/\*\*

\* 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:

\*

\* ```js

\* const tls = require('tls');

\* ```

\* @see [source](https://github.com/nodejs/node/blob/v17.0.0/lib/tls.js)

\*/

declare module 'tls' {

import { X509Certificate } from 'node:crypto';

import \* as net from 'node:net';

import \* as stream from 'stream';

const CLIENT\_RENEG\_LIMIT: number;

const CLIENT\_RENEG\_WINDOW: number;

interface Certificate {

/\*\*

\* Country code.

\*/

C: string;

/\*\*

\* Street.

\*/

ST: string;

/\*\*

\* Locality.

\*/

L: string;

/\*\*

\* Organization.

\*/

O: string;

/\*\*

\* Organizational unit.

\*/

OU: string;

/\*\*

\* Common name.

\*/

CN: string;

}

interface PeerCertificate {

subject: Certificate;

issuer: Certificate;

subjectaltname: string;

infoAccess: NodeJS.Dict<string[]>;

modulus: string;

exponent: string;

valid\_from: string;

valid\_to: string;

fingerprint: string;

fingerprint256: string;

ext\_key\_usage: string[];

serialNumber: string;

raw: Buffer;

}

interface DetailedPeerCertificate extends PeerCertificate {

issuerCertificate: DetailedPeerCertificate;

}

interface CipherNameAndProtocol {

/\*\*

\* The cipher name.

\*/

name: string;

/\*\*

\* SSL/TLS protocol version.

\*/

version: string;

/\*\*

\* IETF name for the cipher suite.

\*/

standardName: string;

}

interface EphemeralKeyInfo {

/\*\*

\* The supported types are 'DH' and 'ECDH'.

\*/

type: string;

/\*\*

\* The name property is available only when type is 'ECDH'.

\*/

name?: string | undefined;

/\*\*

\* The size of parameter of an ephemeral key exchange.

\*/

size: number;

}

interface KeyObject {

/\*\*

\* Private keys in PEM format.

\*/

pem: string | Buffer;

/\*\*

\* Optional passphrase.

\*/

passphrase?: string | undefined;

}

interface PxfObject {

/\*\*

\* PFX or PKCS12 encoded private key and certificate chain.

\*/

buf: string | Buffer;

/\*\*

\* Optional passphrase.

\*/

passphrase?: string | undefined;

}

interface TLSSocketOptions extends SecureContextOptions, CommonConnectionOptions {

/\*\*

\* If true the TLS socket will be instantiated in server-mode.

\* Defaults to false.

\*/

isServer?: boolean | undefined;

/\*\*

\* An optional net.Server instance.

\*/

server?: net.Server | undefined;

/\*\*

\* An optional Buffer instance containing a TLS session.

\*/

session?: Buffer | undefined;

/\*\*

\* 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

\*/

requestOCSP?: boolean | undefined;

}

/\*\*

\* 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.{@link TLSSocket.getPeerCertificate} will only return data while the

\* connection is open.

\* @since v0.11.4

\*/

class TLSSocket extends net.Socket {

/\*\*

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

\*/

constructor(socket: net.Socket, options?: TLSSocketOptions);

/\*\*

\* Returns `true` if the peer certificate was signed by one of the CAs specified

\* when creating the `tls.TLSSocket` instance, otherwise `false`.

\* @since v0.11.4

\*/

authorized: boolean;

/\*\*

\* Returns the reason why the peer's certificate was not been verified. This

\* property is set only when `tlsSocket.authorized === false`.

\* @since v0.11.4

\*/

authorizationError: Error;

/\*\*

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

\* @since v0.11.4

\*/

encrypted: boolean;

/\*\*

\* String containing the selected ALPN protocol.

\* Before a handshake has completed, this value is always null.

\* When a handshake is completed but not ALPN protocol was selected, tlsSocket.alpnProtocol equals false.

