ssl — TLS/SSL wrapper for socket objects

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 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 methods such as getpeercert(), which retrieves the certificate of the other side of the connection, and cipher(), which retrieves the cipher being used for the secure connection.

Functions, Constants, and Exceptions

Socket creation

Client socket example with default context and IPv4/IPv6 dual stack:

```
import socket

import sslhostname = 'www.python.org'

context = ssl.create_default_context()with socket.create_connection((hostname, 443)) as sock:
with context.wrap_socket(sock, server_hostname=hostname) as ssock:
        print(ssock.version())
```

Client socket example with custom context and IPv4:

```
hostname = 'www.python.org'

# PROTOCOL_TLS_CLIENT requires valid cert chain and hostname

context =
ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)context.load_verify_locations('path/to/cabundle.pem')
with socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0) as sock:

        with context.wrap_socket(sock, server_hostname=hostname) as ssock:

                print(ssock.version())
```

Server socket example listening on localhost IPv4:

```
context =
ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)context.load_cert_chain('/path/to/certchain.pem',
'/path/to/private.key')

with socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0) as sock:
        sock.bind(('127.0.0.1', 8443)) sock.listen(5)   with context.wrap_socket(sock,
server_side=True) as ssock:         conn, addr = ssock.accept()...
```

Context creation

A convenience function helps create SSLContext objects for common purposes.

ssl.create_default_context(purpose=Purpose.SERVER_AUTH, cafile=None, capath=None, cadata=None)

Return a new SSLContext object with default settings for the given purpose. The settings are chosen by the ssl module, and usually represent a higher security level than when calling the SSLContext constructor directly.

cafile, capath, cadata represent optional CA certificates to trust for certificate verification, as in SSLContext.load_verify_locations(). If all three are None, this function can choose to trust the system's default CA certificates instead.

The settings are: PROTOCOL_TLS, OP_NO_SSLv2, and OP_NO_SSLv3 with high encryption cipher suites without RC4 and without unauthenticated cipher suites. Passing SERVER_AUTH as purpose sets verify_mode to CERT_REQUIRED and either loads CA certificates (when at least one of cafile, capath or cadata is given) or uses SSLContext.load_default_certs() to load default CA certificates.

When keylog_filename is supported and the environment variable SSLKEYLOGFILE is set, create_default_context() enables key logging.

Exceptions

exception ssl.SSLError

Raised to signal an error from the underlying SSL implementation (currently provided by the OpenSSL library). 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 OSError. The error code and message of SSLError instances are provided by the OpenSSL library.

library

A string mnemonic designating the OpenSSL submodule in which the error occurred, such as SSL, PEM or X509. The range of possible values depends on the OpenSSL version.

reason

A string mnemonic designating the reason this error occurred, for example CERTIFICATE_VERIFY_FAILED. The range of possible values depends on the OpenSSL version.

exception ssl.SSLZeroReturnError

A subclass of SSLError raised when trying to read or write and the SSL connection has been closed cleanly. Note that this doesn't mean that the underlying transport (read TCP) has been closed.

exception ssl.SSLWantReadError

A subclass of SSLError raised by a non-blocking SSL socket when trying to read or write data, but more data needs to be received on the underlying TCP transport before the request can be fulfilled.

exception ssl.SSLWantWriteError

A subclass of SSLError raised by a non-blocking SSL socket when trying to read or write data, but more data needs to be sent on the underlying TCP transport before the request can be fulfilled.

exception ssl.SSLSyscallError

A subclass of SSLError raised when a system error was encountered while trying to fulfill an operation on a SSL socket. Unfortunately, there is no easy way to inspect the original errno number.

exception ssl.SSLEOFError

A subclass of SSLError raised when the SSL connection has been terminated abruptly. Generally, you shouldn't try to reuse the underlying transport when this error is encountered.

exception ssl.SSLCertVerificationError

A subclass of SSLError raised when certificate validation has failed.

verify_code

A numeric error number that denotes the verification error.

verify_message

A human readable string of the verification error.

Random generation

ssl.RAND_bytes(num)

Return num cryptographically strong pseudo-random bytes. Raises an SSLError if the PRNG has not been seeded with enough data or if the operation is not supported by the current RAND method. RAND_status() can be used to check the status of the PRNG and RAND_add() can be used to seed the PRNG.

ssl.RAND_pseudo_bytes(num)

Return (bytes, is_cryptographic): bytes are num pseudo-random bytes, is_cryptographic is True if the bytes generated are cryptographically strong. Raises an SSLError if the operation is not supported by the current RAND method.

