TLS (SSL)

Stability: 2 - Stable

The `tls` module provides an implementation of the Transport Layer Security (TLS) and Secure Socket Layer (SSL) protocols that is built on top of OpenSSL. The module can be accessed using:

```
const tls = require('tls');
```

Determining if crypto support is unavailable

It is possible for Node.js to be built without including support for the `crypto` module. In such cases, attempting to `import` from `tls` or calling `require('tls')` will result in an error being thrown.

When using CommonJS, the error thrown can be caught using try/catch:

```
let tls;
try {
  tls = require('tls');
} catch (err) {
  console.log('tls support is disabled!');
}
```

When using the lexical ESM `import` keyword, the error can only be caught if a handler for `process.on('uncaughtException')` is registered *before* any attempt to load the module is made (using, for instance, a preload module).

When using ESM, if there is a chance that the code may be run on a build of Node.js where crypto support is not enabled, consider using the `import()` function instead of the lexical `import` keyword:

```
let tls;
try {
  tls = await import('tls');
} catch (err) {
  console.log('tls support is disabled!');
}
```

TLS/SSL concepts

TLS/SSL is a set of protocols that rely on a public key infrastructure (PKI) to enable secure communication between a client and a server. For most common cases, each server must have a private key.

Private keys can be generated in multiple ways. The example below illustrates use of the OpenSSL command-line interface to generate a 2048-bit RSA private key:

```
openssl genrsa -out ryans-key.pem 2048
```

With TLS/SSL, all servers (and some clients) must have a *certificate*. Certificates are *public keys* that correspond to a private key, and that are digitally signed either by a Certificate Authority or by the owner of the private key (such certificates are referred to as "self-signed"). The first step to obtaining a certificate is to create a *Certificate Signing Request* (CSR) file.

The OpenSSL command-line interface can be used to generate a CSR for a private key:

```
openssl req -new -sha256 -key ryans-key.pem -out ryans-csr.pem
```

Once the CSR file is generated, it can either be sent to a Certificate Authority for signing or used to generate a self-signed certificate.

Creating a self-signed certificate using the OpenSSL command-line interface is illustrated in the example below:

```
openssl x509 -req -in ryans-csr.pem -signkey ryans-key.pem -out ryans-cert.pem
```

Once the certificate is generated, it can be used to generate a `.pfx` or `.p12` file:

```
openssl pkcs12 -export -in ryans-cert.pem -inkey ryans-key.pem \
      -certfile ca-cert.pem -out ryans.pfx
```

Where:

- `in`: is the signed certificate
- `inkey`: is the associated private key
- `certfile`: is a concatenation of all Certificate Authority (CA) certs into a single file, e.g. `cat ca1-cert.pem ca2-cert.pem > ca-cert.pem`

Perfect forward secrecy

The term *forward secrecy* or *perfect forward secrecy* describes a feature of key-agreement (i.e., key-exchange) methods. That is, the server and client keys are used to negotiate new temporary keys that are used specifically and only for the current communication session. Practically, this means that even if the server's private key is compromised, communication can only be decrypted by eavesdroppers if the attacker manages to obtain the key-pair specifically generated for the session.

Perfect forward secrecy is achieved by randomly generating a key pair for key-agreement on every TLS/SSL handshake (in contrast to using the same key for all sessions). Methods implementing this technique are called "ephemeral".

Currently two methods are commonly used to achieve perfect forward secrecy (note the character "E" appended to the traditional abbreviations):

- DHE: An ephemeral version of the Diffie-Hellman key-agreement protocol.

- ECDHE: An ephemeral version of the Elliptic Curve Diffie-Hellman key-agreement protocol.

To use perfect forward secrecy using `DHE` with the `tls` module, it is required to generate Diffie-Hellman parameters and specify them with the `dhparam` option to `tls.createSecureContext()`. The following illustrates the use of the OpenSSL command-line interface to generate such parameters:

```
openssl dhparam -outform PEM -out dhparam.pem 2048
```

If using perfect forward secrecy using `ECDHE`, Diffie-Hellman parameters are not required and a default ECDHE curve will be used. The `ecdhCurve` property can be used when creating a TLS Server to specify the list of names of supported curves to use, see `tls.createServer()` for more info.

Perfect forward secrecy was optional up to TLSv1.2, but it is not optional for TLSv1.3, because all TLSv1.3 cipher suites use ECDHE.

ALPN and SNI

ALPN (Application-Layer Protocol Negotiation Extension) and SNI (Server Name Indication) are TLS handshake extensions:

- ALPN: Allows the use of one TLS server for multiple protocols (HTTP, HTTP/2)
- SNI: Allows the use of one TLS server for multiple hostnames with different certificates.

Pre-shared keys

TLS-PSK support is available as an alternative to normal certificate-based authentication. It uses a pre-shared key instead of certificates to authenticate a TLS connection, providing mutual authentication. TLS-PSK and public key infrastructure are not mutually exclusive. Clients and servers can accommodate both, choosing either of them during the normal cipher negotiation step.

TLS-PSK is only a good choice where means exist to securely share a key with every connecting machine, so it does not replace the public key infrastructure (PKI) for the majority of TLS uses. The TLS-PSK implementation in OpenSSL has seen many security flaws in recent years, mostly because it is used only by a minority of applications. Please consider all alternative solutions before switching to PSK ciphers. Upon generating PSK it is of critical importance to use sufficient entropy as discussed in RFC 4086. Deriving a shared secret from a password or other low-entropy sources is not secure.

PSK ciphers are disabled by default, and using TLS-PSK thus requires explicitly specifying a cipher suite with the `ciphers` option. The list of available ciphers can be retrieved via `openssl ciphers -v 'PSK'`. All TLS 1.3 ciphers are eligible for PSK but currently only those that use SHA256 digest are supported they can be retrieved via `openssl ciphers -v -s -tls1_3 -psk`.

According to the RFC 4279, PSK identities up to 128 bytes in length and PSKs up to 64 bytes in length must be supported. As of OpenSSL 1.1.0 maximum identity size is 128 bytes, and maximum PSK length is 256 bytes.

The current implementation doesn't support asynchronous PSK callbacks due to the limitations of the underlying OpenSSL API.

Client-initiated renegotiation attack mitigation

The TLS protocol allows clients to renegotiate certain aspects of the TLS session. Unfortunately, session renegotiation requires a disproportionate amount of server-side resources, making it a potential vector for denial-of-service attacks.

To mitigate the risk, renegotiation is limited to three times every ten minutes. An `'error'` event is emitted on the `tls.TLSSocket` instance when this threshold is exceeded. The limits are configurable:

- `tls.CLIENT_RENEG_LIMIT` {number} Specifies the number of renegotiation requests. **Default:** `3`.
- `tls.CLIENT_RENEG_WINDOW` {number} Specifies the time renegotiation window in seconds. **Default:** `600` (10 minutes).

The default renegotiation limits should not be modified without a full understanding of the implications and risks.

TLSv1.3 does not support renegotiation.

Session resumption

Establishing a TLS session can be relatively slow. The process can be sped up by saving and later reusing the session state. There are several mechanisms to do so, discussed here from oldest to newest (and preferred).

Session identifiers Servers generate a unique ID for new connections and send it to the client. Clients and servers save the session state. When reconnecting, clients send the ID of their saved session state and if the server also has the state for that ID, it can agree to use it. Otherwise, the server will create a new session. See RFC 2246 for more information, page 23 and 30.

Resumption using session identifiers is supported by most web browsers when making HTTPS requests.

For Node.js, clients wait for the `'session'` event to get the session data, and provide the data to the `session` option of a subsequent `tls.connect()` to reuse the session. Servers must implement handlers for the `'newSession'` and `'resumeSession'` events to save and restore the session data using the session ID as the lookup key to reuse sessions. To reuse sessions across load balancers or cluster workers, servers must use a shared session cache (such as Redis) in their session handlers.

Session tickets The servers encrypt the entire session state and send it to the client as a "ticket". When reconnecting, the state is sent to the server in the initial connection. This mechanism avoids the need for server-side session cache. If the server doesn't use the ticket, for any reason (failure to decrypt it, it's too old, etc.), it will create a new session and send a new ticket. See RFC 5077 for more information.

Resumption using session tickets is becoming commonly supported by many web browsers when making HTTPS requests.

For Node.js, clients use the same APIs for resumption with session identifiers as for resumption with session tickets. For debugging, if `tls.TLSSocket.getTLSTicket()` returns a value, the session data contains a ticket, otherwise it contains client-side session state.

