

SETTINGS_TIMEOUT (0x4): The endpoint sent a SETTINGS frame but did not receive a response in a timely manner. See Section 6.5.3 ("Settings Synchronization").

STREAM_CLOSED (0x5): The endpoint received a frame after a stream was half-closed.

FRAME_SIZE_ERROR (0x6): The endpoint received a frame with an invalid size.

REFUSED_STREAM (0x7): The endpoint refused the stream prior to performing any application processing (see Section 8.1.4 for details).

CANCEL (0x8): Used by the endpoint to indicate that the stream is no longer needed.

COMPRESSION_ERROR (0x9): The endpoint is unable to maintain the header compression context for the connection.

CONNECT_ERROR (0xa): The connection established in response to a CONNECT request (Section 8.3) was reset or abnormally closed.

ENHANCE_YOUR_CALM (0xb): The endpoint detected that its peer is exhibiting a behavior that might be generating excessive load.

INADEQUATE_SECURITY (0xc): The underlying transport has properties that do not meet minimum security requirements (see Section 9.2).

HTTP_1_1_REQUIRED (0xd): The endpoint requires that HTTP/1.1 be used instead of HTTP/2.

Unknown or unsupported error codes MUST NOT trigger any special behavior. These MAY be treated by an implementation as being equivalent to INTERNAL_ERROR.

8. HTTP Message Exchanges

HTTP/2 is intended to be as compatible as possible with current uses of HTTP. This means that, from the application perspective, the features of the protocol are largely unchanged. To achieve this, all request and response semantics are preserved, although the syntax of conveying those semantics has changed.

Thus, the specification and requirements of HTTP/1.1 Semantics and Content [RFC7231], Conditional Requests [RFC7232], Range Requests [RFC7233], Caching [RFC7234], and Authentication [RFC7235] are applicable to HTTP/2. Selected portions of HTTP/1.1 Message Syntax

and Routing [RFC7230], such as the HTTP and HTTPS URI schemes, are also applicable in HTTP/2, but the expression of those semantics for this protocol are defined in the sections below.

8.1. HTTP Request/Response Exchange

A client sends an HTTP request on a new stream, using a previously unused stream identifier (Section 5.1.1). A server sends an HTTP response on the same stream as the request.

An HTTP message (request or response) consists of:

1. for a response only, zero or more HEADERS frames (each followed by zero or more CONTINUATION frames) containing the message headers of informational (1xx) HTTP responses (see [RFC7230], Section 3.2 and [RFC7231], Section 6.2),
2. one HEADERS frame (followed by zero or more CONTINUATION frames) containing the message headers (see [RFC7230], Section 3.2),
3. zero or more DATA frames containing the payload body (see [RFC7230], Section 3.3), and
4. optionally, one HEADERS frame, followed by zero or more CONTINUATION frames containing the trailer-part, if present (see [RFC7230], Section 4.1.2).

The last frame in the sequence bears an END_STREAM flag, noting that a HEADERS frame bearing the END_STREAM flag can be followed by CONTINUATION frames that carry any remaining portions of the header block.

Other frames (from any stream) MUST NOT occur between the HEADERS frame and any CONTINUATION frames that might follow.

HTTP/2 uses DATA frames to carry message payloads. The "chunked" transfer encoding defined in Section 4.1 of [RFC7230] MUST NOT be used in HTTP/2.

Trailing header fields are carried in a header block that also terminates the stream. Such a header block is a sequence starting with a HEADERS frame, followed by zero or more CONTINUATION frames, where the HEADERS frame bears an END_STREAM flag. Header blocks after the first that do not terminate the stream are not part of an HTTP request or response.

A HEADERS frame (and associated CONTINUATION frames) can only appear at the start or end of a stream. An endpoint that receives a HEADERS frame without the END_STREAM flag set after receiving a final (non-informational) status code MUST treat the corresponding request or response as malformed (Section 8.1.2.6).

An HTTP request/response exchange fully consumes a single stream. A request starts with the HEADERS frame that puts the stream into an "open" state. The request ends with a frame bearing END_STREAM, which causes the stream to become "half-closed (local)" for the client and "half-closed (remote)" for the server. A response starts with a HEADERS frame and ends with a frame bearing END_STREAM, which places the stream in the "closed" state.

An HTTP response is complete after the server sends -- or the client receives -- a frame with the END_STREAM flag set (including any CONTINUATION frames needed to complete a header block). A server can send a complete response prior to the client sending an entire request if the response does not depend on any portion of the request that has not been sent and received. When this is true, a server MAY request that the client abort transmission of a request without error by sending a RST_STREAM with an error code of NO_ERROR after sending a complete response (i.e., a frame with the END_STREAM flag). Clients MUST NOT discard responses as a result of receiving such a RST_STREAM, though clients can always discard responses at their discretion for other reasons.

8.1.1. Upgrading from HTTP/2

HTTP/2 removes support for the 101 (Switching Protocols) informational status code ([RFC7231], Section 6.2.2).

The semantics of 101 (Switching Protocols) aren't applicable to a multiplexed protocol. Alternative protocols are able to use the same mechanisms that HTTP/2 uses to negotiate their use (see Section 3).

8.1.2. HTTP Header Fields

HTTP header fields carry information as a series of key-value pairs. For a listing of registered HTTP headers, see the "Message Header Field" registry maintained at <<https://www.iana.org/assignments/message-headers>>.

Just as in HTTP/1.x, header field names are strings of ASCII characters that are compared in a case-insensitive fashion. However, header field names MUST be converted to lowercase prior to their encoding in HTTP/2. A request or response containing uppercase header field names MUST be treated as malformed (Section 8.1.2.6).

8.1.2.1. Pseudo-Header Fields

While HTTP/1.x used the message start-line (see [RFC7230], Section 3.1) to convey the target URI, the method of the request, and the status code for the response, HTTP/2 uses special pseudo-header fields beginning with ':' character (ASCII 0x3a) for this purpose.

Pseudo-header fields are not HTTP header fields. Endpoints MUST NOT generate pseudo-header fields other than those defined in this document.

Pseudo-header fields are only valid in the context in which they are defined. Pseudo-header fields defined for requests MUST NOT appear in responses; pseudo-header fields defined for responses MUST NOT appear in requests. Pseudo-header fields MUST NOT appear in trailers. Endpoints MUST treat a request or response that contains undefined or invalid pseudo-header fields as malformed (Section 8.1.2.6).

All pseudo-header fields MUST appear in the header block before regular header fields. Any request or response that contains a pseudo-header field that appears in a header block after a regular header field MUST be treated as malformed (Section 8.1.2.6).

8.1.2.2. Connection-Specific Header Fields

HTTP/2 does not use the Connection header field to indicate connection-specific header fields; in this protocol, connection-specific metadata is conveyed by other means. An endpoint MUST NOT generate an HTTP/2 message containing connection-specific header fields; any message containing connection-specific header fields MUST be treated as malformed (Section 8.1.2.6).

The only exception to this is the TE header field, which MAY be present in an HTTP/2 request; when it is, it MUST NOT contain any value other than "trailers".

This means that an intermediary transforming an HTTP/1.x message to HTTP/2 will need to remove any header fields nominated by the Connection header field, along with the Connection header field itself. Such intermediaries SHOULD also remove other connection-specific header fields, such as Keep-Alive, Proxy-Connection, Transfer-Encoding, and Upgrade, even if they are not nominated by the Connection header field.

Note: HTTP/2 purposefully does not support upgrade to another protocol. The handshake methods described in Section 3 are believed sufficient to negotiate the use of alternative protocols.

8.1.2.3. Request Pseudo-Header Fields

The following pseudo-header fields are defined for HTTP/2 requests:

- o The `:method` pseudo-header field includes the HTTP method ([RFC7231], Section 4).
- o The `:scheme` pseudo-header field includes the scheme portion of the target URI ([RFC3986], Section 3.1).

`:scheme` is not restricted to `http` and `https` schemed URIs. A proxy or gateway can translate requests for non-HTTP schemes, enabling the use of HTTP to interact with non-HTTP services.

- o The `:authority` pseudo-header field includes the authority portion of the target URI ([RFC3986], Section 3.2). The authority **MUST NOT** include the deprecated `userinfo` subcomponent for `http` or `https` schemed URIs.

To ensure that the HTTP/1.1 request line can be reproduced accurately, this pseudo-header field **MUST** be omitted when translating from an HTTP/1.1 request that has a request target in origin or asterisk form (see [RFC7230], Section 5.3). Clients that generate HTTP/2 requests directly **SHOULD** use the `:authority` pseudo-header field instead of the Host header field. An intermediary that converts an HTTP/2 request to HTTP/1.1 **MUST** create a Host header field if one is not present in a request by copying the value of the `:authority` pseudo-header field.

- o The `:path` pseudo-header field includes the path and query parts of the target URI (the `path-absolute` production and optionally a `'?'` character followed by the `query` production (see Sections 3.3 and 3.4 of [RFC3986])). A request in asterisk form includes the value `'*'` for the `:path` pseudo-header field.

This pseudo-header field **MUST NOT** be empty for `http` or `https` URIs; `http` or `https` URIs that do not contain a path component **MUST** include a value of `'/'`. The exception to this rule is an **OPTIONS** request for an `http` or `https` URI that does not include a path component; these **MUST** include a `:path` pseudo-header field with a value of `'*'` (see [RFC7230], Section 5.3.4).

