HTTPS

> *Stability: 2 - Stable*

HTTPS is the HTTP protocol over TLS/SSL. In Node.js this is implemented as a separate module.

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 `https` or calling `require('https')` will result in an error being thrown.

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

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

Class: `https.Agent`

An `Agent` object for HTTPS similar to `http.Agent`. See `https.request()` for more information.

`new Agent([options])`

- `options` {Object} Set of configurable options to set on the agent. Can have the same fields as for `http.Agent(options)`, and
  - `maxCachedSessions` {number} maximum number of TLS cached sessions. Use `0` to disable TLS session caching. **Default:** `100`.
  - `servername` {string} the value of Server Name Indication extension to be sent to the server. Use empty string `''` to disable sending the extension. **Default:** host name of the target server, unless the target server is specified using an IP address, in which case the default is `''` (no extension).

    See `Session Resumption` for information about TLS session reuse.

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 managed by this agent (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:

```
// ...
https.globalAgent.on('keylog', (line, tlsSocket) => {
  fs.appendFileSync('/tmp/ssl-keys.log', line, { mode: 0o600 });
});
```

Class: `https.Server`

- Extends: {tls.Server}

See `http.Server` for more information.

`server.close([callback])`

- `callback` {Function}
- Returns: {https.Server}

See `server.close()` from the HTTP module for details.

`server.headersTimeout`

- {number} **Default:** `60000`

See `http.Server#headersTimeout`.

`server.listen()`

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

`server.maxHeadersCount`

- {number} **Default:** `2000`

See `http.Server#maxHeadersCount`.

`server.requestTimeout`

- {number} **Default:** `0`

See `http.Server#requestTimeout`.

`server.setTimeout([msecs][, callback])`

- `msecs` {number} **Default:** `120000` (2 minutes)
- `callback` {Function}
- Returns: {https.Server}

See `http.Server#setTimeout()`.

`server.timeout`

- {number} **Default:** 0 (no timeout)

See `http.Server#timeout`.

`server.keepAliveTimeout`

- {number} **Default:** `5000` (5 seconds)

See `http.Server#keepAliveTimeout`.

`https.createServer([options][, requestListener])`

- `options` {Object} Accepts `options` from `tls.createServer()`, `tls.createSecureContext()` and `http.createServer()`.
- `requestListener` {Function} A listener to be added to the `'request'` event.
- Returns: {https.Server}

```
// curl -k https://localhost:8000/
const https = require('https');
const fs = require('fs');

const options = {
  key: fs.readFileSync('test/fixtures/keys/agent2-key.pem'),
  cert: fs.readFileSync('test/fixtures/keys/agent2-cert.pem')
};

https.createServer(options, (req, res) => {
  res.writeHead(200);
  res.end('hello world\n');
}).listen(8000);
```

Or

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

const options = {
  pfx: fs.readFileSync('test/fixtures/test_cert.pfx'),
  passphrase: 'sample'
};

https.createServer(options, (req, res) => {
  res.writeHead(200);
```

```
  res.end('hello world\n');
}).listen(8000);
```

`https.get(options[, callback])`

`https.get(url[, options][, callback])`

- `url` {string | URL}
- `options` {Object | string | URL} Accepts the same `options` as `https.request()`, with the `method` always set to `GET`.
- `callback` {Function}

Like `http.get()` but for HTTPS.

`options` can be an object, a string, or a `URL` object. If `options` is a string, it is automatically parsed with `new URL()`. If it is a `URL` object, it will be automatically converted to an ordinary `options` object.

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

https.get('https://encrypted.google.com/', (res) => {
  console.log('statusCode:', res.statusCode);
  console.log('headers:', res.headers);

  res.on('data', (d) => {
    process.stdout.write(d);
  });

}).on('error', (e) => {
  console.error(e);
});
```

`https.globalAgent`

Global instance of `https.Agent` for all HTTPS client requests.

`https.request(options[, callback])`

`https.request(url[, options][, callback])`

- `url` {string | URL}
- `options` {Object | string | URL} Accepts all `options` from `http.request()`, with some differences in default values:
  - `protocol` **Default:** `'https:'`
  - `port` **Default:** `443`
  - `agent` **Default:** `https.globalAgent`
- `callback` {Function}
- Returns: {http.ClientRequest}

Makes a request to a secure web server.

The following additional `options` from `tls.connect()` are also accepted: `ca`, `cert`, `ciphers`, `clientCertEngine`, `crl`, `dhparam`, `ecdhCurve`, `honorCipherOrder`, `key`, `passphrase`, `pfx`, `rejectUnauthorized`, `secureOptions`, `secureProtocol`, `servername`, `sessionIdContext`, `highWaterMark`.

`options` can be an object, a string, or a `URL` object. If `options` is a string, it is automatically parsed with `new URL()`. If it is a `URL` object, it will be automatically converted to an ordinary `options` object.

`https.request()` returns an instance of the `http.ClientRequest` class. The `ClientRequest` instance is a writable stream. If one needs to upload a file with a POST request, then write to the `ClientRequest` object.

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

const options = {
  hostname: 'encrypted.google.com',
  port: 443,
  path: '/',
  method: 'GET'
};

const req = https.request(options, (res) => {
  console.log('statusCode:', res.statusCode);
  console.log('headers:', res.headers);

  res.on('data', (d) => {
    process.stdout.write(d);
  });
});

req.on('error', (e) => {
  console.error(e);
});
req.end();
```

Example using options from `tls.connect()`:

```
const options = {
  hostname: 'encrypted.google.com',
  port: 443,
  path: '/',
  method: 'GET',
  key: fs.readFileSync('test/fixtures/keys/agent2-key.pem'),
  cert: fs.readFileSync('test/fixtures/keys/agent2-cert.pem')
};
options.agent = new https.Agent(options);

const req = https.request(options, (res) => {
  // ...
});
```

Alternatively, opt out of connection pooling by not using an `Agent`.