With TLSv1.3, be aware that multiple tickets may be sent by the server, resulting in multiple `'session'` events, see `'session'` for more information.

Single process servers need no specific implementation to use session tickets. To use session tickets across server restarts or load balancers, servers must all have the same ticket keys. There are three 16-byte keys internally, but the tls API exposes them as a single 48-byte buffer for convenience.

It's possible to get the ticket keys by calling `server.getTicketKeys()` on one server instance and then distribute them, but it is more reasonable to securely generate 48 bytes of secure random data and set them with the `ticketKeys` option of `tls.createServer()`. The keys should be regularly regenerated and server's keys can be reset with `server.setTicketKeys()`.

Session ticket keys are cryptographic keys, and they ***must be stored securely***. With TLS 1.2 and below, if they are compromised all sessions that used tickets encrypted with them can be decrypted. They should not be stored on disk, and they should be regenerated regularly.

If clients advertise support for tickets, the server will send them. The server can disable tickets by supplying `require('constants').SSL_OP_NO_TICKET` in `secureOptions`.

Both session identifiers and session tickets timeout, causing the server to create new sessions. The timeout can be configured with the `sessionTimeout` option of `tls.createServer()`.

For all the mechanisms, when resumption fails, servers will create new sessions. Since failing to resume the session does not cause TLS/HTTPS connection failures, it is easy to not notice unnecessarily poor TLS performance. The OpenSSL CLI can be used to verify that servers are resuming sessions. Use the `-reconnect` option to `openssl s_client`, for example:

```
$ openssl s_client -connect localhost:443 -reconnect
```

Read through the debug output. The first connection should say "New", for example:

```
New, TLSv1.2, Cipher is ECDHE-RSA-AES128-GCM-SHA256
```

Subsequent connections should say "Reused", for example:

```
Reused, TLSv1.2, Cipher is ECDHE-RSA-AES128-GCM-SHA256
```

Modifying the default TLS cipher suite

Node.js is built with a default suite of enabled and disabled TLS ciphers. This default cipher list can be configured when building Node.js to allow distributions to provide their own default list.

The following command can be used to show the default cipher suite:

```
node -p crypto.constants.defaultCoreCipherList | tr ':' '\n'
TLS_AES_256_GCM_SHA384
TLS_CHACHA20_POLY1305_SHA256
TLS_AES_128_GCM_SHA256
ECDHE-RSA-AES128-GCM-SHA256
ECDHE-ECDSA-AES128-GCM-SHA256
ECDHE-RSA-AES256-GCM-SHA384
ECDHE-ECDSA-AES256-GCM-SHA384
DHE-RSA-AES128-GCM-SHA256
ECDHE-RSA-AES128-SHA256
DHE-RSA-AES128-SHA256
ECDHE-RSA-AES256-SHA384
DHE-RSA-AES256-SHA384
ECDHE-RSA-AES256-SHA256
DHE-RSA-AES256-SHA256
HIGH
!aNULL
!eNULL
!EXPORT
!DES
!RC4
!MD5
!PSK
!SRP
!CAMELLIA
```

This default can be replaced entirely using the `--tls-cipher-list` command-line switch (directly, or via the `NODE_OPTIONS` environment variable). For instance, the following makes `ECDHE-RSA-AES128-GCM-SHA256:!RC4` the default TLS cipher suite:

```
node --tls-cipher-list='ECDHE-RSA-AES128-GCM-SHA256:!RC4' server.js
```

```
export NODE_OPTIONS=--tls-cipher-list='ECDHE-RSA-AES128-GCM-SHA256:!RC4'
node server.js
```

The default can also be replaced on a per client or server basis using the `ciphers` option from `tls.createSecureContext()`, which is also available in `tls.createServer()`, `tls.connect()`, and when creating new `tls.TLSSocket`s.

The ciphers list can contain a mixture of TLSv1.3 cipher suite names, the ones that start with `'TLS_'`, and specifications for TLSv1.2 and below cipher suites. The TLSv1.2 ciphers support a legacy specification format, consult the OpenSSL cipher list format documentation for details, but those specifications do *not* apply to TLSv1.3 ciphers. The TLSv1.3 suites can only be enabled by including their full name in the cipher list. They cannot, for example, be enabled or disabled by using the legacy TLSv1.2 `'EECDH'` or `'!EECDH'` specification.

Despite the relative order of TLSv1.3 and TLSv1.2 cipher suites, the TLSv1.3 protocol is significantly more secure than TLSv1.2, and will always be chosen over TLSv1.2 if the handshake indicates it is supported, and if any TLSv1.3 cipher suites are enabled.

The default cipher suite included within Node.js has been carefully selected to reflect current security best practices and risk mitigation. Changing the default cipher suite can have a significant impact on the security of an application. The `--tls-cipher-list` switch and `ciphers` option should by used only if absolutely necessary.

The default cipher suite prefers GCM ciphers for Chrome's 'modern cryptography' setting and also prefers ECDHE and DHE ciphers for perfect forward secrecy, while offering *some* backward compatibility.

Old clients that rely on insecure and deprecated RC4 or DES-based ciphers (like Internet Explorer 6) cannot complete the handshaking process with the default configuration. If these clients *must* be supported, the TLS recommendations may offer a compatible cipher suite. For more details on the format, see the OpenSSL cipher list format documentation.

There are only five TLSv1.3 cipher suites:

- `'TLS_AES_256_GCM_SHA384'`
- `'TLS_CHACHA20_POLY1305_SHA256'`
- `'TLS_AES_128_GCM_SHA256'`
- `'TLS_AES_128_CCM_SHA256'`
- `'TLS_AES_128_CCM_8_SHA256'`

The first three are enabled by default. The two `CCM`-based suites are supported by TLSv1.3 because they may be more performant on constrained systems, but they are not enabled by default since they offer less security.

X509 certificate error codes

Multiple functions can fail due to certificate errors that are reported by OpenSSL. In such a case, the function provides an {Error} via its callback that has the property `code` which can take one of the following values:

- `'UNABLE_TO_GET_ISSUER_CERT'`: Unable to get issuer certificate.
- `'UNABLE_TO_GET_CRL'`: Unable to get certificate CRL.
- `'UNABLE_TO_DECRYPT_CERT_SIGNATURE'`: Unable to decrypt certificate's signature.
- `'UNABLE_TO_DECRYPT_CRL_SIGNATURE'`: Unable to decrypt CRL's signature.
- `'UNABLE_TO_DECODE_ISSUER_PUBLIC_KEY'`: Unable to decode issuer public key.
- `'CERT_SIGNATURE_FAILURE'`: Certificate signature failure.
- `'CRL_SIGNATURE_FAILURE'`: CRL signature failure.
- `'CERT_NOT_YET_VALID'`: Certificate is not yet valid.
- `'CERT_HAS_EXPIRED'`: Certificate has expired.
- `'CRL_NOT_YET_VALID'`: CRL is not yet valid.
- `'CRL_HAS_EXPIRED'`: CRL has expired.
- `'ERROR_IN_CERT_NOT_BEFORE_FIELD'`: Format error in certificate's notBefore field.
- `'ERROR_IN_CERT_NOT_AFTER_FIELD'`: Format error in certificate's notAfter field.
- `'ERROR_IN_CRL_LAST_UPDATE_FIELD'`: Format error in CRL's lastUpdate field.
- `'ERROR_IN_CRL_NEXT_UPDATE_FIELD'`: Format error in CRL's nextUpdate field.
- `'OUT_OF_MEM'`: Out of memory.
- `'DEPTH_ZERO_SELF_SIGNED_CERT'`: Self signed certificate.
- `'SELF_SIGNED_CERT_IN_CHAIN'`: Self signed certificate in certificate chain.
- `'UNABLE_TO_GET_ISSUER_CERT_LOCALLY'`: Unable to get local issuer certificate.
- `'UNABLE_TO_VERIFY_LEAF_SIGNATURE'`: Unable to verify the first certificate.
- `'CERT_CHAIN_TOO_LONG'`: Certificate chain too long.
- `'CERT_REVOKED'`: Certificate revoked.
- `'INVALID_CA'`: Invalid CA certificate.
- `'PATH_LENGTH_EXCEEDED'`: Path length constraint exceeded.
- `'INVALID_PURPOSE'`: Unsupported certificate purpose.
- `'CERT_UNTRUSTED'`: Certificate not trusted.
- `'CERT_REJECTED'`: Certificate rejected.
- `'HOSTNAME_MISMATCH'`: Hostname mismatch.