All HTTP/2 requests MUST include exactly one valid value for the ":method", ":scheme", and ":path" pseudo-header fields, unless it is a CONNECT request (Section 8.3). An HTTP request that omits mandatory pseudo-header fields is malformed (Section 8.1.2.6).

HTTP/2 does not define a way to carry the version identifier that is included in the HTTP/1.1 request line.

8.1.2.4. Response Pseudo-Header Fields

For HTTP/2 responses, a single ":status" pseudo-header field is defined that carries the HTTP status code field (see [RFC7231], Section 6). This pseudo-header field MUST be included in all responses; otherwise, the response is malformed (Section 8.1.2.6).

HTTP/2 does not define a way to carry the version or reason phrase that is included in an HTTP/1.1 status line.

8.1.2.5. Compressing the Cookie Header Field

The Cookie header field [COOKIE] uses a semi-colon (";") to delimit cookie-pairs (or "crumbs"). This header field doesn't follow the list construction rules in HTTP (see [RFC7230], Section 3.2.2), which prevents cookie-pairs from being separated into different name-value pairs. This can significantly reduce compression efficiency as individual cookie-pairs are updated.

To allow for better compression efficiency, the Cookie header field MAY be split into separate header fields, each with one or more cookie-pairs. If there are multiple Cookie header fields after decompression, these MUST be concatenated into a single octet string using the two-octet delimiter of 0x3B, 0x20 (the ASCII string "; ") before being passed into a non-HTTP/2 context, such as an HTTP/1.1 connection, or a generic HTTP server application.

Therefore, the following two lists of Cookie header fields are semantically equivalent.

```
cookie: a=b; c=d; e=f
```

```
cookie: a=b  
cookie: c=d  
cookie: e=f
```

8.1.2.6. Malformed Requests and Responses

A malformed request or response is one that is an otherwise valid sequence of HTTP/2 frames but is invalid due to the presence of extraneous frames, prohibited header fields, the absence of mandatory header fields, or the inclusion of uppercase header field names.

A request or response that includes a payload body can include a content-length header field. A request or response is also malformed if the value of a content-length header field does not equal the sum of the DATA frame payload lengths that form the body. A response that is defined to have no payload, as described in [RFC7230], Section 3.3.2, can have a non-zero content-length header field, even though no content is included in DATA frames.

Intermediaries that process HTTP requests or responses (i.e., any intermediary not acting as a tunnel) MUST NOT forward a malformed request or response. Malformed requests or responses that are detected MUST be treated as a stream error (Section 5.4.2) of type `PROTOCOL_ERROR`.

For malformed requests, a server MAY send an HTTP response prior to closing or resetting the stream. Clients MUST NOT accept a malformed response. Note that these requirements are intended to protect against several types of common attacks against HTTP; they are deliberately strict because being permissive can expose implementations to these vulnerabilities.

8.1.3. Examples

This section shows HTTP/1.1 requests and responses, with illustrations of equivalent HTTP/2 requests and responses.

An HTTP GET request includes request header fields and no payload body and is therefore transmitted as a single HEADERS frame, followed by zero or more CONTINUATION frames containing the serialized block of request header fields. The HEADERS frame in the following has both the `END_HEADERS` and `END_STREAM` flags set; no CONTINUATION frames are sent.

GET /resource HTTP/1.1		HEADERS
Host: example.org	==>	+ END_STREAM
Accept: image/jpeg		+ END_HEADERS
		:method = GET
		:scheme = https
		:path = /resource
		host = example.org
		accept = image/jpeg

Similarly, a response that includes only response header fields is transmitted as a HEADERS frame (again, followed by zero or more CONTINUATION frames) containing the serialized block of response header fields.

```

HTTP/1.1 304 Not Modified      HEADERS
ETag: "xyzzy"                  ==>  + END_STREAM
Expires: Thu, 23 Jan ...      + END_HEADERS
                               :status = 304
                               etag = "xyzzy"
                               expires = Thu, 23 Jan ...

```

An HTTP POST request that includes request header fields and payload data is transmitted as one HEADERS frame, followed by zero or more CONTINUATION frames containing the request header fields, followed by one or more DATA frames, with the last CONTINUATION (or HEADERS) frame having the END_HEADERS flag set and the final DATA frame having the END_STREAM flag set:

```

POST /resource HTTP/1.1      HEADERS
Host: example.org            ==>  - END_STREAM
Content-Type: image/jpeg     - END_HEADERS
Content-Length: 123          :method = POST
                               :path = /resource
                               :scheme = https
{binary data}

CONTINUATION
+ END_HEADERS
content-type = image/jpeg
host = example.org
content-length = 123

DATA
+ END_STREAM
{binary data}

```

Note that data contributing to any given header field could be spread between header block fragments. The allocation of header fields to frames in this example is illustrative only.

A response that includes header fields and payload data is transmitted as a HEADERS frame, followed by zero or more CONTINUATION frames, followed by one or more DATA frames, with the last DATA frame in the sequence having the END_STREAM flag set:

```

HTTP/1.1 200 OK           HEADERS
Content-Type: image/jpeg  ==>  - END_STREAM
Content-Length: 123        + END_HEADERS
                           :status = 200
                           content-type = image/jpeg
                           content-length = 123

                           DATA
                           + END_STREAM
                           {binary data}

```

An informational response using a 1xx status code other than 101 is transmitted as a HEADERS frame, followed by zero or more CONTINUATION frames.

Trailing header fields are sent as a header block after both the request or response header block and all the DATA frames have been sent. The HEADERS frame starting the trailers header block has the END_STREAM flag set.

The following example includes both a 100 (Continue) status code, which is sent in response to a request containing a "100-continue" token in the Expect header field, and trailing header fields:

```

HTTP/1.1 100 Continue     HEADERS
Extension-Field: bar      ==>  - END_STREAM
                           + END_HEADERS
                           :status = 100
                           extension-field = bar

HTTP/1.1 200 OK           HEADERS
Content-Type: image/jpeg  ==>  - END_STREAM
Transfer-Encoding: chunked + END_HEADERS
Trailer: Foo              :status = 200
                           content-length = 123
                           content-type = image/jpeg
                           trailer = Foo

123
{binary data}
0
Foo: bar

                           DATA
                           - END_STREAM
                           {binary data}

                           HEADERS
                           + END_STREAM
                           + END_HEADERS
                           foo = bar

```

8.1.4. Request Reliability Mechanisms in HTTP/2

In HTTP/1.1, an HTTP client is unable to retry a non-idempotent request when an error occurs because there is no means to determine the nature of the error. It is possible that some server processing occurred prior to the error, which could result in undesirable effects if the request were reattempted.

HTTP/2 provides two mechanisms for providing a guarantee to a client that a request has not been processed:

- o The GOAWAY frame indicates the highest stream number that might have been processed. Requests on streams with higher numbers are therefore guaranteed to be safe to retry.
- o The REFUSED_STREAM error code can be included in a RST_STREAM frame to indicate that the stream is being closed prior to any processing having occurred. Any request that was sent on the reset stream can be safely retried.

Requests that have not been processed have not failed; clients MAY automatically retry them, even those with non-idempotent methods.

A server MUST NOT indicate that a stream has not been processed unless it can guarantee that fact. If frames that are on a stream are passed to the application layer for any stream, then REFUSED_STREAM MUST NOT be used for that stream, and a GOAWAY frame MUST include a stream identifier that is greater than or equal to the given stream identifier.

In addition to these mechanisms, the PING frame provides a way for a client to easily test a connection. Connections that remain idle can become broken as some middleboxes (for instance, network address translators or load balancers) silently discard connection bindings. The PING frame allows a client to safely test whether a connection is still active without sending a request.

8.2. Server Push

HTTP/2 allows a server to pre-emptively send (or "push") responses (along with corresponding "promised" requests) to a client in association with a previous client-initiated request. This can be useful when the server knows the client will need to have those responses available in order to fully process the response to the original request.

A client can request that server push be disabled, though this is negotiated for each hop independently. The `SETTINGS_ENABLE_PUSH` setting can be set to 0 to indicate that server push is disabled.

Promised requests **MUST** be cacheable (see [RFC7231], Section 4.2.3), **MUST** be safe (see [RFC7231], Section 4.2.1), and **MUST NOT** include a request body. Clients that receive a promised request that is not cacheable, that is not known to be safe, or that indicates the presence of a request body **MUST** reset the promised stream with a stream error (Section 5.4.2) of type `PROTOCOL_ERROR`. Note this could result in the promised stream being reset if the client does not recognize a newly defined method as being safe.

Pushed responses that are cacheable (see [RFC7234], Section 3) can be stored by the client, if it implements an HTTP cache. Pushed responses are considered successfully validated on the origin server (e.g., if the "no-cache" cache response directive is present ([RFC7234], Section 5.2.2)) while the stream identified by the promised stream ID is still open.

Pushed responses that are not cacheable **MUST NOT** be stored by any HTTP cache. They **MAY** be made available to the application separately.

The server **MUST** include a value in the `:authority` pseudo-header field for which the server is authoritative (see Section 10.1). A client **MUST** treat a `PUSH_PROMISE` for which the server is not authoritative as a stream error (Section 5.4.2) of type `PROTOCOL_ERROR`.