Generated pseudo-random byte sequences will be unique if they are of sufficient length, but are not necessarily unpredictable. They can be used for non-cryptographic purposes and for certain purposes in cryptographic protocols, but usually not for key generation etc.

ssl.RAND_status()

Return 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.

ssl.RAND_add(bytes, entropy)

Mix 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.

Certificate handling

ssl.match_hostname(cert, hostname)

Verify that cert (in decoded format as returned by SSLSocket.getpeercert()) matches the given hostname. The rules applied are those for checking the identity of HTTPS servers as outlined in RFC 2818, RFC 5280 and RFC 6125. In addition to HTTPS, this function should be suitable for checking the identity of servers in various SSL-based protocols such as FTPS, IMAPS, POPS and others.

CertificateError is raised on failure. On success, the function returns nothing:

```
>>>cert={'subject':((('commonName','example.com'),),)}
>>>ssl.match_hostname(cert,"example.com")

>>>ssl.match_hostname(cert, "example.org")
```

```
Traceback (most recent call last):File "<stdin>", line 1, in <module>File "/home/py3k/Lib/ssl.py", line 130, in match_hostnamessl.CertificateError: hostname 'example.org' doesn't match 'example.com'
```

ssl.cert_time_to_seconds(cert_time)

Return the time in seconds since the Epoch, given the cert_time string representing the “notBefore” or “notAfter” date from a certificate in "%b %d %H:%M:%S %Y %Z" strptime format (C locale).

Here’s an example:

```
>>> import ssl
>>> timestamp = ssl.cert_time_to_seconds("Jan  5 09:34:43 2018 GMT")

>>> timestamp

1515144883

>>> from datetime import datetime

>>> print(datetime.utcfromtimestamp(timestamp))

2018-01-05 09:34:43
```

ssl.get_server_certificate(addr, ssl_version=PROTOCOL_TLS, 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 SSLContext.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.enum_crls(store_name)

Retrieve CRLs from Windows' system cert store. store_name may be one of CA, ROOT or MY. Windows may provide additional cert stores, too.

The function returns a list of (cert_bytes, encoding_type, trust) tuples. The encoding_type specifies the encoding of cert_bytes. It is either x509_asn for X.509 ASN.1 data or pkcs_7_asn for PKCS#7 ASN.1 data.

Constants

All constants are now enum.IntEnum or enum.IntFlag collections.

ssl.CERT_NONE

Possible value for SSLContext.verify_mode, or the cert_reqs parameter to wrap_socket(). Except for PROTOCOL_TLS_CLIENT, it is the default mode. With client-side sockets, just about any cert is accepted. Validation errors, such as untrusted or expired cert, are ignored and do not abort the TLS/SSL handshake.

In server mode, no certificate is requested from the client, so the client does not send any for client cert authentication.

ssl.CERT_OPTIONAL

Possible value for SSLContext.verify_mode, or the cert_reqs parameter to wrap_socket(). In client mode, CERT_OPTIONAL has the same meaning as CERT_REQUIRED. It is recommended to use CERT_REQUIRED for client-side sockets instead.

In server mode, a client certificate request is sent to the client. The client may either ignore the request or send a certificate in order perform TLS client cert authentication. If the client chooses to send a certificate, it is verified. Any verification error immediately aborts the TLS handshake.

Use of this setting requires a valid set of CA certificates to be passed, either to SSLContext.load_verify_locations() or as a value of the ca_certs parameter to wrap_socket().

ssl.CERT_REQUIRED

Possible value for SSLContext.verify_mode, or the cert_reqs parameter to wrap_socket(). In this mode, certificates are required from the other side of the socket connection; an SSLError will be raised if no certificate is provided, or if its validation fails. This mode is not sufficient to verify a certificate in client mode as it does not match hostnames. check_hostname must be enabled as well to verify the authenticity of a cert. PROTOCOL_TLS_CLIENT uses CERT_REQUIRED and enables check_hostname by default.

With server socket, this mode provides mandatory TLS client cert authentication. A client certificate request is sent to the client and the client must provide a valid and trusted certificate.