\*/

alpnProtocol: string | false | null;

/\*\*

\* Returns an object representing the local certificate. The returned object has

\* some properties corresponding to the fields of the certificate.

\*

\* See {@link 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.

\* @since v11.2.0

\*/

getCertificate(): PeerCertificate | object | null;

/\*\*

\* Returns an object containing information on the negotiated cipher suite.

\*

\* For example:

\*

\* ```json

\* {

\* "name": "AES128-SHA256",

\* "standardName": "TLS\_RSA\_WITH\_AES\_128\_CBC\_SHA256",

\* "version": "TLSv1.2"

\* }

\* ```

\*

\* See [SSL\\_CIPHER\\_get\\_name](https://www.openssl.org/docs/man1.1.1/man3/SSL\_CIPHER\_get\_name.html) for more information.

\* @since v0.11.4

\*/

getCipher(): CipherNameAndProtocol;

/\*\*

\* 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 }`.

\* @since v5.0.0

\*/

getEphemeralKeyInfo(): EphemeralKeyInfo | object | null;

/\*\*

\* 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](https://tools.ietf.org/html/rfc5929).

\* @since v9.9.0

\* @return 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.

\*/

getFinished(): Buffer | undefined;

/\*\*

\* 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.

\* @since v0.11.4

\* @param detailed Include the full certificate chain if `true`, otherwise include just the peer's certificate.

\* @return A certificate object.

\*/

getPeerCertificate(detailed: true): DetailedPeerCertificate;

getPeerCertificate(detailed?: false): PeerCertificate;

getPeerCertificate(detailed?: boolean): PeerCertificate | DetailedPeerCertificate;

/\*\*

\* 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](https://tools.ietf.org/html/rfc5929).

\* @since v9.9.0

\* @return 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.

\*/

getPeerFinished(): Buffer | undefined;

/\*\*

\* 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`](https://www.openssl.org/docs/man1.1.1/man3/SSL\_get\_version.html) documentation for more information.

\* @since v5.7.0

\*/

getProtocol(): string | null;

/\*\*

\* 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 {@link 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).

\* @since v0.11.4

\*/

getSession(): Buffer | undefined;

/\*\*

\* See [SSL\\_get\\_shared\\_sigalgs](https://www.openssl.org/docs/man1.1.1/man3/SSL\_get\_shared\_sigalgs.html) for more information.

\* @since v12.11.0

\* @return List of signature algorithms shared between the server and the client in the order of decreasing preference.

\*/

getSharedSigalgs(): string[];

/\*\*

\* 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.

\* @since v0.11.4

\*/

getTLSTicket(): Buffer | undefined;

/\*\*

\* See `Session Resumption` for more information.

\* @since v0.5.6

\* @return `true` if the session was reused, `false` otherwise.

\*/

isSessionReused(): boolean;

/\*\*

\* 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.

\* @since v0.11.8

\* @param callback 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.

\* @return `true` if renegotiation was initiated, `false` otherwise.

\*/

renegotiate(

options: {

rejectUnauthorized?: boolean | undefined;

requestCert?: boolean | undefined;

},

callback: (err: Error | null) => void

): undefined | 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.

\* @since v0.11.11

\* @param [size=16384] The maximum TLS fragment size. The maximum value is `16384`.

\*/

setMaxSendFragment(size: number): boolean;

/\*\*

\* Disables TLS renegotiation for this `TLSSocket` instance. Once called, attempts

\* to renegotiate will trigger an `'error'` event on the `TLSSocket`.

\* @since v8.4.0

\*/

disableRenegotiation(): void;

/\*\*

\* When enabled, TLS packet trace information is written to `stderr`. This can be

\* used to debug TLS connection problems.

\*

\* Note: 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.

\* @since v12.2.0

\*/

enableTrace(): void;

/\*\*

\* Returns the peer certificate as an `X509Certificate` object.

\*

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

\* @since v15.9.0

\*/

getPeerX509Certificate(): X509Certificate | undefined;

/\*\*

\* Returns the local certificate as an `X509Certificate` object.

