

ssl — TLS/SSL wrapper for socket objects

Source code: [Lib/ssl.py](#)

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, macOS, 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. For example, TLSv1.3 comes with OpenSSL version 1.1.1.

Warning: Don’t use this module without reading the [Security considerations](#). Doing so may lead to a false sense of security, as the default settings of the ssl module are not necessarily appropriate for your application.

[Availability](#): not WASI.

This module does not work or is not available on WebAssembly. See [WebAssembly platforms](#) for more information.

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 methods such as `getpeercert()`, which retrieves the certificate of the other side of the connection, `cipher()`, which retrieves the cipher being used for the secure connection or `get_verified_chain()`, `get_unverified_chain()` which retrieves certificate chain.

For more sophisticated applications, the [`ssl.SSLContext`](#) class helps manage settings and certificates, which can then be inherited by SSL sockets created through the [`SSLContext.wrap_socket\(\)`](#) method.

- Changed in version 3.5.3:* Updated to support linking with OpenSSL 1.1.0
- Changed in version 3.6:* OpenSSL 0.9.8, 1.0.0 and 1.0.1 are deprecated and no longer supported. In the future the ssl module will require at least OpenSSL 1.0.2 or 1.1.0.
- Changed in version 3.10:* [PEP 644](#) has been implemented. The ssl module requires OpenSSL 1.1.1 or newer.
- Use of deprecated constants and functions result in deprecation warnings.

Functions, Constants, and Exceptions

Socket creation

Instances of [`SSLSocket`](#) must be created using the [`SSLContext.wrap_socket\(\)`](#) method. The helper function [`create_default_context\(\)`](#) returns a new context with secure default settings.

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

```
import socket
import ssl

hostname = '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_CLIENT](#) or [PROTOCOL_TLS_SERVER](#), [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.

The default settings for this context include [VERIFY_X509_PARTIAL_CHAIN](#) and [VERIFY_X509_STRICT](#). These make the underlying OpenSSL implementation behave more like a conforming implementation of [RFC 5280](#), in exchange for a small amount of incompatibility with older X.509 certificates.

Note: The protocol, options, cipher and other settings may change to more restrictive values anytime without prior deprecation. The values represent a fair balance between compatibility and security.

If your application needs specific settings, you should create a [SSLContext](#) and apply the settings yourself.

Note: If you find that when certain older clients or servers attempt to connect with a [SSLContext](#) created by this function that they get an error stating “Protocol or cipher suite mismatch”, it may be that they only support SSL3.0 which this function excludes using the [OP_NO_SSLv3](#). SSL3.0 is widely considered to be [completely broken](#). If you still wish to continue to use this function but still allow SSL 3.0 connections you can re-enable them using:

```
ctx = ssl.create_default_context(Purpose.CLIENT_AUTH)
ctx.options |= ~ssl.OP_NO_SSLv3
```

Note: This context enables [VERIFY_X509_STRICT](#) by default, which may reject pre-[RFC 5280](#) or malformed certificates that the underlying OpenSSL implementation otherwise would accept. While disabling this is not recommended, you can do so using:

```
ctx = ssl.create_default_context()
ctx.verify_flags |= ~ssl.VIFY_X509_STRICT
```

Added in version 3.4.

Changed in version 3.4.4: RC4 was dropped from the default cipher string.

Changed in version 3.6: ChaCha20/Poly1305 was added to the default cipher string.

3DES was dropped from the default cipher string.

Changed in version 3.8: Support for key logging to SSLKEYLOGFILE was added.

Changed in version 3.10: The context now uses [PROTOCOL_TLS_CLIENT](#) or [PROTOCOL_TLS_SERVER](#) protocol instead of generic [PROTOCOL_TLS](#).

Changed in version 3.13: The context now uses [VERIFY_X509_PARTIAL_CHAIN](#) and [VERIFY_X509_STRICT](#) in its default verify flags.

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.

Changed in version 3.3: [SSLError](#) used to be a subtype of [socket.error](#).

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.

Added in version 3.3.

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.

Added in version 3.3.

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.

Added in version 3.3.

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.

Added in version 3.3.

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.

Added in version 3.3.

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.

Added in version 3.3.

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.

Added in version 3.3.

exception ssl.SSLCertVerificationError

A subclass of [SSLError](#) raised when certificate validation has failed.

Added in version 3.7.

verify_code

A numeric error number that denotes the verification error.

verify_message

A human readable string of the verification error.

exception ssl.CertificateError

An alias for [SSLCertVerificationError](#).

Changed in version 3.7: The exception is now an alias for [SSLCertVerificationError](#).

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.

For almost all applications [os.urandom\(\)](#) is preferable.

Read the Wikipedia article, [Cryptographically secure pseudorandom number generator \(CSPRNG\)](#), to get the requirements of a cryptographically strong generator.

Added in version 3.3.

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_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.

Changed in version 3.5: Writable [bytes-like object](#) is now accepted.

Certificate handling

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
```

“notBefore” or “notAfter” dates must use GMT ([RFC 5280](#)).

Changed in version 3.5: Interpret the input time as a time in UTC as specified by ‘GMT’ timezone in the input string. Local timezone was used previously. Return an integer (no fractions of a second in the input format)

`ssl.get_server_certificate(addr, ssl_version=PROTOCOL_TLS_CLIENT, ca_certs=None[, timeout])`

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 *cafile* parameter in [SSLContext.load_verify_locations\(\)](#). The call will attempt to validate the server certificate against that set of root certificates, and will fail if the validation attempt fails. A timeout can be specified with the `timeout` parameter.

Changed in version 3.3: This function is now IPv6-compatible.

Changed in version 3.5: The default `ssl_version` is changed from [PROTOCOL_SSLv3](#) to [PROTOCOL_TLS](#) for maximum compatibility with modern servers.

Changed in version 3.10: The `timeout` parameter was added.

`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.get_default_verify_paths()`

Returns a named tuple with paths to OpenSSL’s default cafile and capath. The paths are the same as used by [SSLContext.set_default_verify_paths\(\)](#). The return value is a [named tuple](#) `DefaultVerifyPaths`:

- `cafile` - resolved path to cafile or `None` if the file doesn’t exist,
- `capath` - resolved path to capath or `None` if the directory doesn’t exist,
- `openssl_cafile_env` - OpenSSL’s environment key that points to a cafile,
- `openssl_cafile` - hard coded path to a cafile,
- `openssl_capath_env` - OpenSSL’s environment key that points to a capath,
- `openssl_capath` - hard coded path to a capath directory

Added in version 3.4.

`ssl.enum_certificates(store_name)`

Retrieve certificates 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. Trust specifies the purpose of the certificate as a set of OIDS or exactly `True` if the certificate is trustworthy for all purposes.

Example:

```
>>> ssl.enum_certificates("CA")
[(b'data...', 'x509_asn', {'1.3.6.1.5.5.7.3.1', '1.3.6.1.5.5.7.3.2'}),
 (b'data...', 'x509_asn', True)]
```

[Availability](#): Windows.

Added in version 3.4.

`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.

[Availability](#): Windows.

Added in version 3.4.

Constants

All constants are now [enum.IntEnum](#) or [enum.IntFlag](#) collections.

Added in version 3.6.

ssl.CERT_NONE

Possible value for [SSLContext.verify_mode](#). 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.

See the discussion of [Security considerations](#) below.

ssl.CERT_OPTIONAL

Possible value for [SSLContext.verify_mode](#). 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 to [SSLContext.load_verify_locations\(\)](#).

ssl.CERT_REQUIRED

Possible value for [SSLContext.verify_mode](#). 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 to [SSLContext.load_verify_locations\(\)](#).

class ssl.VerifyMode

[enum.IntEnum](#) collection of CERT_* constants.

Added in version 3.6.

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.

Added in version 3.4.

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.

Added in version 3.4.

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.

Added in version 3.4.

ssl.VERIFY_X509_STRICT

Possible value for [SSLContext.verify_flags](#) to disable workarounds for broken X.509 certificates.

Added in version 3.4.

ssl.VERIFY_ALLOW_PROXY_CERTS

Possible value for [SSLContext.verify_flags](#) to enables proxy certificate verification.

Added in version 3.10.

ssl.VERIFY_X509_TRUSTED_FIRST

Possible value for [SSLContext.verify_flags](#). It instructs OpenSSL to prefer trusted certificates when building the trust chain to validate a certificate. This flag is enabled by default.

Added in version 3.4.4.

ssl.VERIFY_X509_PARTIAL_CHAIN

Possible value for [SSLContext.verify_flags](#). It instructs OpenSSL to accept intermediate CAs in the trust store to be treated as trust-anchors, in the same way as the self-signed root CA certificates. This makes it possible to trust certificates issued by an intermediate CA without having to trust its ancestor root CA.