```
const options = {
  hostname: 'encrypted.google.com',
  port: 443,
  path: '/',
  method: 'GET',
  key: fs.readFileSync('test/fixtures/keys/agent2-key.pem'),
  cert: fs.readFileSync('test/fixtures/keys/agent2-cert.pem'),
  agent: false
};

const req = https.request(options, (res) => {
  // ...
});
```

Example using a `URL` as `options`:

```
const options = new URL('https://abc:xyz@example.com');

const req = https.request(options, (res) => {
  // ...
});
```

Example pinning on certificate fingerprint, or the public key (similar to `pin-sha256`):

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

function sha256(s) {
  return crypto.createHash('sha256').update(s).digest('base64');
}
const options = {
  hostname: 'github.com',
  port: 443,
  path: '/',
  method: 'GET',
  checkServerIdentity: function(host, cert) {
    // Make sure the certificate is issued to the host we are connected to
    const err = tls.checkServerIdentity(host, cert);
    if (err) {
      return err;
    }

    // Pin the public key, similar to HPKP pin-sha25 pinning
    const pubkey256 = 'pL1+qb9HTMRZJmuC/bB/ZI9d302BYrrqiVuRyW+DGrU=';
    if (sha256(cert.pubkey) !== pubkey256) {
      const msg = 'Certificate verification error: ' +
```

```
        `The public key of '${cert.subject.CN}' ` +
        'does not match our pinned fingerprint';
      return new Error(msg);
    }

    // Pin the exact certificate, rather than the pub key
    const cert256 = '25:FE:39:32:D9:63:8C:8A:FC:A1:9A:29:87:' +
      'D8:3E:4C:1D:98:DB:71:E4:1A:48:03:98:EA:22:6A:BD:8B:93:16';
    if (cert.fingerprint256 !== cert256) {
      const msg = 'Certificate verification error: ' +
        `The certificate of '${cert.subject.CN}' ` +
        'does not match our pinned fingerprint';
      return new Error(msg);
    }

    // This loop is informational only.
    // Print the certificate and public key fingerprints of all certs in the
    // chain. Its common to pin the public key of the issuer on the public
    // internet, while pinning the public key of the service in sensitive
    // environments.
    do {
      console.log('Subject Common Name:', cert.subject.CN);
      console.log('  Certificate SHA256 fingerprint:', cert.fingerprint256);

      hash = crypto.createHash('sha256');
      console.log('  Public key ping-sha256:', sha256(cert.pubkey));

      lastprint256 = cert.fingerprint256;
      cert = cert.issuerCertificate;
    } while (cert.fingerprint256 !== lastprint256);

  },
};

options.agent = new https.Agent(options);
const req = https.request(options, (res) => {
  console.log('All OK. Server matched our pinned cert or public key');
  console.log('statusCode:', res.statusCode);
  // Print the HPKP values
  console.log('headers:', res.headers['public-key-pins']);

  res.on('data', (d) => {});
});

req.on('error', (e) => {
  console.error(e.message);
});
req.end();
```

Outputs for example:

```
Subject Common Name: github.com
  Certificate SHA256 fingerprint:
25:FE:39:32:D9:63:8C:8A:FC:A1:9A:29:87:D8:3E:4C:1D:98:DB:71:E4:1A:48:03:98:EA:22:6A:BD

  Public key ping-sha256: pL1+qb9HTMRZJmuC/bB/ZI9d302BYrrqiVuRyW+DGrU=
Subject Common Name: DigiCert SHA2 Extended Validation Server CA
  Certificate SHA256 fingerprint:
40:3E:06:2A:26:53:05:91:13:28:5B:AF:80:A0:D4:AE:42:2C:84:8C:9F:78:FA:D0:1F:C9:4B:C5:B8

  Public key ping-sha256: RRM1dGqnDFsCJXBTHky16vi1obOlCgFFn/yOhI/y+ho=
Subject Common Name: DigiCert High Assurance EV Root CA
  Certificate SHA256 fingerprint:
74:31:E5:F4:C3:C1:CE:46:90:77:4F:0B:61:E0:54:40:88:3B:A9:A0:1E:D0:0B:A6:AB:D7:80:6E:D3

  Public key ping-sha256: WoiWRyIOVNa9ihaBciRSC7XHjliYS9VwUGOIud4PB18=
All OK. Server matched our pinned cert or public key
statusCode: 200
headers: max-age=0; pin-sha256="WoiWRyIOVNa9ihaBciRSC7XHjliYS9VwUGOIud4PB18="; pin-
sha256="RRM1dGqnDFsCJXBTHky16vi1obOlCgFFn/yOhI/y+ho="; pin-
sha256="k2v657xBsOVe1PQRwOsHsw3bsGT2VzIqz5K+59sNQws="; pin-
sha256="K87oWBWM9UZfyddvDfoxL+8lpNyoUB2ptGtn0fv6G2Q="; pin-
sha256="IQBnNBEiFuhj+8x6X8XLgh01V9Ic5/V3IRQLNFFc7v4="; pin-
sha256="iie1VXtL7HzAMF+/PVPR9xzT80kQxdZeJ+zduCB3uj0="; pin-
sha256="LvRiGEjRqfzurezaWuj8Wie2gyHMrW5Q06LspMnox7A="; includeSubDomains
```