\*

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

\* @since v15.9.0

\*/

getX509Certificate(): X509Certificate | undefined;

/\*\*

\* 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

\*

\* ```js

\* 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`](https://www.openssl.org/docs/man1.1.1/man3/SSL\_export\_keying\_material.html) documentation for more

\* information.

\* @since v13.10.0, v12.17.0

\* @param length number of bytes to retrieve from keying material

\* @param label an application specific label, typically this will be a value from the [IANA Exporter Label

\* Registry](https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml#exporter-labels).

\* @param context Optionally provide a context.

\* @return requested bytes of the keying material

\*/

exportKeyingMaterial(length: number, label: string, context: Buffer): Buffer;

addListener(event: string, listener: (...args: any[]) => void): this;

addListener(event: 'OCSPResponse', listener: (response: Buffer) => void): this;

addListener(event: 'secureConnect', listener: () => void): this;

addListener(event: 'session', listener: (session: Buffer) => void): this;

addListener(event: 'keylog', listener: (line: Buffer) => void): this;

emit(event: string | symbol, ...args: any[]): boolean;

emit(event: 'OCSPResponse', response: Buffer): boolean;

emit(event: 'secureConnect'): boolean;

emit(event: 'session', session: Buffer): boolean;

emit(event: 'keylog', line: Buffer): boolean;

on(event: string, listener: (...args: any[]) => void): this;

on(event: 'OCSPResponse', listener: (response: Buffer) => void): this;

on(event: 'secureConnect', listener: () => void): this;

on(event: 'session', listener: (session: Buffer) => void): this;

on(event: 'keylog', listener: (line: Buffer) => void): this;

once(event: string, listener: (...args: any[]) => void): this;

once(event: 'OCSPResponse', listener: (response: Buffer) => void): this;

once(event: 'secureConnect', listener: () => void): this;

once(event: 'session', listener: (session: Buffer) => void): this;

once(event: 'keylog', listener: (line: Buffer) => void): this;

prependListener(event: string, listener: (...args: any[]) => void): this;

prependListener(event: 'OCSPResponse', listener: (response: Buffer) => void): this;

prependListener(event: 'secureConnect', listener: () => void): this;

prependListener(event: 'session', listener: (session: Buffer) => void): this;

prependListener(event: 'keylog', listener: (line: Buffer) => void): this;

prependOnceListener(event: string, listener: (...args: any[]) => void): this;

prependOnceListener(event: 'OCSPResponse', listener: (response: Buffer) => void): this;

prependOnceListener(event: 'secureConnect', listener: () => void): this;

prependOnceListener(event: 'session', listener: (session: Buffer) => void): this;

prependOnceListener(event: 'keylog', listener: (line: Buffer) => void): this;

}

interface CommonConnectionOptions {

/\*\*

\* An optional TLS context object from tls.createSecureContext()

\*/

secureContext?: SecureContext | undefined;

/\*\*

\* When enabled, TLS packet trace information is written to `stderr`. This can be

\* used to debug TLS connection problems.

\* @default false

\*/

enableTrace?: boolean | undefined;

/\*\*

\* If true the server will request a certificate from clients that

\* connect and attempt to verify that certificate. Defaults to

\* false.

\*/

requestCert?: boolean | undefined;

/\*\*

\* An array of strings or a Buffer naming possible ALPN protocols.

\* (Protocols should be ordered by their priority.)

\*/

ALPNProtocols?: string[] | Uint8Array[] | Uint8Array | undefined;

/\*\*

\* SNICallback(servername, cb) <Function> A function that will be

\* called if the client supports SNI TLS extension. Two arguments

\* will be passed when called: servername and cb. SNICallback should

\* invoke cb(null, ctx), where ctx is a SecureContext instance.

\* (tls.createSecureContext(...) can be used to get a proper

\* SecureContext.) If SNICallback wasn't provided the default callback

\* with high-level API will be used (see below).