Class: `tls.CryptoStream`

Stability: 0 - Deprecated: Use `tls.TLSSocket` instead.

The `tls.CryptoStream` class represents a stream of encrypted data. This class is deprecated and should no longer be used.

`cryptoStream.bytesWritten`

The `cryptoStream.bytesWritten` property returns the total number of bytes written to the underlying socket *including* the bytes required for the implementation of the TLS protocol.

Class: `tls.SecurePair`

Stability: 0 - Deprecated: Use `tls.TLSSocket` instead.

Returned by `tls.createSecurePair()`.

Event: `'secure'`

The `'secure'` event is emitted by the `SecurePair` object once a secure connection has been established.

As with checking for the server `'secureConnection'` event, `pair.cleartext.authorized` should be inspected to confirm whether the certificate used is properly authorized.

Class: `tls.Server`

- Extends: {net.Server}

Accepts encrypted connections using TLS or SSL.

Event: `'connection'`

- `socket` {stream.Duplex}

This event is emitted when a new TCP stream is established, before the TLS handshake begins. `socket` is typically an object of type `net.Socket` but will not receive events unlike the socket created from the `net.Server` `'connection'` event. Usually users will not want to access this event.

This event can also be explicitly emitted by users to inject connections into the TLS server. In that case, any `Duplex` stream can be passed.

Event: `'keylog'`

- `line` {Buffer} Line of ASCII text, in NSS `SSLKEYLOGFILE` format.
- `tlsSocket` {tls.TLSSocket} The `tls.TLSSocket` instance on which it was generated.

The `keylog` event is emitted when key material is generated or received by a connection to this server (typically before handshake has completed, but not necessarily). This keying material can be stored for debugging, as it allows

captured TLS traffic to be decrypted. It may be emitted multiple times for each socket.

A typical use case is to append received lines to a common text file, which is later used by software (such as Wireshark) to decrypt the traffic:

```
const logFile = fs.createWriteStream('/tmp/ssl-keys.log', { flags: 'a' });
// ...
server.on('keylog', (line, tlsSocket) => {
  if (tlsSocket.remoteAddress !== '...')
    return; // Only log keys for a particular IP
  logFile.write(line);
});
```

Event: `'newSession'`

The `'newSession'` event is emitted upon creation of a new TLS session. This may be used to store sessions in external storage. The data should be provided to the `'resumeSession'` callback.

The listener callback is passed three arguments when called:

- `sessionId` {Buffer} The TLS session identifier
- `sessionData` {Buffer} The TLS session data
- `callback` {Function} A callback function taking no arguments that must be invoked in order for data to be sent or received over the secure connection.

Listening for this event will have an effect only on connections established after the addition of the event listener.

Event: `'OCSPRequest'`

The `'OCSPRequest'` event is emitted when the client sends a certificate status request. The listener callback is passed three arguments when called:

- `certificate` {Buffer} The server certificate
- `issuer` {Buffer} The issuer's certificate
- `callback` {Function} A callback function that must be invoked to provide the results of the OCSP request.

The server's current certificate can be parsed to obtain the OCSP URL and certificate ID; after obtaining an OCSP response, `callback(null, resp)` is then invoked, where `resp` is a `Buffer` instance containing the OCSP response. Both `certificate` and `issuer` are `Buffer` DER-representations of the primary and issuer's certificates. These can be used to obtain the OCSP certificate ID and OCSP endpoint URL.

Alternatively, `callback(null, null)` may be called, indicating that there was no OCSP response.

Calling `callback(err)` will result in a `socket.destroy(err)` call.

The typical flow of an OCSP request is as follows:

1. Client connects to the server and sends an `'OCSPRequest'` (via the status info extension in ClientHello).
2. Server receives the request and emits the `'OCSPRequest'` event, calling the listener if registered.
3. Server extracts the OCSP URL from either the `certificate` or `issuer` and performs an OCSP request to the CA.
4. Server receives `'OCSPResponse'` from the CA and sends it back to the client via the `callback` argument
5. Client validates the response and either destroys the socket or performs a handshake.

The `issuer` can be `null` if the certificate is either self-signed or the issuer is not in the root certificates list. (An issuer may be provided via the `ca` option when establishing the TLS connection.)

Listening for this event will have an effect only on connections established after the addition of the event listener.

An npm module like asn1.js may be used to parse the certificates.

Event: `'resumeSession'`

The `'resumeSession'` event is emitted when the client requests to resume a previous TLS session. The listener callback is passed two arguments when called:

- `sessionId` {Buffer} The TLS session identifier
- `callback` {Function} A callback function to be called when the prior session has been recovered: `callback([err[, sessionData]])`
  - `err` {Error}
  - `sessionData` {Buffer}

The event listener should perform a lookup in external storage for the `sessionData` saved by the `'newSession'` event handler using the given `sessionId`. If found, call `callback(null, sessionData)` to resume the session. If not found, the session cannot be resumed. `callback()` must be called without `sessionData` so that the handshake can continue and a new session can be created. It is possible to call `callback(err)` to terminate the incoming connection and destroy the socket.

Listening for this event will have an effect only on connections established after the addition of the event listener.

The following illustrates resuming a TLS session:

```
const tlsSessionStore = {};
server.on('newSession', (id, data, cb) => {
  tlsSessionStore[id.toString('hex')] = data;
  cb();
```

```
});
server.on('resumeSession', (id, cb) => {
  cb(null, tlsSessionStore[id.toString('hex')] || null);
});
```

Event: `'secureConnection'`

The `'secureConnection'` event is emitted after the handshaking process for a new connection has successfully completed. The listener callback is passed a single argument when called:

- `tlsSocket` {tls.TLSSocket} The established TLS socket.

The `tlsSocket.authorized` property is a `boolean` indicating whether the client has been verified by one of the supplied Certificate Authorities for the server. If `tlsSocket.authorized` is `false`, then `socket.authorizationError` is set to describe how authorization failed. Depending on the settings of the TLS server, unauthorized connections may still be accepted.

The `tlsSocket.alpnProtocol` property is a string that contains the selected ALPN protocol. When ALPN has no selected protocol, `tlsSocket.alpnProtocol` equals `false`.

The `tlsSocket.servername` property is a string containing the server name requested via SNI.

Event: `'tlsClientError'`

The `'tlsClientError'` event is emitted when an error occurs before a secure connection is established. The listener callback is passed two arguments when called:

- `exception` {Error} The `Error` object describing the error
- `tlsSocket` {tls.TLSSocket} The `tls.TLSSocket` instance from which the error originated.

`server.addContext(hostname, context)`

- `hostname` {string} A SNI host name or wildcard (e.g. `'*'`)
- `context` {Object} An object containing any of the possible properties from the `tls.createSecureContext()` `options` arguments (e.g. `key`, `cert`, `ca`, etc).

The `server.addContext()` method adds a secure context that will be used if the client request's SNI name matches the supplied `hostname` (or wildcard).

When there are multiple matching contexts, the most recently added one is used.

`server.address()`

- Returns: {Object}

Returns the bound address, the address family name, and port of the server as reported by the operating system. See `net.Server.address()` for more information.

`server.close([callback])`

- `callback` {Function} A listener callback that will be registered to listen for the server instance's `'close'` event.
- Returns: {tls.Server}

The `server.close()` method stops the server from accepting new connections.

This function operates asynchronously. The `'close'` event will be emitted when the server has no more open connections.

`server.getTicketKeys()`

- Returns: {Buffer} A 48-byte buffer containing the session ticket keys.

Returns the session ticket keys.

See Session Resumption for more information.

`server.listen()`

Starts the server listening for encrypted connections. This method is identical to `server.listen()` from `net.Server`.

`server.setSecureContext(options)`

- `options` {Object} An object containing any of the possible properties from the `tls.createSecureContext()` `options` arguments (e.g. `key`, `cert`, `ca`, etc).

The `server.setSecureContext()` method replaces the secure context of an existing server. Existing connections to the server are not interrupted.

`server.setTicketKeys(keys)`

- `keys` {Buffer|TypedArray|DataView} A 48-byte buffer containing the session ticket keys.

Sets the session ticket keys.

Changes to the ticket keys are effective only for future server connections. Existing or currently pending server connections will use the previous keys.

See Session Resumption for more information.

Class: `tls.TLSSocket`

- Extends: {net.Socket}

Performs transparent encryption of written data and all required TLS negotiation.

Instances of `tls.TLSSocket` implement the duplex Stream interface.