Added in version 3.10.

`class ssl.VerifyFlags`
`enum.IntFlag` collection of VERIFY_* constants.

Added in version 3.6.

`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.

Added in version 3.6.

Deprecated since version 3.10: TLS clients and servers require different default settings for secure communication. The generic TLS protocol constant is deprecated in favor of `PROTOCOL_TLS_CLIENT` and `PROTOCOL_TLS_SERVER`.

`ssl.PROTOCOL_TLS_CLIENT`
Auto-negotiate the highest protocol version that both the client and server support, and configure the context client-side connections. The protocol enables `CERT_REQUIRED` and `check_hostname` by default.

Added in version 3.6.

`ssl.PROTOCOL_TLS_SERVER`
Auto-negotiate the highest protocol version that both the client and server support, and configure the context server-side connections.

Added in version 3.6.

`ssl.PROTOCOL_SSLv23`
Alias for `PROTOCOL_TLS`.

Deprecated since version 3.6: Use `PROTOCOL_TLS` instead.

`ssl.PROTOCOL_SSLv3`
Selects SSL version 3 as the channel encryption protocol.

This protocol is not available if OpenSSL is compiled with the `no-ssl3` option.

Warning: SSL version 3 is insecure. Its use is highly discouraged.

Deprecated since version 3.6: OpenSSL has deprecated all version specific protocols. Use the default protocol `PROTOCOL_TLS_SERVER` or `PROTOCOL_TLS_CLIENT` with `SSLContext.minimum_version` and `SSLContext.maximum_version` instead.

`ssl.PROTOCOL_TLSv1`
Selects TLS version 1.0 as the channel encryption protocol.

Deprecated since version 3.6: OpenSSL has deprecated all version specific protocols.

`ssl.PROTOCOL_TLSv1_1`
Selects TLS version 1.1 as the channel encryption protocol. Available only with openssl version 1.0.1+.

Added in version 3.4.

Deprecated since version 3.6: OpenSSL has deprecated all version specific protocols.

`ssl.PROTOCOL_TLSv1_2`
Selects TLS version 1.2 as the channel encryption protocol. Available only with openssl version 1.0.1+.

Added in version 3.4.

Deprecated since version 3.6: OpenSSL has deprecated all version specific protocols.

`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.

Added in version 3.2.

`ssl.OP_NO_SSLv2`
Prevents an SSLv2 connection. This option is only applicable in conjunction with `PROTOCOL_TLS`. It prevents the peers from choosing SSLv2 as the protocol version.

Added in version 3.2.

Deprecated since version 3.6: SSLv2 is deprecated

`ssl.OP_NO_SSLv3`
Prevents an SSLv3 connection. This option is only applicable in conjunction with `PROTOCOL_TLS`. It prevents the peers from choosing SSLv3 as the protocol version.

Added in version 3.2.

Deprecated since version 3.6: SSLv3 is deprecated

ssl.OP_NO_TLSv1

Prevents a TLSv1 connection. This option is only applicable in conjunction with [PROTOCOL_TLS](#). It prevents the peers from choosing TLSv1 as the protocol version.

Added in version 3.2.

Deprecated since version 3.7: The option is deprecated since OpenSSL 1.1.0, use the new [SSLContext.minimum_version](#) and [SSLContext.maximum_version](#) instead.

ssl.OP_NO_TLSv1_1

Prevents a TLSv1.1 connection. This option is only applicable in conjunction with [PROTOCOL_TLS](#). It prevents the peers from choosing TLSv1.1 as the protocol version. Available only with openssl version 1.0.1+.

Added in version 3.4.

Deprecated since version 3.7: The option is deprecated since OpenSSL 1.1.0.

ssl.OP_NO_TLSv1_2

Prevents a TLSv1.2 connection. This option is only applicable in conjunction with [PROTOCOL_TLS](#). It prevents the peers from choosing TLSv1.2 as the protocol version. Available only with openssl version 1.0.1+.

Added in version 3.4.

Deprecated since version 3.7: The option is deprecated since OpenSSL 1.1.0.

ssl.OP_NO_TLSv1_3

Prevents a TLSv1.3 connection. This option is only applicable in conjunction with [PROTOCOL_TLS](#). It prevents the peers from choosing TLSv1.3 as the protocol version. TLS 1.3 is available with OpenSSL 1.1.1 or later. When Python has been compiled against an older version of OpenSSL, the flag defaults to 0.

Added in version 3.6.3.

Deprecated since version 3.7: The option is deprecated since OpenSSL 1.1.0. It was added to 2.7.15 and 3.6.3 for backwards compatibility with OpenSSL 1.0.2.

ssl.OP_NO_RENEGOTIATION

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

This option is only available with OpenSSL 1.1.0h and later.

Added in version 3.7.

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.

Added in version 3.3.

ssl.OP_SINGLE_DH_USE

Prevents reuse of the same DH key for distinct SSL sessions. This improves forward secrecy but requires more computational resources. This option only applies to server sockets.

Added in version 3.3.

ssl.OP_SINGLE_ECDH_USE

Prevents reuse of the same ECDH key for distinct SSL sessions. This improves forward secrecy but requires more computational resources. This option only applies to server sockets.

Added in version 3.3.

ssl.OP_ENABLE_MIDDLEBOX_COMPAT

Send dummy Change Cipher Spec (CCS) messages in TLS 1.3 handshake to make a TLS 1.3 connection look more like a TLS 1.2 connection.

This option is only available with OpenSSL 1.1.1 and later.

Added in version 3.8.

ssl.OP_NO_COMPRESSION

Disable compression on the SSL channel. This is useful if the application protocol supports its own compression scheme.

Added in version 3.3.

class ssl.Options

[enum.IntFlag](#) collection of OP_* constants.

ssl.OP_NO_TICKET

Prevent client side from requesting a session ticket.

Added in version 3.6.

`ssl.OP_IGNORE_UNEXPECTED_EOF`

Ignore unexpected shutdown of TLS connections.

This option is only available with OpenSSL 3.0.0 and later.

Added in version 3.10.

`ssl.OP_ENABLE_KTLS`

Enable the use of the kernel TLS. To benefit from the feature, OpenSSL must have been compiled with support for it, and the negotiated cipher suites and extensions must be supported by it (a list of supported ones may vary by platform and kernel version).

Note that with enabled kernel TLS some cryptographic operations are performed by the kernel directly and not via any available OpenSSL Providers. This might be undesirable if, for example, the application requires all cryptographic operations to be performed by the FIPS provider.

This option is only available with OpenSSL 3.0.0 and later.

Added in version 3.12.

`ssl.OP_LEGACY_SERVER_CONNECT`

Allow legacy insecure renegotiation between OpenSSL and unpatched servers only.

Added in version 3.12.

`ssl.HAS_ALPN`

Whether the OpenSSL library has built-in support for the *Application-Layer Protocol Negotiation* TLS extension as described in [RFC 7301](#).

Added in version 3.5.

`ssl.HAS_NEVER_CHECK_COMMON_NAME`

Whether the OpenSSL library has built-in support not checking subject common name and `SSLContext.hostname_checks_common_name` is writeable.

Added in version 3.7.

`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.

Added in version 3.3.

`ssl.HAS_SNI`

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

Added in version 3.2.

`ssl.HAS_NPN`

Whether the OpenSSL library has built-in support for the *Next Protocol Negotiation* as described in the [Application Layer Protocol Negotiation](#). When true, you can use the `SSLContext.set_npn_protocols()` method to advertise which protocols you want to support.

Added in version 3.3.

`ssl.HAS_SSLv2`

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

Added in version 3.7.

`ssl.HAS_SSLv3`

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

Added in version 3.7.

`ssl.HAS_TLSv1`

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

Added in version 3.7.

`ssl.HAS_TLSv1_1`

Whether the OpenSSL library has built-in support for the TLS 1.1 protocol.

Added in version 3.7.

`ssl.HAS_TLSv1_2`

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

Added in version 3.7.

`ssl.HAS_TLSv1_3`

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

Added in version 3.7.

ssl.HAS_PSK

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

Added in version 3.13.

ssl.CHANNEL_BINDING_TYPES

List of supported TLS channel binding types. Strings in this list can be used as arguments to [SSLSocket.get_channel_binding\(\)](#).