\*/

SNICallback?: ((servername: string, cb: (err: Error | null, ctx?: SecureContext) => void) => void) | undefined;

/\*\*

\* If true 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

\*/

rejectUnauthorized?: boolean | undefined;

}

interface TlsOptions extends SecureContextOptions, CommonConnectionOptions, net.ServerOpts {

/\*\*

\* 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).

\*/

handshakeTimeout?: number | undefined;

/\*\*

\* 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.

\*/

sessionTimeout?: number | undefined;

/\*\*

\* 48-bytes of cryptographically strong pseudo-random data.

\*/

ticketKeys?: Buffer | undefined;

/\*\*

\*

\* @param socket

\* @param identity identity parameter sent from the client.

\* @return 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.

\*/

pskCallback?(socket: TLSSocket, identity: string): DataView | NodeJS.TypedArray | null;

/\*\*

\* 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.

\*/

pskIdentityHint?: string | undefined;

}

interface PSKCallbackNegotation {

psk: DataView | NodeJS.TypedArray;

identity: string;

}

interface ConnectionOptions extends SecureContextOptions, CommonConnectionOptions {

host?: string | undefined;

port?: number | undefined;

path?: string | undefined; // Creates unix socket connection to path. If this option is specified, `host` and `port` are ignored.

socket?: stream.Duplex | undefined; // Establish secure connection on a given socket rather than creating a new socket

checkServerIdentity?: typeof checkServerIdentity | undefined;

servername?: string | undefined; // SNI TLS Extension

session?: Buffer | undefined;

minDHSize?: number | undefined;

lookup?: net.LookupFunction | undefined;

timeout?: number | undefined;

/\*\*

\* 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 hostname/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.

\*

\* @param hint message sent from the server to help client

\* decide which identity to use during negotiation.

\* Always `null` if TLS 1.3 is used.

\* @returns Return `null` to stop the negotiation process. `psk` must be

\* compatible with the selected cipher's digest.

\* `identity` must use UTF-8 encoding.

\*/

pskCallback?(hint: string | null): PSKCallbackNegotation | null;

}

/\*\*

\* Accepts encrypted connections using TLS or SSL.

\* @since v0.3.2

\*/

class Server extends net.Server {

constructor(secureConnectionListener?: (socket: TLSSocket) => void);

constructor(options: TlsOptions, secureConnectionListener?: (socket: TLSSocket) => void);

/\*\*

\* 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.

\* @since v0.5.3

\* @param hostname A SNI host name or wildcard (e.g. `'\*'`)

\* @param context An object containing any of the possible properties from the {@link createSecureContext} `options` arguments (e.g. `key`, `cert`, `ca`, etc).

\*/

addContext(hostname: string, context: SecureContextOptions): void;

/\*\*

\* Returns the session ticket keys.

\*

\* See `Session Resumption` for more information.

\* @since v3.0.0

\* @return A 48-byte buffer containing the session ticket keys.

\*/

getTicketKeys(): Buffer;

/\*\*

\* The `server.setSecureContext()` method replaces the secure context of an

\* existing server. Existing connections to the server are not interrupted.

\* @since v11.0.0

\* @param options An object containing any of the possible properties from the {@link createSecureContext} `options` arguments (e.g. `key`, `cert`, `ca`, etc).

\*/

setSecureContext(options: SecureContextOptions): void;

/\*\*

\* 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.

\* @since v3.0.0

\* @param keys A 48-byte buffer containing the session ticket keys.