Methods that return TLS connection metadata (e.g. `tls.TLSSocket.getPeerCertificate()` will only return data while the connection is open.

`new tls.TLSSocket(socket[, options])`

- `socket` {net.Socket|stream.Duplex} On the server side, any `Duplex` stream. On the client side, any instance of `net.Socket` (for generic `Duplex` stream support on the client side, `tls.connect()` must be used).
- `options` {Object}
  - `enableTrace`: See `tls.createServer()`
  - `isServer`: The SSL/TLS protocol is asymmetrical, TLSSockets must know if they are to behave as a server or a client. If `true` the TLS socket will be instantiated as a server. **Default:** `false`.
  - `server` {net.Server} A `net.Server` instance.
  - `requestCert`: Whether to authenticate the remote peer by requesting a certificate. Clients always request a server certificate. Servers (`isServer` is true) may set `requestCert` to true to request a client certificate.
  - `rejectUnauthorized`: See `tls.createServer()`
  - `ALPNProtocols`: See `tls.createServer()`
  - `SNICallback`: See `tls.createServer()`
  - `session` {Buffer} A `Buffer` instance containing a TLS session.
  - `requestOCSP` {boolean} If `true`, specifies that the OCSP status request extension will be added to the client hello and an `'OCSPResponse'` event will be emitted on the socket before establishing a secure communication
  - `secureContext`: TLS context object created with `tls.createSecureContext()`. If a `secureContext` is *not* provided, one will be created by passing the entire `options` object to `tls.createSecureContext()`.
  - `...`: `tls.createSecureContext()` options that are used if the `secureContext` option is missing. Otherwise, they are ignored.

Construct a new `tls.TLSSocket` object from an existing TCP socket.

Event: `'keylog'`

- `line` {Buffer} Line of ASCII text, in NSS `SSLKEYLOGFILE` format.

The `keylog` event is emitted on a `tls.TLSSocket` when key material is generated or received by the socket. This keying material can be stored for debugging,

as it allows captured TLS traffic to be decrypted. It may be emitted multiple times, before or after the handshake completes.

A typical use case is to append received lines to a common text file, which is later used by software (such as Wireshark) to decrypt the traffic:

```
const logFile = fs.createWriteStream('/tmp/ssl-keys.log', { flags: 'a' });
// ...
tlsSocket.on('keylog', (line) => logFile.write(line));
```

Event: `'OCSPResponse'`

The `'OCSPResponse'` event is emitted if the `requestOCSP` option was set when the `tls.TLSSocket` was created and an OCSP response has been received. The listener callback is passed a single argument when called:

- `response` {Buffer} The server's OCSP response

Typically, the `response` is a digitally signed object from the server's CA that contains information about server's certificate revocation status.

Event: `'secureConnect'`

The `'secureConnect'` event is emitted after the handshaking process for a new connection has successfully completed. The listener callback will be called regardless of whether or not the server's certificate has been authorized. It is the client's responsibility to check the `tlsSocket.authorized` property to determine if the server certificate was signed by one of the specified CAs. If `tlsSocket.authorized === false`, then the error can be found by examining the `tlsSocket.authorizationError` property. If ALPN was used, the `tlsSocket.alpnProtocol` property can be checked to determine the negotiated protocol.

The `'secureConnect'` event is not emitted when a {tls.TLSSocket} is created using the `new tls.TLSSocket()` constructor.

Event: `'session'`

- `session` {Buffer}

The `'session'` event is emitted on a client `tls.TLSSocket` when a new session or TLS ticket is available. This may or may not be before the handshake is complete, depending on the TLS protocol version that was negotiated. The event is not emitted on the server, or if a new session was not created, for example, when the connection was resumed. For some TLS protocol versions the event may be emitted multiple times, in which case all the sessions can be used for resumption.

On the client, the `session` can be provided to the `session` option of `tls.connect()` to resume the connection.

See Session Resumption for more information.

For TLSv1.2 and below, `tls.TLSSocket.getSession()` can be called once the handshake is complete. For TLSv1.3, only ticket-based resumption is allowed by the protocol, multiple tickets are sent, and the tickets aren't sent until after the handshake completes. So it is necessary to wait for the `'session'` event to get a resumable session. Applications should use the `'session'` event instead of `getSession()` to ensure they will work for all TLS versions. Applications that only expect to get or use one session should listen for this event only once:

```
tlsSocket.once('session', (session) => {
  // The session can be used immediately or later.
  tls.connect({
    session: session,
    // Other connect options...
  });
});
```

`tlsSocket.address()`

- Returns: {Object}

Returns the bound `address`, the address `family` name, and `port` of the underlying socket as reported by the operating system: `{ port: 12346, family: 'IPv4', address: '127.0.0.1' }`.

`tlsSocket.authorizationError`

Returns the reason why the peer's certificate was not been verified. This property is set only when `tlsSocket.authorized === false`.

`tlsSocket.authorized`

- Returns: {boolean}

Returns `true` if the peer certificate was signed by one of the CAs specified when creating the `tls.TLSSocket` instance, otherwise `false`.

`tlsSocket.disableRenegotiation()`

Disables TLS renegotiation for this `TLSSocket` instance. Once called, attempts to renegotiate will trigger an `'error'` event on the `TLSSocket`.

`tlsSocket.enableTrace()`

When enabled, TLS packet trace information is written to `stderr`. This can be used to debug TLS connection problems.

The format of the output is identical to the output of `openssl s_client -trace` or `openssl s_server -trace`. While it is produced by OpenSSL's

`SSL_trace()` function, the format is undocumented, can change without notice, and should not be relied on.

`tlsSocket.encrypted`

Always returns `true`. This may be used to distinguish TLS sockets from regular `net.Socket` instances.

`tlsSocket.exportKeyingMaterial(length, label[, context])`

- `length` {number} number of bytes to retrieve from keying material
- `label` {string} an application specific label, typically this will be a value from the IANA Exporter Label Registry.
- `context` {Buffer} Optionally provide a context.
- Returns: {Buffer} requested bytes of the keying material

Keying material is used for validations to prevent different kind of attacks in network protocols, for example in the specifications of IEEE 802.1X.

Example

```
const keyingMaterial = tlsSocket.exportKeyingMaterial(
  128,
  'client finished');

/*
 Example return value of keyingMaterial:
 <Buffer 76 26 af 99 c5 56 8e 42 09 91 ef 9f 93 cb ad 6c 7b 65 f8 53 f1 d8 d9
    12 5a 33 b8 b5 25 df 7b 37 9f e0 e2 4f b8 67 83 a3 2f cd 5d 41 42 4c 91
    74 ef 2c ... 78 more bytes>
*/
```

See the OpenSSL `SSL_export_keying_material` documentation for more information.

`tlsSocket.getCertificate()`

- Returns: {Object}

Returns an object representing the local certificate. The returned object has some properties corresponding to the fields of the certificate.

See `tls.TLSSocket.getPeerCertificate()` for an example of the certificate structure.

If there is no local certificate, an empty object will be returned. If the socket has been destroyed, `null` will be returned.

`tlsSocket.getCipher()`

- Returns: {Object}
  - `name` {string} OpenSSL name for the cipher suite.
  - `standardName` {string} IETF name for the cipher suite.
  - `version` {string} The minimum TLS protocol version supported by this cipher suite.

Returns an object containing information on the negotiated cipher suite.

For example:

```
{
    "name": "AES128-SHA256",
    "standardName": "TLS_RSA_WITH_AES_128_CBC_SHA256",
    "version": "TLSv1.2"
}
```

See SSL_CIPHER_get_name for more information.

`tlsSocket.getEphemeralKeyInfo()`

- Returns: {Object}

Returns an object representing the type, name, and size of parameter of an ephemeral key exchange in perfect forward secrecy on a client connection. It returns an empty object when the key exchange is not ephemeral. As this is only supported on a client socket; `null` is returned if called on a server socket. The supported types are `'DH'` and `'ECDH'`. The `name` property is available only when type is `'ECDH'`.

For example: `{ type: 'ECDH', name: 'prime256v1', size: 256 }`.

`tlsSocket.getFinished()`

- Returns: {Buffer|undefined} The latest `Finished` message that has been sent to the socket as part of a SSL/TLS handshake, or `undefined` if no `Finished` message has been sent yet.

As the `Finished` messages are message digests of the complete handshake (with a total of 192 bits for TLS 1.0 and more for SSL 3.0), they can be used for external authentication procedures when the authentication provided by SSL/TLS is not desired or is not enough.