Added in version 3.3.

ssl.OPENSLL_VERSION

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

```
>>> ssl.OPENSLL_VERSION
'OpenSSL 1.0.2k  26 Jan 2017'
```

Added in version 3.2.

ssl.OPENSLL_VERSION_INFO

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

```
>>> ssl.OPENSLL_VERSION_INFO
(1, 0, 2, 11, 15)
```

Added in version 3.2.

ssl.OPENSLL_VERSION_NUMBER

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

```
>>> ssl.OPENSLL_VERSION_NUMBER
268443839
>>> hex(ssl.OPENSLL_VERSION_NUMBER)
'0x100020bf'
```

Added in version 3.2.

ssl.ALERT_DESCRIPTION_HANDSHAKE_FAILURE

ssl.ALERT_DESCRIPTION_INTERNAL_ERROR

ALERT_DESCRIPTION_*

Alert Descriptions from [RFC 5246](#) and others. The [IANA TLS Alert Registry](#) contains this list and references to the RFCs where their meaning is defined.

Used as the return value of the callback function in [SSLContext.set_servername_callback\(\)](#).

Added in version 3.4.

class ssl.AlertDescription

[enum.IntEnum](#) collection of ALERT_DESCRIPTION_* constants.

Added in version 3.6.

Purpose.SERVER_AUTH

Option for [create_default_context\(\)](#) and [SSLContext.load_default_certs\(\)](#). This value indicates that the context may be used to authenticate web servers (therefore, it will be used to create client-side sockets).

Added in version 3.4.

Purpose.CLIENT_AUTH

Option for [create_default_context\(\)](#) and [SSLContext.load_default_certs\(\)](#). This value indicates that the context may be used to authenticate web clients (therefore, it will be used to create server-side sockets).

Added in version 3.4.

class ssl.SSLErrorNumber

[enum.IntEnum](#) collection of SSL_ERROR_* constants.

Added in version 3.6.

class ssl.TLSVersion

[enum.IntEnum](#) collection of SSL and TLS versions for [SSLContext.maximum_version](#) and [SSLContext.minimum_version](#).

Added in version 3.7.

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.

Deprecated since version 3.10: All [TLSVersion](#) members except [TLSVersion.TLSv1_2](#) and [TLSVersion.TLSv1_3](#) are deprecated.

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.

Changed in version 3.5: The `sendfile()` method was added.

Changed in version 3.5: The `shutdown()` does not reset the socket timeout each time bytes are received or sent. The socket timeout is now the maximum total duration of the shutdown.

Deprecated since version 3.6: It is deprecated to create a [SSLSocket](#) instance directly, use [SSLContext.wrap_socket\(\)](#) to wrap a socket.

Changed in version 3.7: [SSLSocket](#) instances must to created with [wrap_socket\(\)](#). In earlier versions, it was possible to create instances directly. This was never documented or officially supported.

Changed in version 3.10: Python now uses `SSL_read_ex` and `SSL_write_ex` internally. The functions support reading and writing of data larger than 2 GB. Writing zero-length data no longer fails with a protocol violation error.

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.

Changed in version 3.5: The socket timeout is no longer reset each time bytes are received or sent. The socket timeout is now the maximum total duration to read up to *len* bytes.

Deprecated since version 3.6: Use `recv()` instead of [read\(\)](#).

`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.

Changed in version 3.5: The socket timeout is no longer reset each time bytes are received or sent. The socket timeout is now the maximum total duration to write *buf*.

Deprecated since version 3.6: Use `send()` instead of [write\(\)](#).

Note: The `read()` and `write()` methods are the low-level methods that read and write unencrypted, application-level data and decrypt/encrypt it to encrypted, wire-level data. These methods require an active SSL connection, i.e. the handshake was completed and `SSLSocket.unwrap()` was not called.

Normally you should use the socket API methods like `recv()` and `send()` instead of these methods.

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.
- Changed in version 3.5:* The socket timeout is no longer reset each time bytes are received or sent. The socket timeout is now the maximum total duration of the handshake.
- Changed in version 3.7:* Hostname or IP address is matched by OpenSSL during handshake. The function `match_hostname()` is no longer used. In case OpenSSL refuses a hostname or IP address, the handshake is aborted early and a TLS alert message is sent to the peer.

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}
```

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. Whether the peer provides a certificate depends on the SSL socket's role:

- for a client SSL socket, the server will always provide a certificate, regardless of whether validation was required;
- for a server SSL socket, the client will only provide a certificate when requested by the server; therefore `getpeercert()` will return `None` if you used `CERT_NONE` (rather than `CERT_OPTIONAL` or `CERT_REQUIRED`).

See also [SSLContext.check_hostname](#).

- Changed in version 3.2:* The returned dictionary includes additional items such as `issuer` and `notBefore`.
- Changed in version 3.4:* `ValueError` is raised when the handshake isn't done. The returned dictionary includes additional X509v3 extension items such as `crlDistributionPoints`, `caIssuers` and `OCSP` URIs.
- Changed in version 3.9:* IPv6 address strings no longer have a trailing new line.

SSLSocket.get_verified_chain()

Returns verified certificate chain provided by the other end of the SSL channel as a list of DER-encoded bytes. If certificate verification was disabled method acts the same as [get_unverified_chain\(\)](#).

Added in version 3.13.

SSLSocket.get_unverified_chain()

Returns raw certificate chain provided by the other end of the SSL channel as a list of DER-encoded bytes.

Added in version 3.13.

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 available in both the client and server. 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.

Added in version 3.5.

`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.

Added in version 3.3.

`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.

Added in version 3.3.

`SSLSocket.selected_alpn_protocol()`

Return the protocol that was selected during the TLS handshake. If [SSLContext.set_alpn_protocols\(\)](#) was not called, if the other party does not support ALPN, if this socket does not support any of the client's proposed protocols, or if the handshake has not happened yet, `None` is returned.

Added in version 3.5.

`SSLSocket.selected_npn_protocol()`

Return the higher-level protocol that was selected during the TLS/SSL handshake. If [SSLContext.set_npn_protocols\(\)](#) was not called, or if the other party does not support NPN, or if the handshake has not yet happened, this will return `None`.

Added in version 3.3.

Deprecated since version 3.10: NPN has been superseded by ALPN

`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 returned socket should always be used for further communication with the other side of the connection, rather than the original socket.

`SSLSocket.verify_client_post_handshake()`

Requests post-handshake authentication (PHA) from a TLS 1.3 client. PHA can only be initiated for a TLS 1.3 connection from a server-side socket, after the initial TLS handshake and with PHA enabled on both sides, see [SSLContext.post_handshake_auth](#).

The method does not perform a cert exchange immediately. The server-side sends a CertificateRequest during the next write event and expects the client to respond with a certificate on the next read event.

If any precondition isn't met (e.g. not TLS 1.3, PHA not enabled), an [SSLError](#) is raised.

Note: Only available with OpenSSL 1.1.1 and TLS 1.3 enabled. Without TLS 1.3 support, the method raises [NotImplementedError](#).

Added in version 3.8.

`SSLSocket.version()`

Return the actual SSL protocol version negotiated by the connection as a string, or `None` if no secure connection is established. As of this writing, possible return values include `"SSLv2"`, `"SSLv3"`, `"TLSv1"`, `"TLSv1.1"` and `"TLSv1.2"`. Recent OpenSSL versions may define more return values.

Added in version 3.5.

`SSLSocket.pending()`

Returns the number of already decrypted bytes available for read, pending on the connection.

`SSLSocket.context`

The [SSLContext](#) object this SSL socket is tied to.

Added in version 3.2.

`SSLSocket.server_side`

A boolean which is `True` for server-side sockets and `False` for client-side sockets.

Added in version 3.2.

SSLSocket.server_hostname

Hostname of the server: `str` type, or `None` for server-side socket or if the hostname was not specified in the constructor.

Added in version 3.2.

Changed in version 3.7: The attribute is now always ASCII text. When `server_hostname` is an internationalized domain name (IDN), this attribute now stores the A-label form ("`xn--pythn-mua.org`"), rather than the U-label form ("`pythön.org`").

SSLSocket.session

The [SSLSession](#) for this SSL connection. The session is available for client and server side sockets after the TLS handshake has been performed. For client sockets the session can be set before [do_handshake\(\)](#) has been called to reuse a session.

Added in version 3.6.

SSLSocket.session_reused

Added in version 3.6.

SSL Contexts

Added in version 3.2.

An SSL context holds various data longer-lived than single SSL connections, such as SSL configuration options, certificate(s) and private key(s). It also manages a cache of SSL sessions for server-side sockets, in order to speed up repeated connections from the same clients.

`class ssl.SSLContext(protocol=None)`

Create a new SSL context. You may pass *protocol* which must be one of the `PROTOCOL_*` constants defined in this module. The parameter 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, the default is [PROTOCOL_TLS](#); it provides 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):

<i>client / server</i>	SSLv2	SSLv3	TLS [3]	TLSv1	TLSv1.1	TLSv1.2
SSLv2	yes	no	no [1]	no	no	no
SSLv3	no	yes	no [2]	no	no	no
TLS (SSLv23) [3]	no [1]	no [2]	yes	yes	yes	yes
TLSv1	no	no	yes	yes	no	no
TLSv1.1	no	no	yes	no	yes	no
TLSv1.2	no	no	yes	no	no	yes

Footnotes

[1](1,2) [SSLContext](#) disables SSLv2 with [OP_NO_SSLv2](#) by default.

[2](1,2) [SSLContext](#) disables SSLv3 with [OP_NO_SSLv3](#) by default.

[3](1,2) TLS 1.3 protocol will be available with [PROTOCOL_TLS](#) in OpenSSL >= 1.1.1. There is no dedicated PROTOCOL constant for just TLS 1.3.

See also: [create_default_context\(\)](#) lets the `ssl` module choose security settings for a given purpose.

Changed in version 3.6: The context is created with secure default values. The options [OP_NO_COMPRESSION](#), [OP_CIPHER_SERVER_PREFERENCE](#), [OP_SINGLE_DH_USE](#), [OP_SINGLE_ECDH_USE](#), [OP_NO_SSLv2](#), and [OP_NO_SSLv3](#) (except for [PROTOCOL_SSLv3](#)) are set by default. The initial cipher suite list contains only HIGH ciphers, no NULL ciphers and no MD5 ciphers.

Deprecated since version 3.10: [SSLContext](#) without protocol argument is deprecated. The context class will either require [PROTOCOL_TLS_CLIENT](#) or [PROTOCOL_TLS_SERVER](#) protocol in the future.

Changed in version 3.10: The default cipher suites now include only secure AES and ChaCha20 ciphers with forward secrecy and security level 2. RSA and DH keys with less than 2048 bits and ECC keys with less than 224 bits are prohibited. [PROTOCOL_TLS](#), [PROTOCOL_TLS_CLIENT](#), and [PROTOCOL_TLS_SERVER](#) use TLS 1.2 as minimum TLS version.

Note: [SSLContext](#) only supports limited mutation once it has been used by a connection. Adding new certificates to the internal trust store is allowed, but changing ciphers, verification settings, or mTLS certificates may result in surprising behavior.

Note: [SSLContext](#) is designed to be shared and used by multiple connections. Thus, it is thread-safe as long as it is not reconfigured after being used by a connection.

[SSLContext](#) objects have the following methods and attributes:

SSLContext.cert_store_stats()

Get statistics about quantities of loaded X.509 certificates, count of X.509 certificates flagged as CA certificates and certificate revocation lists as dictionary.

Example for a context with one CA cert and one other cert:

```
>>> context.cert_store_stats()
{'crl': 0, 'x509_ca': 1, 'x509': 2}
```

Added in version 3.4.

SSLContext.load_cert_chain(certfile, keyfile=None, password=None)

Load a private key and the corresponding certificate. The *certfile* string must be the path to a single file in PEM format containing the certificate as well as any number of CA certificates needed to establish the certificate’s authenticity. The *keyfile* string, if present, must point to a file containing the private key. Otherwise the private key will be taken from *certfile* as well. See the discussion of [Certificates](#) for more information on how the certificate is stored in the *certfile*.

The *password* argument may be a function to call to get the password for decrypting the private key. It will only be called if the private key is encrypted and a password is necessary. It will be called with no arguments, and it should return a string, bytes, or bytearray. If the return value is a string it will be encoded as UTF-8 before using it to decrypt the key. Alternatively a string, bytes, or bytearray value may be supplied directly as the *password* argument. It will be ignored if the private key is not encrypted and no password is needed.

If the *password* argument is not specified and a password is required, OpenSSL’s built-in password prompting mechanism will be used to interactively prompt the user for a password.

An [SSLError](#) is raised if the private key doesn’t match with the certificate.

Changed in version 3.3: New optional argument *password*.

SSLContext.load_default_certs(purpose=Purpose.SERVER_AUTH)

Load a set of default “certification authority” (CA) certificates from default locations. On Windows it loads CA certs from the CA and ROOT system stores. On all systems it calls [SSLContext.set_default_verify_paths\(\)](#). In the future the method may load CA certificates from other locations, too.

The *purpose* flag specifies what kind of CA certificates are loaded. The default settings [Purpose.SERVER_AUTH](#) loads certificates, that are flagged and trusted for TLS web server authentication (client side sockets). [Purpose.CLIENT_AUTH](#) loads CA certificates for client certificate verification on the server side.

Added in version 3.4.

SSLContext.load_verify_locations(cafile=None, capath=None, cadata=None)

Load a set of “certification authority” (CA) certificates used to validate other peers’ certificates when [verify_mode](#) is other than [CERT_NONE](#). At least one of *cafile* or *capath* must be specified.

This method can also load certification revocation lists (CRLs) in PEM or DER format. In order to make use of CRLs, [SSLContext.verify_flags](#) must be configured properly.

The *cafile* string, if present, is the path to a file of concatenated CA certificates in PEM format. See the discussion of [Certificates](#) for more information about how to arrange the certificates in this file.

The *capath* string, if present, is the path to a directory containing several CA certificates in PEM format, following an [OpenSSL specific layout](#).

The *cadata* object, if present, is either an ASCII string of one or more PEM-encoded certificates or a [bytes-like object](#) of DER-encoded certificates. Like with *capath* extra lines around PEM-encoded certificates are ignored but at least one certificate must be present.

Changed in version 3.4: New optional argument *cadata*

SSLContext.get_ca_certs(binary_form=False)

Get a list of loaded “certification authority” (CA) certificates. If the *binary_form* parameter is [False](#) each list entry is a dict like the output of [SSLSocket.getpeercert\(\)](#). Otherwise the method returns a list of DER-encoded certificates. The returned list does not contain certificates from *capath* unless a certificate was requested and loaded by a SSL connection.

Note: Certificates in a *capath* directory aren’t loaded unless they have been used at least once.

Added in version 3.4.

SSLContext.get_ciphers()

Get a list of enabled ciphers. The list is in order of cipher priority. See [SSLContext.set_ciphers\(\)](#).

Example:

```
>>> ctx = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
>>> ctx.set_ciphers('ECDHE+AESGCM:!ECDSA')
>>> ctx.get_ciphers()
[{'aead': True,
  'alg_bits': 256,
  'auth': 'auth-rsa',
  'description': 'ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH Au=RSA '}]
```