An intermediary can receive pushes from the server and choose not to forward them on to the client. In other words, how to make use of the pushed information is up to that intermediary. Equally, the intermediary might choose to make additional pushes to the client, without any action taken by the server.

A client cannot push. Thus, servers **MUST** treat the receipt of a `PUSH_PROMISE` frame as a connection error (Section 5.4.1) of type `PROTOCOL_ERROR`. Clients **MUST** reject any attempt to change the `SETTINGS_ENABLE_PUSH` setting to a value other than 0 by treating the message as a connection error (Section 5.4.1) of type `PROTOCOL_ERROR`.

8.2.1. Push Requests

Server push is semantically equivalent to a server responding to a request; however, in this case, that request is also sent by the server, as a `PUSH_PROMISE` frame.

The PUSH_PROMISE frame includes a header block that contains a complete set of request header fields that the server attributes to the request. It is not possible to push a response to a request that includes a request body.

Pushed responses are always associated with an explicit request from the client. The PUSH_PROMISE frames sent by the server are sent on that explicit request's stream. The PUSH_PROMISE frame also includes a promised stream identifier, chosen from the stream identifiers available to the server (see Section 5.1.1).

The header fields in PUSH_PROMISE and any subsequent CONTINUATION frames MUST be a valid and complete set of request header fields (Section 8.1.2.3). The server MUST include a method in the ":method" pseudo-header field that is safe and cacheable. If a client receives a PUSH_PROMISE that does not include a complete and valid set of header fields or the ":method" pseudo-header field identifies a method that is not safe, it MUST respond with a stream error (Section 5.4.2) of type `PROTOCOL_ERROR`.

The server SHOULD send PUSH_PROMISE (Section 6.6) frames prior to sending any frames that reference the promised responses. This avoids a race where clients issue requests prior to receiving any PUSH_PROMISE frames.

For example, if the server receives a request for a document containing embedded links to multiple image files and the server chooses to push those additional images to the client, sending PUSH_PROMISE frames before the DATA frames that contain the image links ensures that the client is able to see that a resource will be pushed before discovering embedded links. Similarly, if the server pushes responses referenced by the header block (for instance, in Link header fields), sending a PUSH_PROMISE before sending the header block ensures that clients do not request those resources.

PUSH_PROMISE frames MUST NOT be sent by the client.

PUSH_PROMISE frames can be sent by the server in response to any client-initiated stream, but the stream MUST be in either the "open" or "half-closed (remote)" state with respect to the server. PUSH_PROMISE frames are interspersed with the frames that comprise a response, though they cannot be interspersed with HEADERS and CONTINUATION frames that comprise a single header block.

Sending a PUSH_PROMISE frame creates a new stream and puts the stream into the "reserved (local)" state for the server and the "reserved (remote)" state for the client.

8.2.2. Push Responses

After sending the `PUSH_PROMISE` frame, the server can begin delivering the pushed response as a response (Section 8.1.2.4) on a server-initiated stream that uses the promised stream identifier. The server uses this stream to transmit an HTTP response, using the same sequence of frames as defined in Section 8.1. This stream becomes "half-closed" to the client (Section 5.1) after the initial `HEADERS` frame is sent.

Once a client receives a `PUSH_PROMISE` frame and chooses to accept the pushed response, the client **SHOULD NOT** issue any requests for the promised response until after the promised stream has closed.

If the client determines, for any reason, that it does not wish to receive the pushed response from the server or if the server takes too long to begin sending the promised response, the client can send a `RST_STREAM` frame, using either the `CANCEL` or `REFUSED_STREAM` code and referencing the pushed stream's identifier.

A client can use the `SETTINGS_MAX_CONCURRENT_STREAMS` setting to limit the number of responses that can be concurrently pushed by a server. Advertising a `SETTINGS_MAX_CONCURRENT_STREAMS` value of zero disables server push by preventing the server from creating the necessary streams. This does not prohibit a server from sending `PUSH_PROMISE` frames; clients need to reset any promised streams that are not wanted.

Clients receiving a pushed response **MUST** validate that either the server is authoritative (see Section 10.1) or the proxy that provided the pushed response is configured for the corresponding request. For example, a server that offers a certificate for only the "example.com" DNS-ID or Common Name is not permitted to push a response for "https://www.example.org/doc".

The response for a `PUSH_PROMISE` stream begins with a `HEADERS` frame, which immediately puts the stream into the "half-closed (remote)" state for the server and "half-closed (local)" state for the client, and ends with a frame bearing `END_STREAM`, which places the stream in the "closed" state.

Note: The client never sends a frame with the `END_STREAM` flag for a server push.

8.3. The CONNECT Method

In HTTP/1.x, the pseudo-method CONNECT ([RFC7231], Section 4.3.6) is used to convert an HTTP connection into a tunnel to a remote host. CONNECT is primarily used with HTTP proxies to establish a TLS session with an origin server for the purposes of interacting with "https" resources.

In HTTP/2, the CONNECT method is used to establish a tunnel over a single HTTP/2 stream to a remote host for similar purposes. The HTTP header field mapping works as defined in Section 8.1.2.3 ("Request Pseudo-Header Fields"), with a few differences. Specifically:

- o The ":method" pseudo-header field is set to "CONNECT".
- o The ":scheme" and ":path" pseudo-header fields MUST be omitted.
- o The ":authority" pseudo-header field contains the host and port to connect to (equivalent to the authority-form of the request-target of CONNECT requests (see [RFC7230], Section 5.3)).

A CONNECT request that does not conform to these restrictions is malformed (Section 8.1.2.6).

A proxy that supports CONNECT establishes a TCP connection [TCP] to the server identified in the ":authority" pseudo-header field. Once this connection is successfully established, the proxy sends a HEADERS frame containing a 2xx series status code to the client, as defined in [RFC7231], Section 4.3.6.

After the initial HEADERS frame sent by each peer, all subsequent DATA frames correspond to data sent on the TCP connection. The payload of any DATA frames sent by the client is transmitted by the proxy to the TCP server; data received from the TCP server is assembled into DATA frames by the proxy. Frame types other than DATA or stream management frames (RST_STREAM, WINDOW_UPDATE, and PRIORITY) MUST NOT be sent on a connected stream and MUST be treated as a stream error (Section 5.4.2) if received.

The TCP connection can be closed by either peer. The END_STREAM flag on a DATA frame is treated as being equivalent to the TCP FIN bit. A client is expected to send a DATA frame with the END_STREAM flag set after receiving a frame bearing the END_STREAM flag. A proxy that receives a DATA frame with the END_STREAM flag set sends the attached data with the FIN bit set on the last TCP segment. A proxy that receives a TCP segment with the FIN bit set sends a DATA frame with the END_STREAM flag set. Note that the final TCP segment or DATA frame could be empty.

A TCP connection error is signaled with RST_STREAM. A proxy treats any error in the TCP connection, which includes receiving a TCP segment with the RST bit set, as a stream error (Section 5.4.2) of type CONNECT_ERROR. Correspondingly, a proxy MUST send a TCP segment with the RST bit set if it detects an error with the stream or the HTTP/2 connection.

9. Additional HTTP Requirements/Considerations

This section outlines attributes of the HTTP protocol that improve interoperability, reduce exposure to known security vulnerabilities, or reduce the potential for implementation variation.

9.1. Connection Management

HTTP/2 connections are persistent. For best performance, it is expected that clients will not close connections until it is determined that no further communication with a server is necessary (for example, when a user navigates away from a particular web page) or until the server closes the connection.

Clients SHOULD NOT open more than one HTTP/2 connection to a given host and port pair, where the host is derived from a URI, a selected alternative service [ALT-SVC], or a configured proxy.

A client can create additional connections as replacements, either to replace connections that are near to exhausting the available stream identifier space (Section 5.1.1), to refresh the keying material for a TLS connection, or to replace connections that have encountered errors (Section 5.4.1).

A client MAY open multiple connections to the same IP address and TCP port using different Server Name Indication [TLS-EXT] values or to provide different TLS client certificates but SHOULD avoid creating multiple connections with the same configuration.

Servers are encouraged to maintain open connections for as long as possible but are permitted to terminate idle connections if necessary. When either endpoint chooses to close the transport-layer TCP connection, the terminating endpoint SHOULD first send a GOAWAY (Section 6.8) frame so that both endpoints can reliably determine whether previously sent frames have been processed and gracefully complete or terminate any necessary remaining tasks.

9.1.1. Connection Reuse

Connections that are made to an origin server, either directly or through a tunnel created using the CONNECT method (Section 8.3), MAY be reused for requests with multiple different URI authority components. A connection can be reused as long as the origin server is authoritative (Section 10.1). For TCP connections without TLS, this depends on the host having resolved to the same IP address.

For "https" resources, connection reuse additionally depends on having a certificate that is valid for the host in the URI. The certificate presented by the server MUST satisfy any checks that the client would perform when forming a new TLS connection for the host in the URI.

An origin server might offer a certificate with multiple "subjectAltName" attributes or names with wildcards, one of which is valid for the authority in the URI. For example, a certificate with a "subjectAltName" of "*.example.com" might permit the use of the same connection for requests to URIs starting with "https://a.example.com/" and "https://b.example.com/".

In some deployments, reusing a connection for multiple origins can result in requests being directed to the wrong origin server. For example, TLS termination might be performed by a middlebox that uses the TLS Server Name Indication (SNI) [TLS-EXT] extension to select an origin server. This means that it is possible for clients to send confidential information to servers that might not be the intended target for the request, even though the server is otherwise authoritative.