Corresponds to the `SSL_get_finished` routine in OpenSSL and may be used to implement the `tls-unique` channel binding from RFC 5929.

`tlsSocket.getPeerCertificate([detailed])`

- `detailed` {boolean} Include the full certificate chain if `true`, otherwise include just the peer's certificate.

- Returns: {Object} A certificate object.

Returns an object representing the peer's certificate. If the peer does not provide a certificate, an empty object will be returned. If the socket has been destroyed, `null` will be returned.

If the full certificate chain was requested, each certificate will include an `issuerCertificate` property containing an object representing its issuer's certificate.

Certificate object A certificate object has properties corresponding to the fields of the certificate.

- `raw` {Buffer} The DER encoded X.509 certificate data.
- `subject` {Object} The certificate subject, described in terms of Country (`C`), StateOrProvince (`ST`), Locality (`L`), Organization (`O`), OrganizationalUnit (`OU`), and CommonName (`CN`). The CommonName is typically a DNS name with TLS certificates. Example: `{C: 'UK', ST: 'BC', L: 'Metro', O: 'Node Fans', OU: 'Docs', CN: 'example.com'}`.
- `issuer` {Object} The certificate issuer, described in the same terms as the `subject`.
- `valid_from` {string} The date-time the certificate is valid from.
- `valid_to` {string} The date-time the certificate is valid to.
- `serialNumber` {string} The certificate serial number, as a hex string. Example: `'B9B0D332A1AA5635'`.
- `fingerprint` {string} The SHA-1 digest of the DER encoded certificate. It is returned as a `:` separated hexadecimal string. Example: `'2A:7A:C2:DD:...'`.
- `fingerprint256` {string} The SHA-256 digest of the DER encoded certificate. It is returned as a `:` separated hexadecimal string. Example: `'2A:7A:C2:DD:...'`.
- `fingerprint512` {string} The SHA-512 digest of the DER encoded certificate. It is returned as a `:` separated hexadecimal string. Example: `'2A:7A:C2:DD:...'`.
- `ext_key_usage` {Array} (Optional) The extended key usage, a set of OIDs.
- `subjectaltname` {string} (Optional) A string containing concatenated names for the subject, an alternative to the `subject` names.
- `infoAccess` {Array} (Optional) An array describing the AuthorityInfoAccess, used with OCSP.
- `issuerCertificate` {Object} (Optional) The issuer certificate object. For self-signed certificates, this may be a circular reference.

The certificate may contain information about the public key, depending on the key type.

For RSA keys, the following properties may be defined:

- `bits` {number} The RSA bit size. Example: `1024`.

- `exponent` {string} The RSA exponent, as a string in hexadecimal number notation. Example: `'0x010001'`.
- `modulus` {string} The RSA modulus, as a hexadecimal string. Example: `'B56CE45CB7...'`.
- `pubkey` {Buffer} The public key.

For EC keys, the following properties may be defined:

- `pubkey` {Buffer} The public key.
- `bits` {number} The key size in bits. Example: `256`.
- `asn1Curve` {string} (Optional) The ASN.1 name of the OID of the elliptic curve. Well-known curves are identified by an OID. While it is unusual, it is possible that the curve is identified by its mathematical properties, in which case it will not have an OID. Example: `'prime256v1'`.
- `nistCurve` {string} (Optional) The NIST name for the elliptic curve, if it has one (not all well-known curves have been assigned names by NIST). Example: `'P-256'`.

Example certificate:

```
{ subject:
   { OU: [ 'Domain Control Validated', 'PositiveSSL Wildcard' ],
     CN: '*.nodejs.org' },
  issuer:
   { C: 'GB',
     ST: 'Greater Manchester',
     L: 'Salford',
     O: 'COMODO CA Limited',
     CN: 'COMODO RSA Domain Validation Secure Server CA' },
  subjectaltname: 'DNS:*.nodejs.org, DNS:nodejs.org',
  infoAccess:
   { 'CA Issuers - URI':
      [ 'http://crt.comodoca.com/COMODORSADomainValidationSecureServerCA.crt' ],
     'OCSP - URI': [ 'http://ocsp.comodoca.com' ] },
  modulus: 'B56CE45CB740B09A13F64AC543B712FF9EE8E4C284B542A1708A27E82A8D151CA178153E12E6DDA1
  exponent: '0x10001',
  pubkey: <Buffer ... >,
  valid_from: 'Aug 14 00:00:00 2017 GMT',
  valid_to: 'Nov 20 23:59:59 2019 GMT',
  fingerprint: '01:02:59:D9:C3:D2:0D:08:F7:82:4E:44:A4:B4:53:C5:E2:3A:87:4D',
  fingerprint256: '69:AE:1A:6A:D4:3D:C6:C1:1B:EA:C6:23:DE:BA:2A:14:62:62:93:5C:7A:EA:06:41:9
  fingerprint512: '19:2B:3E:C3:B3:5B:32:E8:AE:BB:78:97:27:E4:BA:6C:39:C9:92:79:4F:31:46:39:B
  ext_key_usage: [ '1.3.6.1.5.5.7.3.1', '1.3.6.1.5.5.7.3.2' ],
  serialNumber: '66593D57F20CBC573E433381B5FEC280',
  raw: <Buffer ... > }
```

`tlsSocket.getPeerFinished()`

- Returns: {Buffer|undefined} The latest `Finished` message that is expected or has actually been received from the socket as part of a SSL/TLS handshake, or `undefined` if there is no `Finished` message so far.

As the `Finished` messages are message digests of the complete handshake (with a total of 192 bits for TLS 1.0 and more for SSL 3.0), they can be used for external authentication procedures when the authentication provided by SSL/TLS is not desired or is not enough.

Corresponds to the `SSL_get_peer_finished` routine in OpenSSL and may be used to implement the `tls-unique` channel binding from RFC 5929.

`tlsSocket.getPeerX509Certificate()`

- Returns: {X509Certificate}

Returns the peer certificate as an {X509Certificate} object.

If there is no peer certificate, or the socket has been destroyed, `undefined` will be returned.

`tlsSocket.getProtocol()`

- Returns: {string|null}

Returns a string containing the negotiated SSL/TLS protocol version of the current connection. The value `'unknown'` will be returned for connected sockets that have not completed the handshaking process. The value `null` will be returned for server sockets or disconnected client sockets.

Protocol versions are:

- `'SSLv3'`
- `'TLSv1'`
- `'TLSv1.1'`
- `'TLSv1.2'`
- `'TLSv1.3'`

See the OpenSSL `SSL_get_version` documentation for more information.

`tlsSocket.getSession()`

- {Buffer}

Returns the TLS session data or `undefined` if no session was negotiated. On the client, the data can be provided to the `session` option of `tls.connect()` to resume the connection. On the server, it may be useful for debugging.

See Session Resumption for more information.

Note: `getSession()` works only for TLSv1.2 and below. For TLSv1.3, applications must use the `'session'` event (it also works for TLSv1.2 and below).

`tlsSocket.getSharedSigalgs()`

- Returns: {Array} List of signature algorithms shared between the server and the client in the order of decreasing preference.

See SSL_get_shared_sigalgs for more information.

`tlsSocket.getTLSTicket()`

- {Buffer}

For a client, returns the TLS session ticket if one is available, or `undefined`. For a server, always returns `undefined`.

It may be useful for debugging.

See Session Resumption for more information.

`tlsSocket.getX509Certificate()`

- Returns: {X509Certificate}

Returns the local certificate as an {X509Certificate} object.

If there is no local certificate, or the socket has been destroyed, `undefined` will be returned.

`tlsSocket.isSessionReused()`

- Returns: {boolean} `true` if the session was reused, `false` otherwise.

See Session Resumption for more information.

`tlsSocket.localAddress`

- {string}

Returns the string representation of the local IP address.

`tlsSocket.localPort`

- {integer}

Returns the numeric representation of the local port.

`tlsSocket.remoteAddress`

- {string}

Returns the string representation of the remote IP address. For example, `'74.125.127.100'` or `'2001:4860:a005::68'`.

`tlsSocket.remoteFamily`

- {string}

Returns the string representation of the remote IP family. `'IPv4'` or `'IPv6'`.

`tlsSocket.remotePort`

- {integer}

Returns the numeric representation of the remote port. For example, `443`.