```
        'Enc=AESGCM(256) Mac=AEAD',
        'digest': None,
        'id': 50380848,
        'kea': 'kx-ecdhe',
        'name': 'ECDHE-RSA-AES256-GCM-SHA384',
        'protocol': 'TLSv1.2',
        'strength_bits': 256,
        'symmetric': 'aes-256-gcm'},
    {'aead': True,
     'alg_bits': 128,
     'auth': 'auth-rsa',
     'description': 'ECDHE-RSA-AES128-GCM-SHA256 TLSv1.2 Kx=ECDH    Au=RSA    '
                    'Enc=AESGCM(128) Mac=AEAD',
     'digest': None,
     'id': 50380847,
     'kea': 'kx-ecdhe',
     'name': 'ECDHE-RSA-AES128-GCM-SHA256',
     'protocol': 'TLSv1.2',
     'strength_bits': 128,
     'symmetric': 'aes-128-gcm'}]]
```

Added in version 3.6.

SSLContext.set_default_verify_paths()

Load a set of default “certification authority” (CA) certificates from a filesystem path defined when building the OpenSSL library. Unfortunately, there’s no easy way to know whether this method succeeds: no error is returned if no certificates are to be found. When the OpenSSL library is provided as part of the operating system, though, it is likely to be configured properly.

SSLContext.set_ciphers(ciphers)

Set the available ciphers for sockets created with this context. It should be a string in the [OpenSSL cipher list format](#). If no cipher can be selected (because compile-time options or other configuration forbids use of all the specified ciphers), an [SSL](#)[Error](#) will be raised.

Note: when connected, the [SSL](#)[Socket.cipher\(\)](#) method of SSL sockets will give the currently selected cipher.

TLS 1.3 cipher suites cannot be disabled with [set_ciphers\(\)](#).

SSLContext.set_alpn_protocols(protocols)

Specify which protocols the socket should advertise during the SSL/TLS handshake. It should be a list of ASCII strings, like `['http/1.1', 'spdy/2']`, ordered by preference. The selection of a protocol will happen during the handshake, and will play out according to [RFC 7301](#). After a successful handshake, the [SSL](#)[Socket.selected_alpn_protocol\(\)](#) method will return the agreed-upon protocol.

This method will raise [NotImplementedError](#) if [HAS_ALPN](#) is `False`.

Added in version 3.5.

SSLContext.set_npn_protocols(protocols)

Specify which protocols the socket should advertise during the SSL/TLS handshake. It should be a list of strings, like `['http/1.1', 'spdy/2']`, ordered by preference. The selection of a protocol will happen during the handshake, and will play out according to the [Application Layer Protocol Negotiation](#). After a successful handshake, the [SSL](#)[Socket.selected_npn_protocol\(\)](#) method will return the agreed-upon protocol.

This method will raise [NotImplementedError](#) if [HAS_NPN](#) is `False`.

Added in version 3.3.

Deprecated since version 3.10: NPN has been superseded by ALPN

SSLContext.sni_callback

Register a callback function that will be called after the TLS Client Hello handshake message has been received by the SSL/TLS server when the TLS client specifies a server name indication. The server name indication mechanism is specified in [RFC 6066](#) section 3 - Server Name Indication.

Only one callback can be set per `SSLContext`. If `sni_callback` is set to `None` then the callback is disabled. Calling this function a subsequent time will disable the previously registered callback.

The callback function will be called with three arguments; the first being the [ssl.SSL](#)[Socket](#), the second is a string that represents the server name that the client is intending to communicate (or `None` if the TLS Client Hello does not contain a server name) and the third argument is the original [SSLContext](#). The server name argument is text. For internationalized domain name, the server name is an IDN A-label (`"xn--pythn-mua.org"`).

A typical use of this callback is to change the [ssl.SSL](#)[Socket](#)’s [SSL](#)[Socket.context](#) attribute to a new object of type [SSLContext](#) representing a certificate chain that matches the server name.

Due to the early negotiation phase of the TLS connection, only limited methods and attributes are usable like [SSL](#)[Socket.selected_alpn_protocol\(\)](#) and [SSL](#)[Socket.context](#). The [SSL](#)[Socket.getpeercert\(\)](#), [SSL](#)[Socket.get_verified_chain\(\)](#), [SSL](#)[Socket.get_unverified_chain\(\)](#) [SSL](#)[Socket.cipher\(\)](#) and [SSL](#)[Socket.compression\(\)](#) methods require that the TLS connection has progressed beyond the TLS Client Hello and therefore will not return meaningful values nor can they be called safely.

The `sni_callback` function must return `None` to allow the TLS negotiation to continue. If a TLS failure is required, a constant `ALERT_DESCRIPTION_*` can be returned. Other return values will result in a TLS fatal error with `ALERT_DESCRIPTION_INTERNAL_ERROR`.

If an exception is raised from the `sni_callback` function the TLS connection will terminate with a fatal TLS alert message `ALERT_DESCRIPTION_HANDSHAKE_FAILURE`.

This method will raise `NotImplementedError` if the OpenSSL library had `OPENSSL_NO_TLSEXT` defined when it was built.

Added in version 3.7.

SSLContext.set_servername_callback(*server_name_callback*)

This is a legacy API retained for backwards compatibility. When possible, you should use `sni_callback` instead. The given `server_name_callback` is similar to `sni_callback`, except that when the server hostname is an IDN-encoded internationalized domain name, the `server_name_callback` receives a decoded U-label (`"python.org"`).

If there is a decoding error on the server name, the TLS connection will terminate with an `ALERT_DESCRIPTION_INTERNAL_ERROR` fatal TLS alert message to the client.

Added in version 3.4.

SSLContext.load_dh_params(*dhfile*)

Load the key generation parameters for Diffie-Hellman (DH) key exchange. Using DH key exchange improves forward secrecy at the expense of computational resources (both on the server and on the client). The `dhfile` parameter should be the path to a file containing DH parameters in PEM format.

This setting doesn't apply to client sockets. You can also use the `OP_SINGLE_DH_USE` option to further improve security.

Added in version 3.3.

SSLContext.set_ecdh_curve(*curve_name*)

Set the curve name for Elliptic Curve-based Diffie-Hellman (ECDH) key exchange. ECDH is significantly faster than regular DH while arguably as secure. The `curve_name` parameter should be a string describing a well-known elliptic curve, for example `prime256v1` for a widely supported curve.

This setting doesn't apply to client sockets. You can also use the `OP_SINGLE_ECDH_USE` option to further improve security.

This method is not available if `HAS_ECDH` is `False`.

Added in version 3.3.

See also:

[SSL/TLS & Perfect Forward Secrecy](#)

Vincent Bernat.

SSLContext.wrap_socket(*sock*, *server_side=False*, *do_handshake_on_connect=True*, *suppress_ragged_eofs=True*, *server_hostname=None*, *session=None*)

Wrap an existing Python socket `sock` and return an instance of `SSLContext.sslsocket_class` (default `SSLSocket`). The returned SSL socket is tied to the context, its settings and certificates. `sock` must be a `SOCK_STREAM` socket; other socket types are unsupported.

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

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. The method may raise `SSLError`.

On client connections, the optional parameter `server_hostname` specifies the hostname of the service which we are connecting to. This allows a single server to host multiple SSL-based services with distinct certificates, quite similarly to HTTP virtual hosts. Specifying `server_hostname` will raise a `ValueError` if `server_side` is true.

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.recv()` method should signal unexpected EOF from the other end of the connection. If specified as `True` (the default), it returns a normal EOF (an empty bytes object) in response to unexpected EOF errors raised from the underlying socket; if `False`, it will raise the exceptions back to the caller.

session, see [session](#).

To wrap an `SSLSocket` in another `SSLSocket`, use `SSLContext.wrap_bio()`.

Changed in version 3.5: Always allow a `server_hostname` to be passed, even if OpenSSL does not have SNI.

Changed in version 3.6: `session` argument was added.

Changed in version 3.7: The method returns an instance of `SSLContext.sslsocket_class` instead of hard-coded `SSLSocket`.

SSLContext.**sslsocket_class**

The return type of [SSLContext.wrap_socket\(\)](#), defaults to [SSLSocket](#). The attribute can be overridden on instance of class in order to return a custom subclass of [SSLSocket](#).

Added in version 3.7.

SSLContext.**wrap_bio**(*incoming, outgoing, server_side=False, server_hostname=None, session=None*)

Wrap the BIO objects *incoming* and *outgoing* and return an instance of [SSLContext.sslobject_class](#) (default [SSLObject](#)). The SSL routines will read input data from the incoming BIO and write data to the outgoing BIO.

The *server_side*, *server_hostname* and *session* parameters have the same meaning as in [SSLContext.wrap_socket\(\)](#).

Changed in version 3.6: *session* argument was added.

Changed in version 3.7: The method returns an instance of [SSLContext.sslobject_class](#) instead of hard-coded [SSLObject](#).

SSLContext.**sslobject_class**

The return type of [SSLContext.wrap_bio\(\)](#), defaults to [SSLObject](#). The attribute can be overridden on instance of class in order to return a custom subclass of [SSLObject](#).

Added in version 3.7.

SSLContext.**session_stats**()

Get statistics about the SSL sessions created or managed by this context. A dictionary is returned which maps the names of each [piece of information](#) to their numeric values. For example, here is the total number of hits and misses in the session cache since the context was created:

```
>>> stats = context.session_stats()
>>> stats['hits'], stats['misses']
(0, 0)
```

SSLContext.**check_hostname**

Whether to match the peer cert’s hostname in [SSLSocket.do_handshake\(\)](#). The context’s [verify_mode](#) must be set to [CERT_OPTIONAL](#) or [CERT_REQUIRED](#), and you must pass *server_hostname* to [wrap_socket\(\)](#) in order to match the hostname. Enabling hostname checking automatically sets [verify_mode](#) from [CERT_NONE](#) to [CERT_REQUIRED](#). It cannot be set back to [CERT_NONE](#) as long as hostname checking is enabled. The [PROTOCOL_TLS_CLIENT](#) protocol enables hostname checking by default. With other protocols, hostname checking must be enabled explicitly.

Example:

```
import socket, ssl

context = ssl.SSLContext(ssl.PROTOCOL_TLSv1_2)
context.verify_mode = ssl.CERT_REQUIRED
context.check_hostname = True
context.load_default_certs()

