The "SNI" Alt-Svc Parameter
draft-bishop-httplib-sni-altsvc-02

Abstract

HTTP Alternative Services provides a mechanism for an origin to declare that its content is accessible via some other combination of host, port, and protocol. In the process of using such an alternative, an observer can identify that the client is requesting resources from a particular hostname.

This document extends HTTP Alternative Services, in combination with Secondary Certificate Authentication, to enable clients not to disclose the origin to which they intend to connect.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on November 25, 2018.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect
1. Introduction

Confidentiality and authentication during communication are primary goals of using TLS to secure traffic on the Internet. However, due to the nature of TLS, certain information is inherently not confidential – notably, the hostname and the corresponding certificate of the origin to which the client is connecting are transferred unencrypted in the Server Name Indication extension [SNI] and the server’s Certificate message [TLS12].

While the client identity can be obscured by using TLS renegotiation immediately after the handshake (in TLS 1.2) or by using TLS 1.3 [TLS13], the server is not afforded such privacy considerations.

Servers may also have wildcard certificates which do not enumerate specific subdomains, but clients will disclose the first subdomain used on a connection via the SNI extension when establishing the connection.

[SNIEncryption] discusses a potential solution to these issues in Section 3, HTTP Co-Tenancy Fronting, but notes both discoverability and server authentication issues with that approach. This document provides a mechanism to address both limitations.
1.1. Usage

In [AltSvc], once a client has received a validated Alternative Service record for an origin, it "SHOULD use that alternative service for all requests to the associated origin as soon as it is available, provided the alternative service information is fresh (Section 2.2) and the security properties of the alternative service protocol are desirable, as compared to the existing connection." However, the client "MUST have reasonable assurances that the alternative service is under control of and valid for the whole origin ... established through use of a TLS-based protocol with the certificate checks defined in [RFC2818]." This causes the origin to be disclosed in the SNI extension while connecting to the alternative, and the origin’s certificate to be returned by the alternative, creating the same privacy issues as connecting directly to the origin.

The extension described in Section 2 enables an origin to declare that reasonable assurances should be obtained, not by requesting the desired hostname in the TLS handshake, but by requesting it via [SecondaryCerts]. The validation checks from [RFC2818] are applied to this certificate.

Because the entire exchange happens inside TLS, a passive observer cannot identify the hostname(s) the client might be requesting.

1.2. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

The key words "MUST (BUT WE KNOW YOU WON’T)", "SHOULD CONSIDER", "REALLY SHOULD NOT", "OUGHT TO", "WOULD PROBABLY", "MAY WISH TO", "COULD", "POSSIBLE", and "MIGHT" in this document are to be interpreted as described in [RFC6919].

Field definitions are given in Augmented Backus-Naur Form (ABNF), as defined in [RFC5234].

2. The "sni" Alt-Svc Extension

When an origin wishes to nominate a "fronting server", it includes the "sni" parameter in its alternative service entry.

Syntax:
sni = ( reg-name / empty-string )
empty-string = DQUOTE DQUOTE

"reg-name" is defined in Section 3.2.2 of [RFC3986].

When processing such an alternative, clients SHOULD present the hostname given in the "sni" parameter in the SNI extension during the TLS handshake. If the hostname given is an empty string, clients SHOULD omit the SNI extension from the TLS handshake. The server MUST return a valid certificate which covers at least one of the following:

- The hostname indicated in the SNI extension
- The hostname of the origin that published the alternative
- The hostname used for connecting to the alternative

The client MUST validate the certificate in the handshake for authenticity according to [RFC2818] and ensure that it is valid for at least one of these names. Clients SHOULD NOT accept certificates issued to the IP address of the alternative unless the alternative is specified as an IP literal.

If the certificate is not valid for the origin’s hostname, the client MUST NOT make requests to any origin corresponding to this certificate. In this case, the client SHOULD send a "CERTIFICATE_REQUEST" frame including an SNI extension indicating the origin which published the alternative service immediately upon connecting. If no corresponding "CERTIFICATE" frame is presented by the server after a reasonable timeout, or if the server’s SETTINGS frame does not include the "SETTINGS_HTTP_CERT_AUTH" setting, the client MUST consider the alternative connection to have failed.

3. Examples

3.1. SNI of Colocated Domain

Suppose a client has received the following Alt-Svc entry for sensitive.example.com in the past:

h2="innocence.org:443";ma=2635200;persist=true;sni=innocence.org

If the client now wishes to make a request to https://sensitive.example.com/private, it would perform a DNS resolution for innocence.org. The client would then open a TCP connection to the resulting IP address and begin a TLS handshake.
In the client’s TLS handshake, it would request a certificate for the hostname "innocence.org". The TLS server would present such a certificate, issued by an authority trusted by the client. The client will validate the certificate for the name "sensitive.example.com". When validation fails, the client will try to validate the certificate for the name "innocence.org", which will succeed. After validation succeeds, the client will send a "CERTIFICATE_REQUEST" frame asking that the server also authenticate with a certificate for sensitive.example.com.

After receiving the "CERTIFICATE" frame proving possession of a certificate for sensitive.example.com, the client will verify that this certificate is trusted. If so, the client will proceed to send HTTP/2 requests to the server requesting the resource https://sensitive.example.com/private.

3.2. Wildcard Subdomains

Suppose a client has received the following Alt-Svc entry for sensitive.example.com in the past:

h2="www.example.com:443";ma=2635200;persist=true;sni=www.example.com

If the client now wishes to make a request to https://sensitive.example.com/private, it would perform a DNS resolution for www.example.com, the specified alternative. The client would then open a TCP connection to the resulting IP address and begin a TLS handshake.

In the client’s TLS handshake, it would request a certificate for the hostname www.example.com. The TLS server would present a certificate which included www.example.com as one of the covered hostnames.

Suppose that the certificate with which the server authenticated also contained a Subject Alternative Name of "*.example.com". Because the certificate covers the desired origin, the client would perform validity checks on this certificate.

If the certificate is trusted, the client will proceed to send HTTP/2 requests to the server requesting the resource https://sensitive.example.com/private.

3.3. Omitting SNI

Suppose a client has received the following Alt-Svc entry for sensitive.example.com in the past:

h2="alternative.example.com:443";ma=2635200;persist=true;sni="
If the client now wishes to make a request to https://sensitive.example.com/private, it would perform a DNS resolution for alternative.example.com, the specified alternative. The client would then open a TCP connection to the resulting IP address and begin a TLS handshake.

In the client’s TLS handshake, it would omit the Server Name Indication extension. The TLS server would present a certificate according to its configured defaults.

The server would supply a certificate that covers sensitive.example.com, for example because it contains a Subject Alternative Name of "*.example.com", and the client would perform validity checks on this certificate.

If the supplied certificate does not cover sensitive.example.com, or is not valid, the client will terminate the connection.

3.4. SNI of Unrelated Domain

Suppose a client has received the following Alt-Svc entry for sensitive.example.com in the past:

h2=":443";ma=2635200;persist=true;sni=other.example

If the client now wishes to make a request to https://sensitive.example.com/private, it would perform a DNS resolution for sensitive.example.com (the Alt-Svc entry does not specify a different hostname). The client would then open a TCP connection to the resulting IP address and begin a TLS handshake.

In the client’s TLS handshake, it would request a certificate for the hostname "other.example". The TLS server does not have a certificate for this hostname, but it would return a certificate for sensitive.example.com, issued by an authority trusted by the client, and the client will successfully validate the certificate for the name "sensitive.example.com".

Note that an active attacker could identify this server by sending a Client Hello with the same SNI value and observing the certificate the server uses to authenticate. The server could mitigate this by authenticating with a certificate for other.example.

4. Security Considerations

[AltSvc] permits clients to ignore unrecognized parameters. As a result, servers publishing records with the "sni" parameter cannot be assured that clients will not include their origin in the SNI header.
when connecting to the nominated alternative. If, for security reasons, an origin wishes its identity never to be disclosed when the alternative is being used, an alternative mechanism would be required to ascertain client support before generating the Alt-Svc record.

Clients will need to connect directly to the origin at least once in order to receive the Alt-Svc entry via an HTTP header or "ALTSVC" frame, thus disclosing their use of the origin to the network on the first connection. This could be mitigated by future work defining a way to publish alternative services in a mechanism which can be retrieved confidentially, such as via DNS in combination with [RFC7858] or [DoH].

However, servers which publish Alt-Svc records over unencrypted channels (HTTP connections without TLS) or channels without client authorization (DNS, or publicly accessible HTTP resources) enable active observers to build a map of fronting servers by collecting Alt-Svc advertisements. Servers SHOULD CONSIDER this trade-off in deciding when and how to make Alt-Svc records available to unauthenticated parties.

While concealing information from passive observers is beneficial, low-effort active attacks still exist. If an attacker can collect the actual server identity by sending a Client Hello with the same SNI value, the usefulness of this technique is limited. Server deployments SHOULD reserve sensitive domains for use with Secondary Certificates or conceal them inside wildcards in order to mitigate this.

5. IANA Considerations

The "Hypertext Transfer Protocol (HTTP) Alt-Svc Parameter Registry" defines the name space for parameters, as described in [AltSvc]. It is maintained at http://www.iana.org/assignments/http-alt-svc-parameters [1].

This document registers the following parameter:

Name: "sni"

Specification: This document

6. References
6.1. Normative References


Bishop Expires November 25, 2018 [Page 8]
6.2. Informative References


6.3. URIs


Appendix A. Acknowledgements

Conversations with Benjamin Schwartz helped to flesh out this idea.

Author’s Address

Mike Bishop
Akamai

Email: mbishop@evequefou.be
Cache Digests for HTTP/2
draft-ietf-httpbis-cache-digest-05

Abstract

This specification defines a HTTP/2 frame type to allow clients to inform the server of their cache’s contents. Servers can then use this to inform their choices of what to push to clients.

Note to Readers

Discussion of this draft takes place on the HTTP working group mailing list (ietf-http-wg@w3.org), which is archived at https://lists.w3.org/Archives/Public/ietf-http-wg/.

Working Group information can be found at http://httpwg.github.io/; source code and issues list for this draft can be found at https://github.com/httpwg/http-extensions/labels/cache-digest.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on January 3, 2019.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.
1. Introduction .......................... 2
   1.1. Notational Conventions .......... 3
2. The CACHE_DIGEST Frame .............. 3
   2.1. Client Behavior ................. 4
       2.1.1. Creating a digest .......... 4
       2.1.2. Adding a URL to the Digest-Value 5
       2.1.3. Removing a URL to the Digest-Value 7
       2.1.4. Computing a fingerprint value ... 8
       2.1.5. Computing the key ........... 9
       2.1.6. Computing a Hash Value ....... 9
       2.1.7. Computing an Alternative Hash Value 9
       2.2. Server Behavior ............... 10
           2.2.1. Querying the Digest for a Value 10
3. The SETTINGS_SENDING_CACHE_DIGEST SETTINGS Parameter ....... 11
4. The SETTINGS_ACCEPT_CACHE_DIGEST SETTINGS Parameter ....... 12
5. IANA Considerations .................. 12
6. Security Considerations .............. 13
7. References ........................... 13
   7.1. Normative References ............ 13
   7.2. Informative References .......... 14
Appendix A. Encoding the CACHE_DIGEST frame as an HTTP Header . 15
Appendix B. Changes ........................ 16
   B.1. Since draft-ietf-httpbis-cache-digest-04 .......... 16
   B.2. Since draft-ietf-httpbis-cache-digest-03 .......... 16
   B.3. Since draft-ietf-httpbis-cache-digest-02 .......... 16
   B.4. Since draft-ietf-httpbis-cache-digest-01 .......... 16
   B.5. Since draft-ietf-httpbis-cache-digest-00 .......... 17
Appendix C. Acknowledgements ............ 17
Authors’ Addresses ........................ 17

1. Introduction

HTTP/2 [RFC7540] allows a server to "push" synthetic request/response pairs into a client’s cache optimistically. While there is strong interest in using this facility to improve perceived Web browsing
performance, it is sometimes counterproductive because the client
might already have cached the "pushed" response.

When this is the case, the bandwidth used to "push" the response is
effectively wasted, and represents opportunity cost, because it could
be used by other, more relevant responses. HTTP/2 allows a stream to
be cancelled by a client using a RST_STREAM frame in this situation,
but there is still at least one round trip of potentially wasted
capacity even then.

This specification defines a HTTP/2 frame type to allow clients to
inform the server of their freshly cached contents using a Cuckoo-
filter [Cuckoo] based digest. Servers can then use this to inform
their choices of what to push to clients.

1.1. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
"SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this
document are to be interpreted as described in [RFC2119].

2. The CACHE_DIGEST Frame

The CACHE_DIGEST frame type is 0xd (decimal 13).

+-------------------------------+-------------------------------+
|         Origin-Len (16)       | Origin? (\*)                ...
+-------------------------------+-------------------------------+
|                   Digest-Value? (\*)                        ...
+---------------------------------------------------------------+

The CACHE_DIGEST frame payload has the following fields:

Origin-Len: An unsigned, 16-bit integer indicating the length, in
octets, of the Origin field.

Origin: A sequence of characters containing the ASCII serialization
of an origin ([RFC6454], Section 6.2) that the Digest-Value
applies to.

Digest-Value: A sequence of octets containing the digest as computed
in Section 2.1.1 and Section 2.1.2.

The CACHE_DIGEST frame defines the following flags:

- *RESET* (0x1): When set, indicates that any and all cache digests
  for the applicable origin held by the recipient MUST be considered
  invalid.
o  *COMPLETE* (0x2): When set, indicates that the currently valid set of cache digests held by the server constitutes a complete representation of the cache’s state regarding that origin.

2.1. Client Behavior

A CACHE_DIGEST frame MUST be sent from a client to a server on stream 0, and conveys a digest of the contents of the client’s cache for the indicated origin.

In typical use, a client will send one or more CACHE_DIGESTs immediately after the first request on a connection for a given origin, on the same stream, because there is usually a short period of inactivity then, and servers can benefit most when they understand the state of the cache before they begin pushing associated assets (e.g., CSS, JavaScript and images). Clients MAY send CACHE_DIGEST at other times.

If the cache’s state is cleared, lost, or the client otherwise wishes the server to stop using previously sent CACHE_DIGESTs, it can send a CACHE_DIGEST with the RESET flag set.

When generating CACHE_DIGEST, a client MUST NOT include stale-cached responses or responses whose URLs do not share origins [RFC6454] with the indicated origin. Clients MUST NOT send CACHE_DIGEST frames on connections that are not authoritative (as defined in [RFC7540], 10.1) for the indicated origin.

When the CACHE_DIGEST frames sent represent the complete set of stored responses, the last such frame SHOULD have a COMPLETE flag set, to indicate to the server that it has all relevant state. Note that for the purposes of COMPLETE, responses cached since the beginning of the connection or the last RESET flag on a CACHE_DIGEST frame need not be included.

CACHE_DIGEST has no defined meaning when sent from servers, and SHOULD be ignored by clients.

2.1.1. Creating a digest

Given the following inputs:

- "P", an integer smaller than 256, that indicates the probability of a false positive that is acceptable, expressed as "1/2**\*P".