`tlsSocket.renegotiate(options, callback)`

- `options` {Object}
  - `rejectUnauthorized` {boolean} If not `false`, the server certificate is verified against the list of supplied CAs. An `'error'` event is emitted if verification fails; `err.code` contains the OpenSSL error code. **Default:** `true`.
  - `requestCert`
- `callback` {Function} If `renegotiate()` returned `true`, callback is attached once to the `'secure'` event. If `renegotiate()` returned `false`, `callback` will be called in the next tick with an error, unless the `tlsSocket` has been destroyed, in which case `callback` will not be called at all.
- Returns: {boolean} `true` if renegotiation was initiated, `false` otherwise.

The `tlsSocket.renegotiate()` method initiates a TLS renegotiation process. Upon completion, the `callback` function will be passed a single argument that is either an `Error` (if the request failed) or `null`.

This method can be used to request a peer's certificate after the secure connection has been established.

When running as the server, the socket will be destroyed with an error after `handshakeTimeout` timeout.

For TLSv1.3, renegotiation cannot be initiated, it is not supported by the protocol.

`tlsSocket.setMaxSendFragment(size)`

- `size` {number} The maximum TLS fragment size. The maximum value is `16384`. **Default:** `16384`.
- Returns: {boolean}

The `tlsSocket.setMaxSendFragment()` method sets the maximum TLS fragment size. Returns `true` if setting the limit succeeded; `false` otherwise.

Smaller fragment sizes decrease the buffering latency on the client: larger fragments are buffered by the TLS layer until the entire fragment is received and its integrity is verified; large fragments can span multiple roundtrips and their processing can be delayed due to packet loss or reordering. However, smaller fragments add extra TLS framing bytes and CPU overhead, which may decrease overall server throughput.

`tls.checkServerIdentity(hostname, cert)`

- `hostname` {string} The host name or IP address to verify the certificate against.
- `cert` {Object} A certificate object representing the peer's certificate.
- Returns: {Error|undefined}

Verifies the certificate `cert` is issued to `hostname`.

Returns {Error} object, populating it with `reason`, `host`, and `cert` on failure. On success, returns {undefined}.

This function is intended to be used in combination with the `checkServerIdentity` option that can be passed to `tls.connect()` and as such operates on a certificate object. For other purposes, consider using `x509.checkHost()` instead.

This function can be overwritten by providing an alternative function as the `options.checkServerIdentity` option that is passed to `tls.connect()`. The overwriting function can call `tls.checkServerIdentity()` of course, to augment the checks done with additional verification.

This function is only called if the certificate passed all other checks, such as being issued by trusted CA (`options.ca`).

Earlier versions of Node.js incorrectly accepted certificates for a given `hostname` if a matching `uniformResourceIdentifier` subject alternative name was present (see CVE-2021-44531). Applications that wish to accept `uniformResourceIdentifier` subject alternative names can use a custom `options.checkServerIdentity` function that implements the desired behavior.

`tls.connect(options[, callback])`

- `options` {Object}
  - `enableTrace`: See `tls.createServer()`

- `host` {string} Host the client should connect to. **Default:** `'localhost'`.
- `port` {number} Port the client should connect to.
- `path` {string} Creates Unix socket connection to path. If this option is specified, `host` and `port` are ignored.
- `socket` {stream.Duplex} Establish secure connection on a given socket rather than creating a new socket. Typically, this is an instance of `net.Socket`, but any `Duplex` stream is allowed. If this option is specified, `path`, `host` and `port` are ignored, except for certificate validation. Usually, a socket is already connected when passed to `tls.connect()`, but it can be connected later. Connection/disconnection/destruction of `socket` is the user's responsibility; calling `tls.connect()` will not cause `net.connect()` to be called.
- `allowHalfOpen` {boolean} If set to `false`, then the socket will automatically end the writable side when the readable side ends. If the `socket` option is set, this option has no effect. See the `allowHalfOpen` option of `net.Socket` for details. **Default:** `false`.
- `rejectUnauthorized` {boolean} If not `false`, the server certificate is verified against the list of supplied CAs. An `'error'` event is emitted if verification fails; `err.code` contains the OpenSSL error code. **Default:** `true`.
- `pskCallback` {Function}
  - hint: {string} optional message sent from the server to help client decide which identity to use during negotiation. Always `null` if TLS 1.3 is used.
  - Returns: {Object} in the form `{ psk: <Buffer|TypedArray|DataView>, identity: <string> }` or `null` to stop the negotiation process. `psk` must be compatible with the selected cipher's digest. `identity` must use UTF-8 encoding.

  When negotiating TLS-PSK (pre-shared keys), this function is called with optional identity `hint` provided by the server or `null` in case of TLS 1.3 where `hint` was removed. It will be necessary to provide a custom `tls.checkServerIdentity()` for the connection as the default one will try to check host name/IP of the server against the certificate but that's not applicable for PSK because there won't be a certificate present. More information can be found in the RFC 4279.
- `ALPNProtocols`: {string[]|Buffer[]|TypedArray[]|DataView[]|Buffer|TypedArray|DataView} An array of strings, `Buffer`s or `TypedArray`s or `DataView`s, or a single `Buffer` or `TypedArray` or `DataView` containing the supported ALPN protocols. `Buffer`s should have the format `[len][name][len][name]...` e.g. `'\x08http/1.1\x08http/1.0'`, where the `len` byte is the length of the next protocol name. Passing an array is usually much simpler, e.g. `['http/1.1', 'http/1.0']`. Protocols earlier in the list have higher preference than those later.
- `servername`: {string} Server name for the SNI (Server Name Indication) TLS extension. It is the name of the host being connected to,

and must be a host name, and not an IP address. It can be used by a multi-homed server to choose the correct certificate to present to the client, see the `SNICallback` option to `tls.createServer()`.
  - `checkServerIdentity(servername, cert)` {Function} A callback function to be used (instead of the builtin `tls.checkServerIdentity()` function) when checking the server's host name (or the provided `servername` when explicitly set) against the certificate. This should return an {Error} if verification fails. The method should return `undefined` if the `servername` and `cert` are verified.
  - `session` {Buffer} A `Buffer` instance, containing TLS session.
  - `minDHSize` {number} Minimum size of the DH parameter in bits to accept a TLS connection. When a server offers a DH parameter with a size less than `minDHSize`, the TLS connection is destroyed and an error is thrown. **Default:** `1024`.
  - `highWaterMark`: {number} Consistent with the readable stream `highWaterMark` parameter. **Default:** `16 * 1024`.
  - `secureContext`: TLS context object created with `tls.createSecureContext()`. If a `secureContext` is *not* provided, one will be created by passing the entire `options` object to `tls.createSecureContext()`.
  - `onread` {Object} If the `socket` option is missing, incoming data is stored in a single `buffer` and passed to the supplied `callback` when data arrives on the socket, otherwise the option is ignored. See the `onread` option of `net.Socket` for details.
  - `...`: `tls.createSecureContext()` options that are used if the `secureContext` option is missing, otherwise they are ignored.
  - `...`: Any `socket.connect()` option not already listed.
- `callback` {Function}
- Returns: {tls.TLSSocket}

The `callback` function, if specified, will be added as a listener for the `'secureConnect'` event.

`tls.connect()` returns a `tls.TLSSocket` object.

Unlike the `https` API, `tls.connect()` does not enable the SNI (Server Name Indication) extension by default, which may cause some servers to return an incorrect certificate or reject the connection altogether. To enable SNI, set the `servername` option in addition to `host`.

The following illustrates a client for the echo server example from `tls.createServer()`:

```
// Assumes an echo server that is listening on port 8000.
const tls = require('tls');
const fs = require('fs');

const options = {
  // Necessary only if the server requires client certificate authentication.
```

```
  key: fs.readFileSync('client-key.pem'),
  cert: fs.readFileSync('client-cert.pem'),

  // Necessary only if the server uses a self-signed certificate.
  ca: [ fs.readFileSync('server-cert.pem') ],

  // Necessary only if the server's cert isn't for "localhost".
  checkServerIdentity: () => { return null; },
};

const socket = tls.connect(8000, options, () => {
  console.log('client connected',
              socket.authorized ? 'authorized' : 'unauthorized');
  process.stdin.pipe(socket);
  process.stdin.resume();
});
socket.setEncoding('utf8');
socket.on('data', (data) => {
  console.log(data);
});
socket.on('end', () => {
  console.log('server ends connection');
});
```

`tls.connect(path[, options][, callback])`

- `path` {string} Default value for `options.path`.
- `options` {Object} See `tls.connect()`.
- `callback` {Function} See `tls.connect()`.
- Returns: {tls.TLSSocket}

Same as `tls.connect()` except that `path` can be provided as an argument instead of an option.