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
ssl_sock = context.wrap_socket(s, server_hostname='www.verisign.com')
ssl_sock.connect(('www.verisign.com', 443))
```

Added in version 3.4.

Changed in version 3.7: [verify_mode](#) is now automatically changed to [CERT_REQUIRED](#) when hostname checking is enabled and [verify_mode](#) is [CERT_NONE](#). Previously the same operation would have failed with a [ValueError](#).

SSLContext.**keylog_filename**

Write TLS keys to a keylog file, whenever key material is generated or received. The keylog file is designed for debugging purposes only. The file format is specified by NSS and used by many traffic analyzers such as Wireshark. The log file is opened in append-only mode. Writes are synchronized between threads, but not between processes.

Added in version 3.8.

SSLContext.**maximum_version**

A [TLSVersion](#) enum member representing the highest supported TLS version. The value defaults to [TLSVersion.MAXIMUM_SUPPORTED](#). The attribute is read-only for protocols other than [PROTOCOL_TLS](#), [PROTOCOL_TLS_CLIENT](#), and [PROTOCOL_TLS_SERVER](#).

The attributes [maximum_version](#), [minimum_version](#) and [SSLContext.options](#) all affect the supported SSL and TLS versions of the context. The implementation does not prevent invalid combination. For example a context with [OP_NO_TLSv1_2](#) in [options](#) and [maximum_version](#) set to [TLSVersion.TLSv1_2](#) will not be able to establish a TLS 1.2 connection.

Added in version 3.7.

SSLContext.**minimum_version**

Like [SSLContext.maximum_version](#) except it is the lowest supported version or [TLSVersion.MINIMUM_SUPPORTED](#).

Added in version 3.7.

SSLContext.**num_tickets**

Control the number of TLS 1.3 session tickets of a [PROTOCOL_TLS_SERVER](#) context. The setting has no impact on TLS 1.0 to 1.2 connections.

Added in version 3.8.

SSLContext.options

An integer representing the set of SSL options enabled on this context. The default value is [OP_ALL](#), but you can specify other options such as [OP_NO_SSLv2](#) by ORing them together.

Changed in version 3.6: [SSLContext.options](#) returns [Options](#) flags:

```
>>> ssl.create_default_context().options
<Options.OP_ALL|OP_NO_SSLv3|OP_NO_SSLv2|OP_NO_COMPRESSION: 2197947391>
```

Deprecated since version 3.7: All `OP_NO_SSL*` and `OP_NO_TLS*` options have been deprecated since Python 3.7. Use [SSLContext.minimum_version](#) and [SSLContext.maximum_version](#) instead.

SSLContext.post_handshake_auth

Enable TLS 1.3 post-handshake client authentication. Post-handshake auth is disabled by default and a server can only request a TLS client certificate during the initial handshake. When enabled, a server may request a TLS client certificate at any time after the handshake.

When enabled on client-side sockets, the client signals the server that it supports post-handshake authentication.

When enabled on server-side sockets, [SSLContext.verify_mode](#) must be set to [CERT_OPTIONAL](#) or [CERT_REQUIRED](#), too. The actual client cert exchange is delayed until [SSLSocket.verify_client_post_handshake\(\)](#) is called and some I/O is performed.

Added in version 3.8.

SSLContext.protocol

The protocol version chosen when constructing the context. This attribute is read-only.

SSLContext.hostname_checks_common_name

Whether [check_hostname](#) falls back to verify the cert’s subject common name in the absence of a subject alternative name extension (default: true).

Added in version 3.7.

Changed in version 3.10: The flag had no effect with OpenSSL before version 1.1.1l. Python 3.8.9, 3.9.3, and 3.10 include workarounds for previous versions.

SSLContext.security_level

An integer representing the [security level](#) for the context. This attribute is read-only.

Added in version 3.10.

SSLContext.verify_flags

The flags for certificate verification operations. You can set flags like [VERIFY_CRL_CHECK_LEAF](#) by ORing them together. By default OpenSSL does neither require nor verify certificate revocation lists (CRLs).

Added in version 3.4.

Changed in version 3.6: [SSLContext.verify_flags](#) returns [VerifyFlags](#) flags:

```
>>> ssl.create_default_context().verify_flags
<VerifyFlags.VERIFY_X509_TRUSTED_FIRST: 32768>
```

SSLContext.verify_mode

Whether to try to verify other peers’ certificates and how to behave if verification fails. This attribute must be one of [CERT_NONE](#), [CERT_OPTIONAL](#) or [CERT_REQUIRED](#).

Changed in version 3.6: [SSLContext.verify_mode](#) returns [VerifyMode](#) enum:

```
>>> ssl.create_default_context().verify_mode
<VerifyMode.CERT_REQUIRED: 2>
```

SSLContext.set_psk_client_callback(callback)

Enables TLS-PSK (pre-shared key) authentication on a client-side connection.

In general, certificate based authentication should be preferred over this method.

The parameter `callback` is a callable object with the signature: `def callback(hint: str | None) -> tuple[str | None, bytes]`. The `hint` parameter is an optional identity hint sent by the server. The return value is a tuple in the form (client-identity, psk). Client-identity is an optional string which may be used by the server to select a corresponding PSK for the client. The string must be less than or equal to 256 octets when UTF-8 encoded. PSK is a [bytes-like object](#) representing the pre-shared key. Return a zero length PSK to reject the connection.

Setting `callback` to [None](#) removes any existing callback.

- Note:** When using TLS 1.3:
- the `hint` parameter is always [None](#).
 - client-identity must be a non-empty string.

Example usage:

```
context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
context.check_hostname = False
context.verify_mode = ssl.CERT_NONE
context.maximum_version = ssl.TLSVersion.TLSv1_2
context.set_ciphers('PSK')

# A simple lambda:
psk = bytes.fromhex('c0ffee')
context.set_psk_client_callback(lambda hint: (None, psk))

# A table using the hint from the server:
psk_table = { 'ServerId_1': bytes.fromhex('c0ffee'),
              'ServerId_2': bytes.fromhex('facade')
            }
def callback(hint):
    return 'ClientId_1', psk_table.get(hint, b'')
context.set_psk_client_callback(callback)
```

This method will raise [NotImplementedError](#) if [HAS_PSK](#) is `False`.

Added in version 3.13.

SSLContext.set_psk_server_callback(callback, identity_hint=None)

Enables TLS-PSK (pre-shared key) authentication on a server-side connection.

In general, certificate based authentication should be preferred over this method.

The parameter `callback` is a callable object with the signature: `def callback(identity: str | None) -> bytes`. The `identity` parameter is an optional identity sent by the client which can be used to select a corresponding PSK. The return value is a [bytes-like object](#) representing the pre-shared key. Return a zero length PSK to reject the connection.

Setting `callback` to [None](#) removes any existing callback.

The parameter `identity_hint` is an optional identity hint string sent to the client. The string must be less than or equal to 256 octets when UTF-8 encoded.

Note: When using TLS 1.3 the `identity_hint` parameter is not sent to the client.

Example usage:

```
context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
context.maximum_version = ssl.TLSVersion.TLSv1_2
context.set_ciphers('PSK')

# A simple lambda:
psk = bytes.fromhex('c0ffee')
context.set_psk_server_callback(lambda identity: psk)

# A table using the identity of the client:
psk_table = { 'ClientId_1': bytes.fromhex('c0ffee'),
              'ClientId_2': bytes.fromhex('facade')
            }
def callback(identity):
    return psk_table.get(identity, b'')
context.set_psk_server_callback(callback, 'ServerId_1')
```

This method will raise [NotImplementedError](#) if [HAS_PSK](#) is `False`.

Added in version 3.13.

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 they claim 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-----
```

Certificate chains

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-----
```

CA certificates

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. The platform’s certificates file can be used by calling [SSLContext.load_default_certs\(\)](#), this is done automatically with [create_default_context\(\)](#).

Combined key and certificate

Often the private key is stored in the same file as the certificate; in this case, only the `certfile` parameter to [SSLContext.load_cert_chain\(\)](#) needs to be passed. If the private key is stored with the certificate, 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-----
```

Self-signed certificates

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.