- "N", an integer that represents the number of entries - a prime number smaller than 2**32
1. Let "f" be the number of bits per fingerprint, calculated as "P + 3"

2. Let "b" be the bucket size, defined as 4.

3. Let "allocated" be the closest power of 2 that is larger than "N".

4. Let "bytes" be "f"*"allocated"*"b"/8 rounded up to the nearest integer.

5. Add 5 to "bytes".

6. Allocate memory of "bytes" and set it to zero. Assign it to "digest-value".

7. Set the first byte to "P".

8. Set the second till fifth bytes to "N" in big endian form.

9. Return the "digest-value".

Note: "allocated" is necessary due to the nature of the way Cuckoo filters are creating the secondary hash, by XORing the initial hash and the fingerprint’s hash. The XOR operation means that secondary hash can pick an entry beyond the initial number of entries, up to the next power of 2. In order to avoid issues there, we allocate the table appropriately. For increased space efficiency, it is recommended that implementations pick a number of entries that’s close to the next power of 2.

2.1.2. Adding a URL to the Digest-Value

Given the following inputs:

- "URL" a string corresponding to the Effective Request URI ([RFC7230], Section 5.5) of a cached response [RFC7234]
- "maxcount" - max number of cuckoo hops
- "digest-value"

1. Let "f" be the value of the first byte of "digest-value".

2. Let "b" be the bucket size, defined as 4.

3. Let "N" be the value of the second to fifth bytes of "digest-value" in big endian form.
4. Let "key" be the return value of Section 2.1.5 with "URL" as input.

5. Let "h1" be the return value of Section 2.1.6 with "key" and "N" as inputs.

6. Let "dest_fingerprint" be the return value of Section 2.1.4 with "key" and "f" as inputs.

7. Let "h2" be the return value of Section 2.1.7 with "h1", "dest_fingerprint" and "N" as inputs.

8. Let "h" be either "h1" or "h2", picked in random.

9. While "maxcount" is larger than zero:
   1. Let "position_start" be 40 + "h" * "f" * "b".
   2. Let "position_end" be "position_start" + "f" * "b".
   3. While "position_start" < "position_end":
      1. Let "bits" be "f" bits from "digest_value" starting at "position_start".
      2. If "bits" is all zeros, set "bits" to "dest_fingerprint" and terminate these steps.
      3. Add "f" to "position_start".
   4. Let "e" be a random number from 0 to "b".
   5. Subtract "f" * ("b" - "e") from "position_start".
   6. Let "bits" be "f" bits from "digest_value" starting at "position_start".
   7. Let "fingerprint" be the value of bits, read as big endian.
   8. Set "bits" to "dest_fingerprint".
   9. Set "dest_fingerprint" to "fingerprint".
   10. Let "h" be Section 2.1.7 with "h", "dest_fingerprint" and "N" as inputs.
   11. Subtract 1 from "maxcount".
10. Subtract "f" from "position_start".

11. Let "fingerprint" be the "f" bits starting at "position_start".

12. Let "h1" be "h"

13. Subtract 1 from "maxcount".

14. If "maxcount" is zero, return an error.

15. Go to step 7.

2.1.3. Removing a URL to the Digest-Value

Given the following inputs:

  o "URL" a string corresponding to the Effective Request URI ([RFC7230], Section 5.5) of a cached response [RFC7234]

  o "digest-value"

1. Let "f" be the value of the first byte of "digest-value".

2. Let "b" be the bucket size, defined as 4.

3. Let "N" be the value of the second to fifth bytes of "digest-value" in big endian form.

4. Let "key" be the return value of Section 2.1.5 with "URL" as input.

5. Let "h1" be the return value of Section 2.1.6 with "key" and "N" as inputs.

6. Let "fingerprint" be the return value of Section 2.1.4 with "key" and "f" as inputs.

7. Let "h2" be the return value of Section 2.1.7 with "h1", "fingerprint" and "N" as inputs.

8. Let "hashes" be an array containing "h1" and "h2".

9. For each "h" in "hashes":
   1. Let "position_start" be 40 + "h" * "f" * "b".
   2. Let "position_end" be "position_start" + "f" * "b".
3. While "position_start" < "position_end":
   1. Let "bits" be "f" bits from "digest_value" starting at "position_start".
   2. If "bits" is "fingerprint", set "bits" to all zeros and terminate these steps.
   3. Add "f" to "position_start".

2.1.4. Computing a fingerprint value

Given the following inputs:
   o "key", an array of characters
   o "f", an integer indicating the number of output bits

1. Let "hash-value" be the SHA-256 message digest [RFC6234] of "key", expressed as an integer.
2. Let "h" be the number of bits in "hash-value"
3. Let "fingerprint-value" be 0
4. While "fingerprint-value" is 0 and "h" > "f":
   1. Let "fingerprint-value" be the "f" least significant bits of "hash-value".
   2. Let "hash-value" be the "h"-"f" most significant bits of "hash-value".
   3. Subtract "f" from "h".
5. If "fingerprint-value" is 0, let "fingerprint-value" be 1.
6. Return "fingerprint-value".

Note: Step 5 is to handle the extremely unlikely case where a SHA-256 digest of "key" is all zeros. The implications of it means that there’s an infinitesimally larger probability of getting a "fingerprint-value" of 1 compared to all other values. This is not a problem for any practical purpose.
2.1.5. Computing the key

Given the following inputs:
  o "URL", an array of characters

1. Let "key" be "URL" converted to an ASCII string by percent-encoding as appropriate [RFC3986].
2. Return "key"

2.1.6. Computing a Hash Value

Given the following inputs:
  o "key", an array of characters.
  o "N", an integer

"hash-value" can be computed using the following algorithm:

1. Let "hash-value" be the SHA-256 message digest [RFC6234] of "key", truncated to 32 bits, expressed as an integer.
2. Return "hash-value" modulo N.

2.1.7. Computing an Alternative Hash Value

Given the following inputs:
  o "hash1", an integer indicating the previous hash.
  o "fingerprint", an integer indicating the fingerprint value.
  o "N", an integer indicating the number of entries in the digest.

1. Let "fingerprint-string" be the value of "fingerprint" in base 10, expressed as a string.
2. Let "hash2" be the return value of Section 2.1.6 with "fingerprint-string" and "N" as inputs, XORed with "hash1".
3. Return "hash2".
2.2. Server Behavior

In typical use, a server will query (as per Section 2.2.1) the CACHE_DIGESTs received on a given connection to inform what it pushes to that client;

- If a given URL has a match in a current CACHE_DIGEST, a complete response need not be pushed; The server MAY push a 304 response for that resource, indicating the client that it hasn’t changed.

- If a given URL has no match in any current CACHE_DIGEST, the client does not have a cached copy, and a complete response can be pushed.

Servers MAY use all CACHE_DIGESTs received for a given origin as current, as long as they do not have the RESET flag set; a CACHE_DIGEST frame with the RESET flag set MUST clear any previously stored CACHE_DIGESTs for its origin. Servers MUST treat an empty Digest-Value with a RESET flag set as effectively clearing all stored digests for that origin.

Clients are not likely to send updates to CACHE_DIGEST over the lifetime of a connection; it is expected that servers will separately track what cacheable responses have been sent previously on the same connection, using that knowledge in conjunction with that provided by CACHE_DIGEST.

Servers MUST ignore CACHE_DIGEST frames sent on a stream other than 0.

2.2.1. Querying the Digest for a Value

Given the following inputs:

- "URL" a string corresponding to the Effective Request URI ([RFC7230], Section 5.5) of a cached response [RFC7234].

- "digest-value", an array of bits.

1. Let "f" be the value of the first byte of "digest-value".

2. Let "b" be the bucket size, defined as 4.

3. Let "N" be the value of the second to fifth bytes of "digest-value" in big endian form.

4. Let "key" be the return value of Section 2.1.5 with "URL" as input.
5. Let "h1" be the return value of Section 2.1.6 with "key" and "N" as inputs.

6. Let "fingerprint" be the return value of Section 2.1.4 with "key" and "f" as inputs.

7. Let "h2" be the return value of Section 2.1.7 with "h1", "fingerprint" and "N" as inputs.

8. Let "hashes" be an array containing "h1" and "h2".

9. For each "h" in "hashes":
   1. Let "position_start" be 40 + "h" * "f" * "b".
   2. Let "position_end" be "position_start" + "f" * "b".
   3. While "position_start" < "position_end":
      1. Let "bits" be "f" bits from "digest_value" starting at "position_start".
      2. If "bits" is "fingerprint", return true
      3. Add "f" to "position_start".

10. Return false.

3. The SETTINGS_SENDING_CACHE_DIGEST SETTINGS Parameter

   A Client SHOULD notify its support for CACHE_DIGEST frames by sending the SETTINGS_SENDING_CACHE_DIGEST (0xXXX) SETTINGS parameter.

   The value of the parameter is a bit-field of which the following bits are defined:

   DIGEST_PENDING (0x1): When set it indicates that the client has a digest to send, and the server may choose to wait for a digest in order to make server push decisions.

   Rest of the bits MUST be ignored and MUST be left unset when sending.

   The initial value of the parameter is zero (0x0) meaning that the client has no digest to send the server.
4. The SETTINGS_ACCEPT_CACHE_DIGEST SETTINGS Parameter

A server can notify its support for CACHE_DIGEST frame by sending the SETTINGS_ACCEPT_CACHE_DIGEST (0x7) SETTINGS parameter. If the server is tempted to making optimizations based on CACHE_DIGEST frames, it SHOULD send the SETTINGS parameter immediately after the connection is established.

The value of the parameter is a bit-field of which the following bits are defined:

- ACCEPT (0x1): When set, it indicates that the server is willing to make use of a digest of cached responses.

- Rest of the bits MUST be ignored and MUST be left unset when sending.

- The initial value of the parameter is zero (0x0) meaning that the server is not interested in seeing a CACHE_DIGEST frame.

Some underlying transports allow the server’s first flight of application data to reach the client at around the same time when the client sends its first flight data. When such transport (e.g., TLS 1.3 [I-D.ietf-tls-tls13] in full-handshake mode) is used, a client can postpone sending the CACHE_DIGEST frame until it receives a SETTINGS_ACCEPT_CACHE_DIGEST settings value.

- When the underlying transport does not have such property (e.g., TLS 1.3 in 0-RTT mode), a client can reuse the settings value found in previous connections to that origin [RFC6454] to make assumptions.

5. IANA Considerations

This document registers the following entry in the Permanent Message Headers Registry, as per [RFC3864]:

- Header field name: Cache-Digest
- Applicable protocol: http
- Status: experimental
- Author/Change controller: IESG
- Specification document(s): [this document]

This document registers the following entry in the HTTP/2 Frame Type Registry, as per [RFC7540]:

Oku & Weiss
Expires January 3, 2019
This document registers the following entry in the HTTP/2 Settings Registry, as per [RFC7540]:

- **Code**: 0x7
- **Name**: SETTINGS_ACCEPT_CACHE_DIGEST
- **Initial Value**: 0x0
- **Reference**: [this document]

6. Security Considerations

The contents of a User Agent’s cache can be used to re-identify or "fingerprint" the user over time, even when other identifiers (e.g., Cookies [RFC6265]) are cleared.

CACHE_DIGEST allows such cache-based fingerprinting to become passive, since it allows the server to discover the state of the client’s cache without any visible change in server behaviour.

As a result, clients MUST mitigate for this threat when the user attempts to remove identifiers (e.g., "clearing cookies"). This could be achieved in a number of ways; for example: by clearing the cache, by changing one or both of N and P, or by adding new, synthetic entries to the digest to change its contents.

TODO: discuss how effective the suggested mitigations actually would be.

Additionally, User Agents SHOULD NOT send CACHE_DIGEST when in "privacy mode."

7. References

7.1. Normative References

Internet-Draft          Cache Digests for HTTP/2               July 2018


7.2. Informative References


Appendix A. Encoding the CACHE_DIGEST frame as an HTTP Header

On some web browsers that support Service Workers [Service-Workers] but not Cache Digests (yet), it is possible to achieve the benefit of using Cache Digests by emulating the frame using HTTP Headers.

For the sake of interoperability with such clients, this appendix defines how a CACHE_DIGEST frame can be encoded as an HTTP header named "Cache-Digest".

The definition uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234] with the list rule extension defined in [RFC7230], Section 7.

\[
\text{Cache-Digest} = 1#\text{digest-entity} \\
\text{digest-entity} = \text{digest-value} "(\text{OWS ;} \text{ OWS digest-flag}) \\
\text{digest-value} = \text{<Digest-Value encoded using base64url>} \\
\text{digest-flag} = \text{token}
\]

A Cache-Digest request header is defined as a list construct of cache-digest-entities. Each cache-digest-entity corresponds to a CACHE_DIGEST frame.
Digest-Value is encoded using base64url [RFC4648], Section 5. Flags that are set are encoded as digest-flags by their names that are compared case-insensitively.

Origin is omitted in the header form. The value is implied from the value of the ":authority" pseudo header. Client MUST only send Cache-Digest headers containing digests that belong to the origin specified by the HTTP request.

The example below contains a digest of one resource and has only the "COMPLETE" flag set.

Cache-Digest: AfdA; complete

Clients MUST associate Cache-Digest headers to every HTTP request, since Fetch [Fetch] - the HTTP API supported by Service Workers - does not define the order in which the issued requests will be sent to the server nor guarantees that all the requests will be transmitted using a single HTTP/2 connection.

Also, due to the fact that any header that is supplied to Fetch is required to be end-to-end, there is an ambiguity in what a Cache-Digest header represents when a request is transmitted through a proxy. The header may represent the cache state of a client or that of a proxy, depending on how the proxy handles the header.

Appendix B. Changes

B.1. Since draft-ietf-httpbis-cache-digest-04
  o Remove ETag from the digest key calculations.
  o Add SETTINGS_ prefix to parameter names.

B.2. Since draft-ietf-httpbis-cache-digest-03
  o Yoav becomes an author; Mark steps down.

B.3. Since draft-ietf-httpbis-cache-digest-02
  o Switch to Cuckoo Filter.

B.4. Since draft-ietf-httpbis-cache-digest-01
  o Added definition of the Cache-Digest header.
  o Introduce ACCEPT_CACHE_DIGEST SETTINGS parameter.
o Change intended status from Standard to Experimental.

B.5. Since draft-ietf-httpbis-cache-digest-00

o Make the scope of a digest frame explicit and shift to stream 0.

Appendix C. Acknowledgements

+{:numbered="false"}

Thanks to Stefan Eissing for his suggestions.

Authors’ Addresses

Kazuho Oku
Fastly

Email: kazuhooku@gmail.com

Yoav Weiss
Akamai

Email: yoav@yoav.ws
URI: https://blog.yoav.ws/
HTTP Client Hints
draft-ietf-httpbis-client-hints-06

Abstract

An increasing diversity of Web-connected devices and software capabilities has created a need to deliver optimized content for each device.

This specification defines an extensible and configurable set of HTTP request header fields, colloquially known as Client Hints, to address this. They are intended to be used as input to proactive content negotiation; just as the Accept header field allows user agents to indicate what formats they prefer, Client Hints allow user agents to indicate device and agent specific preferences.

Note to Readers

Discussion of this draft takes place on the HTTP working group mailing list (ietf-http-wg@w3.org), which is archived at https://lists.w3.org/Archives/Public/ietf-http-wg/ [1].

Working Group information can be found at http://httpwg.github.io/ [2]; source code and issues list for this draft can be found at https://github.com/httpwg/http-extensions/labels/client-hints [3].

