust. Again, this file just contains these chains concatenated together. For validation, Python will use the first chain it finds in the file which matches.

Some "standard" root certificates are available from various certification authorities: CACert.org, Thawte, Verisign, Positive SSL (used by python.org), Equifax and GeoTrust.

In general, if you are using SSL3 or TLS1, you don't need to put the full chain in your "CA certs" file; you only need the root certificates, and the remote peer is supposed to furnish the other certificates necessary to chain from its certificate to a root certificate. See **RFC 4158** for more discussion of the way in which certification chains can be built.

If you are going to create a server that provides SSL-encrypted connection services, you will need to acquire a certificate for that service. There are many ways of acquiring appropriate certificates, such as buying one from a certification authority. Another common practice is to generate a self-signed certificate. The simplest way to do this is with the OpenSSL package, using something like the following:

```
% openssl req -new -x509 -days 365 -nodes -out cert.pem -keyout cert.pem
Generating a 1024 bit RSA private key
.......++++++
............................++++++
writing new private key to 'cert.pem'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:MyState
Locality Name (eg, city) []:Some City
Organization Name (eg, company) [Internet Widgits Pty Ltd]:My Organization, Inc.
Organizational Unit Name (eg, section) []:My Group
Common Name (eg, YOUR name) []:myserver.mygroup.myorganization.com
```

```
Email Address []:ops@myserver.mygroup.myorganization.com
%
```

The disadvantage of a self-signed certificate is that it is its own root certificate, and no one else will have it in their cache of known (and trusted) root certificates.

18.3.4. Examples¶

18.3.4.1. Testing for SSL support¶

To test for the presence of SSL support in a Python installation, user code should use the following idiom:

```
try:
  import ssl
except ImportError:
  pass
else:
  [ do something that requires SSL support ]
```

18.3.4.2. Client-side operation¶

This example connects to an SSL server, prints the server's address and certificate, sends some bytes, and reads part of the response:

```
import socket, ssl, pprint

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# require a certificate from the server
ssl_sock = ssl.wrap_socket(s,
                           ca_certs="/etc/ca_certs_file",
                           cert_reqs=ssl.CERT_REQUIRED)

ssl_sock.connect(('www.verisign.com', 443))

print repr(ssl_sock.getpeername())
print ssl_sock.cipher()
print pprint.pformat(ssl_sock.getpeercert())

# Set a simple HTTP request -- use httplib in actual code.
ssl_sock.write("""GET / HTTP/1.0\r
Host: www.verisign.com\r\n\r\n""")

# Read a chunk of data.  Will not necessarily
# read all the data returned by the server.
data = ssl_sock.read()

# note that closing the SSLSocket will also close the underlying socket
ssl_sock.close()
```

As of September 6, 2007, the certificate printed by this program looked like this:

```
{'notAfter': 'May  8 23:59:59 2009 GMT',
'subject': ((('serialNumber', u'2497886'),),
            (('1.3.6.1.4.1.311.60.2.1.3', u'US'),),
            (('1.3.6.1.4.1.311.60.2.1.2', u'Delaware'),),
            (('countryName', u'US'),),
            (('postalCode', u'94043'),),
            (('stateOrProvinceName', u'California'),),
            (('localityName', u'Mountain View'),),
            (('streetAddress', u'487 East Middlefield Road'),),
            (('organizationName', u'VeriSign, Inc.'),),
            (('organizationalUnitName',
              u'Production Security Services'),),
            (('organizationalUnitName',
              u'Terms of use at www.verisign.com/rpa (c)06'),),
            (('commonName', u'www.verisign.com'),))}
```

which is a fairly poorly-formed `subject` field.

18.3.4.3. Server-side operation¶

For server operation, typically you'd need to have a server certificate, and private key, each in a file. You'd open a socket, bind it to a port, call `listen()` on it, then start waiting for clients to connect:

```
import socket, ssl

bindsocket = socket.socket()
bindsocket.bind(('myaddr.mydomain.com', 10023))
bindsocket.listen(5)
```

When one did, you'd call `accept()` on the socket to get the new socket from the other end, and use `wrap_socket()` to create a server-side SSL context for it:

```
while True:
  newsocket, fromaddr = bindsocket.accept()
  connstream = ssl.wrap_socket(newsocket,
                               server_side=True,
                               certfile="mycertfile",
                               keyfile="mykeyfile",
                               ssl_version=ssl.PROTOCOL_TLSv1)
  deal_with_client(connstream)
```

Then you'd read data from the `connstream` and do something with it till you are finished with the client (or the client is finished with you):

```
def deal_with_client(connstream):

  data = connstream.read()
  # null data means the client is finished with us
  while data:
     if not do_something(connstream, data):
        # we'll assume do_something returns False
        # when we're finished with client
        break
     data = connstream.read()
  # finished with client
  connstream.close()
```

And go back to listening for new client connections.

See also

Class `socket.socket`

Documentation of underlying `socket` class

Introducing SSL and Certificates using OpenSSL

Frederick J. Hirsch

RFC 1422: Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management

Steve Kent

RFC 1750: Randomness Recommendations for Security

D. Eastlake et. al.

RFC 3280: Internet X.509 Public Key Infrastructure Certificate and CRL Profile

Housley et. al.

Table Of Contents

Previous topic

18.2. `socket` — Low-level networking interface

Next topic

This Page

- Show Source

Navigation