Examples

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
```

Client-side operation

This example creates a SSL context with the recommended security settings for client sockets, including automatic certificate verification:

```
>>> context = ssl.create_default_context()
```

If you prefer to tune security settings yourself, you might create a context from scratch (but beware that you might not get the settings right):

```
>>> context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
>>> context.load_verify_locations("/etc/ssl/certs/ca-bundle.crt")
```

(this snippet assumes your operating system places a bundle of all CA certificates in `/etc/ssl/certs/ca-bundle.crt`; if not, you'll get an error and have to adjust the location)

The `PROTOCOL_TLS_CLIENT` protocol configures the context for cert validation and hostname verification. `verify_mode` is set to `CERT_REQUIRED` and `check_hostname` is set to `True`. All other protocols create SSL contexts with insecure defaults.

When you use the context to connect to a server, `CERT_REQUIRED` and `check_hostname` validate the server certificate: it ensures that the server certificate was signed with one of the CA certificates, checks the signature for correctness, and verifies other properties like validity and identity of the hostname:

```
>>> conn = context.wrap_socket(socket.socket(socket.AF_INET),
...                             server_hostname="www.python.org")
>>> conn.connect(("www.python.org", 443))
```

You may then fetch the certificate:

```
>>> cert = conn.getpeercert()
```

Visual inspection shows that the certificate does identify the desired service (that is, the HTTPS host `www.python.org`):

```
>>> pprint.pprint(cert)
{'OCSP': ('http://ocsp.digicert.com',),
 'caIssuers': ('http://cacerts.digicert.com/DigiCertSHA2ExtendedValidationServerCA.crt',),
 'crlDistributionPoints': ('http://crl3.digicert.com/sha2-ev-server-g1.crl',
                           'http://crl4.digicert.com/sha2-ev-server-g1.crl'),
 'issuer': (((('countryName', 'US'),),
               (('organizationName', 'DigiCert Inc'),),
               (('organizationalUnitName', 'www.digicert.com'),),
               (('commonName', 'DigiCert SHA2 Extended Validation Server CA'),)),
 'notAfter': 'Sep  9 12:00:00 2016 GMT',
 'notBefore': 'Sep  5 00:00:00 2014 GMT',
 'serialNumber': '01BB6F00122B177F36CAB49CEA8B6B26',
 'subject': (((('businessCategory', 'Private Organization'),),
                (('1.3.6.1.4.1.311.60.2.1.3', 'US'),),
                (('1.3.6.1.4.1.311.60.2.1.2', 'Delaware'),),
                (('serialNumber', '3359300'),),
                (('streetAddress', '16 Allen Rd'),),
                (('postalCode', '03894-4801'),),
                (('countryName', 'US'),),
                (('stateOrProvinceName', 'NH'),),
                (('localityName', 'Wolfeboro'),),
                (('organizationName', 'Python Software Foundation'),),
                (('commonName', 'www.python.org'),)),
 'subjectAltName': (('DNS', 'www.python.org'),
                    ('DNS', 'python.org'),
                    ('DNS', 'pypi.org'),
                    ('DNS', 'docs.python.org'),
                    ('DNS', 'testpypi.org'),
                    ('DNS', 'bugs.python.org'),
                    ('DNS', 'wiki.python.org'),
                    ('DNS', 'hg.python.org'),
                    ('DNS', 'mail.python.org'),
                    ('DNS', 'packaging.python.org'),
                    ('DNS', 'pythonhosted.org'),
                    ('DNS', 'www.pythonhosted.org'),
                    ('DNS', 'test.pythonhosted.org'),
                    ('DNS', 'us.pycon.org'),
                    ('DNS', 'id.python.org')),
 'version': 3}
```

Now the SSL channel is established and the certificate verified, you can proceed to talk with the server:

```
>>> conn.sendall(b"HEAD / HTTP/1.0\r\nHost: linuxfr.org\r\n\r\n")
>>> pprint.pprint(conn.recv(1024).split(b"\r\n"))
[b'HTTP/1.1 200 OK',
 b'Date: Sat, 18 Oct 2014 18:27:20 GMT',
 b'Server: nginx',
 b'Content-Type: text/html; charset=utf-8',
 b'X-Frame-Options: SAMEORIGIN',
 b'Content-Length: 45679',
 b'Accept-Ranges: bytes',
 b'Via: 1.1 varnish',
 b'Age: 2188',
 b'X-Served-By: cache-lcy1134-LCY',
 b'X-Cache: HIT',
```

```
b'X-Cache-Hits: 11',
b'Vary: Cookie',
b'Strict-Transport-Security: max-age=63072000; includeSubDomains',
b'Connection: close',
b'',
b'']
```

See the discussion of [Security considerations](#) below.

Server-side operation

For server operation, typically you'll need to have a server certificate, and private key, each in a file. You'll first create a context holding the key and the certificate, so that clients can check your authenticity. Then you'll open a socket, bind it to a port, call `listen()` on it, and start waiting for clients to connect:

```
import socket, ssl

context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
context.load_cert_chain(certfile="mycertfile", keyfile="mykeyfile")

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

When a client connects, you'll call `accept()` on the socket to get the new socket from the other end, and use the context's [SSLContext.wrap_socket\(\)](#) method to create a server-side SSL socket for the connection:

```
while True:
    newsocket, fromaddr = bindsocket.accept()
    connstream = context.wrap_socket(newsocket, server_side=True)
    try:
        deal_with_client(connstream)
    finally:
        connstream.shutdown(socket.SHUT_RDWR)
        connstream.close()
```

Then you'll 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.recv(1024)
    # empty 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.recv(1024)
    # finished with client
```

And go back to listening for new client connections (of course, a real server would probably handle each client connection in a separate thread, or put the sockets in [non-blocking mode](#) and use an event loop).

Notes on non-blocking sockets

SSL sockets behave slightly different than regular sockets in non-blocking mode. When working with non-blocking sockets, there are thus several things you need to be aware of:

- Most [SSLSocket](#) methods will raise either [SSLWantWriteError](#) or [SSLWantReadError](#) instead of [BlockingIOError](#) if an I/O operation would block. [SSLWantReadError](#) will be raised if a read operation on the underlying socket is necessary, and [SSLWantWriteError](#) for a write operation on the underlying socket. Note that attempts to *write* to an SSL socket may require *reading* from the underlying socket first, and attempts to *read* from the SSL socket may require a prior *write* to the underlying socket.

Changed in version 3.5: In earlier Python versions, the `SSLSocket.send()` method returned zero instead of raising [SSLWantWriteError](#) or [SSLWantReadError](#).

- Calling [select\(\)](#) tells you that the OS-level socket can be read from (or written to), but it does not imply that there is sufficient data at the upper SSL layer. For example, only part of an SSL frame might have arrived. Therefore, you must be ready to handle `SSLSocket.recv()` and `SSLSocket.send()` failures, and retry after another call to [select\(\)](#).
- Conversely, since the SSL layer has its own framing, a SSL socket may still have data available for reading without [select\(\)](#) being aware of it. Therefore, you should first call `SSLSocket.recv()` to drain any potentially available data, and then only block on a [select\(\)](#) call if still necessary.

(of course, similar provisions apply when using other primitives such as [poll\(\)](#), or those in the [selectors](#) module)

- The SSL handshake itself will be non-blocking: the [SSLSocket.do_handshake\(\)](#) method has to be retried until it returns successfully. Here is a synopsis using [select\(\)](#) to wait for the socket's readiness:

```
while True:
    try:
        sock.do_handshake()
        break
    except ssl.SSLWantReadError:
        select.select([sock], [], [])
```