Use of this setting requires a valid set of CA certificates to be passed, either to SSLContext.load_verify_locations() or as a value of the ca_certs parameter to wrap_socket().

class ssl.VerifyMode

enum.IntEnum collection of CERT_* constants.

ssl.VERIFY_DEFAULT

Possible value for SSLContext.verify_flags. In this mode, certificate revocation lists (CRLs) are not checked. By default OpenSSL does neither require nor verify CRLs.

ssl.VERIFY_CRL_CHECK_LEAF

Possible value for SSLContext.verify_flags. In this mode, only the peer cert is checked but none of the intermediate CA certificates. The mode requires a valid CRL that is signed by the peer cert's issuer (its direct ancestor CA). If no proper CRL has been loaded with SSLContext.load_verify_locations, validation will fail.

ssl.VERIFY_CRL_CHECK_CHAIN

Possible value for SSLContext.verify_flags. In this mode, CRLs of all certificates in the peer cert chain are checked.

ssl.VERIFY_X509_STRICT

Possible value for SSLContext.verify_flags to disable workarounds for broken X.509 certificates.

ssl.VERIFY_X509_TRUSTED_FIRST

Possible value for SSLContext.verify_flags. It instructs OpenSSL to prefer trusted certificates when
class ssl.VerifyFlags

enum.IntFlag collection of VERIFY_* constants.

building the trust chain to validate a certificate. This flag is enabled by default.

ssl.PROTOCOL_TLS

Selects the highest protocol version that both the client and server support. Despite the name, this option can select both “SSL” and “TLS” protocols.

ssl.PROTOCOL_TLS_CLIENT

Auto-negotiate the highest protocol version like PROTOCOL_TLS, but only support client-side SSLSocket connections. The protocol enables CERT_REQUIRED and check_hostname by default.

ssl.PROTOCOL_TLS_SERVER

Auto-negotiate the highest protocol version like PROTOCOL_TLS, but only support server-side SSLSocket connections.

ssl.OP_ALL

Enables workarounds for various bugs present in other SSL implementations. This option is set by default. It does not necessarily set the same flags as OpenSSL’s SSL_OP_ALL constant.

ssl.OP_NO_RENEGOTIATION

Disable all renegotiation in TLSv1.2 and earlier. Do not send HelloRequest messages, and ignore renegotiation requests via ClientHello.

ssl.OP_CIPHER_SERVER_PREFERENCE

Use the server’s cipher ordering preference, rather than the client’s. This option has no effect on client sockets and SSLv2 server sockets.

ssl.HAS_ECDH

Whether the OpenSSL library has built-in support for the Elliptic Curve-based Diffie-Hellman key exchange. This should be true unless the feature was explicitly disabled by the distributor.

ssl.HAS_SNI

Whether the OpenSSL library has built-in support for the Server Name Indication extension (as defined in RFC 6066).

ssl.HAS_SSLv2

Whether the OpenSSL library has built-in support for the SSL 2.0 protocol.

ssl.HAS_SSLv3

Whether the OpenSSL library has built-in support for the SSL 3.0 protocol.

ssl.HAS_TLSv1

Whether the OpenSSL library has built-in support for the TLS 1.0 protocol.

ssl.HAS_TLSv1_2

Whether the OpenSSL library has built-in support for the TLS 1.2 protocol.

ssl.HAS_TLSv1_3

Whether the OpenSSL library has built-in support for the TLS 1.3 protocol.

ssl.OPENSSL_VERSION

The version string of the OpenSSL library loaded by the interpreter:

```
>>> ssl.OPENSSL_VERSION

'OpenSSL 1.0.2k  26 Jan 2017'
```

ssl.OPENSSL_VERSION_INFO

A tuple of five integers representing version information about the OpenSSL library:

```
>>> ssl.OPENSSL_VERSION_INFO

(1, 0, 2, 11, 15)
```

ssl.OPENSSL_VERSION_NUMBER

The raw version number of the OpenSSL library, as a single integer:

```
>>>                                          ssl.OPENSSL_VERSION_NUMBER268443839>>>
hex(ssl.OPENSSL_VERSION_NUMBER)'0x100020bf'
```

TLSVersion.MINIMUM_SUPPORTED

TLSVersion.MAXIMUM_SUPPORTED

The minimum or maximum supported SSL or TLS version. These are magic constants. Their values don’t reflect the lowest and highest available TLS/SSL versions.