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on January 17, 2019.
Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the
document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal
Provisions Relating to IETF Documents
(https://trustee.ietf.org/license-info) in effect on the date of
publication of this document. Please review these documents
carefully, as they describe your rights and restrictions with respect
to this document. Code Components extracted from this document must
include Simplified BSD License text as described in Section 4.e of
the Trust Legal Provisions and are provided without warranty as
described in the Simplified BSD License.

Table of Contents

1.  Introduction ........................................... 3
   1.1.  Notational Conventions .......................... 4
2.  Client Hint Request Header Fields .......................... 4
   2.1.  Sending Client Hints .............................. 4
   2.2.  Server Processing of Client Hints ................. 4
         2.2.1.  Advertising Support via Accept-CH Header Field ... 5
         2.2.2.  The Accept-CH-Lifetime Header Field ............ 5
         2.2.3.  Interaction with Caches ...................... 6
3.  Client Hints ........................................... 6
   3.1.  The DPR Header Field .............................. 6
       3.1.1.  Confirming Selected DPR ...................... 7
   3.2.  The Width Header Field ............................ 7
   3.3.  The Viewport-Width Header Field .................. 7
4.  Examples ............................................... 8
5.  Security Considerations .................................. 8
6.  IANA Considerations ..................................... 9
   6.1.  Accept-CH ....................................... 9
   6.2.  Accept-CH-Lifetime .............................. 9
   6.3.  Content-DPR ..................................... 10
   6.4.  DPR ........................................ 10
   6.5.  Viewport-Width .................................. 10
   6.6.  Width ......................................... 10
7.  References ............................................. 10
   7.1.  Normative References ............................ 10
   7.2.  Informative References ........................... 12
   7.3.  URIs .......................................... 12
Appendix A.  Interaction with Key Response Header Field .... 12
Appendix B.  Changes ...................................... 12
   B.1.  Since -00 ..................................... 12
   B.2.  Since -01 ..................................... 13
   B.3.  Since -02 ..................................... 13
1. Introduction

There are thousands of different devices accessing the web, each with different device capabilities and preference information. These device capabilities include hardware and software characteristics, as well as dynamic user and client preferences.

One way to infer some of these capabilities is through User-Agent (Section 5.5.3 of [RFC7231]) header field detection against an established database of client signatures. However, this technique requires acquiring such a database, integrating it into the serving path, and keeping it up to date. However, even once this infrastructure is deployed, user agent sniffing has numerous limitations:

- User agent detection cannot reliably identify all static variables
- User agent detection cannot infer any dynamic client preferences
- User agent detection requires an external device database
- User agent detection is not cache friendly

A popular alternative strategy is to use HTTP cookies ([RFC6265]) to communicate some information about the user agent. However, this approach is also not cache friendly, bound by same origin policy, and imposes additional client-side latency by requiring JavaScript execution to create and manage HTTP cookies.

This document defines a set of new request header fields that allow user agent to perform proactive content negotiation (Section 3.4.1 of [RFC7231]) by indicating device and agent specific preferences, through a mechanism similar to the Accept header field which is used to indicate preferred response formats.

Client Hints does not supersede or replace the User-Agent header field. Existing device detection mechanisms can continue to use both mechanisms if necessary. By advertising its capabilities within a request header field, Client Hints allows for cache friendly and proactive content negotiation.
1.1. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

This document uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234] with the list rule extension defined in [RFC7230], Appendix B. It includes by reference the DIGIT rule from [RFC5234] and the OWS and field-name rules from [RFC7230].

2. Client Hint Request Header Fields

A Client Hint request header field is a HTTP header field that is used by HTTP clients to indicate configuration data that can be used by the server to select an appropriate response. Each one conveys client preferences that the server can use to adapt and optimize the response.

2.1. Sending Client Hints

Clients control which Client Hints are sent in requests, based on their default settings, user configuration, and server preferences. The client and server can use an opt-in mechanism outlined below to negotiate which fields should be sent to allow for efficient content adaption, and optionally use additional mechanisms to negotiate delegation policies that control access of third parties to same fields.

Implementers should be aware of the passive fingerprinting implications when implementing support for Client Hints, and follow the considerations outlined in "Security Considerations" section of this document.

2.2. Server Processing of Client Hints

When presented with a request that contains one or more client hint header fields, servers can optimize the response based upon the information in them. When doing so, and if the resource is cacheable, the server MUST also generate a Vary response header field (Section 7.1.4 of [RFC7231]) to indicate which hints can affect the selected response and whether the selected response is appropriate for a later request.

Further, depending on the hint used, the server can generate additional response header fields to convey related values to aid
client processing. For example, this specification defines the "Content-DPR" response header field that needs to be returned by the server when the "DPR" hint is used to select the response.

2.2.1. Advertising Support via Accept-CH Header Field

Servers can advertise support for Client Hints using the Accept-CH header field or an equivalent HTML meta element with http-equiv attribute ([HTML5]).

Accept-CH = #field-name

For example:

Accept-CH: DPR, Width, Viewport-Width

When a client receives an HTTP response advertising support for Client Hints, it should process it as origin ([RFC6454]) opt-in to receive Client Hint header fields advertised in the field-value. The opt-in MUST be delivered over a secure transport.

For example, based on Accept-CH example above, a user agent could append DPR, Width, and Viewport-Width header fields to all same-origin resource requests initiated by the page constructed from the response.

2.2.2. The Accept-CH-Lifetime Header Field

Servers can ask the client to remember the set of Client Hints that the server supports for a specified period of time, to enable delivery of Client Hints on subsequent requests to the server’s origin ([RFC6454]).

Accept-CH-Lifetime = #delta-seconds

When a client receives an HTTP response that contains Accept-CH-Lifetime header field, the field-value indicates that the Accept-CH preference SHOULD be persisted and bound to the origin, and be considered stale after response’s age ([RFC7234], section 4.2) is greater than the specified number of seconds. The preference MUST be delivered over a secure transport, and MUST NOT be persisted for an origin that isn’t HTTPS.

Accept-CH: DPR, Width
Accept-CH: Viewport-Width
Accept-CH-Lifetime: 86400
For example, based on the Accept-CH and Accept-CH-Lifetime example above, which is received in response to a user agent navigating to "https://example.com", and delivered over a secure transport: a user agent SHOULD persist an Accept-CH preference bound to "https://example.com" for up to 86400 seconds (1 day), and use it for user agent navigations to "https://example.com" and any same-origin resource requests initiated by the page constructed from the navigation’s response. This preference SHOULD NOT extend to resource requests initiated to "https://example.com" from other origins.

If Accept-CH-Lifetime occurs in a message more than once, the last value overrides all previous occurrences.

2.2.3. Interaction with Caches

When selecting an optimized response based on one or more Client Hints, and if the resource is cacheable, the server needs to generate a Vary response header field ([RFC7234]) to indicate which hints can affect the selected response and whether the selected response is appropriate for a later request.

Vary: DPR

Above example indicates that the cache key needs to include the DPR header field.

Vary: DPR, Width

Above example indicates that the cache key needs to include the DPR and Width header fields.

3. Client Hints

3.1. The DPR Header Field

The "DPR" request header field is a number that indicates the client’s current Device Pixel Ratio (DPR), which is the ratio of physical pixels over CSS px (Section 5.2 of [CSSVAL]) of the layout viewport (Section 9.1.1 of [CSS2]) on the device.

$$DPR = 1^{DIGIT} \ [ \ . \ ] 1^{DIGIT}$$

If DPR occurs in a message more than once, the last value overrides all previous occurrences.
3.1.1. Confirming Selected DPR

The "Content-DPR" response header field is a number that indicates the ratio between physical pixels over CSS px of the selected image response.

Content-DPR = 1*DIGIT [ "." 1*DIGIT ]

DPR ratio affects the calculation of intrinsic size of image resources on the client - i.e. typically, the client automatically scales the natural size of the image by the DPR ratio to derive its display dimensions. As a result, the server MUST explicitly indicate the DPR of the selected image response whenever the DPR hint is used, and the client MUST use the DPR value returned by the server to perform its calculations. In case the server returned Content-DPR value contradicts previous client-side DPR indication, the server returned value MUST take precedence.

Note that DPR confirmation is only required for image responses, and the server does not need to confirm the resource width as this value can be derived from the resource itself once it is decoded by the client.

If Content-DPR occurs in a message more than once, the last value overrides all previous occurrences.

3.2. The Width Header Field

The "Width" request header field is a number that indicates the desired resource width in physical px (i.e. intrinsic size of an image). The provided physical px value is a number rounded to the smallest following integer (i.e. ceiling value).

Width = 1*DIGIT

If the desired resource width is not known at the time of the request or the resource does not have a display width, the Width header field can be omitted. If Width occurs in a message more than once, the last value overrides all previous occurrences.

3.3. The Viewport-Width Header Field

The "Viewport-Width" request header field is a number that indicates the layout viewport width in CSS px. The provided CSS px value is a number rounded to the smallest following integer (i.e. ceiling value).

Viewport-Width = 1*DIGIT

Grigorik                Expires January 17, 2019                [Page 7]
If Viewport-Width occurs in a message more than once, the last value overrides all previous occurrences.

4. Examples

For example, given the following request header fields:

```
DPR: 2.0
Width: 320
Viewport-Width: 320
```

The server knows that the device pixel ratio is 2.0, that the intended display width of the requested resource is 160 CSS px (320 physical pixels at 2x resolution), and that the viewport width is 320 CSS px.

If the server uses above hints to perform resource selection for an image asset, it must confirm its selection via the Content-DPR response header to allow the client to calculate the appropriate intrinsic size of the image response. The server does not need to confirm resource width, only the ratio between physical pixels and CSS px of the selected image resource:

```
Content-DPR: 1.0
```

The Content-DPR response header field indicates to the client that the server has selected resource with DPR ratio of 1.0. The client can use this information to perform additional processing on the resource - for example, calculate the appropriate intrinsic size of the image resource such that it is displayed at the correct resolution.

5. Security Considerations

The request header fields defined in this specification, and those that extend it, expose information about the user’s environment to enable proactive content negotiation. Such information may reveal new information about the user and implementers ought to consider the following considerations, recommendations, and best practices.

Transmitted Client Hints header fields SHOULD NOT provide new information that is otherwise not available to the application via other means, such as using HTML, CSS, or JavaScript. Further, sending highly granular data, such as image and viewport width may help identify users across multiple requests. Reducing the set of field values that can be expressed, or restricting them to an enumerated range where the advertised value is close but is not an exact representation of the current value, can improve privacy and
reduce risk of linkability by ensuring that the same value is sent by multiple users. However, such precautions can still be insufficient for some types of data, especially data that can change over time.

Implementers ought to consider both user and server controlled mechanisms and policies to control which Client Hints header fields are advertised:

- Implementers SHOULD restrict delivery of some or all Client Hints header fields to the opt-in origin only, unless the opt-in origin has explicitly delegated permission to another origin to request Client Hints header fields.
- Implementers MAY provide user choice mechanisms so that users may balance privacy concerns with bandwidth limitations. However, implementers should also be aware that explaining the privacy implications of passive fingerprinting to users may be challenging.
- Implementations specific to certain use cases or threat models MAY avoid transmitting some or all of Client Hints header fields. For example, avoid transmission of header fields that can carry higher risks of linkability.

Implementers SHOULD support Client Hints opt-in mechanisms and MUST clear persisted opt-in preferences when site data, browsing history, browsing cache, or similar, are cleared.

6. IANA Considerations


6.1. Accept-CH

- Header field name: Accept-CH
- Applicable protocol: HTTP
- Status: standard
- Author/Change controller: IETF
- Specification document(s): Section 2.2.1 of this document
- Related information: for Client Hints

6.2. Accept-CH-Lifetime

- Header field name: Accept-CH-Lifetime
- Applicable protocol: HTTP
- Status: standard
- Author/Change controller: IETF
6.3. Content-DPR

- Header field name: Content-DPR
- Applicable protocol: HTTP
- Status: standard
- Author/Change controller: IETF
- Specification document(s): Section 3.1.1 of this document
- Related information: for Client Hints

6.4. DPR

- Header field name: DPR
- Applicable protocol: HTTP
- Status: standard
- Author/Change controller: IETF
- Specification document(s): Section 3.1 of this document
- Related information: for Client Hints

6.5. Viewport-Width

- Header field name: Viewport-Width
- Applicable protocol: HTTP
- Status: standard
- Author/Change controller: IETF
- Specification document(s): Section 3.3 of this document
- Related information: for Client Hints

6.6. Width

- Header field name: Width
- Applicable protocol: HTTP
- Status: standard
- Author/Change controller: IETF
- Specification document(s): Section 3.2 of this document
- Related information: for Client Hints

7. References

7.1. Normative References


7.2. Informative References


7.3. URIs

[1]  https://lists.w3.org/Archives/Public/ietf-http-wg/


Appendix A. Interaction with Key Response Header Field

Client Hints may be combined with Key response header field ([KEY]) to enable fine-grained control of the cache key for improved cache efficiency. For example, the server can return the following set of instructions:

Key: DPR;partition=1.5:2.5:4.0

Above example indicates that the cache key needs to include the value of the DPR header field with three segments: less than 1.5, 1.5 to less than 2.5, and 4.0 or greater.

Key: Width;div=320

Above example indicates that the cache key needs to include the value of the Width header field and be partitioned into groups of 320: 0-320, 320-640, and so on.

Appendix B. Changes

B.1. Since -00

- Issue 168 (make Save-Data extensible) updated ABNF.
- Issue 163 (CH review feedback) editorial feedback from httpwg list.
- Issue 153 (NetInfo API citation) added normative reference.
B.2. Since -01
   - Issue 200: Moved Key reference to informative.
   - Issue 215: Extended passive fingerprinting and mitigation considerations.
   - Changed document status to experimental.

B.3. Since -02
   - Issue 239: Updated reference to CR-css-values-3
   - Issue 240: Updated reference for Network Information API
   - Issue 241: Consistency in IANA considerations
   - Issue 250: Clarified Accept-CH

B.4. Since -03
   - Issue 284: Extended guidance for Accept-CH
   - Issue 308: Editorial cleanup
   - Issue 306: Define Accept-CH-Lifetime

B.5. Since -04
   - Issue 361: Removed Downlink
   - Issue 361: Moved Key to appendix, plus other editorial feedback

B.6. Since -05
   - Issue 372: Scoped CH opt-in and delivery to secure transports
   - Issue 373: Bind CH opt-in to origin

B.7. Since -06
   - Issue 524: Save-Data is now defined by NetInfo spec, dropping

Acknowledgements

Thanks to Mark Nottingham, Julian Reschke, Chris Bentzel, Yoav Weiss, Ben Greenstein, Tarun Bansal, Roy Fielding, Vasilii Faronov, Ted Hardie, Jonas Sicking, and numerous other members of the IETF HTTP Working Group for invaluable help and feedback.

Author’s Address

Ilya Grigorik
Google

Email: ilya@igvita.com
URI: https://www.igvita.com/
Expect-CT Extension for HTTP
draft-ietf-httpbis-expect-ct-07

Abstract

This document defines a new HTTP header field, named Expect-CT, that allows web host operators to instruct user agents to expect valid Signed Certificate Timestamps (SCTs) to be served on connections to these hosts. Expect-CT allows web host operators to discover misconfigurations in their Certificate Transparency deployments and ensure that misissued certificates accepted by UAs are discoverable in Certificate Transparency logs.