\*/

setTicketKeys(keys: Buffer): void;

/\*\*

\* events.EventEmitter

\* 1. tlsClientError

\* 2. newSession

\* 3. OCSPRequest

\* 4. resumeSession

\* 5. secureConnection

\* 6. keylog

\*/

addListener(event: string, listener: (...args: any[]) => void): this;

addListener(event: 'tlsClientError', listener: (err: Error, tlsSocket: TLSSocket) => void): this;

addListener(event: 'newSession', listener: (sessionId: Buffer, sessionData: Buffer, callback: (err: Error, resp: Buffer) => void) => void): this;

addListener(event: 'OCSPRequest', listener: (certificate: Buffer, issuer: Buffer, callback: (err: Error | null, resp: Buffer) => void) => void): this;

addListener(event: 'resumeSession', listener: (sessionId: Buffer, callback: (err: Error, sessionData: Buffer) => void) => void): this;

addListener(event: 'secureConnection', listener: (tlsSocket: TLSSocket) => void): this;

addListener(event: 'keylog', listener: (line: Buffer, tlsSocket: TLSSocket) => void): this;

emit(event: string | symbol, ...args: any[]): boolean;

emit(event: 'tlsClientError', err: Error, tlsSocket: TLSSocket): boolean;

emit(event: 'newSession', sessionId: Buffer, sessionData: Buffer, callback: (err: Error, resp: Buffer) => void): boolean;

emit(event: 'OCSPRequest', certificate: Buffer, issuer: Buffer, callback: (err: Error | null, resp: Buffer) => void): boolean;

emit(event: 'resumeSession', sessionId: Buffer, callback: (err: Error, sessionData: Buffer) => void): boolean;

emit(event: 'secureConnection', tlsSocket: TLSSocket): boolean;

emit(event: 'keylog', line: Buffer, tlsSocket: TLSSocket): boolean;

on(event: string, listener: (...args: any[]) => void): this;

on(event: 'tlsClientError', listener: (err: Error, tlsSocket: TLSSocket) => void): this;

on(event: 'newSession', listener: (sessionId: Buffer, sessionData: Buffer, callback: (err: Error, resp: Buffer) => void) => void): this;

on(event: 'OCSPRequest', listener: (certificate: Buffer, issuer: Buffer, callback: (err: Error | null, resp: Buffer) => void) => void): this;

on(event: 'resumeSession', listener: (sessionId: Buffer, callback: (err: Error, sessionData: Buffer) => void) => void): this;

on(event: 'secureConnection', listener: (tlsSocket: TLSSocket) => void): this;

on(event: 'keylog', listener: (line: Buffer, tlsSocket: TLSSocket) => void): this;

once(event: string, listener: (...args: any[]) => void): this;

once(event: 'tlsClientError', listener: (err: Error, tlsSocket: TLSSocket) => void): this;

once(event: 'newSession', listener: (sessionId: Buffer, sessionData: Buffer, callback: (err: Error, resp: Buffer) => void) => void): this;

once(event: 'OCSPRequest', listener: (certificate: Buffer, issuer: Buffer, callback: (err: Error | null, resp: Buffer) => void) => void): this;

once(event: 'resumeSession', listener: (sessionId: Buffer, callback: (err: Error, sessionData: Buffer) => void) => void): this;

once(event: 'secureConnection', listener: (tlsSocket: TLSSocket) => void): this;

once(event: 'keylog', listener: (line: Buffer, tlsSocket: TLSSocket) => void): this;

prependListener(event: string, listener: (...args: any[]) => void): this;

prependListener(event: 'tlsClientError', listener: (err: Error, tlsSocket: TLSSocket) => void): this;

prependListener(event: 'newSession', listener: (sessionId: Buffer, sessionData: Buffer, callback: (err: Error, resp: Buffer) => void) => void): this;

prependListener(event: 'OCSPRequest', listener: (certificate: Buffer, issuer: Buffer, callback: (err: Error | null, resp: Buffer) => void) => void): this;

prependListener(event: 'resumeSession', listener: (sessionId: Buffer, callback: (err: Error, sessionData: Buffer) => void) => void): this;

prependListener(event: 'secureConnection', listener: (tlsSocket: TLSSocket) => void): this;

prependListener(event: 'keylog', listener: (line: Buffer, tlsSocket: TLSSocket) => void): this;

prependOnceListener(event: string, listener: (...args: any[]) => void): this;