A server that does not wish clients to reuse connections can indicate that it is not authoritative for a request by sending a 421 (Misdirected Request) status code in response to the request (see Section 9.1.2).

A client that is configured to use a proxy over HTTP/2 directs requests to that proxy through a single connection. That is, all requests sent via a proxy reuse the connection to the proxy.

9.1.2. The 421 (Misdirected Request) Status Code

The 421 (Misdirected Request) status code indicates that the request was directed at a server that is not able to produce a response. This can be sent by a server that is not configured to produce responses for the combination of scheme and authority that are included in the request URI.

Clients receiving a 421 (Misdirected Request) response from a server MAY retry the request -- whether the request method is idempotent or not -- over a different connection. This is possible if a connection is reused (Section 9.1.1) or if an alternative service is selected [ALT-SVC].

This status code MUST NOT be generated by proxies.

A 421 response is cacheable by default, i.e., unless otherwise indicated by the method definition or explicit cache controls (see Section 4.2.2 of [RFC7234]).

9.2. Use of TLS Features

Implementations of HTTP/2 MUST use TLS version 1.2 [TLS12] or higher for HTTP/2 over TLS. The general TLS usage guidance in [TLSBCP] SHOULD be followed, with some additional restrictions that are specific to HTTP/2.

The TLS implementation MUST support the Server Name Indication (SNI) [TLS-EXT] extension to TLS. HTTP/2 clients MUST indicate the target domain name when negotiating TLS.

Deployments of HTTP/2 that negotiate TLS 1.3 or higher need only support and use the SNI extension; deployments of TLS 1.2 are subject to the requirements in the following sections. Implementations are encouraged to provide defaults that comply, but it is recognized that deployments are ultimately responsible for compliance.

9.2.1. TLS 1.2 Features

This section describes restrictions on the TLS 1.2 feature set that can be used with HTTP/2. Due to deployment limitations, it might not be possible to fail TLS negotiation when these restrictions are not met. An endpoint MAY immediately terminate an HTTP/2 connection that does not meet these TLS requirements with a connection error (Section 5.4.1) of type `INADEQUATE_SECURITY`.

A deployment of HTTP/2 over TLS 1.2 MUST disable compression. TLS compression can lead to the exposure of information that would not otherwise be revealed [RFC3749]. Generic compression is unnecessary since HTTP/2 provides compression features that are more aware of context and therefore likely to be more appropriate for use for performance, security, or other reasons.

A deployment of HTTP/2 over TLS 1.2 MUST disable renegotiation. An endpoint MUST treat a TLS renegotiation as a connection error (Section 5.4.1) of type `PROTOCOL_ERROR`. Note that disabling

renegotiation can result in long-lived connections becoming unusable due to limits on the number of messages the underlying cipher suite can encipher.

An endpoint MAY use renegotiation to provide confidentiality protection for client credentials offered in the handshake, but any renegotiation MUST occur prior to sending the connection preface. A server SHOULD request a client certificate if it sees a renegotiation request immediately after establishing a connection.

This effectively prevents the use of renegotiation in response to a request for a specific protected resource. A future specification might provide a way to support this use case. Alternatively, a server might use an error (Section 5.4) of type HTTP_1_1_REQUIRED to request the client use a protocol that supports renegotiation.

Implementations MUST support ephemeral key exchange sizes of at least 2048 bits for cipher suites that use ephemeral finite field Diffie-Hellman (DHE) [TLS12] and 224 bits for cipher suites that use ephemeral elliptic curve Diffie-Hellman (ECDHE) [RFC4492]. Clients MUST accept DHE sizes of up to 4096 bits. Endpoints MAY treat negotiation of key sizes smaller than the lower limits as a connection error (Section 5.4.1) of type INADEQUATE_SECURITY.

9.2.2. TLS 1.2 Cipher Suites

A deployment of HTTP/2 over TLS 1.2 SHOULD NOT use any of the cipher suites that are listed in the cipher suite black list (Appendix A).

Endpoints MAY choose to generate a connection error (Section 5.4.1) of type INADEQUATE_SECURITY if one of the cipher suites from the black list is negotiated. A deployment that chooses to use a black-listed cipher suite risks triggering a connection error unless the set of potential peers is known to accept that cipher suite.

Implementations MUST NOT generate this error in reaction to the negotiation of a cipher suite that is not on the black list. Consequently, when clients offer a cipher suite that is not on the black list, they have to be prepared to use that cipher suite with HTTP/2.

The black list includes the cipher suite that TLS 1.2 makes mandatory, which means that TLS 1.2 deployments could have non-intersecting sets of permitted cipher suites. To avoid this problem causing TLS handshake failures, deployments of HTTP/2 that use TLS 1.2 MUST support TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 [TLS-ECDHE] with the P-256 elliptic curve [FIPS186].

Note that clients might advertise support of cipher suites that are on the black list in order to allow for connection to servers that do not support HTTP/2. This allows servers to select HTTP/1.1 with a cipher suite that is on the HTTP/2 black list. However, this can result in HTTP/2 being negotiated with a black-listed cipher suite if the application protocol and cipher suite are independently selected.

10. Security Considerations

10.1. Server Authority

HTTP/2 relies on the HTTP/1.1 definition of authority for determining whether a server is authoritative in providing a given response (see [RFC7230], Section 9.1). This relies on local name resolution for the "http" URI scheme and the authenticated server identity for the "https" scheme (see [RFC2818], Section 3).

10.2. Cross-Protocol Attacks

In a cross-protocol attack, an attacker causes a client to initiate a transaction in one protocol toward a server that understands a different protocol. An attacker might be able to cause the transaction to appear as a valid transaction in the second protocol. In combination with the capabilities of the web context, this can be used to interact with poorly protected servers in private networks.

Completing a TLS handshake with an ALPN identifier for HTTP/2 can be considered sufficient protection against cross-protocol attacks. ALPN provides a positive indication that a server is willing to proceed with HTTP/2, which prevents attacks on other TLS-based protocols.

The encryption in TLS makes it difficult for attackers to control the data that could be used in a cross-protocol attack on a cleartext protocol.

The cleartext version of HTTP/2 has minimal protection against cross-protocol attacks. The connection preface (Section 3.5) contains a string that is designed to confuse HTTP/1.1 servers, but no special protection is offered for other protocols. A server that is willing to ignore parts of an HTTP/1.1 request containing an Upgrade header field in addition to the client connection preface could be exposed to a cross-protocol attack.

10.3. Intermediary Encapsulation Attacks

The HTTP/2 header field encoding allows the expression of names that are not valid field names in the Internet Message Syntax used by HTTP/1.1. Requests or responses containing invalid header field names **MUST** be treated as malformed (Section 8.1.2.6). An intermediary therefore cannot translate an HTTP/2 request or response containing an invalid field name into an HTTP/1.1 message.

Similarly, HTTP/2 allows header field values that are not valid. While most of the values that can be encoded will not alter header field parsing, carriage return (CR, ASCII 0xd), line feed (LF, ASCII 0xa), and the zero character (NUL, ASCII 0x0) might be exploited by an attacker if they are translated verbatim. Any request or response that contains a character not permitted in a header field value **MUST** be treated as malformed (Section 8.1.2.6). Valid characters are defined by the "field-content" ABNF rule in Section 3.2 of [RFC7230].

10.4. Cacheability of Pushed Responses

Pushed responses do not have an explicit request from the client; the request is provided by the server in the PUSH_PROMISE frame.

Caching responses that are pushed is possible based on the guidance provided by the origin server in the Cache-Control header field. However, this can cause issues if a single server hosts more than one tenant. For example, a server might offer multiple users each a small portion of its URI space.

Where multiple tenants share space on the same server, that server **MUST** ensure that tenants are not able to push representations of resources that they do not have authority over. Failure to enforce this would allow a tenant to provide a representation that would be served out of cache, overriding the actual representation that the authoritative tenant provides.

Pushed responses for which an origin server is not authoritative (see Section 10.1) **MUST NOT** be used or cached.

10.5. Denial-of-Service Considerations

An HTTP/2 connection can demand a greater commitment of resources to operate than an HTTP/1.1 connection. The use of header compression and flow control depend on a commitment of resources for storing a greater amount of state. Settings for these features ensure that memory commitments for these features are strictly bounded.

The number of PUSH_PROMISE frames is not constrained in the same fashion. A client that accepts server push SHOULD limit the number of streams it allows to be in the "reserved (remote)" state. An excessive number of server push streams can be treated as a stream error (Section 5.4.2) of type ENHANCE_YOUR_CALM.

Processing capacity cannot be guarded as effectively as state capacity.

The SETTINGS frame can be abused to cause a peer to expend additional processing time. This might be done by pointlessly changing SETTINGS parameters, setting multiple undefined parameters, or changing the same setting multiple times in the same frame. WINDOW_UPDATE or PRIORITY frames can be abused to cause an unnecessary waste of resources.

Large numbers of small or empty frames can be abused to cause a peer to expend time processing frame headers. Note, however, that some uses are entirely legitimate, such as the sending of an empty DATA or CONTINUATION frame at the end of a stream.

Header compression also offers some opportunities to waste processing resources; see Section 7 of [COMPRESSION] for more details on potential abuses.