Note to Readers

Discussion of this draft takes place on the HTTP working group mailing list (ietf-http-wg@w3.org), which is archived at https://lists.w3.org/Archives/Public/ietf-http-wg/ [1].

Working Group information can be found at http://httpwg.github.io/ [2]; source code and issues list for this draft can be found at https://github.com/httpwg/http-extensions/labels/expect-ct [3].

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on January 17, 2019.
1.  Introduction ......................................................... 3
  1.1.  Requirements Language ........................................ 4
  1.2.  Terminology ..................................................... 4
2.  Server and Client Behavior ........................................ 5
  2.1.  Response Header Field Syntax .................................. 5
        2.1.1.  The report-uri Directive ............................... 6
        2.1.2.  The enforce Directive .................................. 6
        2.1.3.  The max-age Directive .................................. 7
        2.1.4.  Examples ................................................ 7
  2.2.  Server Processing Model ........................................ 7
        2.2.1.  HTTP-over-Secure-Transport Request Type ............... 8
        2.2.2.  HTTP Request Type ..................................... 8
  2.3.  User Agent Processing Model ................................... 8
        2.3.1.  Missing or Malformed Expect-CT Header Fields ........... 8
        2.3.2.  Expect-CT Header Field Processing ...................... 8
        2.3.3.  Reporting ............................................... 10
  2.4.  Evaluating Expect-CT Connections for CT Compliance .......... 10
        2.4.1.  Skipping CT compliance checks .......................... 11
3.  Reporting Expect-CT Failure ........................................ 11
  3.1.  Generating a violation report .................................. 12
  3.2.  Sending a violation report .................................... 13
  3.3.  Receiving a violation report .................................. 14
4.  Security Considerations ............................................ 14
  4.1.  Maximum max-age .............................................. 15
  4.2.  Avoiding amplification attacks ................................ 15
5.  Privacy Considerations ............................................. 15
6.  IANA Considerations ............................................... 16
  6.1.  Header Field Registry ......................................... 16
  6.2.  Media Types Registry .......................................... 16
7.  Usability Considerations ........................................... 17
8.  Authoring Considerations .......................................... 17
1. Introduction

This document defines a new HTTP header field that enables UAs to identify web hosts that expect the presence of Signed Certificate Timestamps (SCTs) [I-D.ietf-trans-rfc6962-bis] in future Transport Layer Security (TLS) [RFC5246] connections.

Web hosts that serve the Expect-CT HTTP header field are noted by the UA as Known Expect-CT Hosts. The UA evaluates each connection to a Known Expect-CT Host for compliance with the UA’s Certificate Transparency (CT) Policy. If the connection violates the CT Policy, the UA sends a report to a URI configured by the Expect-CT Host and/or fails the connection, depending on the configuration that the Expect-CT Host has chosen.

If misconfigured, Expect-CT can cause unwanted connection failures (for example, if a host deploys Expect-CT but then switches to a legitimate certificate that is not logged in Certificate Transparency logs, or if a web host operator believes their certificate to conform to all UAs’ CT policies but is mistaken). Web host operators are advised to deploy Expect-CT with caution, by using the reporting feature and gradually increasing the interval where the UA remembers the host as a Known Expect-CT Host. These precautions can help web host operators gain confidence that their Expect-CT deployment is not causing unwanted connection failures.

Expect-CT is a trust-on-first-use (TOFU) mechanism. The first time a UA connects to a host, it lacks the information necessary to require SCTs for the connection. Thus, the UA will not be able to detect and thwart an attack on the UA’s first connection to the host. Still, Expect-CT provides value by 1) allowing UAs to detect the use of unlogged certificates after the initial communication, and 2) allowing web hosts to be confident that UAs are only trusting publicly-auditable certificates.
1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

1.2. Terminology

Terminology is defined in this section.

- "Certificate Transparency Policy" is a policy defined by the UA concerning the number, sources, and delivery mechanisms of Signed Certificate Timestamps that are served on TLS connections. The policy defines the properties of a connection that must be met in order for the UA to consider it CT-qualified.

- "Certificate Transparency Qualified" describes a TLS connection for which the UA has determined that a sufficient quantity and quality of Signed Certificate Timestamps have been provided.

- "CT-qualified" is an abbreviation for "Certificate Transparency Qualified".

- "CT Policy" is an abbreviation for "Certificate Transparency Policy".

- "Effective Expect-CT Date" is the time at which a UA observed a valid Expect-CT header field for a given host.

- "Expect-CT Host" is a conformant host implementing the HTTP server aspects of Expect-CT. This means that an Expect-CT Host returns the "Expect-CT" HTTP response header field in its HTTP response messages sent over secure transport. The term "host" is equivalent to "server" in this specification.

- "Known Expect-CT Host" is an Expect-CT Host that the UA has noted as such. See Section 2.3.2.1 for particulars.

- UA is an acronym for "user agent". For the purposes of this specification, a UA is an HTTP client application typically actively manipulated by a user [RFC7230].

- "Unknown Expect-CT Host" is an Expect-CT Host that the UA has not noted.
2. Server and Client Behavior

2.1. Response Header Field Syntax

The "Expect-CT" response header field is a new field defined in this specification. It is used by a server to indicate that UAs should evaluate connections to the host emitting the header field for CT compliance (Section 2.4).

Figure 1 describes the syntax (Augmented Backus-Naur Form) of the header field, using the grammar defined in [RFC5234] and the rules defined in Section 3.2 of [RFC7230].

```
Expect-CT           = #expect-ct-directive
expect-ct-directive = directive-name [ "=" directive-value ]
directive-name      = token
directive-value     = token / quoted-string
```

Figure 1: Syntax of the Expect-CT header field

Optional white space ("OWS") is used as defined in Section 3.2.3 of [RFC7230]. "token" and "quoted-string" are used as defined in Section 3.2.6 of [RFC7230].

The directives defined in this specification are described below. The overall requirements for directives are:

1. The order of appearance of directives is not significant.

2. A given directive MUST NOT appear more than once in a given header field. Directives are either optional or required, as stipulated in their definitions.

3. Directive names are case insensitive.

4. UAs MUST ignore any header fields containing directives, or other header field value data, that do not conform to the syntax defined in this specification. In particular, UAs must not attempt to fix malformed header fields.

5. If a header field contains any directive(s) the UA does not recognize, the UA MUST ignore those directives.

6. If the Expect-CT header field otherwise satisfies the above requirements (1 through 5), the UA MUST process the directives it recognizes.
2.1.1. The report-uri Directive

The OPTIONAL "report-uri" directive indicates the URI to which the UA SHOULD report Expect-CT failures (Section 2.4). The UA POSTs the reports to the given URI as described in Section 3.

The "report-uri" directive is REQUIRED to have a directive value, for which the syntax is defined in Figure 2.

report-uri-value = absolute-URI

Figure 2: Syntax of the report-uri directive value

"absolute-URI" is defined in Section 4.3 of [RFC3986].

Hosts may set "report-uri"s that use HTTP or HTTPS. If the scheme in the "report-uri" is one that uses TLS (e.g., HTTPS), UAs MUST check Expect-CT compliance when the host in the "report-uri" is a Known Expect-CT Host; similarly, UAs MUST apply HSTS if the host in the "report-uri" is a Known HSTS Host.

Note that the report-uri need not necessarily be in the same Internet domain or web origin as the host being reported about.

UAs SHOULD make their best effort to report Expect-CT failures to the "report-uri", but they may fail to report in exceptional conditions. For example, if connecting to the "report-uri" itself incurs an Expect-CT failure or other certificate validation failure, the UA MUST cancel the connection. Similarly, if Expect-CT Host A sets a "report-uri" referring to Expect-CT Host B, and if B sets a "report-uri" referring to A, and if both hosts fail to comply to the UA's CT Policy, the UA SHOULD detect and break the loop by failing to send reports to and about those hosts.

UAs SHOULD limit the rate at which they send reports. For example, it is unnecessary to send the same report to the same "report-uri" more than once.

2.1.2. The enforce Directive

The OPTIONAL "enforce" directive is a valueless directive that, if present (i.e., it is "asserted"), signals to the UA that compliance to the CT Policy should be enforced (rather than report-only) and that the UA should refuse future connections that violate its CT Policy. When both the "enforce" directive and "report-uri" directive (as defined in Figure 2) are present, the configuration is referred to as an "enforce-and-report" configuration, signalling to the UA...
both that compliance to the CT Policy should be enforced and that violations should be reported.

2.1.3. The max-age Directive

The "max-age" directive specifies the number of seconds after the reception of the Expect-CT header field during which the UA SHOULD regard the host from whom the message was received as a Known Expect-CT Host.

The "max-age" directive is REQUIRED to be present within an "Expect-CT" header field. The "max-age" directive is REQUIRED to have a directive value, for which the syntax (after quoted-string unescaping, if necessary) is defined in Figure 3.

max-age-value = delta-seconds
delta-seconds = 1*DIGIT

Figure 3: Syntax of the max-age directive value

"delta-seconds" is used as defined in Section 1.2.1 of [RFC7234].

2.1.4. Examples

The following three examples demonstrate valid Expect-CT response header fields (where the second splits the directives into two field instances):

Expect-CT: max-age=86400, enforce

Expect-CT: max-age=86400,enforce

Expect-CT: report-uri="https://foo.example/report"

Expect-CT: max-age=86400,report-uri="https://foo.example/report"

Figure 4: Examples of valid Expect-CT response header fields

2.2. Server Processing Model

This section describes the processing model that Expect-CT Hosts implement. The model has 2 parts: (1) the processing rules for HTTP request messages received over a secure transport (e.g., authenticated, non-anonymous TLS); and (2) the processing rules for HTTP request messages received over non-secure transports, such as TCP.
2.2.1. HTTP-over-Secure-Transport Request Type

An Expect-CT Host includes an Expect-CT header field in its response. The header field MUST satisfy the grammar specified in Section 2.1.

Establishing a given host as an Expect-CT Host, in the context of a given UA, is accomplished as follows:

1. Over the HTTP protocol running over secure transport, by correctly returning (per this specification) a valid Expect-CT header field to the UA.

2. Through other mechanisms, such as a client-side preloaded Expect-CT Host list.

2.2.2. HTTP Request Type

Expect-CT Hosts SHOULD NOT include the Expect-CT header field in HTTP responses conveyed over non-secure transport. UAs MUST ignore any Expect-CT header field received in an HTTP response conveyed over non-secure transport.

2.3. User Agent Processing Model

The UA processing model relies on parsing domain names. Note that internationalized domain names SHALL be canonicalized according to the scheme in Section 10 of [RFC6797].

The UA stores Known Expect-CT Hosts and their associated Expect-CT directives. This data is collectively known as a host’s ”Expect-CT” metadata”.

2.3.1. Missing or Malformed Expect-CT Header Fields

If an HTTP response does not include an Expect-CT header field that conforms to the grammar specified in Section 2.1, then the UA MUST NOT update any Expect-CT metadata.

2.3.2. Expect-CT Header Field Processing

If the UA receives, over a secure transport, an HTTP response that includes an Expect-CT header field conforming to the grammar specified in Section 2.1, the UA MUST evaluate the connection on which the header field was received for compliance with the UA’s CT Policy, and then process the Expect-CT header field as follows.

If the connection does not comply with the UA’s CT Policy (i.e., the connection is not CT-qualified), then the UA MUST NOT update any
Expect-CT metadata. If the header field includes a "report-uri" directive, the UA SHOULD send a report to the specified "report-uri" (Section 2.3.3).

If the connection complies with the UA’s CT Policy (i.e., the connection is CT-qualified), then the UA MUST either:

- Note the host as a Known Expect-CT Host if it is not already so noted (see Section 2.3.2.1), or
- Update the UA’s cached information for the Known Expect-CT Host if the "enforce", "max-age", or "report-uri" header field value directives convey information different from that already maintained by the UA. If the "max-age" directive has a value of 0, the UA MUST remove its cached Expect-CT information if the host was previously noted as a Known Expect-CT Host, and MUST NOT note this host as a Known Expect-CT Host if it is not already noted.

2.3.2.1. Noting Expect-CT

Upon receipt of the Expect-CT response header field over an error-free TLS connection (including the validation adding in Section 2.4), the UA MUST note the host as a Known Expect-CT Host, storing the host’s domain name and its associated Expect-CT directives in non-volatile storage.

To note a host as a Known Expect-CT Host, the UA MUST set its Expect-CT metadata given in the most recently received valid Expect-CT header field, as specified in Section 2.3.2.2.

For forward compatibility, the UA MUST ignore any unrecognized Expect-CT header field directives, while still processing those directives it does recognize. Section 2.1 specifies the directives "enforce", "max-age", and "report-uri", but future specifications and implementations might use additional directives.

2.3.2.2. Storage Model

If the substring matching the host production from the Request-URI (of the message to which the host responded) does not congruently match an existing Known Expect-CT Host’s domain name, per the matching procedure specified in Section 8.2 of [RFC6797], then the UA MUST add this host to the Known Expect-CT Host cache. The UA caches:

- the Expect-CT Host’s domain name,
- whether the "enforce" directive is present
the Effective Expiration Date, which is the Effective Expect-CT Date plus the value of the "max-age" directive. Alternatively, the UA MAY cache enough information to calculate the Effective Expiration Date. The Effective Expiration Date is calculated from when the UA observed the Expect-CT header field and is independent of when the response was generated.

If any other metadata from optional or future Expect-CT header directives are present in the Expect-CT header field, and the UA understands them, the UA MAY note them as well.

UAs MAY set an upper limit on the value of max-age, so that UAs that have noted erroneous Expect-CT hosts (whether by accident or due to attack) have some chance of recovering over time. If the server sets a max-age greater than the UA’s upper limit, the UA MAY behave as if the server set the max-age to the UA’s upper limit. For example, if the UA caps max-age at 5,184,000 seconds (60 days), and an Expect-CT Host sets a max-age directive of 90 days in its Expect-CT header field, the UA MAY behave as if the max-age were effectively 60 days. (One way to achieve this behavior is for the UA to simply store a value of 60 days instead of the 90-day value provided by the Expect-CT host.)

2.3.3. Reporting

If the UA receives, over a secure transport, an HTTP response that includes an Expect-CT header field with a "report-uri" directive, and the connection does not comply with the UA’s CT Policy (i.e., the connection is not CT-qualified), and the UA has not already sent an Expect-CT report for this connection, then the UA SHOULD send a report to the specified "report-uri" as specified in Section 3.

2.4. Evaluating Expect-CT Connections for CT Compliance

When a UA sets up a TLS connection, the UA determines whether the host is a Known Expect-CT Host according to its Known Expect-CT Host cache. An Expect-CT Host is "expired" if the effective expiration date refers to a date in the past. The UA MUST ignore any expired Expect-CT Hosts in its cache and not treat such hosts as Known Expect-CT hosts.