A path option, if specified, will take precedence over the path argument.

`tls.connect(port[, host][, options][, callback])`

- `port` {number} Default value for `options.port`.
- `host` {string} Default value for `options.host`.
- `options` {Object} See `tls.connect()`.
- `callback` {Function} See `tls.connect()`.
- Returns: {tls.TLSSocket}

Same as `tls.connect()` except that `port` and `host` can be provided as arguments instead of options.

A port or host option, if specified, will take precedence over any port or host argument.

`tls.createSecureContext([options])`

- `options` {Object}
  - `ca` {string|string[]|Buffer|Buffer[]} Optionally override the trusted CA certificates. Default is to trust the well-known CAs curated by Mozilla. Mozilla's CAs are completely replaced when CAs are explicitly specified using this option. The value can be a string or `Buffer`, or an `Array` of strings and/or `Buffer`s. Any string or `Buffer` can contain multiple PEM CAs concatenated together. The peer's certificate must be chainable to a CA trusted by the server for the connection to be authenticated. When using certificates that are not chainable to a well-known CA, the certificate's CA must be explicitly specified as a trusted or the connection will fail to authenticate. If the peer uses a certificate that doesn't match or chain to one of the default CAs, use the `ca` option to provide a CA certificate that the peer's certificate can match or chain to. For self-signed certificates, the certificate is its own CA, and must be provided. For PEM encoded certificates, supported types are "TRUSTED CERTIFICATE", "X509 CERTIFICATE", and "CERTIFICATE". See also `tls.rootCertificates`.
  - `cert` {string|string[]|Buffer|Buffer[]} Cert chains in PEM format. One cert chain should be provided per private key. Each cert chain should consist of the PEM formatted certificate for a provided private `key`, followed by the PEM formatted intermediate certificates (if any), in order, and not including the root CA (the root CA must be pre-known to the peer, see `ca`). When providing multiple cert chains, they do not have to be in the same order as their private keys in `key`. If the intermediate certificates are not provided, the peer will not be able to validate the certificate, and the handshake will fail.
  - `sigalgs` {string} Colon-separated list of supported signature algorithms. The list can contain digest algorithms (`SHA256`, `MD5` etc.), public key algorithms (`RSA-PSS`, `ECDSA` etc.), combination of both (e.g 'RSA+SHA384') or TLS v1.3 scheme names (e.g. `rsa_pss_pss_sha512`). See OpenSSL man pages for more info.
  - `ciphers` {string} Cipher suite specification, replacing the default. For more information, see Modifying the default TLS cipher suite. Permitted ciphers can be obtained via `tls.getCiphers()`. Cipher names must be uppercased in order for OpenSSL to accept them.
  - `clientCertEngine` {string} Name of an OpenSSL engine which can provide the client certificate.
  - `crl` {string|string[]|Buffer|Buffer[]} PEM formatted CRLs (Certificate Revocation Lists).
  - `dhparam` {string|Buffer} Diffie-Hellman parameters, required for perfect forward secrecy. Use `openssl dhparam` to create the parameters.

The key length must be greater than or equal to 1024 bits or else an error will be thrown. Although 1024 bits is permissible, use 2048 bits or larger for stronger security. If omitted or invalid, the parameters are silently discarded and DHE ciphers will not be available.

- `ecdhCurve` {string} A string describing a named curve or a colon separated list of curve NIDs or names, for example `P-521:P-384:P-256`, to use for ECDH key agreement. Set to `auto` to select the curve automatically. Use `crypto.getCurves()` to obtain a list of available curve names. On recent releases, `openssl ecparam -list_curves` will also display the name and description of each available elliptic curve. **Default:** `tls.DEFAULT_ECDH_CURVE`.
- `honorCipherOrder` {boolean} Attempt to use the server's cipher suite preferences instead of the client's. When `true`, causes `SSL_OP_CIPHER_SERVER_PREFERENCE` to be set in `secureOptions`, see OpenSSL Options for more information.
- `key` {string|string[]|Buffer|Buffer[]|Object[]} Private keys in PEM format. PEM allows the option of private keys being encrypted. Encrypted keys will be decrypted with `options.passphrase`. Multiple keys using different algorithms can be provided either as an array of unencrypted key strings or buffers, or an array of objects in the form `{pem: <string|buffer>[, passphrase: <string>]}`. The object form can only occur in an array. `object.passphrase` is optional. Encrypted keys will be decrypted with `object.passphrase` if provided, or `options.passphrase` if it is not.
- `privateKeyEngine` {string} Name of an OpenSSL engine to get private key from. Should be used together with `privateKeyIdentifier`.
- `privateKeyIdentifier` {string} Identifier of a private key managed by an OpenSSL engine. Should be used together with `privateKeyEngine`. Should not be set together with `key`, because both options define a private key in different ways.
- `maxVersion` {string} Optionally set the maximum TLS version to allow. One of `'TLSv1.3'`, `'TLSv1.2'`, `'TLSv1.1'`, or `'TLSv1'`. Cannot be specified along with the `secureProtocol` option; use one or the other. **Default:** `tls.DEFAULT_MAX_VERSION`.
- `minVersion` {string} Optionally set the minimum TLS version to allow. One of `'TLSv1.3'`, `'TLSv1.2'`, `'TLSv1.1'`, or `'TLSv1'`. Cannot be specified along with the `secureProtocol` option; use one or the other. Avoid setting to less than TLSv1.2, but it may be required for interoperability. **Default:** `tls.DEFAULT_MIN_VERSION`.
- `passphrase` {string} Shared passphrase used for a single private key and/or a PFX.
- `pfx` {string|string[]|Buffer|Buffer[]|Object[]} PFX or PKCS12 encoded private key and certificate chain. `pfx` is an alternative to providing `key` and `cert` individually. PFX is usually encrypted, if it is, `passphrase` will be used to decrypt it. Multiple PFX can be provided either as an array of unencrypted PFX buffers, or an array of objects

in the form {buf: <string|buffer>[, passphrase: <string>]}. The object form can only occur in an array. object.passphrase is optional. Encrypted PFX will be decrypted with object.passphrase if provided, or options.passphrase if it is not.

- secureOptions {number} Optionally affect the OpenSSL protocol behavior, which is not usually necessary. This should be used carefully if at all! Value is a numeric bitmask of the SSL_OP_* options from OpenSSL Options.
- secureProtocol {string} Legacy mechanism to select the TLS protocol version to use, it does not support independent control of the minimum and maximum version, and does not support limiting the protocol to TLSv1.3. Use minVersion and maxVersion instead. The possible values are listed as SSL_METHODS, use the function names as strings. For example, use 'TLSv1_1_method' to force TLS version 1.1, or 'TLS_method' to allow any TLS protocol version up to TLSv1.3. It is not recommended to use TLS versions less than 1.2, but it may be required for interoperability. **Default:** none, see minVersion.
- sessionIdContext {string} Opaque identifier used by servers to ensure session state is not shared between applications. Unused by clients.
- ticketKeys: {Buffer} 48-bytes of cryptographically strong pseudo-random data. See Session Resumption for more information.
- sessionTimeout {number} The number of seconds after which a TLS session created by the server will no longer be resumable. See Session Resumption for more information. **Default:** 300.

tls.createServer() sets the default value of the honorCipherOrder option to true, other APIs that create secure contexts leave it unset.

tls.createServer() uses a 128 bit truncated SHA1 hash value generated from process.argv as the default value of the sessionIdContext option, other APIs that create secure contexts have no default value.

The tls.createSecureContext() method creates a SecureContext object. It is usable as an argument to several tls APIs, such as tls.createServer() and server.addContext(), but has no public methods.

A key is *required* for ciphers that use certificates. Either key or pfx can be used to provide it.

If the ca option is not given, then Node.js will default to using Mozilla's publicly trusted list of CAs.

tls.createSecurePair([context][, isServer][, requestCert][, rejectUnauthorized][, options])

Stability: 0 - Deprecated: Use tls.TLSSocket instead.