prependOnceListener(event: 'tlsClientError', listener: (err: Error, tlsSocket: TLSSocket) => void): this;

prependOnceListener(event: 'newSession', listener: (sessionId: Buffer, sessionData: Buffer, callback: (err: Error, resp: Buffer) => void) => void): this;

prependOnceListener(event: 'OCSPRequest', listener: (certificate: Buffer, issuer: Buffer, callback: (err: Error | null, resp: Buffer) => void) => void): this;

prependOnceListener(event: 'resumeSession', listener: (sessionId: Buffer, callback: (err: Error, sessionData: Buffer) => void) => void): this;

prependOnceListener(event: 'secureConnection', listener: (tlsSocket: TLSSocket) => void): this;

prependOnceListener(event: 'keylog', listener: (line: Buffer, tlsSocket: TLSSocket) => void): this;

}

/\*\*

\* @deprecated since v0.11.3 Use `tls.TLSSocket` instead.

\*/

interface SecurePair {

encrypted: TLSSocket;

cleartext: TLSSocket;

}

type SecureVersion = 'TLSv1.3' | 'TLSv1.2' | 'TLSv1.1' | 'TLSv1';

interface SecureContextOptions {

/\*\*

\* 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.

\*/

ca?: string | Buffer | Array<string | Buffer> | undefined;

/\*\*

\* 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.

\*/

cert?: string | Buffer | Array<string | Buffer> | undefined;

/\*\*

\* 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).

\*/

sigalgs?: string | undefined;

/\*\*

\* Cipher suite specification, replacing the default. For more

\* information, see modifying the default cipher suite. Permitted

\* ciphers can be obtained via tls.getCiphers(). Cipher names must be

\* uppercased in order for OpenSSL to accept them.

\*/

ciphers?: string | undefined;

/\*\*

\* Name of an OpenSSL engine which can provide the client certificate.

\*/

clientCertEngine?: string | undefined;

/\*\*

\* PEM formatted CRLs (Certificate Revocation Lists).

\*/

crl?: string | Buffer | Array<string | Buffer> | undefined;

/\*\*

\* 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.

\*/

dhparam?: string | Buffer | undefined;

/\*\*

\* 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.

\*/

ecdhCurve?: string | undefined;

/\*\*

\* 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

\*/

honorCipherOrder?: boolean | undefined;

/\*\*

\* 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.

\*/

key?: string | Buffer | Array<Buffer | KeyObject> | undefined;

/\*\*

\* Name of an OpenSSL engine to get private key from. Should be used

\* together with privateKeyIdentifier.

\*/

privateKeyEngine?: string | undefined;

/\*\*

\* 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.

\*/

privateKeyIdentifier?: string | undefined;

/\*\*

\* 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:\*\* `'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.

\*/

maxVersion?: SecureVersion | undefined;

/\*\*

\* 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. It is not recommended to use

\* less than TLSv1.2, but it may be required for interoperability.

\* \*\*Default:\*\* `'TLSv1.2'`, unless changed using CLI options. Using

\* `--tls-v1.0` sets the default to `'TLSv1'`. Using `--tls-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.

\*/

minVersion?: SecureVersion | undefined;

/\*\*

\* Shared passphrase used for a single private key and/or a PFX.

\*/

passphrase?: string | undefined;

/\*\*

\* 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.

\*/

pfx?: string | Buffer | Array<string | Buffer | PxfObject> | undefined;

/\*\*

\* 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

\*/

secureOptions?: number | undefined; // Value is a numeric bitmask of the `SSL\_OP\_\*` options

/\*\*

\* 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.

\*/

secureProtocol?: string | undefined;

/\*\*

\* Opaque identifier used by servers to ensure session state is not

\* shared between applications. Unused by clients.

\*/

sessionIdContext?: string | undefined;

/\*\*

\* 48-bytes of cryptographically strong pseudo-random data.

\* See Session Resumption for more information.