When a UA connects to a Known Expect-CT Host using a TLS connection, if the TLS connection has no errors, then the UA will apply an additional correctness check: compliance with a CT Policy. A UA should evaluate compliance with its CT Policy whenever connecting to a Known Expect-CT Host, as soon as possible. However, the check can
be skipped for local policy reasons (as discussed in Section 2.4.1),
or in the event that other checks cause the UA to terminate the
connection before CT compliance is evaluated. For example, a Public
Key Pinning failure [RFC7469] could cause the UA to terminate the
connection before CT compliance is checked. Similarly, if the UA
terminates the connection due to an Expect-CT failure, this could
cause the UA to skip subsequent correctness checks. When the CT
compliance check is skipped or bypassed, Expect-CT reports
(Section 3) will not be sent.

When CT compliance is evaluated for a Known Expect-CT Host, the UA
MUST evaluate compliance when setting up the TLS session, before
beginning an HTTP conversation over the TLS channel.

If a connection to a Known Expect-CT Host violates the UA’s CT policy
(i.e., the connection is not CT-qualified), and if the Known Expect-
CT Host’s Expect-CT metadata indicates an "enforce" configuration,
the UA MUST treat the CT compliance failure as an error.

If a connection to a Known Expect-CT Host violates the UA’s CT
policy, and if the Known Expect-CT Host’s Expect-CT metadata includes
a "report-uri", the UA SHOULD send an Expect-CT report to that
"report-uri" (Section 3).

2.4.1. Skipping CT compliance checks

It is acceptable for a UA to skip CT compliance checks for some hosts
according to local policy. For example, a UA may disable CT
compliance checks for hosts whose validated certificate chain
terminates at a user-defined trust anchor, rather than a trust anchor
built-in to the UA (or underlying platform).

If the UA does not evaluate CT compliance, e.g., because the user has
elected to disable it, or because a presented certificate chain
chains up to a user-defined trust anchor, UAs SHOULD NOT send Expect-
CT reports.

3. Reporting Expect-CT Failure

When the UA attempts to connect to a Known Expect-CT Host and the
connection is not CT-qualified, the UA SHOULD report Expect-CT
failures to the "report-uri", if any, in the Known Expect-CT Host’s
Expect-CT metadata.

When the UA receives an Expect-CT response header field over a
connection that is not CT-qualified, if the UA has not already sent
an Expect-CT report for this connection, then the UA SHOULD report
Expect-CT failures to the configured "report-uri", if any.
3.1. Generating a violation report

To generate a violation report object, the UA constructs a JSON [RFC8259] object with the following keys and values:

- **"date-time"**: the value for this key indicates the UTC time that the UA observed the CT compliance failure. The value is a string formatted according to Section 5.6, "Internet Date/Time Format", of [RFC3339].

- **"hostname"**: the value is the hostname to which the UA made the original request that failed the CT compliance check. The value is provided as a string.

- **"port"**: the value is the port to which the UA made the original request that failed the CT compliance check. The value is provided as an integer.

- **"effective-expiration-date"**: the value indicates the Effective Expiration Date (see Section 2.3.2.2) for the Expect-CT Host that failed the CT compliance check, in UTC. The value is provided as a string formatted according to Section 5.6 of [RFC3339] ("Internet Date/Time Format").

- **"served-certificate-chain"**: the value is the certificate chain as served by the Expect-CT Host during TLS session setup. The value is provided as an array of strings, which MUST appear in the order that the certificates were served; each string in the array is the Privacy-Enhanced Mail (PEM) representation of each X.509 certificate as described in [RFC7468].

- **"validated-certificate-chain"**: the value is the certificate chain as constructed by the UA during certificate chain verification. (This may differ from the value of the "served-certificate-chain" key.) The value is provided as an array of strings, which MUST appear in the order matching the chain that the UA validated; each string in the array is the Privacy-Enhanced Mail (PEM) representation of each X.509 certificate as described in [RFC7468].

- **"scts"**: the value represents the SCTs (if any) that the UA received for the Expect-CT host and their validation statuses. The value is provided as an array of JSON objects. The SCTs may appear in any order. Each JSON object in the array has the following keys:

  * A "version" key, with an integer value. The UA MUST set this value to "1" if the SCT is in the format defined in Section 3.2
The "status" key, with a string value that the UA MUST set to one of the following values: "unknown" (indicating that the UA does not have or does not trust the public key of the log from which the SCT was issued), "valid" (indicating that the UA successfully validated the SCT as described in Section 5.2 of [RFC6962] or Section 8.2.3 of [I-D.ietf-trans-rfc6962-bis]), or "invalid" (indicating that the SCT validation failed because of, e.g., a bad signature).

* The "source" key, with a string value that indicates from where the UA obtained the SCT, as defined in Section 3 of [RFC6962] and Section 6 of [I-D.ietf-trans-rfc6962-bis]. The UA MUST set the value to one of "tls-extension", "ocsp", or "embedded".

* The "serialized_sct" key, with a string value. If the value of the "version" key is "1", the UA MUST set this value to the base64 encoded [RFC4648] serialized "SignedCertificateTimestamp" structure from Section 3.2 of [RFC6962]. If the value of the "version" key is "2", the UA MUST set this value to the base64 encoded [RFC4648] serialized "TransItem" structure representing the SCT, as defined in Section 4.5 of [I-D.ietf-trans-rfc6962-bis].

3.2. Sending a violation report

The UA SHOULD report an Expect-CT failure when a connection to a Known Expect-CT Host does not comply with the UA’s CT Policy and the host’s Expect-CT metadata contains a "report-uri". Additionally, the UA SHOULD report an Expect-CT failure when it receives an Expect-CT header field which contains the "report-uri" directive over a connection that does not comply with the UA’s CT Policy.

The steps to report an Expect-CT failure are as follows.
1. Prepare a JSON object "report object" with the single key "expect-ct-report", whose value is the result of generating a violation report object as described in Section 3.1.

2. Let "report body" be the JSON stringification of "report object".

3. Let "report-uri" be the value of the "report-uri" directive in the Expect-CT header field.

4. Send an HTTP POST request to "report-uri" with a "Content-Type" header field of "application/expect-ct-report+json", and an entity body consisting of "report body".

The UA MAY perform other operations as part of sending the HTTP POST request, for example sending a CORS preflight as part of [FETCH].

3.3. Receiving a violation report

Upon receiving an Expect-CT violation report, the report server MUST respond with a 2xx (Successful) status code if it can parse the request body as valid JSON and recognizes the hostname in the "hostname" field of the report. If the report body cannot be parsed or the report server does not expect to receive reports for the hostname in the "hostname" field, the report server MUST respond with a 4xx (Client Error) status code.

If the report’s "test-report" key is set to true, the server MAY discard the report without further processing but MUST still return a 2xx (Successful) status code.

4. Security Considerations

When UAs support the Expect-CT header field, it becomes a potential vector for hostile header attacks against site owners. If a site owner uses a certificate issued by a certificate authority which does not embed SCTs nor serve SCTs via OCSP or TLS extension, a malicious server operator or attacker could temporarily reconfigure the host to comply with the UA’s CT policy, and add the Expect-CT header field in enforcing mode with a long "max-age". Implementing user agents would note this as an Expect-CT Host (see Section 2.3.2.1). After having done this, the configuration could then be reverted to not comply with the CT policy, prompting failures. Note this scenario would require the attacker to have substantial control over the infrastructure in question, being able to obtain different certificates, change server software, or act as a man-in-the-middle in connections.
Site operators could themselves only cure this situation by one of: reconfiguring their web server to transmit SCTs using the TLS extension defined in Section 6.5 of [I-D.ietf-trans-rfc6962-bis], obtaining a certificate from an alternative certificate authority which provides SCTs by one of the other methods, or by waiting for the user agents’ persisted notation of this as an Expect-CT host to reach its "max-age". User agents may choose to implement mechanisms for users to cure this situation, as noted in Section 7.

4.1. Maximum max-age

There is a security trade-off in that low maximum values provide a narrow window of protection for users that visit the Known Expect-CT Host only infrequently, while high maximum values might result in a denial of service to a UA in the event of a hostile header attack, or simply an error on the part of the site-owner.

There is probably no ideal maximum for the "max-age" directive. Since Expect-CT is primarily a policy-expansion and investigation technology rather than an end-user protection, a value on the order of 30 days (2,592,000 seconds) may be considered a balance between these competing security concerns.

4.2. Avoiding amplification attacks

Another kind of hostile header attack uses the "report-uri" mechanism on many hosts not currently exposing SCTs as a method to cause a denial-of-service to the host receiving the reports. If some highly-trafficked websites emitted a non-enforcing Expect-CT header field with a "report-uri", implementing UAs’ reports could flood the reporting host. It is noted in Section 2.1.1 that UAs should limit the rate at which they emit reports, but an attacker may alter the Expect-CT header’s fields to induce UAs to submit different reports to different URIs to still cause the same effect.

5. Privacy Considerations

Expect-CT can be used to infer what Certificate Transparency policy is in use, by attempting to retrieve specially-configured websites which pass one user agents’ policies but not another’s. Note that this consideration is true of UAs which enforce CT policies without Expect-CT as well.

Additionally, reports submitted to the "report-uri" could reveal information to a third party about which webpage is being accessed and by which IP address, by using individual "report-uri" values for individually-tracked pages. This information could be leaked even if client-side scripting were disabled.
Implementations must store state about Known Expect-CT Hosts, and hence which domains the UA has contacted.

Violation reports, as noted in Section 3, contain information about the certificate chain that has violated the CT policy. In some cases, such as organization-wide compromise of the end-to-end security of TLS, this may include information about the interception tools and design used by the organization that the organization would otherwise prefer not be disclosed.

Because Expect-CT causes remotely-detectable behavior, it’s advisable that UAs offer a way for privacy-sensitive users to clear currently noted Expect-CT hosts, and allow users to query the current state of Known Expect-CT Hosts.

6. IANA Considerations

6.1. Header Field Registry

This document registers the "Expect-CT" header field in the "Permanent Message Header Field Names" registry located at https://www.iana.org/assignments/message-headers [4].

Header field name: Expect-CT
Applicable protocol: http
Status: standard
Author/Change controller: IETF
Specification document(s): This document
Related information: (empty)

6.2. Media Types Registry

The MIME media type for Expect-CT violation reports is "application/expect-ct-report+json" (which uses the suffix established in [RFC6839]).

Type name: application
Subtype name: expect-ct-report+json
Required parameters: n/a
Optional parameters: n/a
Encoding considerations: binary

Security considerations: See Section 4

Interoperability considerations: n/a

Published specification: This document

Applications that use this media type: UAs that implement Certificate Transparency compliance checks and reporting

Additional information:

  Deprecated alias names for this type: n/a

  Magic number(s): n/a

  File extension(s): n/a

  Macintosh file type code(s): n/a

Person & email address to contact for further information: Emily Stark (estark@google.com)

Intended usage: COMMON

Restrictions on usage: none

Author: Emily Stark (estark@google.com)

Change controller: IETF

7. Usability Considerations

When the UA detects a Known Expect-CT Host in violation of the UA’s CT Policy, users will experience denials of service. It is advisable for UAs to explain the reason why.

8. Authoring Considerations

Expect-CT could be specified as a TLS extension or X.509 certificate extension instead of an HTTP response header field. Using an HTTP header field as the mechanism for Expect-CT introduces a layering mismatch: for example, the software that terminates TLS and validates Certificate Transparency information might know nothing about HTTP. Nevertheless, an HTTP header field was chosen primarily for ease of deployment. In practice, deploying new certificate extensions requires certificate authorities to support them, and new TLS
extensions require server software updates, including possibly to servers outside of the site owner’s direct control (such as in the case of a third-party CDN). Ease of deployment is a high priority for Expect-CT because it is intended as a temporary transition mechanism for user agents that are transitioning to universal Certificate Transparency requirements.

9. References

9.1. Normative References

[I-D.ietf-trans-rfc6962-bis]


9.2. Informative References


9.3. URIs

[1] https://lists.w3.org/Archives/Public/ietf-http-wg/


Appendix A. Changes

A.1. Since -06
   - Editorial changes

A.2. Since -05
   - Remove SHOULD requirement that UAs disallow certificate error
     overrides for Known Expect-CT Hosts.
   - Remove restriction that Expect-CT Hosts cannot be IP addresses.
   - Editorial changes

A.3. Since -04
   - Editorial changes

A.4. Since -03
   - Editorial changes

A.5. Since -02
   - Add concept of test reports and specify that servers must respond
     with 2xx status codes to valid reports.
   - Add "failure-mode" key to reports to allow report servers to
     distinguish report-only from enforced failures.

A.6. Since -01
   - Change SCT reporting format to support both RFC 6962 and 6962-bis
     SCTs.

A.7. Since -00
   - Editorial changes
   - Change Content-Type header of reports to 'application/expect-ct-
     report+json'
   - Update header field syntax to match convention (issue #327)
   - Reference RFC 6962-bis instead of RFC 6962
Author’s Address

Emily Stark
Google

Email: estark@google.com
Abstract

This document defines a mechanism for running the WebSocket Protocol (RFC 6455) over a single stream of an HTTP/2 connection.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on December 20, 2018.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.
1. Introduction

The Hypertext Transfer Protocol (HTTP) [RFC7230] provides compatible resource-level semantics across different versions but it does not offer compatibility at the connection management level. Other protocols, such as WebSockets, that rely on connection management details of HTTP must be updated for new versions of HTTP.

The WebSocket Protocol [RFC6455] uses the HTTP/1.1 Upgrade mechanism (Section 6.7 of [RFC7230]) to transition a TCP connection from HTTP into a WebSocket connection. A different approach must be taken with HTTP/2 [RFC7540]. HTTP/2 does not allow connection-wide header fields and status codes such as the Upgrade and Connection request header fields or the 101 (Switching Protocols) response code due to its multiplexing nature. These are all required by the [RFC6455] opening handshake.

Being able to bootstrap WebSockets from HTTP/2 allows one TCP connection to be shared by both protocols and extends HTTP/2’s more efficient use of the network to WebSockets.

This document extends the HTTP CONNECT method (as specified for HTTP/2 in Section 8.3 of [RFC7540]). The extension allows the substitution of a new protocol name to connect to rather than the external host normally used by CONNECT. The result is a tunnel on a single HTTP/2 stream that can carry data for WebSockets (or any other protocol). The other streams on the connection may carry more extended CONNECT tunnels, traditional HTTP/2 data, or a mixture of both.
This tunneled stream will be multiplexed with other regular streams on the connection and enjoys the normal priority, cancellation, and flow control features of HTTP/2.

Streams that successfully establish a WebSocket connection using a tunneled stream and the modifications to the opening handshake defined in this document then use the traditional WebSocket Protocol, treating the stream as if were the TCP connection in that specification.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. The SETTINGS_ENABLE_CONNECT_PROTOCOL SETTINGS Parameter

This document adds a new SETTINGS Parameter to those defined by [RFC7540], Section 6.5.2.

The new parameter name is SETTINGS_ENABLE_CONNECT_PROTOCOL. The value of the parameter MUST be 0 or 1.

Upon receipt of SETTINGS_ENABLE_CONNECT_PROTOCOL with a value of 1, a client MAY use the Extended CONNECT definition of this document when creating new streams. Receipt of this parameter by a server does not have any impact.

A sender MUST NOT send a SETTINGS_ENABLE_CONNECT_PROTOCOL parameter with the value of 0 after previously sending a value of 1.