TLSVersion.SSLv3

TLSVersion.TLSv1

TLSVersion.TLSv1_1

TLSVersion.TLSv1_2

TLSVersion.TLSv1_3

SSL 3.0 to TLS 1.3.

SSL Sockets

class ssl.SSLSocket(socket.socket)

SSL sockets provide the following methods of Socket Objects:

- accept()
- bind()
- close()
- connect()
- detach()
- fileno()
- getpeername(), getsockname()
- getsockopt(), setsockopt()
- gettimeout(), settimeout(), setblocking()
- listen()
- makefile()
- recv(), recv_into() (but passing a non-zero flags argument is not allowed)
- send(), sendall() (with the same limitation)
- sendfile() (but os.sendfile will be used for plain-text sockets only, else send() will be used)
- shutdown()

- However, since the SSL (and TLS) protocol has its own framing atop of TCP, the SSL sockets abstraction can, in certain respects, diverge from the specification of normal, OS-level sockets. See especially the notes on non-blocking sockets.

- Instances of SSLSocket must be created using the SSLContext.wrap_socket() method.

SSL sockets also have the following additional methods and attributes:

SSLSocket.read(len=1024, buffer=None)

Read up to len bytes of data from the SSL socket and return the result as a bytes instance. If buffer is specified, then read into the buffer instead, and return the number of bytes read.

Raise SSLWantReadError or SSLWantWriteError if the socket is non-blocking and the read would block.

As at any time a re-negotiation is possible, a call to read() can also cause write operations.

SSLSocket.write(buf)

Write buf to the SSL socket and return the number of bytes written. The buf argument must be an object supporting the buffer interface.

Raise SSLWantReadError or SSLWantWriteError if the socket is non-blocking and the write would block.

As at any time a re-negotiation is possible, a call to write() can also cause read operations.

SSLSocket.do_handshake()

Perform the SSL setup handshake.

Changed in version 3.4: The handshake method also performs match_hostname() when the check_hostname attribute of the socket's context is true.

SSLSocket.getpeercert(binary_form=False)

If there is no certificate for the peer on the other end of the connection, return None. If the SSL handshake hasn't been done yet, raise ValueError.

If the binary_form parameter 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 several keys, amongst them subject (the principal for which the certificate was issued) and issuer (the principal issuing the certificate). 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 and issuer fields are tuples containing the sequence of relative distinguished names (RDNs) given in the certificate's data structure for the respective fields, and each RDN is a sequence of name-value pairs. Here is a real-world example:

```
{'issuer': ((('countryName', 'IL'),),

(('organizationName', 'StartCom Ltd.'),),

        (('organizationalUnitName',

'Secure Digital Certificate Signing'),),

        (('commonName',           'StartCom Class 2 Primary Intermediate Server CA'),)), 'notAfter': 'Nov
22 08:15:19 2013 GMT','notBefore': 'Nov 21 03:09:52 2011 GMT','serialNumber':'95F0',

'subject': ((('description', '571208-SLe257oHY9fVQ07Z'),),

        (('countryName','US'),),

      (('stateOrProvinceName', 'California'),),

        (('localityName', 'San Francisco'),),
    (('organizationName', 'Electronic Frontier Foundation, Inc.'),),
    (('commonName','*.eff.org'),),
    (('emailAddress','hostmaster@eff.org'),)),
'subjectAltName': (('DNS', '*.eff.org'),
('DNS', 'eff.org')), 'version': 3}
```

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.shared_ciphers()

Return the list of ciphers shared by the client during the handshake. Each entry of the returned list is a three-value tuple containing the name of the cipher, the version of the SSL protocol that defines its use, and the number of secret bits the cipher uses. shared_ciphers() returns None if no connection has been established or the socket is a client socket.

SSLSocket.compression()

Return the compression algorithm being used as a string, or None if the connection isn't compressed.

If the higher-level protocol supports its own compression mechanism, you can use OP_NO_COMPRESSION to disable SSL-level compression.

SSLSocket.get_channel_binding(cb_type="tls-unique")

Get channel binding data for current connection, as a bytes object. Returns None if not connected or the handshake has not been completed.

The cb_type parameter allow selection of the desired channel binding type. Valid channel binding types are listed in the CHANNEL_BINDING_TYPES list. Currently only the 'tls-unique' channel binding, defined by RFC 5929, is supported. ValueError will be raised if an unsupported channel binding type is requested.