\*/

ticketKeys?: Buffer | undefined;

/\*\*

\* 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.

\*/

sessionTimeout?: number | undefined;

}

interface SecureContext {

context: any;

}

/\*\*

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

\*

\* Returns [Error](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/Error) object, populating it with `reason`, `host`, and `cert` on

\* failure. On success, returns [undefined](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data\_structures#Undefined\_type).

\*

\* This function can be overwritten by providing alternative function as part of

\* the `options.checkServerIdentity` option 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`).

\* @since v0.8.4

\* @param hostname The host name or IP address to verify the certificate against.

\* @param cert A `certificate object` representing the peer's certificate.

\*/

function checkServerIdentity(hostname: string, cert: PeerCertificate): Error | undefined;

/\*\*

\* Creates a new {@link 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:

\*

\* ```js

\* 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 {@link connect}.

\* @since v0.3.2

\*/

function createServer(secureConnectionListener?: (socket: TLSSocket) => void): Server;

function createServer(options: TlsOptions, secureConnectionListener?: (socket: TLSSocket) => void): Server;

/\*\*

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

\*

\* `tls.connect()` returns a {@link 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 {@link createServer}:

\*

\* ```js

\* // 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');

\* });

\* ```

\* @since v0.11.3

\*/

function connect(options: ConnectionOptions, secureConnectListener?: () => void): TLSSocket;

function connect(port: number, host?: string, options?: ConnectionOptions, secureConnectListener?: () => void): TLSSocket;

function connect(port: number, options?: ConnectionOptions, secureConnectListener?: () => void): TLSSocket;

/\*\*

\* 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 {@link TLSSocket}.

\*

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

\*

\* ```js

\* pair = tls.createSecurePair(// ... );

\* pair.encrypted.pipe(socket);

\* socket.pipe(pair.encrypted);

\* ```

\*

\* can be replaced by:

\*

\* ```js

\* secureSocket = tls.TLSSocket(socket, options);

\* ```

\*

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

\* @since v0.3.2

\* @deprecated Since v0.11.3 - Use {@link TLSSocket} instead.

\* @param context A secure context object as returned by `tls.createSecureContext()`

\* @param isServer `true` to specify that this TLS connection should be opened as a server.

\* @param requestCert `true` to specify whether a server should request a certificate from a connecting client. Only applies when `isServer` is `true`.

\* @param rejectUnauthorized If not `false` a server automatically reject clients with invalid certificates. Only applies when `isServer` is `true`.

\*/

function createSecurePair(context?: SecureContext, isServer?: boolean, requestCert?: boolean, rejectUnauthorized?: boolean): SecurePair;

/\*\*

\* {@link createServer} sets the default value of the `honorCipherOrder` option

\* to `true`, other APIs that create secure contexts leave it unset.

\*

\* {@link 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 {@link 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](https://hg.mozilla.org/mozilla-central/raw-file/tip/security/nss/lib/ckfw/builtins/certdata.txt).

\* @since v0.11.13

\*/

function createSecureContext(options?: SecureContextOptions): SecureContext;

/\*\*

\* 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 {@link createSecureContext}.

\*

\* Cipher names that start with `'tls\_'` are for TLSv1.3, all the others are for

\* TLSv1.2 and below.

\*

\* ```js

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

\* ```

\* @since v0.10.2

\*/

function getCiphers(): string[];

/\*\*

\* 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.

\*/

let DEFAULT\_ECDH\_CURVE: 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.

\*/

let DEFAULT\_MAX\_VERSION: SecureVersion;

/\*\*

\* 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.

\*/

let DEFAULT\_MIN\_VERSION: SecureVersion;

/\*\*

\* An immutable array of strings representing the root certificates (in PEM

\* format) used for verifying peer certificates. This is the default value

\* of the ca option to tls.createSecureContext().

\*/

const rootCertificates: ReadonlyArray<string>;

}

declare module 'node:tls' {

export \* from 'tls';

}