Limits in SETTINGS parameters cannot be reduced instantaneously, which leaves an endpoint exposed to behavior from a peer that could exceed the new limits. In particular, immediately after establishing a connection, limits set by a server are not known to clients and could be exceeded without being an obvious protocol violation.

All these features -- i.e., SETTINGS changes, small frames, header compression -- have legitimate uses. These features become a burden only when they are used unnecessarily or to excess.

An endpoint that doesn't monitor this behavior exposes itself to a risk of denial-of-service attack. Implementations SHOULD track the use of these features and set limits on their use. An endpoint MAY treat activity that is suspicious as a connection error (Section 5.4.1) of type ENHANCE_YOUR_CALM.

10.5.1. Limits on Header Block Size

A large header block (Section 4.3) can cause an implementation to commit a large amount of state. Header fields that are critical for routing can appear toward the end of a header block, which prevents streaming of header fields to their ultimate destination. This ordering and other reasons, such as ensuring cache correctness, mean

that an endpoint might need to buffer the entire header block. Since there is no hard limit to the size of a header block, some endpoints could be forced to commit a large amount of available memory for header fields.

An endpoint can use the `SETTINGS_MAX_HEADER_LIST_SIZE` to advise peers of limits that might apply on the size of header blocks. This setting is only advisory, so endpoints MAY choose to send header blocks that exceed this limit and risk having the request or response being treated as malformed. This setting is specific to a connection, so any request or response could encounter a hop with a lower, unknown limit. An intermediary can attempt to avoid this problem by passing on values presented by different peers, but they are not obligated to do so.

A server that receives a larger header block than it is willing to handle can send an HTTP 431 (Request Header Fields Too Large) status code [RFC6585]. A client can discard responses that it cannot process. The header block MUST be processed to ensure a consistent connection state, unless the connection is closed.

10.5.2. CONNECT Issues

The CONNECT method can be used to create disproportionate load on an proxy, since stream creation is relatively inexpensive when compared to the creation and maintenance of a TCP connection. A proxy might also maintain some resources for a TCP connection beyond the closing of the stream that carries the CONNECT request, since the outgoing TCP connection remains in the TIME_WAIT state. Therefore, a proxy cannot rely on `SETTINGS_MAX_CONCURRENT_STREAMS` alone to limit the resources consumed by CONNECT requests.

10.6. Use of Compression

Compression can allow an attacker to recover secret data when it is compressed in the same context as data under attacker control. HTTP/2 enables compression of header fields (Section 4.3); the following concerns also apply to the use of HTTP compressed content-codings ([RFC7231], Section 3.1.2.1).

There are demonstrable attacks on compression that exploit the characteristics of the web (e.g., [BREACH]). The attacker induces multiple requests containing varying plaintext, observing the length of the resulting ciphertext in each, which reveals a shorter length when a guess about the secret is correct.

Implementations communicating on a secure channel MUST NOT compress content that includes both confidential and attacker-controlled data unless separate compression dictionaries are used for each source of data. Compression MUST NOT be used if the source of data cannot be reliably determined. Generic stream compression, such as that provided by TLS, MUST NOT be used with HTTP/2 (see Section 9.2).

Further considerations regarding the compression of header fields are described in [COMPRESSION].

10.7. Use of Padding

Padding within HTTP/2 is not intended as a replacement for general purpose padding, such as might be provided by TLS [TLS12]. Redundant padding could even be counterproductive. Correct application can depend on having specific knowledge of the data that is being padded.

To mitigate attacks that rely on compression, disabling or limiting compression might be preferable to padding as a countermeasure.

Padding can be used to obscure the exact size of frame content and is provided to mitigate specific attacks within HTTP, for example, attacks where compressed content includes both attacker-controlled plaintext and secret data (e.g., [BREACH]).

Use of padding can result in less protection than might seem immediately obvious. At best, padding only makes it more difficult for an attacker to infer length information by increasing the number of frames an attacker has to observe. Incorrectly implemented padding schemes can be easily defeated. In particular, randomized padding with a predictable distribution provides very little protection; similarly, padding payloads to a fixed size exposes information as payload sizes cross the fixed-sized boundary, which could be possible if an attacker can control plaintext.

Intermediaries SHOULD retain padding for DATA frames but MAY drop padding for HEADERS and PUSH_PROMISE frames. A valid reason for an intermediary to change the amount of padding of frames is to improve the protections that padding provides.

10.8. Privacy Considerations

Several characteristics of HTTP/2 provide an observer an opportunity to correlate actions of a single client or server over time. These include the value of settings, the manner in which flow-control windows are managed, the way priorities are allocated to streams, the timing of reactions to stimulus, and the handling of any features that are controlled by settings.

As far as these create observable differences in behavior, they could be used as a basis for fingerprinting a specific client, as defined in Section 1.8 of [HTML5].

HTTP/2's preference for using a single TCP connection allows correlation of a user's activity on a site. Reusing connections for different origins allows tracking across those origins.

Because the PING and SETTINGS frames solicit immediate responses, they can be used by an endpoint to measure latency to their peer. This might have privacy implications in certain scenarios.

11. IANA Considerations

A string for identifying HTTP/2 is entered into the "Application-Layer Protocol Negotiation (ALPN) Protocol IDs" registry established in [TLS-ALPN].

This document establishes a registry for frame types, settings, and error codes. These new registries appear in the new "Hypertext Transfer Protocol version 2 (HTTP/2) Parameters" section.

This document registers the HTTP2-Settings header field for use in HTTP; it also registers the 421 (Misdirected Request) status code.

This document registers the "PRI" method for use in HTTP to avoid collisions with the connection preface (Section 3.5).

11.1. Registration of HTTP/2 Identification Strings

This document creates two registrations for the identification of HTTP/2 (see Section 3.3) in the "Application-Layer Protocol Negotiation (ALPN) Protocol IDs" registry established in [TLS-ALPN].

The "h2" string identifies HTTP/2 when used over TLS:

Protocol: HTTP/2 over TLS

Identification Sequence: 0x68 0x32 ("h2")

Specification: This document

The "h2c" string identifies HTTP/2 when used over cleartext TCP:

Protocol: HTTP/2 over TCP

Identification Sequence: 0x68 0x32 0x63 ("h2c")

Specification: This document

11.2. Frame Type Registry

This document establishes a registry for HTTP/2 frame type codes. The "HTTP/2 Frame Type" registry manages an 8-bit space. The "HTTP/2 Frame Type" registry operates under either of the "IETF Review" or "IESG Approval" policies [RFC5226] for values between 0x00 and 0xef, with values between 0xf0 and 0xff being reserved for Experimental Use.

New entries in this registry require the following information:

Frame Type: A name or label for the frame type.

Code: The 8-bit code assigned to the frame type.

Specification: A reference to a specification that includes a description of the frame layout, its semantics, and flags that the frame type uses, including any parts of the frame that are conditionally present based on the value of flags.

The entries in the following table are registered by this document.

Frame Type	Code	Section
DATA	0x0	Section 6.1
HEADERS	0x1	Section 6.2
PRIORITY	0x2	Section 6.3
RST_STREAM	0x3	Section 6.4
SETTINGS	0x4	Section 6.5
PUSH_PROMISE	0x5	Section 6.6
PING	0x6	Section 6.7
GOAWAY	0x7	Section 6.8
WINDOW_UPDATE	0x8	Section 6.9
CONTINUATION	0x9	Section 6.10

11.3. Settings Registry

This document establishes a registry for HTTP/2 settings. The "HTTP/2 Settings" registry manages a 16-bit space. The "HTTP/2 Settings" registry operates under the "Expert Review" policy [RFC5226] for values in the range from 0x0000 to 0xffff, with values between 0xf000 and 0xffff being reserved for Experimental Use.

New registrations are advised to provide the following information:

Name: A symbolic name for the setting. Specifying a setting name is optional.

Code: The 16-bit code assigned to the setting.

Initial Value: An initial value for the setting.

Specification: An optional reference to a specification that describes the use of the setting.

The entries in the following table are registered by this document.

Name	Code	Initial Value	Specification
HEADER_TABLE_SIZE	0x1	4096	Section 6.5.2
ENABLE_PUSH	0x2	1	Section 6.5.2
MAX_CONCURRENT_STREAMS	0x3	(infinite)	Section 6.5.2
INITIAL_WINDOW_SIZE	0x4	65535	Section 6.5.2
MAX_FRAME_SIZE	0x5	16384	Section 6.5.2
MAX_HEADER_LIST_SIZE	0x6	(infinite)	Section 6.5.2

11.4. Error Code Registry

This document establishes a registry for HTTP/2 error codes. The "HTTP/2 Error Code" registry manages a 32-bit space. The "HTTP/2 Error Code" registry operates under the "Expert Review" policy [RFC5226].

Registrations for error codes are required to include a description of the error code. An expert reviewer is advised to examine new registrations for possible duplication with existing error codes. Use of existing registrations is to be encouraged, but not mandated.

New registrations are advised to provide the following information:

Name: A name for the error code. Specifying an error code name is optional.

Code: The 32-bit error code value.

Description: A brief description of the error code semantics, longer if no detailed specification is provided.

Specification: An optional reference for a specification that defines the error code.

The entries in the following table are registered by this document.