- `context` {Object} A secure context object as returned by `tls.createSecureContext()`
- `isServer` {boolean} `true` to specify that this TLS connection should be opened as a server.
- `requestCert` {boolean} `true` to specify whether a server should request a certificate from a connecting client. Only applies when `isServer` is `true`.
- `rejectUnauthorized` {boolean} If not `false` a server automatically reject clients with invalid certificates. Only applies when `isServer` is `true`.
- `options`
  - `enableTrace`: See `tls.createServer()`
  - `secureContext`: A TLS context object from `tls.createSecureContext()`
  - `isServer`: If `true` the TLS socket will be instantiated in server-mode. **Default:** `false`.
  - `server` {net.Server} A `net.Server` instance
  - `requestCert`: See `tls.createServer()`
  - `rejectUnauthorized`: See `tls.createServer()`
  - `ALPNProtocols`: See `tls.createServer()`
  - `SNICallback`: See `tls.createServer()`
  - `session` {Buffer} A `Buffer` instance containing a TLS session.
  - `requestOCSP` {boolean} If `true`, specifies that the OCSP status request extension will be added to the client hello and an `'OCSPResponse'` event will be emitted on the socket before establishing a secure communication.

Creates a new secure pair object with two streams, one of which reads and writes the encrypted data and the other of which reads and writes the cleartext data. Generally, the encrypted stream is piped to/from an incoming encrypted data stream and the cleartext one is used as a replacement for the initial encrypted stream.

`tls.createSecurePair()` returns a `tls.SecurePair` object with `cleartext` and `encrypted` stream properties.

Using `cleartext` has the same API as `tls.TLSSocket`.

The `tls.createSecurePair()` method is now deprecated in favor of `tls.TLSSocket()`. For example, the code:

```
pair = tls.createSecurePair(/* ... */);
pair.encrypted.pipe(socket);
socket.pipe(pair.encrypted);
```

can be replaced by:

```
secureSocket = tls.TLSSocket(socket, options);
```

where `secureSocket` has the same API as `pair.cleartext`.

`tls.createServer([options][, secureConnectionListener])`

- `options` {Object}

- `ALPNProtocols`: {string[]|Buffer[]|TypedArray[]|DataView[]|Buffer|TypedArray|DataView} An array of strings, `Buffer`s or `TypedArray`s or `DataView`s, or a single `Buffer` or `TypedArray` or `DataView` containing the supported ALPN protocols. `Buffer`s should have the format `[len][name][len][name]...` e.g. `0x05hello0x05world`, where the first byte is the length of the next protocol name. Passing an array is usually much simpler, e.g. `['hello', 'world']`. (Protocols should be ordered by their priority.)
- `clientCertEngine` {string} Name of an OpenSSL engine which can provide the client certificate.
- `enableTrace` {boolean} If `true`, `tls.TLSSocket.enableTrace()` will be called on new connections. Tracing can be enabled after the secure connection is established, but this option must be used to trace the secure connection setup. **Default:** `false`.
- `handshakeTimeout` {number} Abort the connection if the SSL/TLS handshake does not finish in the specified number of milliseconds. A `'tlsClientError'` is emitted on the `tls.Server` object whenever a handshake times out. **Default:** `120000` (120 seconds).
- `rejectUnauthorized` {boolean} If not `false` the server will reject any connection which is not authorized with the list of supplied CAs. This option only has an effect if `requestCert` is `true`. **Default:** `true`.
- `requestCert` {boolean} If `true` the server will request a certificate from clients that connect and attempt to verify that certificate. **Default:** `false`.
- `sessionTimeout` {number} The number of seconds after which a TLS session created by the server will no longer be resumable. See Session Resumption for more information. **Default:** `300`.
- `SNICallback(servername, callback)` {Function} A function that will be called if the client supports SNI TLS extension. Two arguments will be passed when called: `servername` and `callback`. `callback` is an error-first callback that takes two optional arguments: `error` and `ctx`. `ctx`, if provided, is a `SecureContext` instance. `tls.createSecureContext()` can be used to get a proper `SecureContext`. If `callback` is called with a falsy `ctx` argument, the default secure context of the server will be used. If `SNICallback` wasn't provided the default callback with high-level API will be used (see below).
- `ticketKeys`: {Buffer} 48-bytes of cryptographically strong pseudo-random data. See Session Resumption for more information.
- `pskCallback` {Function}
  * socket: {tls.TLSSocket} the server `tls.TLSSocket` instance for this connection.
  * identity: {string} identity parameter sent from the client.
  * Returns: {Buffer|TypedArray|DataView} pre-shared key that must either be a buffer or `null` to stop the negotiation process.

Returned PSK must be compatible with the selected cipher's digest.

When negotiating TLS-PSK (pre-shared keys), this function is called with the identity provided by the client. If the return value is `null` the negotiation process will stop and an "unknown_psk_identity" alert message will be sent to the other party. If the server wishes to hide the fact that the PSK identity was not known, the callback must provide some random data as `psk` to make the connection fail with "decrypt_error" before negotiation is finished. PSK ciphers are disabled by default, and using TLS-PSK thus requires explicitly specifying a cipher suite with the `ciphers` option. More information can be found in the RFC 4279.

- `pskIdentityHint` {string} optional hint to send to a client to help with selecting the identity during TLS-PSK negotiation. Will be ignored in TLS 1.3. Upon failing to set pskIdentityHint `'tlsClientError'` will be emitted with `'ERR_TLS_PSK_SET_IDENTIY_HINT_FAILED'` code.
- ...: Any `tls.createSecureContext()` option can be provided. For servers, the identity options (`pfx`, `key`/`cert` or `pskCallback`) are usually required.
- ...: Any `net.createServer()` option can be provided.

- `secureConnectionListener` {Function}
- Returns: {tls.Server}

Creates a new `tls.Server`. The `secureConnectionListener`, if provided, is automatically set as a listener for the `'secureConnection'` event.

The `ticketKeys` options is automatically shared between `cluster` module workers.

The following illustrates a simple echo server:

```
const tls = require('tls');
const fs = require('fs');

const options = {
  key: fs.readFileSync('server-key.pem'),
  cert: fs.readFileSync('server-cert.pem'),

  // This is necessary only if using client certificate authentication.
  requestCert: true,

  // This is necessary only if the client uses a self-signed certificate.
  ca: [ fs.readFileSync('client-cert.pem') ]
};

const server = tls.createServer(options, (socket) => {
  console.log('server connected',
```

```
              socket.authorized ? 'authorized' : 'unauthorized');
  socket.write('welcome!\n');
  socket.setEncoding('utf8');
  socket.pipe(socket);
});
server.listen(8000, () => {
  console.log('server bound');
});
```

The server can be tested by connecting to it using the example client from `tls.connect()`.

`tls.getCiphers()`

- Returns: {string[]}

Returns an array with the names of the supported TLS ciphers. The names are lower-case for historical reasons, but must be uppercased to be used in the `ciphers` option of `tls.createSecureContext()`.

Not all supported ciphers are enabled by default. See Modifying the default TLS cipher suite.

Cipher names that start with `'tls_'` are for TLSv1.3, all the others are for TLSv1.2 and below.

```
console.log(tls.getCiphers()); // ['aes128-gcm-sha256', 'aes128-sha', ...]
```

`tls.rootCertificates`

- {string[]}

An immutable array of strings representing the root certificates (in PEM format) from the bundled Mozilla CA store as supplied by current Node.js version.

The bundled CA store, as supplied by Node.js, is a snapshot of Mozilla CA store that is fixed at release time. It is identical on all supported platforms.

`tls.DEFAULT_ECDH_CURVE`

The default curve name to use for ECDH key agreement in a tls server. The default value is `'auto'`. See `tls.createSecureContext()` for further information.

`tls.DEFAULT_MAX_VERSION`

- {string} The default value of the `maxVersion` option of `tls.createSecureContext()`. It can be assigned any of the supported TLS protocol versions, `'TLSv1.3'`, `'TLSv1.2'`, `'TLSv1.1'`, or `'TLSv1'`. **Default:** `'TLSv1.3'`, unless changed using CLI options. Using `--tls-max-v1.2` sets the default to `'TLSv1.2'`.

Using `--tls-max-v1.3` sets the default to `'TLSv1.3'`. If multiple of the options are provided, the highest maximum is used.

`tls.DEFAULT_MIN_VERSION`

- {string} The default value of the `minVersion` option of `tls.createSecureContext()`. It can be assigned any of the supported TLS protocol versions, `'TLSv1.3'`, `'TLSv1.2'`, `'TLSv1.1'`, or `'TLSv1'`. **Default:** `'TLSv1.2'`, unless changed using CLI options. Using `--tls-min-v1.0` sets the default to `'TLSv1'`. Using `--tls-min-v1.1` sets the default to `'TLSv1.1'`. Using `--tls-min-v1.3` sets the default to `'TLSv1.3'`. If multiple of the options are provided, the lowest minimum is used.