```
except ssl.SSLWantWriteError:
    select.select([], [sock], [])
```

See also: The [asyncio](#) module supports [non-blocking SSL sockets](#) and provides a higher level [Streams API](#). It polls for events using the [selectors](#) module and handles [SSLWantWriteError](#), [SSLWantReadError](#) and [BlockingIOError](#) exceptions. It runs the SSL handshake asynchronously as well.

Memory BIO Support

Added in version 3.5.

Ever since the SSL module was introduced in Python 2.6, the [SSLSocket](#) class has provided two related but distinct areas of functionality:

- SSL protocol handling
- Network IO

The network IO API is identical to that provided by [socket.socket](#), from which [SSLSocket](#) also inherits. This allows an SSL socket to be used as a drop-in replacement for a regular socket, making it very easy to add SSL support to an existing application.

Combining SSL protocol handling and network IO usually works well, but there are some cases where it doesn’t. An example is async IO frameworks that want to use a different IO multiplexing model than the “select/poll on a file descriptor” (readiness based) model that is assumed by [socket.socket](#) and by the internal OpenSSL socket IO routines. This is mostly relevant for platforms like Windows where this model is not efficient. For this purpose, a reduced scope variant of [SSLSocket](#) called [SSLObject](#) is provided.

`class ssl.SSLObject`

A reduced-scope variant of [SSLSocket](#) representing an SSL protocol instance that does not contain any network IO methods. This class is typically used by framework authors that want to implement asynchronous IO for SSL through memory buffers.

This class implements an interface on top of a low-level SSL object as implemented by OpenSSL. This object captures the state of an SSL connection but does not provide any network IO itself. IO needs to be performed through separate “BIO” objects which are OpenSSL’s IO abstraction layer.

This class has no public constructor. An [SSLObject](#) instance must be created using the [wrap_bio\(\)](#) method. This method will create the [SSLObject](#) instance and bind it to a pair of BIOs. The *incoming* BIO is used to pass data from Python to the SSL protocol instance, while the *outgoing* BIO is used to pass data the other way around.

The following methods are available:

- [context](#)
- [server_side](#)
- [server_hostname](#)
- [session](#)
- [session_reused](#)
- [read\(\)](#)
- [write\(\)](#)
- [getpeercert\(\)](#)
- [get_verified_chain\(\)](#)
- [get_unverified_chain\(\)](#)
- [selected_alpn_protocol\(\)](#)
- [selected_npn_protocol\(\)](#)
- [cipher\(\)](#)
- [shared_ciphers\(\)](#)
- [compression\(\)](#)
- [pending\(\)](#)
- [do_handshake\(\)](#)
- [verify_client_post_handshake\(\)](#)
- [unwrap\(\)](#)
- [get_channel_binding\(\)](#)
- [version\(\)](#)

When compared to [SSLSocket](#), this object lacks the following features:

- Any form of network IO; `recv()` and `send()` read and write only to the underlying [MemoryBIO](#) buffers.
- There is no *do_handshake_on_connect* machinery. You must always manually call [do_handshake\(\)](#) to start the handshake.
- There is no handling of *suppress_ragged_eofs*. All end-of-file conditions that are in violation of the protocol are reported via the [SSLEOFError](#) exception.
- The method [unwrap\(\)](#) call does not return anything, unlike for an SSL socket where it returns the underlying socket.
- The *server_name_callback* callback passed to [SSLContext.set_servername_callback\(\)](#) will get an [SSLObject](#) instance instead of a [SSLSocket](#) instance as its first parameter.

Some notes related to the use of [SSLObject](#):

- All IO on an [SSLObject](#) is [non-blocking](#). This means that for example [read\(\)](#) will raise an [SSLWantReadError](#) if it needs more data than the incoming BIO has available.

Changed in version 3.7: [SSLObject](#) instances must be created with [wrap_bio\(\)](#). In earlier versions, it was possible to create instances directly. This was never documented or officially supported.

An `SSLObject` communicates with the outside world using memory buffers. The class [MemoryBIO](#) provides a memory buffer that can be used for this purpose. It wraps an OpenSSL memory BIO (Basic IO) object:

class `ssl.MemoryBIO`

A memory buffer that can be used to pass data between Python and an SSL protocol instance.

pending

Return the number of bytes currently in the memory buffer.

eof

A boolean indicating whether the memory BIO is current at the end-of-file position.

read(*n=-1*)

Read up to *n* bytes from the memory buffer. If *n* is not specified or negative, all bytes are returned.

write(*buf*)

Write the bytes from *buf* to the memory BIO. The *buf* argument must be an object supporting the buffer protocol.

The return value is the number of bytes written, which is always equal to the length of *buf*.

write_eof()

Write an EOF marker to the memory BIO. After this method has been called, it is illegal to call [write\(\)](#). The attribute [eof](#) will become true after all data currently in the buffer has been read.

SSL session

Added in version 3.6.

class `ssl.SSLSession`

Session object used by [session](#).

id

time

timeout

ticket_lifetime_hint

has_ticket

Security considerations

Best defaults

For **client use**, if you don't have any special requirements for your security policy, it is highly recommended that you use the [create_default_context\(\)](#) function to create your SSL context. It will load the system's trusted CA certificates, enable certificate validation and hostname checking, and try to choose reasonably secure protocol and cipher settings.

For example, here is how you would use the [smtplib.SMTP](#) class to create a trusted, secure connection to a SMTP server:

```
>>> import ssl, smtplib
>>> smtp = smtplib.SMTP("mail.python.org", port=587)
>>> context = ssl.create_default_context()
>>> smtp.starttls(context=context)
(220, b'2.0.0 Ready to start TLS')
```

If a client certificate is needed for the connection, it can be added with [SSLContext.load_cert_chain\(\)](#).

By contrast, if you create the SSL context by calling the [SSLContext](#) constructor yourself, it will not have certificate validation nor hostname checking enabled by default. If you do so, please read the paragraphs below to achieve a good security level.

Manual settings

Verifying certificates

When calling the [SSLContext](#) constructor directly, [CERT_NONE](#) is the default. Since it does not authenticate the other peer, it can be insecure, especially in client mode where most of the time you would like to ensure the authenticity of the server you're talking to. Therefore, when in client mode, it is highly recommended to use [CERT_REQUIRED](#). However, it is in itself not sufficient; you also have to check that the server certificate, which can be obtained by calling [SSLSocket.getpeercert\(\)](#), matches the desired service. For many protocols and applications, the service can be identified by the hostname. This common check is automatically performed when [SSLContext.check_hostname](#) is enabled.

Changed in version 3.7: Hostname matchings is now performed by OpenSSL. Python no longer uses `match_hostname()`.

In server mode, if you want to authenticate your clients using the SSL layer (rather than using a higher-level authentication mechanism), you’ll also have to specify [CERT_REQUIRED](#) and similarly check the client certificate.

Protocol versions

SSL versions 2 and 3 are considered insecure and are therefore dangerous to use. If you want maximum compatibility between clients and servers, it is recommended to use [PROTOCOL_TLS_CLIENT](#) or [PROTOCOL_TLS_SERVER](#) as the protocol version. SSLv2 and SSLv3 are disabled by default.

```
>>> client_context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
>>> client_context.minimum_version = ssl.TLSVersion.TLSv1_3
>>> client_context.maximum_version = ssl.TLSVersion.TLSv1_3
```

The SSL context created above will only allow TLSv1.3 and later (if supported by your system) connections to a server. [PROTOCOL_TLS_CLIENT](#) implies certificate validation and hostname checks by default. You have to load certificates into the context.

Cipher selection

If you have advanced security requirements, fine-tuning of the ciphers enabled when negotiating a SSL session is possible through the [SSLContext.set_ciphers\(\)](#) method. Starting from Python 3.2.3, the ssl module disables certain weak ciphers by default, but you may want to further restrict the cipher choice. Be sure to read OpenSSL’s documentation about the [cipher list format](#). If you want to check which ciphers are enabled by a given cipher list, use [SSLContext.get_ciphers\(\)](#) or the `openssl ciphers` command on your system.

Multi-processing

If using this module as part of a multi-processed application (using, for example the [multiprocessing](#) or [concurrent.futures](#) modules), be aware that OpenSSL’s internal random number generator does not properly handle forked processes. Applications must change the PRNG state of the parent process if they use any SSL feature with [os.fork\(\)](#). Any successful call of [RAND_add\(\)](#) or [RAND_bytes\(\)](#) is sufficient.

TLS 1.3

Added in version 3.7.

The TLS 1.3 protocol behaves slightly differently than previous version of TLS/SSL. Some new TLS 1.3 features are not yet available.

- TLS 1.3 uses a disjunct set of cipher suites. All AES-GCM and ChaCha20 cipher suites are enabled by default. The method [SSLContext.set_ciphers\(\)](#) cannot enable or disable any TLS 1.3 ciphers yet, but [SSLContext.get_ciphers\(\)](#) returns them.
- Session tickets are no longer sent as part of the initial handshake and are handled differently. [SSLSocket.session](#) and [SSLSession](#) are not compatible with TLS 1.3.
- Client-side certificates are also no longer verified during the initial handshake. A server can request a certificate at any time. Clients process certificate requests while they send or receive application data from the server.
- TLS 1.3 features like early data, deferred TLS client cert request, signature algorithm configuration, and rekeying are not supported yet.

See also:

Class [socket.socket](#)

Documentation of underlying [socket](#) class

[SSL/TLS Strong Encryption: An Introduction](#)

Intro from the Apache HTTP Server documentation

[RFC 1422: Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management](#)

Steve Kent

[RFC 4086: Randomness Requirements for Security](#)

Donald E., Jeffrey I. Schiller

[RFC 5280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List \(CRL\) Profile](#)

D. Cooper

[RFC 5246: The Transport Layer Security \(TLS\) Protocol Version 1.2](#)

T. Dierks et. al.

[RFC 6066: Transport Layer Security \(TLS\) Extensions](#)

D. Eastlake

[IANA TLS: Transport Layer Security \(TLS\) Parameters](#)

IANA

[RFC 7525: Recommendations for Secure Use of Transport Layer Security \(TLS\) and Datagram Transport Layer Security \(DTLS\)](#)

IETF

[Mozilla’s Server Side TLS recommendations](#)

Mozilla