Name	Code	Description	Specification
NO_ERROR	0x0	Graceful shutdown	Section 7
PROTOCOL_ERROR	0x1	Protocol error detected	Section 7
INTERNAL_ERROR	0x2	Implementation fault	Section 7
FLOW_CONTROL_ERROR	0x3	Flow-control limits exceeded	Section 7
SETTINGS_TIMEOUT	0x4	Settings not acknowledged	Section 7
STREAM_CLOSED	0x5	Frame received for closed stream	Section 7
FRAME_SIZE_ERROR	0x6	Frame size incorrect	Section 7
REFUSED_STREAM	0x7	Stream not processed	Section 7
CANCEL	0x8	Stream cancelled	Section 7
COMPRESSION_ERROR	0x9	Compression state not updated	Section 7
CONNECT_ERROR	0xa	TCP connection error for CONNECT method	Section 7
ENHANCE_YOUR_CALM	0xb	Processing capacity exceeded	Section 7
INADEQUATE_SECURITY	0xc	Negotiated TLS parameters not acceptable	Section 7
HTTP_1_1_REQUIRED	0xd	Use HTTP/1.1 for the request	Section 7

11.5. HTTP2-Settings Header Field Registration

This section registers the HTTP2-Settings header field in the "Permanent Message Header Field Names" registry [BCP90].

Header field name: HTTP2-Settings

Applicable protocol: http

Status: standard

Author/Change controller: IETF

Specification document(s): Section 3.2.1 of this document

Related information: This header field is only used by an HTTP/2 client for Upgrade-based negotiation.

11.6. PRI Method Registration

This section registers the "PRI" method in the "HTTP Method Registry" ([RFC7231], Section 8.1).

Method Name: PRI

Safe: Yes

Idempotent: Yes

Specification document(s): Section 3.5 of this document

Related information: This method is never used by an actual client. This method will appear to be used when an HTTP/1.1 server or intermediary attempts to parse an HTTP/2 connection preface.

11.7. The 421 (Misdirected Request) HTTP Status Code

This document registers the 421 (Misdirected Request) HTTP status code in the "HTTP Status Codes" registry ([RFC7231], Section 8.2).

Status Code: 421

Short Description: Misdirected Request

Specification: Section 9.1.2 of this document

11.8. The h2c Upgrade Token

This document registers the "h2c" upgrade token in the "HTTP Upgrade Tokens" registry ([RFC7230], Section 8.6).

Value: h2c

Description: Hypertext Transfer Protocol version 2 (HTTP/2)

Expected Version Tokens: None

Reference: Section 3.2 of this document

12. References

12.1. Normative References

- [COMPRESSION] Peon, R. and H. Ruellan, "HPACK: Header Compression for HTTP/2", RFC 7541, DOI 10.17487/RFC7541, May 2015, <<http://www.rfc-editor.org/info/rfc7541>>.
- [COOKIE] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <<http://www.rfc-editor.org/info/rfc6265>>.
- [FIPS186] NIST, "Digital Signature Standard (DSS)", FIPS PUB 186-4, July 2013, <<http://dx.doi.org/10.6028/NIST.FIPS.186-4>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<http://www.rfc-editor.org/info/rfc2119>>.
- [RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, DOI 10.17487/RFC2818, May 2000, <<http://www.rfc-editor.org/info/rfc2818>>.
- [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <<http://www.rfc-editor.org/info/rfc3986>>.
- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <<http://www.rfc-editor.org/info/rfc4648>>.
- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, DOI 10.17487/RFC5226, May 2008, <<http://www.rfc-editor.org/info/rfc5226>>.
- [RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <<http://www.rfc-editor.org/info/rfc5234>>.
- [RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <<http://www.rfc-editor.org/info/rfc7230>>.

- [RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <<http://www.rfc-editor.org/info/rfc7231>>.
- [RFC7232] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests", RFC 7232, DOI 10.17487/RFC7232, June 2014, <<http://www.rfc-editor.org/info/rfc7232>>.
- [RFC7233] Fielding, R., Ed., Lafon, Y., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Range Requests", RFC 7233, DOI 10.17487/RFC7233, June 2014, <<http://www.rfc-editor.org/info/rfc7233>>.
- [RFC7234] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching", RFC 7234, DOI 10.17487/RFC7234, June 2014, <<http://www.rfc-editor.org/info/rfc7234>>.
- [RFC7235] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Authentication", RFC 7235, DOI 10.17487/RFC7235, June 2014, <<http://www.rfc-editor.org/info/rfc7235>>.
- [TCP] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <<http://www.rfc-editor.org/info/rfc793>>.
- [TLS-ALPN] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <<http://www.rfc-editor.org/info/rfc7301>>.
- [TLS-ECDHE] Rescorla, E., "TLS Elliptic Curve Cipher Suites with SHA-256/384 and AES Galois Counter Mode (GCM)", RFC 5289, DOI 10.17487/RFC5289, August 2008, <<http://www.rfc-editor.org/info/rfc5289>>.
- [TLS-EXT] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <<http://www.rfc-editor.org/info/rfc6066>>.

- [TLS12] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <<http://www.rfc-editor.org/info/rfc5246>>.

12.2. Informative References

- [ALT-SVC] Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", Work in Progress, draft-ietf-httpbis-alt-svc-06, February 2015.
- [BCP90] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, September 2004, <<http://www.rfc-editor.org/info/bcp90>>.
- [BREACH] Gluck, Y., Harris, N., and A. Prado, "BREACH: Reviving the CRIME Attack", July 2013, <<http://breachattack.com/resources/BREACH%20-%20SSL,%20gone%20in%2030%20seconds.pdf>>.
- [HTML5] Hickson, I., Berjon, R., Faulkner, S., Leithead, T., Doyle Navara, E., O'Connor, E., and S. Pfeiffer, "HTML5", W3C Recommendation REC-html5-20141028, October 2014, <<http://www.w3.org/TR/2014/REC-html5-20141028/>>.
- [RFC3749] Hollenbeck, S., "Transport Layer Security Protocol Compression Methods", RFC 3749, DOI 10.17487/RFC3749, May 2004, <<http://www.rfc-editor.org/info/rfc3749>>.
- [RFC4492] Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., and B. Moeller, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)", RFC 4492, DOI 10.17487/RFC4492, May 2006, <<http://www.rfc-editor.org/info/rfc4492>>.
- [RFC6585] Nottingham, M. and R. Fielding, "Additional HTTP Status Codes", RFC 6585, DOI 10.17487/RFC6585, April 2012, <<http://www.rfc-editor.org/info/rfc6585>>.
- [RFC7323] Borman, D., Braden, B., Jacobson, V., and R. Scheffenegger, Ed., "TCP Extensions for High Performance", RFC 7323, DOI 10.17487/RFC7323, September 2014, <<http://www.rfc-editor.org/info/rfc7323>>.
- [TALKING] Huang, L., Chen, E., Barth, A., Rescorla, E., and C. Jackson, "Talking to Yourself for Fun and Profit", 2011, <<http://w2spconf.com/2011/papers/websocket.pdf>>.

[TLSBCP] Sheffer, Y., Holz, R., and P. Saint-Andre,
"Recommendations for Secure Use of Transport Layer
Security (TLS) and Datagram Transport Layer Security
(DTLS)", BCP 195, RFC 7525, DOI 10.17487/RFC7525, May
2015, <<http://www.rfc-editor.org/info/rfc7525>>.

Appendix A. TLS 1.2 Cipher Suite Black List

An HTTP/2 implementation MAY treat the negotiation of any of the following cipher suites with TLS 1.2 as a connection error (Section 5.4.1) of type INADEQUATE_SECURITY:

- TLS_NULL_WITH_NULL_NULL
- TLS_RSA_WITH_NULL_MD5
- TLS_RSA_WITH_NULL_SHA
- TLS_RSA_EXPORT_WITH_RC4_40_MD5
- TLS_RSA_WITH_RC4_128_MD5
- TLS_RSA_WITH_RC4_128_SHA
- TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
- TLS_RSA_WITH_IDEA_CBC_SHA
- TLS_RSA_EXPORT_WITH_DES40_CBC_SHA
- TLS_RSA_WITH_DES_CBC_SHA
- TLS_RSA_WITH_3DES_EDE_CBC_SHA
- TLS_DH_DSS_EXPORT_WITH_DES40_CBC_SHA
- TLS_DH_DSS_WITH_DES_CBC_SHA
- TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA
- TLS_DH_RSA_EXPORT_WITH_DES40_CBC_SHA
- TLS_DH_RSA_WITH_DES_CBC_SHA
- TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA
- TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
- TLS_DHE_DSS_WITH_DES_CBC_SHA
- TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA
- TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA

- TLS_DHE_RSA_WITH_DES_CBC_SHA
- TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA
- TLS_DH_anon_EXPORT_WITH_RC4_40_MD5
- TLS_DH_anon_WITH_RC4_128_MD5
- TLS_DH_anon_EXPORT_WITH_DES40_CBC_SHA
- TLS_DH_anon_WITH_DES_CBC_SHA
- TLS_DH_anon_WITH_3DES_EDE_CBC_SHA
- TLS_KRB5_WITH_DES_CBC_SHA
- TLS_KRB5_WITH_3DES_EDE_CBC_SHA
- TLS_KRB5_WITH_RC4_128_SHA
- TLS_KRB5_WITH_IDEA_CBC_SHA
- TLS_KRB5_WITH_DES_CBC_MD5
- TLS_KRB5_WITH_3DES_EDE_CBC_MD5
- TLS_KRB5_WITH_RC4_128_MD5
- TLS_KRB5_WITH_IDEA_CBC_MD5
- TLS_KRB5_EXPORT_WITH_DES_CBC_40_SHA
- TLS_KRB5_EXPORT_WITH_RC2_CBC_40_SHA
- TLS_KRB5_EXPORT_WITH_RC4_40_SHA
- TLS_KRB5_EXPORT_WITH_DES_CBC_40_MD5
- TLS_KRB5_EXPORT_WITH_RC2_CBC_40_MD5
- TLS_KRB5_EXPORT_WITH_RC4_40_MD5
- TLS_PSK_WITH_NULL_SHA
- TLS_DHE_PSK_WITH_NULL_SHA
- TLS_RSA_PSK_WITH_NULL_SHA

- TLS_RSA_WITH_AES_128_CBC_SHA
- TLS_DH_DSS_WITH_AES_128_CBC_SHA
- TLS_DH_RSA_WITH_AES_128_CBC_SHA
- TLS_DHE_DSS_WITH_AES_128_CBC_SHA
- TLS_DHE_RSA_WITH_AES_128_CBC_SHA
- TLS_DH_anon_WITH_AES_128_CBC_SHA
- TLS_RSA_WITH_AES_256_CBC_SHA
- TLS_DH_DSS_WITH_AES_256_CBC_SHA
- TLS_DH_RSA_WITH_AES_256_CBC_SHA
- TLS_DHE_DSS_WITH_AES_256_CBC_SHA
- TLS_DHE_RSA_WITH_AES_256_CBC_SHA
- TLS_DH_anon_WITH_AES_256_CBC_SHA
- TLS_RSA_WITH_NULL_SHA256
- TLS_RSA_WITH_AES_128_CBC_SHA256
- TLS_RSA_WITH_AES_256_CBC_SHA256
- TLS_DH_DSS_WITH_AES_128_CBC_SHA256
- TLS_DH_RSA_WITH_AES_128_CBC_SHA256
- TLS_DHE_DSS_WITH_AES_128_CBC_SHA256
- TLS_RSA_WITH_CAMELLIA_128_CBC_SHA
- TLS_DH_DSS_WITH_CAMELLIA_128_CBC_SHA
- TLS_DH_RSA_WITH_CAMELLIA_128_CBC_SHA
- TLS_DHE_DSS_WITH_CAMELLIA_128_CBC_SHA
- TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA
- TLS_DH_anon_WITH_CAMELLIA_128_CBC_SHA

- TLS_DHE_RSA_WITH_AES_128_CBC_SHA256
- TLS_DH_DSS_WITH_AES_256_CBC_SHA256
- TLS_DH_RSA_WITH_AES_256_CBC_SHA256
- TLS_DHE_DSS_WITH_AES_256_CBC_SHA256
- TLS_DHE_RSA_WITH_AES_256_CBC_SHA256
- TLS_DH_anon_WITH_AES_128_CBC_SHA256
- TLS_DH_anon_WITH_AES_256_CBC_SHA256
- TLS_RSA_WITH_CAMELLIA_256_CBC_SHA
- TLS_DH_DSS_WITH_CAMELLIA_256_CBC_SHA
- TLS_DH_RSA_WITH_CAMELLIA_256_CBC_SHA
- TLS_DHE_DSS_WITH_CAMELLIA_256_CBC_SHA
- TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA
- TLS_DH_anon_WITH_CAMELLIA_256_CBC_SHA
- TLS_PSK_WITH_RC4_128_SHA
- TLS_PSK_WITH_3DES_EDE_CBC_SHA
- TLS_PSK_WITH_AES_128_CBC_SHA
- TLS_PSK_WITH_AES_256_CBC_SHA
- TLS_DHE_PSK_WITH_RC4_128_SHA
- TLS_DHE_PSK_WITH_3DES_EDE_CBC_SHA
- TLS_DHE_PSK_WITH_AES_128_CBC_SHA
- TLS_DHE_PSK_WITH_AES_256_CBC_SHA
- TLS_RSA_PSK_WITH_RC4_128_SHA
- TLS_RSA_PSK_WITH_3DES_EDE_CBC_SHA
- TLS_RSA_PSK_WITH_AES_128_CBC_SHA

- TLS_RSA_PSK_WITH_AES_256_CBC_SHA
- TLS_RSA_WITH_SEED_CBC_SHA
- TLS_DH_DSS_WITH_SEED_CBC_SHA
- TLS_DH_RSA_WITH_SEED_CBC_SHA
- TLS_DHE_DSS_WITH_SEED_CBC_SHA
- TLS_DHE_RSA_WITH_SEED_CBC_SHA
- TLS_DH_anon_WITH_SEED_CBC_SHA
- TLS_RSA_WITH_AES_128_GCM_SHA256
- TLS_RSA_WITH_AES_256_GCM_SHA384
- TLS_DH_RSA_WITH_AES_128_GCM_SHA256
- TLS_DH_RSA_WITH_AES_256_GCM_SHA384
- TLS_DH_DSS_WITH_AES_128_GCM_SHA256
- TLS_DH_DSS_WITH_AES_256_GCM_SHA384
- TLS_DH_anon_WITH_AES_128_GCM_SHA256
- TLS_DH_anon_WITH_AES_256_GCM_SHA384
- TLS_PSK_WITH_AES_128_GCM_SHA256
- TLS_PSK_WITH_AES_256_GCM_SHA384
- TLS_RSA_PSK_WITH_AES_128_GCM_SHA256
- TLS_RSA_PSK_WITH_AES_256_GCM_SHA384
- TLS_PSK_WITH_AES_128_CBC_SHA256
- TLS_PSK_WITH_AES_256_CBC_SHA384
- TLS_PSK_WITH_NULL_SHA256
- TLS_PSK_WITH_NULL_SHA384
- TLS_DHE_PSK_WITH_AES_128_CBC_SHA256

- TLS_DHE_PSK_WITH_AES_256_CBC_SHA384
- TLS_DHE_PSK_WITH_NULL_SHA256
- TLS_DHE_PSK_WITH_NULL_SHA384
- TLS_RSA_PSK_WITH_AES_128_CBC_SHA256
- TLS_RSA_PSK_WITH_AES_256_CBC_SHA384
- TLS_RSA_PSK_WITH_NULL_SHA256
- TLS_RSA_PSK_WITH_NULL_SHA384
- TLS_RSA_WITH_CAMELLIA_128_CBC_SHA256
- TLS_DH_DSS_WITH_CAMELLIA_128_CBC_SHA256
- TLS_DH_RSA_WITH_CAMELLIA_128_CBC_SHA256
- TLS_DHE_DSS_WITH_CAMELLIA_128_CBC_SHA256
- TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA256
- TLS_DH_anon_WITH_CAMELLIA_128_CBC_SHA256
- TLS_RSA_WITH_CAMELLIA_256_CBC_SHA256
- TLS_DH_DSS_WITH_CAMELLIA_256_CBC_SHA256
- TLS_DH_RSA_WITH_CAMELLIA_256_CBC_SHA256
- TLS_DHE_DSS_WITH_CAMELLIA_256_CBC_SHA256
- TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA256
- TLS_DH_anon_WITH_CAMELLIA_256_CBC_SHA256
- TLS_EMPTY_RENEGOTIATION_INFO_SCSV
- TLS_ECDH_ECDSA_WITH_NULL_SHA
- TLS_ECDH_ECDSA_WITH_RC4_128_SHA
- TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA
- TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA

- TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_NULL_SHA
- TLS_ECDHE_ECDSA_WITH_RC4_128_SHA
- TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
- TLS_ECDH_RSA_WITH_NULL_SHA
- TLS_ECDH_RSA_WITH_RC4_128_SHA
- TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA
- TLS_ECDH_RSA_WITH_AES_128_CBC_SHA
- TLS_ECDH_RSA_WITH_AES_256_CBC_SHA
- TLS_ECDHE_RSA_WITH_NULL_SHA
- TLS_ECDHE_RSA_WITH_RC4_128_SHA
- TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
- TLS_ECDH_anon_WITH_NULL_SHA
- TLS_ECDH_anon_WITH_RC4_128_SHA
- TLS_ECDH_anon_WITH_3DES_EDE_CBC_SHA
- TLS_ECDH_anon_WITH_AES_128_CBC_SHA
- TLS_ECDH_anon_WITH_AES_256_CBC_SHA
- TLS_SRP_SHA_WITH_3DES_EDE_CBC_SHA
- TLS_SRP_SHA_RSA_WITH_3DES_EDE_CBC_SHA
- TLS_SRP_SHA_DSS_WITH_3DES_EDE_CBC_SHA

- TLS_SRP_SHA_WITH_AES_128_CBC_SHA
- TLS_SRP_SHA_RSA_WITH_AES_128_CBC_SHA
- TLS_SRP_SHA_DSS_WITH_AES_128_CBC_SHA
- TLS_SRP_SHA_WITH_AES_256_CBC_SHA
- TLS_SRP_SHA_RSA_WITH_AES_256_CBC_SHA
- TLS_SRP_SHA_DSS_WITH_AES_256_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384
- TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256
- TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
- TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256
- TLS_ECDH_RSA_WITH_AES_256_CBC_SHA384
- TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384
- TLS_ECDH_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDH_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_PSK_WITH_RC4_128_SHA
- TLS_ECDHE_PSK_WITH_3DES_EDE_CBC_SHA
- TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA
- TLS_ECDHE_PSK_WITH_AES_256_CBC_SHA
- TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_PSK_WITH_AES_256_CBC_SHA384

- TLS_ECDHE_PSK_WITH_NULL_SHA
- TLS_ECDHE_PSK_WITH_NULL_SHA256
- TLS_ECDHE_PSK_WITH_NULL_SHA384
- TLS_RSA_WITH_ARIA_128_CBC_SHA256
- TLS_RSA_WITH_ARIA_256_CBC_SHA384
- TLS_DH_DSS_WITH_ARIA_128_CBC_SHA256
- TLS_DH_DSS_WITH_ARIA_256_CBC_SHA384
- TLS_DH_RSA_WITH_ARIA_128_CBC_SHA256
- TLS_DH_RSA_WITH_ARIA_256_CBC_SHA384
- TLS_DHE_DSS_WITH_ARIA_128_CBC_SHA256
- TLS_DHE_DSS_WITH_ARIA_256_CBC_SHA384
- TLS_DHE_RSA_WITH_ARIA_128_CBC_SHA256
- TLS_DHE_RSA_WITH_ARIA_256_CBC_SHA384
- TLS_DH_anon_WITH_ARIA_128_CBC_SHA256
- TLS_DH_anon_WITH_ARIA_256_CBC_SHA384
- TLS_ECDHE_ECDSA_WITH_ARIA_128_CBC_SHA256
- TLS_ECDHE_ECDSA_WITH_ARIA_256_CBC_SHA384
- TLS_ECDH_ECDSA_WITH_ARIA_128_CBC_SHA256
- TLS_ECDH_ECDSA_WITH_ARIA_256_CBC_SHA384
- TLS_ECDHE_RSA_WITH_ARIA_128_CBC_SHA256
- TLS_ECDHE_RSA_WITH_ARIA_256_CBC_SHA384
- TLS_ECDH_RSA_WITH_ARIA_128_CBC_SHA256
- TLS_ECDH_RSA_WITH_ARIA_256_CBC_SHA384
- TLS_RSA_WITH_ARIA_128_GCM_SHA256

- TLS_RSA_WITH_ARIA_256_GCM_SHA384
- TLS_DH_RSA_WITH_ARIA_128_GCM_SHA256
- TLS_DH_RSA_WITH_ARIA_256_GCM_SHA384
- TLS_DH_DSS_WITH_ARIA_128_GCM_SHA256
- TLS_DH_DSS_WITH_ARIA_256_GCM_SHA384
- TLS_DH_anon_WITH_ARIA_128_GCM_SHA256
- TLS_DH_anon_WITH_ARIA_256_GCM_SHA384
- TLS_ECDH_ECDSA_WITH_ARIA_128_GCM_SHA256
- TLS_ECDH_ECDSA_WITH_ARIA_256_GCM_SHA384
- TLS_ECDH_RSA_WITH_ARIA_128_GCM_SHA256
- TLS_ECDH_RSA_WITH_ARIA_256_GCM_SHA384
- TLS_PSK_WITH_ARIA_128_CBC_SHA256
- TLS_PSK_WITH_ARIA_256_CBC_SHA384
- TLS_DHE_PSK_WITH_ARIA_128_CBC_SHA256
- TLS_DHE_PSK_WITH_ARIA_256_CBC_SHA384
- TLS_RSA_PSK_WITH_ARIA_128_CBC_SHA256
- TLS_RSA_PSK_WITH_ARIA_256_CBC_SHA384
- TLS_PSK_WITH_ARIA_128_GCM_SHA256
- TLS_PSK_WITH_ARIA_256_GCM_SHA384
- TLS_RSA_PSK_WITH_ARIA_128_GCM_SHA256
- TLS_RSA_PSK_WITH_ARIA_256_GCM_SHA384
- TLS_ECDHE_PSK_WITH_ARIA_128_CBC_SHA256
- TLS_ECDHE_PSK_WITH_ARIA_256_CBC_SHA384
- TLS_ECDHE_ECDSA_WITH_CAMELLIA_128_CBC_SHA256

- TLS_ECDHE_ECDSA_WITH_CAMELLIA_256_CBC_SHA384
- TLS_ECDH_ECDSA_WITH_CAMELLIA_128_CBC_SHA256
- TLS_ECDH_ECDSA_WITH_CAMELLIA_256_CBC_SHA384
- TLS_ECDHE_RSA_WITH_CAMELLIA_128_CBC_SHA256
- TLS_ECDHE_RSA_WITH_CAMELLIA_256_CBC_SHA384
- TLS_ECDH_RSA_WITH_CAMELLIA_128_CBC_SHA256
- TLS_ECDH_RSA_WITH_CAMELLIA_256_CBC_SHA384
- TLS_RSA_WITH_CAMELLIA_128_GCM_SHA256
- TLS_RSA_WITH_CAMELLIA_256_GCM_SHA384
- TLS_DH_RSA_WITH_CAMELLIA_128_GCM_SHA256
- TLS_DH_RSA_WITH_CAMELLIA_256_GCM_SHA384
- TLS_DH_DSS_WITH_CAMELLIA_128_GCM_SHA256
- TLS_DH_DSS_WITH_CAMELLIA_256_GCM_SHA384
- TLS_DH_anon_WITH_CAMELLIA_128_GCM_SHA256
- TLS_DH_anon_WITH_CAMELLIA_256_GCM_SHA384
- TLS_ECDH_ECDSA_WITH_CAMELLIA_128_GCM_SHA256
- TLS_ECDH_ECDSA_WITH_CAMELLIA_256_GCM_SHA384
- TLS_ECDH_RSA_WITH_CAMELLIA_128_GCM_SHA256
- TLS_ECDH_RSA_WITH_CAMELLIA_256_GCM_SHA384
- TLS_PSK_WITH_CAMELLIA_128_GCM_SHA256
- TLS_PSK_WITH_CAMELLIA_256_GCM_SHA384
- TLS_RSA_PSK_WITH_CAMELLIA_128_GCM_SHA256
- TLS_RSA_PSK_WITH_CAMELLIA_256_GCM_SHA384
- TLS_PSK_WITH_CAMELLIA_128_CBC_SHA256

- TLS_PSK_WITH_CAMELLIA_256_CBC_SHA384
- TLS_DHE_PSK_WITH_CAMELLIA_128_CBC_SHA256
- TLS_DHE_PSK_WITH_CAMELLIA_256_CBC_SHA384
- TLS_RSA_PSK_WITH_CAMELLIA_128_CBC_SHA256
- TLS_RSA_PSK_WITH_CAMELLIA_256_CBC_SHA384
- TLS_ECDHE_PSK_WITH_CAMELLIA_128_CBC_SHA256
- TLS_ECDHE_PSK_WITH_CAMELLIA_256_CBC_SHA384
- TLS_RSA_WITH_AES_128_CCM
- TLS_RSA_WITH_AES_256_CCM
- TLS_RSA_WITH_AES_128_CCM_8
- TLS_RSA_WITH_AES_256_CCM_8
- TLS_PSK_WITH_AES_128_CCM
- TLS_PSK_WITH_AES_256_CCM
- TLS_PSK_WITH_AES_128_CCM_8
- TLS_PSK_WITH_AES_256_CCM_8

Note: This list was assembled from the set of registered TLS cipher suites at the time of writing. This list includes those cipher suites that do not offer an ephemeral key exchange and those that are based on the TLS null, stream, or block cipher type (as defined in Section 6.2.3 of [TLS12]). Additional cipher suites with these properties could be defined; these would not be explicitly prohibited.

Acknowledgements

This document includes substantial input from the following individuals:

- o Adam Langley, Wan-Teh Chang, Jim Morrison, Mark Nottingham, Alyssa Wilk, Costin Manolache, William Chan, Vitaliy Lvin, Joe Chan, Adam Barth, Ryan Hamilton, Gavin Peters, Kent Alstad, Kevin Lindsay, Paul Amer, Fan Yang, and Jonathan Leighton (SPDY contributors).
- o Gabriel Montenegro and Willy Tarreau (Upgrade mechanism).
- o William Chan, Salvatore Loreto, Osama Mazahir, Gabriel Montenegro, Jitu Padhye, Roberto Peon, and Rob Trace (Flow control).
- o Mike Bishop (Extensibility).
- o Mark Nottingham, Julian Reschke, James Snell, Jeff Pinner, Mike Bishop, and Herve Ruellan (Substantial editorial contributions).
- o Kari Hurtta, Tatsuhiro Tsujikawa, Greg Wilkins, Poul-Henning Kamp, and Jonathan Thackray.
- o Alexey Melnikov, who was an editor of this document in 2013.

A substantial proportion of Martin's contribution was supported by Microsoft during his employment there.

The Japanese HTTP/2 community provided invaluable contributions, including a number of implementations as well as numerous technical and editorial contributions.

Authors' Addresses

Mike Belshe
BitGo

E-Mail: mike@belshe.com

Roberto Peon
Google, Inc

E-Mail: fenix@google.com

Martin Thomson (editor)
Mozilla
331 E Evelyn Street
Mountain View, CA 94041
United States

E-Mail: martin.thomson@gmail.com