The use of a SETTINGS Parameter to opt-in to an otherwise incompatible protocol change is a use of "Extending HTTP/2" defined by Section 5.5 of [RFC7540]. Specifically, the addition a new pseudo-header field ":protocol" and the change in meaning of the ":authority" pseudo-header field in Section 4 require opt-in negotiation. If a client were to use the provisions of the extended CONNECT method defined in this document without first receiving a SETTINGS_ENABLE_CONNECT_PROTOCOL parameter, a non-supporting peer would detect a malformed request and generate a stream error (Section 8.1.2.6 of [RFC7540]).
4. The Extended CONNECT Method

Usage of the CONNECT method in HTTP/2 is defined by Section 8.3 of [RFC7540]. This extension modifies the method in the following ways:

- A new pseudo-header field :protocol MAY be included on request HEADERS indicating the desired protocol to be spoken on the tunnel created by CONNECT. The pseudo-header field is single valued and contains a value from the HTTP Upgrade Token Registry located at https://www.iana.org/assignments/http-upgrade-tokens/http-upgrade-tokens.xhtml

- On requests that contain the :protocol pseudo-header field, the :scheme and :path pseudo-header fields of the target URI (See Section 5) MUST also be included.

- On requests bearing the :protocol pseudo-header field, the :authority pseudo-header field is interpreted according to Section 8.1.2.3 of [RFC7540] instead of Section 8.3 of [RFC7540]. In particular, the server MUST NOT create a tunnel to the host indicated by the :authority as it would with a CONNECT method request that was not modified by this extension.

Upon receiving a CONNECT request bearing the :protocol pseudo-header field the server establishes a tunnel to another service of the protocol type indicated by the pseudo-header field. This service may or may not be co-located with the server.

5. Using Extended CONNECT To Bootstrap the WebSocket Protocol

The :protocol pseudo-header field MUST be included in the CONNECT request and it MUST have a value of "websocket" to initiate a WebSocket connection on an HTTP/2 stream. Other HTTP request and response header fields, such as those for manipulating cookies, may be included in the HEADERS with the CONNECT method as usual. This request replaces the GET-based request in [RFC6455] and is used to process the WebSockets opening handshake.

The scheme of the target URI (Section 5.1 of [RFC7230]) MUST be "https" for "wss" schemed WebSockets and "http" for "ws" schemed WebSockets. The remainder of the Target URI is the same as the websocket URI. The websocket URI is still used for proxy autoconfiguration. The security requirements for the HTTP/2 connection used by this specification are established by [RFC7540] for https requests and [RFC8164] for http requests.
[RFC6455] requires the use of Connection and Upgrade header fields that are not part of HTTP/2. They MUST NOT be included in the CONNECT request defined here.

[RFC6455] requires the use of a Host header field which is also not part of HTTP/2. The Host information is conveyed as part of the :authority pseudo-header field which is required on every HTTP/2 transaction.

Implementations using this extended CONNECT to bootstrap WebSockets do not do the processing of the [RFC6455] Sec-WebSocket-Key and Sec-WebSocket-Accept header fields as that functionality has been superseded by the :protocol pseudo-header field.

The Origin [RFC6454], Sec-WebSocket-Version, Sec-WebSocket-Protocol, and Sec-WebSocket-Extensions header fields are used in the CONNECT request and response header fields in the same way as defined in [RFC6455]. Note that HTTP/1 header field names were case-insensitive and HTTP/2 requires they be encoded as lower case.

After successfully processing the opening handshake, the peers should proceed with the WebSocket Protocol [RFC6455] using the HTTP/2 stream from the CONNECT transaction as if it were the TCP connection referred to in [RFC6455]. The state of the WebSocket connection at this point is OPEN as defined by [RFC6455], Section 4.1.

The HTTP/2 stream closure is also analogous to the TCP connection closure of [RFC6455]. Orderly TCP level closures are represented as END_STREAM ([RFC7540], Section 6.1) flags and RST exceptions are represented with the RST_STREAM ([RFC7540], Section 6.4) frame with the CANCEL ([RFC7540], Section 7) error code.

5.1. Example
[[ From Client ]]                        [[ From Server ]]

SETTINGS
SETTINGS_ENABLE_CONNECT_ = 1

HEADERS + END_HEADERS
:method = CONNECT
:protocol = websocket
:scheme = https
:authority = server.example.com
sec-websocket-protocol = chat, superchat
sec-websocket-extensions = permessage-deflate
sec-websocket-version = 13
origin = http://www.example.com

HEADERS + END_HEADERS
:status = 200
sec-websocket-protocol = chat

DATA
WebSocket Data

DATA + END_STREAM
WebSocket Data

6. Design Considerations

A more native integration with HTTP/2 is certainly possible with larger additions to HTTP/2. This design was selected to minimize the solution complexity while still addressing the primary concern of running HTTP/2 and WebSockets concurrently.

7. About Intermediaries

This document does not change how WebSockets interacts with HTTP forward proxies. If a client wishing to speak WebSockets connects via HTTP/2 to an HTTP proxy it should continue to use a traditional (i.e. not with a :protocol pseudo-header field) CONNECT to tunnel through that proxy to the WebSocket server via HTTP.

The resulting version of HTTP on that tunnel determines whether WebSockets is initiated directly or via a modified CONNECT request described in this document.
8. Security Considerations

[RFC6455] ensures that non-WebSockets clients, especially XMLHttpRequest based clients, cannot make a WebSocket connection. Its primary mechanism for doing that is the use of Sec- prefixed request header fields that cannot be created by XMLHttpRequest-based clients. This specification addresses that concern in two ways:

- XMLHttpRequest also prohibits use of the CONNECT method in addition to Sec- prefixed request header fields.
- The use of a pseudo-header field is something that is connection specific and HTTP/2 does not ever allow to be created outside of the protocol stack.

The security considerations of [RFC6455] section 10 continue to apply to the use of the WebSockets Protocol when using this specification with the exception of 10.8. That section is not relevant because it is specific to the bootstrapping handshake that is changed in this document.

9. IANA Considerations

This document establishes an entry for the HTTP/2 Settings Registry that was established by Section 11.3 of [RFC7540].

Name: SETTINGS_ENABLE_CONNECT_PROTOCOL

Code: 0x8

Initial Value: 0

Specification: This document

10. Normative References


Acknowledgments

The 2017 HTTP Workshop had a very productive discussion that helped determine the key problem and acceptable level of solution complexity.

Author’s Address

Patrick McManus
Mozilla

Email: mcmanus@ducksong.com
Structured Headers for HTTP
draft-ietf-httpbis-header-structure-07

Abstract

This document describes a set of data types and algorithms associated with them that are intended to make it easier and safer to define and handle HTTP header fields. It is intended for use by new specifications of HTTP header fields as well as revisions of existing header field specifications when doing so does not cause interoperability issues.

Note to Readers

.Raycast EDITOR: please remove this section before publication_

Discussion of this draft takes place on the HTTP working group mailing list (ietf-http-wg@w3.org), which is archived at https://lists.w3.org/Archives/Public/ietf-http-wg/ [1].

Working Group information can be found at https://httpwg.github.io/ [2]; source code and issues list for this draft can be found at https://github.com/httpwg/http-extensions/labels/header-structure [3].

Tests for implementations are collected at https://github.com/httpwg/structured-header-tests [4].

Implementations are tracked at https://github.com/httpwg/wiki/wiki/Structured-Headers [5].

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.
Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on January 3, 2019.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction .............................................. 3
   1.1. Notational Conventions .......................... 4
2. Defining New Structured Headers .......................... 4
3. Structured Header Data Types ............................ 6
   3.1. Dictionaries .................................... 6
   3.2. Lists .......................................... 6
   3.3. Parameterised Lists .............................. 7
   3.4. Items .......................................... 7
   3.5. Integers ....................................... 8
   3.6. Floats ......................................... 8
   3.7. Strings ........................................ 8
   3.8. Identifiers .................................... 9
   3.9. Binary Content .................................. 9
4. Structured Headers in HTTP/1 .............................. 10
   4.1. Serialising Structured Headers into HTTP/1 .... 10
   4.2. Parsing HTTP/1 Header Fields into Structured Headers.. 14
5. IANA Considerations ....................................... 22
6. Security Considerations ................................... 22
7. References ............................................... 22
   7.1. Normative References ............................. 22
   7.2. Informative References ........................... 23
   7.3. URIs ........................................... 24
Appendix A. Frequently Asked Questions ..................... 24
   A.1. Why not JSON? .................................... 24
1. Introduction

Specifying the syntax of new HTTP header fields is an onerous task; even with the guidance in [RFC7231], Section 8.3.1, there are many decisions — and pitfalls — for a prospective HTTP header field author.

Once a header field is defined, bespoke parsers and serialisers often need to be written, because each header has slightly different handling of what looks like common syntax.

This document introduces a set of common data structures for use in HTTP header field values to address these problems. In particular, it defines a generic, abstract model for header field values, along with a concrete serialisation for expressing that model in HTTP/1 [RFC7230] header fields.

HTTP headers that are defined as "Structured Headers" use the types defined in this specification to define their syntax and basic handling rules, thereby simplifying both their definition by specification writers and handling by implementations.

Additionally, future versions of HTTP can define alternative serialisations of the abstract model of these structures, allowing headers that use it to be transmitted more efficiently without being redefined.

Note that it is not a goal of this document to redefine the syntax of existing HTTP headers; the mechanisms described herein are only intended to be used with headers that explicitly opt into them.

To specify a header field that is a Structured Header, see Section 2.

Section 3 defines a number of abstract data types that can be used in Structured Headers.
Those abstract types can be serialised into and parsed from textual headers - such as those used in HTTP/1 - using the algorithms described in Section 4.

1.1. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

This document uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234], including the VCHAR, DIGIT, ALPHA and DQUOTE rules from that document. It also includes the OWS rule from [RFC7230].

This document uses algorithms to specify parsing and serialisation behaviours, and ABNF to illustrate expected syntax.

For parsing, implementations MUST follow the algorithms, but MAY vary in implementation so as the behaviours are indistinguishable from specified behaviour. If there is disagreement between the parsing algorithms and ABNF, the specified algorithms take precedence.

For serialisation, the ABNF illustrates the range of acceptable wire representations with as much fidelity as possible, and the algorithms define the recommended way to produce them. Implementations MAY vary from the specified behaviour so long as the output still matches the ABNF.

2. Defining New Structured Headers

To define a HTTP header as a structured header, its specification needs to:

- Reference this specification. Recipients and generators of the header need to know that the requirements of this document are in effect.

- Specify the header field’s allowed syntax for values, in terms of the types described in Section 3, along with their associated semantics. Syntax definitions are encouraged to use the ABNF rules beginning with "sh-" defined in this specification.

- Specify any additional constraints upon the syntax of the structured used, as well as the consequences when those constraints are violated. When Structured Headers parsing fails,
the header is discarded (see Section 4.2); in most situations, header-specific constraints should do likewise.

Note that a header field definition cannot relax the requirements of a structure or its processing; they can only add additional constraints, because doing so would preclude handling by generic software.

For example:

```
# Foo-Example Header

The Foo-Example HTTP header field conveys information about how much Foo the message has.

Foo-Example is a Structured Header [RFCxxxx]. Its value MUST be a dictionary ([RFCxxxx], Section Y.Y). Its ABNF is:

   Foo-Example = sh-dictionary

The dictionary MUST contain:

* Exactly one member whose key is "foo", and whose value is an integer ([RFCxxxx], Section Y.Y), indicating the number of foos in the message.
* Exactly one member whose key is "barUrls", and whose value is a string ([RFCxxxx], Section Y.Y), conveying the Bar URLs for the message. See below for processing requirements.

If the parsed header field does not contain both, it MUST be ignored.

"foo" MUST be between 0 and 10, inclusive; other values MUST cause the header to be ignored.

"barUrls" contains a space-separated list of URI-references ([RFC3986], Section 4.1):

   barURLs = URI-reference *( 1*SP URI-reference )

If a member of barURLs is not a valid URI-reference, it MUST cause that value to be ignored.

If a member of barURLs is a relative reference ([RFC3986], Section 4.2), it MUST be resolved ([RFC3986], Section 5) before being used.
Internet-Draft

Structured Headers for HTTP

July 2018

This specification defines minimums for the length or number of
various structures supported by Structured Headers implementations.
It does not specify maximum sizes in most cases, but header authors
should be aware that HTTP implementations do impose various limits on
the size of individual header fields, the total number of fields,
and/or the size of the entire header block.
3.

Structured Header Data Types
This section defines the abstract value types that can be composed
into Structured Headers. The ABNF provided represents the on-wire
format in HTTP/1.

3.1.

Dictionaries

Dictionaries are unordered maps of key-value pairs, where the keys
are identifiers (Section 3.8) and the values are items (Section 3.4).
There can be one or more members, and keys are required to be unique.
The ABNF for dictionaries is:
sh-dictionary
dict-member
member-name
member-value

=
=
=
=

dict-member *( OWS "," OWS dict-member )
member-name "=" member-value
identifier
sh-item

In HTTP/1, keys and values are separated by "=" (without whitespace),
and key/value pairs are separated by a comma with optional
whitespace. For example:
Example-DictHeader: en="Applepie", da=*w4ZibGV0w6ZydGUK=*
Typically,
individual
optional.
unless the

a header field specification will define the semantics of
keys, as well as whether their presence is required or
Recipients MUST ignore keys that are undefined or unknown,
header field’s specification specifically disallows them.

Parsers MUST support dictionaries containing at least 1024 key/value
pairs.
3.2.

Lists

Lists are arrays of items (Section 3.4) with one or more members.
The ABNF for lists is:
sh-list
= list-member *( OWS "," OWS list-member )
list-member = sh-item

Nottingham & Kamp

Expires January 3, 2019

[Page 6]


In HTTP/1, each member is separated by a comma and optional whitespace. For example, a header field whose value is defined as a list of strings could look like:

Example-StrListHeader: "foo", "bar", "It was the best of times."

Header specifications can constrain the types of individual values if necessary.

Parsers MUST support lists containing at least 1024 members.

3.3. Parameterised Lists

Parameterised Lists are arrays of a parameterised identifiers.

A parameterised identifier is an identifier (Section 3.8) with an optional set of parameters, each parameter having a identifier and an optional value that is an item (Section 3.4). Ordering between parameters is not significant, and duplicate parameters MUST cause parsing to fail.

The ABNF for parameterised lists is:

\[
\text{sh-param-list} = \text{param-id} *( \text{OWS }","\text{ OWS param-id} ) \\
\text{param-id} = \text{identifier} \text{"parameter} \\
\text{param} = \text{OWS }";\text{ OWS param-name [ "=" param-value ]} \\
\text{param-name} = \text{identifier} \\
\text{param-value} = \text{sh-item}
\]

In HTTP/1, each param-id is separated by a comma and optional whitespace (as in Lists), and the parameters are separated by semicolons. For example:

Example-ParamListHeader: abc_123;a=1;b=2; cdef_456, ghi;q="9";r=w

Parsers MUST support parameterised lists containing at least 1024 members, and support members with at least 256 parameters.

3.4. Items

An item is can be a integer (Section 3.5), float (Section 3.6), string (Section 3.7), or binary content (Section 3.9).

The ABNF for items is:

\[
\text{sh-item} = \text{sh-integer} / \text{sh-float} / \text{sh-string} / \text{sh-binary}
\]
3.5. Integers

Integers have a range of -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 inclusive (i.e., a 64-bit signed integer).

The ABNF for integers is:

```
sh-integer = ["-"] 1*19DIGIT
```

For example:

```
Example-IntegerHeader: 42
```

3.6. Floats

Floats are integers with a fractional part, that can be stored as IEEE 754 double precision numbers (binary64) ([IEEE754]).

The ABNF for floats is:

```
sh-float    = ["-"] ( DIGIT "." 1*14DIGIT / 2DIGIT "." 1*13DIGIT / 3DIGIT "." 1*12DIGIT / 4DIGIT "." 1*11DIGIT / 5DIGIT "." 1*10DIGIT / 6DIGIT "." 1*9DIGIT / 7DIGIT "." 1*8DIGIT / 8DIGIT "." 1*7DIGIT / 9DIGIT "." 1*6DIGIT / 10DIGIT "." 1*5DIGIT / 11DIGIT "." 1*4DIGIT / 12DIGIT "." 1*3DIGIT / 13DIGIT "." 1*2DIGIT / 14DIGIT "." 1DIGIT )
```

For example, a header whose value is defined as a float could look like:

```
Example-FloatHeader: 4.5
```

3.7. Strings

Strings are zero or more printable ASCII [RFC0020] characters (i.e., the range 0x20 to 0x7E). Note that this excludes tabs, newlines, carriage returns, etc.

The ABNF for strings is:
In HTTP/1 headers, strings are delimited with double quotes, using a backslash ("\") to escape double quotes and backslashes. For example:

Example-StringHeader: "hello world"

Note that strings only use DQUOTE as a delimiter; single quotes do not delimit strings. Furthermore, only DQUOTE and "\" can be escaped; other sequences MUST cause parsing to fail.

Unicode is not directly supported in this document, because it causes a number of interoperability issues, and - with few exceptions - header values do not require it.

When it is necessary for a field value to convey non-ASCII string content, binary content (Section 3.9) SHOULD be specified, along with a character encoding (preferably, UTF-8).

Parsers MUST support strings with at least 1024 characters.

3.8. Identifiers

Identifiers are short textual identifiers; their abstract model is identical to their expression in the textual HTTP serialisation. Parsers MUST support identifiers with at least 64 characters.

The ABNF for identifiers is:

```
identifier = lcalpha *( lcalpha / DIGIT / "_" / "-"/ "*" / "/" )
lcalpha    = %x61-7A ; a-z
```

Note that identifiers can only contain lowercase letters.

3.9. Binary Content

Arbitrary binary content can be conveyed in Structured Headers.

The ABNF for binary content is:

```
sh-binary = "*" *(base64) "*"
base64   = ALPHA / DIGIT / "+" / "/" / ";=
```
In HTTP/1 headers, binary content is delimited with asterisks and encoded using base64 ([RFC4648], Section 4). For example:

Example-BinaryHdr: *cHJldGVuZCB0aGlzIGlzIGJpbmFyeSBjb250ZW50Lg==*

Parsers MUST support binary content with at least 16384 octets after decoding.

4. Structured Headers in HTTP/1

This section defines how to serialise and parse Structured Headers in HTTP/1 textual header fields, and protocols compatible with them (e.g., in HTTP/2 [RFC7540] before HPACK [RFC7541] is applied).

4.1. Serialising Structured Headers into HTTP/1

Given a structured defined in this specification:

1. If the structure is a dictionary, return the result of Serialising a Dictionary (§ser-dictionary).
2. If the structure is a list, return the result of Serialising a List (§ser-list).
3. If the structure is a parameterised list, return the result of Serialising a Parameterised List (§ser-param-list).
4. If the structure is an item, return the result of Serialising an Item (§ser-item).
5. Otherwise, fail serialisation.

4.1.1. Serialising a Dictionary

Given a dictionary as input:

1. Let output be an empty string.
2. For each member mem of input:
   1. Let name be the result of applying Serialising an Identifier Section 4.1.8 to mem’s member-name.
   2. Append name to output.
   3. Append "=" to output.
4. Let value be the result of applying Serialising an Item Section 4.1.4 to mem’s member-value.

5. Append value to output.

3. Return output.

4.1.2. Serialising a List

Given a list as input:

1. Let output be an empty string.

2. For each member mem of input:

   1. Let value be the result of applying Serialising an Item Section 4.1.4 to mem.

   2. Append value to output.

   3. If more members remain in input:

      1. Append a COMMA to output.

      2. Append a single WS to output.

3. Return output.

4.1.3. Serialising a Parameterised List

Given a parameterised list as input:

1. Let output be an empty string.

2. For each member mem of input:

   1. Let id be the result of applying Serialising an Identifier Section 4.1.8 to mem’s identifier.

   2. Append id to output.

   3. For each parameter in mem’s parameters:

      1. Let name be the result of applying Serialising an Identifier Section 4.1.8 to parameter’s param-name.

      2. Append name to output.
3. If parameter has a param-value:
   1. Let value be the result of applying Serialising an Item Section 4.1.4 to parameter’s param-value.
   2. Append "=" to output.
   3. Append value to output.
3. Return output.

4.1.4. Serialising an Item

Given an item as input:
1. If input is a type other than an integer, float, string or binary content, fail serialisation.
2. Let output be an empty string.
3. If input is an integer, let value be the result of applying Serialising an Integer Section 4.1.5 to input.
4. If input is a float, let value be the result of applying Serialising a Float Section 4.1.6 to input.
5. If input is a string, let value be the result of applying Serialising a String Section 4.1.7 to input.
6. If input is binary content, let value be the result of applying Serialising Binary Content Section 4.1.9 to input.
7. Return output.

4.1.5. Serialising an Integer

Given an integer as input:
1. If input is not an integer in the range of -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 inclusive, fail serialisation.
2. Let output be an empty string.
3. If input is less than (but not equal to) 0, append "-" to output.
4. Append input’s numeric value represented in base 10 using only decimal digits to output.
5. Return output.

4.1.6. Serialising a Float

Given a float as input:

1. If input is not a IEEE 754 double precision number, fail serialisation.

2. Let output be an empty string.

3. If input is less than (but not equal to) 0, append "-" to output.

4. Append input’s integer component represented in base 10 using only decimal digits to output; if it is zero, append "0".

5. Append "." to output.

6. Append input’s decimal component represented in base 10 using only decimal digits to output; if it is zero, append "0".

7. Return output.

4.1.7. Serialising a String

Given a string as input:

1. If input is not a sequence of characters, or contains characters outside the range allowed by VCHAR, fail serialisation.

2. Let output be an empty string.

3. Append DQUOTE to output.

4. For each character char in input:

   1. If char is "\" or DQUOTE:

      1. Append "\" to output.

      2. Append char to output, using ASCII encoding [RFC0020].

   5. Append DQUOTE to output.

6. Return output.
4.1.8. Serialising an Identifier

Given an identifier as input:

1. If input is not a sequence of characters, or contains characters not allowed in Section 3.8, fail serialisation.
2. Let output be an empty string.
3. Append input to output, using ASCII encoding [RFC0020].
4. Return output.

4.1.9. Serialising Binary Content

Given binary content as input:

1. If input is not a sequence of bytes, fail serialisation.
2. Let output be an empty string.
3. Append "*" to output.
4. Append the result of base64-encoding input as per [RFC4648], Section 4, taking account of the requirements below.
5. Append "*" to output.
6. Return output.

The encoded data is required to be padded with ", as per [RFC4648], Section 3.2.

Likewise, encoded data SHOULD have pad bits set to zero, as per [RFC4648], Section 3.5, unless it is not possible to do so due to implementation constraints.

4.2. Parsing HTTP/1 Header Fields into Structured Headers

When a receiving implementation parses textual HTTP header fields (e.g., in HTTP/1 or HTTP/2) that are known to be Structured Headers, it is important that care be taken, as there are a number of edge cases that can cause interoperability or even security problems. This section specifies the algorithm for doing so.

Given an ASCII string input_string that represents the chosen header's field-value, and header_type, one of "dictionary", "list", "param-list", or "item", return the parsed header value.
1. Discard any leading OWS from input_string.

2. If header_type is "dictionary", let output be the result of Parsing a Dictionary from Text (Section 4.2.1).

3. If header_type is "list", let output be the result of Parsing a List from Text (Section 4.2.2).

4. If header_type is "param-list", let output be the result of Parsing a Parameterised List from Text (Section 4.2.3).

5. Otherwise, let output be the result of Parsing an Item from Text (Section 4.2.5).

6. Discard any leading OWS from input_string.

7. If input_string is not empty, fail parsing.

8. Otherwise, return output.

When generating input_string, parsers MUST combine all instances of the target header field into one comma-separated field-value, as per [RFC7230], Section 3.2.2; this assures that the header is processed correctly.

For Lists, Parameterised Lists and Dictionaries, this has the effect of correctly concatenating all instances of the header field.

Strings can but SHOULD NOT be split across multiple header instances, because comma(s) inserted upon combination will become part of the string output by the parser.

Integers, Floats and Binary Content cannot be split across multiple headers because the inserted commas will cause parsing to fail.

If parsing fails - including when calling another algorithm - the entire header field’s value MUST be discarded. This is intentionally strict, to improve interoperability and safety, and specifications referencing this document cannot loosen this requirement.

Note that this has the effect of discarding any header field with non-ASCII characters in input_string.

4.2.1. Parsing a Dictionary from Text

Given an ASCII string input_string, return a mapping of (identifier, item). input_string is modified to remove the parsed value.
1. Let dictionary be an empty, unordered mapping.

2. While input_string is not empty:
   1. Let this_key be the result of running Parse Identifier from Text (Section 4.2.8) with input_string.
   2. If dictionary already contains this_key, fail parsing.
   3. Consume the first character of input_string; if it is not "=", fail parsing.
   4. Let this_value be the result of running Parse Item from Text (Section 4.2.5) with input_string.
   5. Add key this_key with value this_value to dictionary.
   6. Discard any leading OWS from input_string.
   7. If input_string is empty, return dictionary.
   8. Consume the first character of input_string; if it is not COMMA, fail parsing.
   9. Discard any leading OWS from input_string.
  10. If input_string is empty, fail parsing.

3. No structured data has been found; fail parsing.

4.2.2. Parsing a List from Text

Given an ASCII string input_string, return a list of items. input_string is modified to remove the parsed value.

1. Let items be an empty array.

2. While input_string is not empty:
   1. Let item be the result of running Parse Item from Text (Section 4.2.5) with input_string.
   2. Append item to items.
   3. Discard any leading OWS from input_string.
   4. If input_string is empty, return items.
5. Consume the first character of input_string; if it is not COMMA, fail parsing.

6. Discard any leading OWS from input_string.

7. If input_string is empty, fail parsing.

3. No structured data has been found; fail parsing.

4.2.3. Parsing a Parameterised List from Text

Given an ASCII string input_string, return a list of parameterised identifiers. input_string is modified to remove the parsed value.

1. Let items be an empty array.

2. While input_string is not empty:

   1. Let item be the result of running Parse Parameterised Identifier from Text (Section 4.2.4) with input_string.

   2. Append item to items.

   3. Discard any leading OWS from input_string.

   4. If input_string is empty, return items.

   5. Consume the first character of input_string; if it is not COMMA, fail parsing.

   6. Discard any leading OWS from input_string.

   7. If input_string is empty, fail parsing.

3. No structured data has been found; fail parsing.

4.2.4. Parsing a Parameterised Identifier from Text

Given an ASCII string input_string, return a identifier with a mapping of parameters. input_string is modified to remove the parsed value.

1. Let primary_identifier be the result of Parsing a Identifier from Text (Section 4.2.8) from input_string.

2. Let parameters be an empty, unordered mapping.

3. In a loop:
1. Discard any leading OWS from input_string.

2. If the first character of input_string is not ";", exit the loop.

3. Consume a ";" character from the beginning of input_string.

4. Discard any leading OWS from input_string.

5. let param_name be the result of Parsing a Identifier from Text (Section 4.2.8) from input_string.

6. If param_name is already present in parameters, fail parsing.

7. Let param_value be a null value.

8. If the first character of input_string is "=":
   1. Consume the "=" character at the beginning of input_string.
   2. Let param_value be the result of Parsing an Item from Text (Section 4.2.5) from input_string.

9. Insert (param_name, param_value) into parameters.

4. Return the tuple (primary_identifier, parameters).

4.2.5. Parsing an Item from Text

Given an ASCII string input_string, return an item. input_string is modified to remove the parsed value.

1. Discard any leading OWS from input_string.

2. If the first character of input_string is a "-" or a DIGIT, process input_string as a number (Section 4.2.6) and return the result.

3. If the first character of input_string is a DQUOTE, process input_string as a string (Section 4.2.7) and return the result.

4. If the first character of input_string is "*", process input_string as binary content (Section 4.2.9) and return the result.

5. Otherwise, fail parsing.
4.2.6. Parsing a Number from Text

NOTE: This algorithm parses both Integers Section 3.5 and Floats Section 3.6, and returns the corresponding structure.

1. Let type be "integer".
2. Let sign be 1.
3. Let input_number be an empty string.
4. If the first character of input_string is "-", remove it from input_string and set sign to -1.
5. If input_string is empty, fail parsing.
6. If the first character of input_string is not a DIGIT, fail parsing.
7. While input_string is not empty:
   1. Let char be the result of removing the first character of input_string.
   2. If char is a DIGIT, append it to input_number.
   3. Else, if type is "integer" and char is ".", append char to input_number and set type to "float".
   4. Otherwise, fail parsing.
5. If type is "integer" and input_number contains more than 19 characters, fail parsing.
6. If type is "float" and input_number contains more than 16 characters, fail parsing.
8. If type is "integer":
   1. Parse input_number as an integer and let output_number be the result.
   2. If output_number is outside the range defined in Section 3.5, fail parsing.
9. Otherwise:
   1. If the final character of input_number is ".", fail parsing.
2. Parse input_number as a float and let output_number be the result.

10. Return the product of output_number and sign.

4.2.7. Parsing a String from Text

Given an ASCII string input_string, return an unquoted string. input_string is modified to remove the parsed value.

1. Let output_string be an empty string.

2. If the first character of input_string is not DQUOTE, fail parsing.

3. Discard the first character of input_string.

4. While input_string is not empty:
   1. Let char be the result of removing the first character of input_string.
   2. If char is a backslash ("\"):
      1. If input_string is now empty, fail parsing.
      2. Else:
         1. Let next_char be the result of removing the first character of input_string.
         2. If next_char is not DQUOTE or "\", fail parsing.
         3. Append next_char to output_string.
      3. Else, if char is DQUOTE, return output_string.
   4. Else, if char is in the range %x00-1f or %x7f (i.e., is not in VCHAR), fail parsing.
   5. Else, append char to output_string.

5. Otherwise, fail parsing.
4.2.8. Parsing an Identifier from Text

Given an ASCII string input_string, return a identifier. input_string is modified to remove the parsed value.

1. If the first character of input_string is not lcalpha, fail parsing.
2. Let output_string be an empty string.
3. While input_string is not empty:
   1. Let char be the result of removing the first character of input_string.
   2. If char is not one of lcalpha, DIGIT, ",", ",", "*" or "/":
      1. Prepend char to input_string.
      2. Return output_string.
   3. Append char to output_string.
4. Return output_string.

4.2.9. Parsing Binary Content from Text

Given an ASCII string input_string, return binary content. input_string is modified to remove the parsed value.

1. If the first character of input_string is not "*", fail parsing.
2. Discard the first character of input_string.
3. Let b64_content be the result of removing content of input_string up to but not including the first instance of the character "*". If there is not a "*" character before the end of input_string, fail parsing.
4. Consume the "*" character at the beginning of input_string.
5. If b64_content contains a character not included in ALPHA, DIGIT, "+", "/" and "=", fail parsing.
6. Let binary_content be the result of Base 64 Decoding [RFC4648] b64_content, synthesising padding if necessary (note the requirements about recipient behaviour below).
7. Return binary_content.

As per [RFC4648], Section 3.2, it is RECOMMENDED that parsers reject encoded data that is not properly padded, although this might not be possible in some base64 implementations.

Because some implementations of base64 do not allow rejection of encoded data that has non-zero pad bits (see [RFC4648], Section 3.5), parsers SHOULD NOT fail when it is present, unless they cannot be configured to handle it.

This specification does not relax the requirements in [RFC4648], Section 3.1 and 3.3; therefore, parsers MUST fail on characters outside the base64 alphabet, and on line feeds in encoded data.

5. IANA Considerations

This draft has no actions for IANA.

6. Security Considerations

The size of most types defined by Structured Headers is not limited; as a result, extremely large header fields could be an attack vector (e.g., for resource consumption). Most HTTP implementations limit the sizes of size of individual header fields as well as the overall header block size to mitigate such attacks.

It is possible for parties with the ability to inject new HTTP header fields to change the meaning of a Structured Headers. In some circumstances, this will cause parsing to fail, but it is not possible to reliably fail in all such circumstances.

7. References

7.1. Normative References


7.2. Informative References


See also http://grouper.ieee.org/groups/754/ [6].


7.3. URIs

[1] https://lists.w3.org/Archives/Public/ietf-http-wg/

Appendix A. Frequently Asked Questions

A.1. Why not JSON?

Earlier proposals for structured headers were based upon JSON [RFC8259]. However, constraining its use to make it suitable for HTTP header fields required senders and recipients to implement specific additional handling.

For example, JSON has specification issues around large numbers and objects with duplicate members. Although advice for avoiding these issues is available (e.g., [RFC7493]), it cannot be relied upon.

Likewise, JSON strings are by default Unicode strings, which have a number of potential interoperability issues (e.g., in comparison). Although implementers can be advised to avoid non-ASCII content where unnecessary, this is difficult to enforce.

Another example is JSON’s ability to nest content to arbitrary depths. Since the resulting memory commitment might be unsuitable (e.g., in embedded and other limited server deployments), it’s necessary to limit it in some fashion; however, existing JSON implementations have no such limits, and even if a limit is specified, it’s likely that some header field definition will find a need to violate it.

Because of JSON’s broad adoption and implementation, it is difficult to impose such additional constraints across all implementations; some deployments would fail to enforce them, thereby harming interoperability.

Since a major goal for Structured Headers is to improve interoperability and simplify implementation, these concerns led to a format that requires a dedicated parser and serialiser.
Additionally, there were widely shared feelings that JSON doesn’t "look right" in HTTP headers.

A.2. Structured Headers don’t "fit" my data.

Structured headers intentionally limits the complexity of data structures, to assure that it can be processed in a performant manner with little overhead. This means that work is necessary to fit some data types into them.

Sometimes, this can be achieved by creating limited substructures in values, and/or using more than one header. For example, consider:

Example-Thing: name="Widget", cost=89.2, descriptions="foo bar"
Example-Description: foo; url="https://example.net"; context=123,
bar; url="https://example.org"; context=456

Since the description contains a list of key/value pairs, we use a Parameterised List to represent them, with the identifier for each item in the list used to identify it in the "descriptions" member of the Example-Thing header.

When specifying more than one header, it’s important to remember to describe what a processor’s behaviour should be when one of the headers is missing.

If you need to fit arbitrarily complex data into a header, Structured Headers is probably a poor fit for your use case.

Appendix B. Changes

_RFC Editor: Please remove this section before publication._

B.1. Since draft-ietf-httpbis-header-structure-06

   - Add a FAQ.
   - Allow non-zero pad bits.
   - Explicitly check for integers that violate constraints.

B.2. Since draft-ietf-httpbis-header-structure-05

   - Reorganise specification to separate parsing out.
   - Allow referencing specs to use ABNF.
   - Define serialisation algorithms.
Refine relationship between ABNF, parsing and serialisation algorithms.

B.3. Since draft-ietf-httpbis-header-structure-04

- Remove identifiers from item.
- Remove most limits on sizes.
- Refine number parsing.

B.4. Since draft-ietf-httpbis-header-structure-03

- Strengthen language around failure handling.

B.5. Since draft-ietf-httpbis-header-structure-02

- Split Numbers into Integers and Floats.
- Define number parsing.
- Tighten up binary parsing and give it an explicit end delimiter.
- Clarify that mappings are unordered.
- Allow zero-length strings.
- Improve string parsing algorithm.
- Improve limits in algorithms.
- Require parsers to combine header fields before processing.
- Throw an error on trailing garbage.


- Replaced with draft-nottingham-structured-headers.

B.7. Since draft-ietf-httpbis-header-structure-00

- Added signed 64bit integer type.
- Drop UTF8, and settle on BCP137 ::EmbeddedUnicodeChar for h1-unicode-string.
- Change h1_blob delimiter to ":" since "’" is valid t_char
Authors’ Addresses

Mark Nottingham
Fastly
Email: mnot@mnot.net
URI: https://www.mnot.net/

Poul-Henning Kamp
The Varnish Cache Project
Email: phk@varnish-cache.org
Secondary Certificate Authentication in HTTP/2

draft-ietf-httpbis-http2-secondary-certs-02

Abstract

A use of TLS Exported Authenticators is described which enables HTTP/2 clients and servers to offer additional certificate-based credentials after the connection is established. The means by which these credentials are used with requests is defined.

Note to Readers

Discussion of this draft takes place on the HTTP working group mailing list (ietf-http-wg@w3.org), which is archived at https://lists.w3.org/Archives/Public/ietf-http-wg/ [1].

Working Group information can be found at http://httpwg.github.io/ [2]; source code and issues list for this draft can be found at https://github.com/httpwg/http-extensions/labels/secondary-certs [3].

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on December 28, 2018.
Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the
document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal
Provisions Relating to IETF Documents
(https://trustee.ietf.org/license-info) in effect on the date of
publication of this document. Please review these documents
carefully, as they describe your rights and restrictions with respect
to this document. Code Components extracted from this document must
include Simplified BSD License text as described in Section 4.e of
the Trust Legal Provisions and are provided without warranty as
described in the Simplified BSD License.

Table of Contents

1.  Introduction .................................................. 3
   1.1. Server Certificate Authentication .......................... 3
   1.2. Client Certificate Authentication .......................... 4
   1.2.1. HTTP/1.1 Using TLS 1.2 and Earlier ....................... 5
   1.2.2. HTTP/1.1 Using TLS 1.3 ................................. 6
   1.2.3. HTTP/2 ................................................... 6
   1.3. HTTP-Layer Certificate Authentication ....................... 7
   1.4. Terminology ................................................ 8
2.  Discovering Additional Certificates at the HTTP/2 Layer ......... 8
   2.1. Indicating Support for HTTP-Layer Certificate
       Authentication .............................................. 8
   2.2. Making Certificates or Requests Available .................. 9
   2.3. Requiring Certificate Authentication ........................ 10
       2.3.1. Requiring Additional Server Certificates ............... 10
       2.3.2. Requiring Additional Client Certificates ............... 11
3.  Certificates Frames for HTTP/2 ................................ 12
   3.1. The CERTIFICATE_NEEDED Frame ............................ 12
   3.2. The USE_CERTIFICATE Frame ................................ 14
   3.3. The CERTIFICATE_REQUEST Frame ............................ 15
       3.3.1. Exported Authenticator Request Characteristics .......... 16
   3.4. The CERTIFICATE Frame .................................... 16
       3.4.1. Exported Authenticator Characteristics .................. 17
4.  Indicating Failures During HTTP-Layer Certificate
    Authentication ................................................ 17
5.  Security Considerations ....................................... 18
   5.1. Impersonation .............................................. 18
   5.2. Fingerprinting ............................................. 19
   5.3. Denial of Service ......................................... 19
   5.4. Confusion About State ..................................... 19
6.  IANA Considerations .......................................... 20
   6.1. HTTP/2 SETTINGS_HTTP_CERT_AUTH Setting ................. 20
1. Introduction

HTTP clients need to know that the content they receive on a connection comes from the origin that they intended to retrieve from. The traditional form of server authentication in HTTP has been in the form of a single X.509 certificate provided during the TLS ([RFC5246], [I-D.ietf-tls-tls13]) handshake.

Many existing HTTP [RFC7230] servers also have authentication requirements for the resources they serve. Of the bountiful authentication options available for authenticating HTTP requests, client certificates present a unique challenge for resource-specific authentication requirements because of the interaction with the underlying TLS layer.

TLS 1.2 [RFC5246] supports one server and one client certificate on a connection. These certificates may contain multiple identities, but only one certificate may be provided.

Many HTTP servers host content from several origins. HTTP/2 permits clients to reuse an existing HTTP connection to a server provided that the secondary origin is also in the certificate provided during the TLS handshake. In many cases, servers choose to maintain separate certificates for different origins but still desire the benefits of a shared HTTP connection.

1.1. Server Certificate Authentication

Section 9.1.1 of [RFC7540] describes how connections may be used to make requests from multiple origins as long as the server is authoritative for both. A server is considered authoritative for an origin if DNS resolves the origin to the IP address of the server and (for TLS) if the certificate presented by the server contains the origin in the Subject Alternative Names field.
[RFC7838] enables a step of abstraction from the DNS resolution. If both hosts have provided an Alternative Service at hostnames which resolve to the IP address of the server, they are considered authoritative just as if DNS resolved the origin itself to that address. However, the server's one TLS certificate is still required to contain the name of each origin in question.

[RFC8336] relaxes the requirement to perform the DNS lookup if already connected to a server with an appropriate certificate which claims support for a particular origin.

Servers which host many origins often would prefer to have separate certificates for some sets of origins. This may be for ease of certificate management (the ability to separately revoke or renew them), due to different sources of certificates (a CDN acting on behalf of multiple origins), or other factors which might drive this administrative decision. Clients connecting to such origins cannot currently reuse connections, even if both client and server would prefer to do so.

Because the TLS SNI extension is exchanged in the clear, clients might also prefer to retrieve certificates inside the encrypted context. When this information is sensitive, it might be advantageous to request a general-purpose certificate or anonymous ciphersuite at the TLS layer, while acquiring the "real" certificate in HTTP after the connection is established.

1.2. Client Certificate Authentication

For servers that wish to use client certificates to authenticate users, they might request client authentication during or immediately after the TLS handshake. However, if not all users or resources need certificate-based authentication, a request for a certificate has the unfortunate consequence of triggering the client to seek a certificate, possibly requiring user interaction, network traffic, or other time-consuming activities. During this time, the connection is stalled in many implementations. Such a request can result in a poor experience, particularly when sent to a client that does not expect the request.

The TLS 1.3 CertificateRequest can be used by servers to give clients hints about which certificate to offer. Servers that rely on certificate-based authentication might request different certificates for different resources. Such a server cannot use contextual information about the resource to construct an appropriate TLS CertificateRequest message during the initial handshake.
Consequently, client certificates are requested at connection establishment time only in cases where all clients are expected or required to have a single certificate that is used for all resources. Many other uses for client certificates are reactive, that is, certificates are requested in response to the client making a request.

1.2.1. HTTP/1.1 Using TLS 1.2 and Earlier

In HTTP/1.1, a server that relies on client authentication for a subset of users or resources does not request a certificate when the connection is established. Instead, it only requests a client certificate when a request is made to a resource that requires a certificate. TLS 1.2 [RFC5246] accommodates this by permitting the server to request a new TLS handshake, in which the server will request the client’s certificate.

Figure 1 shows the server initiating a TLS-layer renegotiation in response to receiving an HTTP/1.1 request to a protected resource.

Client                                      Server
-- (HTTP) GET /protected -------------------> *1
<---------------------- (TLS) HelloRequest -- *2
-- (TLS) ClientHello ---------------------->
<------------------ (TLS) ServerHello, ... --
<----------------- (TLS) CertificateRequest -- *3
-- (TLS) ..., Certificate ------------------> *4
-- (TLS) Finished ------------------------->
<-------------------------- (TLS) Finished --
<--------------------------- (HTTP) 200 OK -- *5

Figure 1: HTTP/1.1 reactive certificate authentication with TLS 1.2

In this example, the server receives a request for a protected resource (at *1 on Figure 1). Upon performing an authorization check, the server determines that the request requires authentication using a client certificate and that no such certificate has been provided.

The server initiates TLS renegotiation by sending a TLS HelloRequest (at *2). The client then initiates a TLS handshake. Note that some TLS messages are elided from the figure for the sake of brevity.

The critical messages for this example are the server requesting a certificate with a TLS CertificateRequest (*3); this request might use information about the request or resource. The client then provides a certificate and proof of possession of the private key in Certificate and CertificateVerify messages (*4).
When the handshake completes, the server performs any authorization checks a second time. With the client certificate available, it then authorizes the request and provides a response (*5).

1.2.2. HTTP/1.1 Using TLS 1.3

TLS 1.3 [I-D.ietf-tls-tls13] introduces a new client authentication mechanism that allows for clients to authenticate after the handshake has been completed. For the purposes of authenticating an HTTP request, this is functionally equivalent to renegotiation. Figure 2 shows the simpler exchange this enables.

Client                                      Server
-- (HTTP) GET /protected ------------------->
<------------------------ (TLS) CertificateRequest --
-- (TLS) Certificate, CertificateVerify, Finished ------------------------>
<---------------------------------- (HTTP) 200 OK --

Figure 2: HTTP/1.1 reactive certificate authentication with TLS 1.3

TLS 1.3 does not support renegotiation, instead supporting direct client authentication. In contrast to the TLS 1.2 example, in TLS 1.3, a server can simply request a certificate.

1.2.3. HTTP/2

An important part of the HTTP/1.1 exchange is that the client is able to easily identify the request that caused the TLS renegotiation. The client is able to assume that the next unanswered request on the connection is responsible. The HTTP stack in the client is then able to direct the certificate request to the application or component that initiated that request. This ensures that the application has the right contextual information for processing the request.

In HTTP/2, a client can have multiple outstanding requests. Without some sort of correlation information, a client is unable to identify which request caused the server to request a certificate.

Thus, the minimum necessary mechanism to support reactive certificate authentication in HTTP/2 is an identifier that can be used to correlate an HTTP request with a request for a certificate. Since streams are used for individual requests, correlation with a stream is sufficient.

[RFC7540] prohibits renegotiation after any application data has been sent. This completely blocks reactive certificate authentication in HTTP/2 using TLS 1.2. If this restriction were relaxed by an
extension or update to HTTP/2, such an identifier could be added to TLS 1.2 by means of an extension to TLS. Unfortunately, many TLS 1.2 implementations do not permit application data to continue during a renegotiation. This is problematic for a multiplexed protocol like HTTP/2.

1.3. HTTP-Layer Certificate Authentication

This draft defines HTTP/2 frames to carry the relevant certificate messages, enabling certificate-based authentication of both clients and servers independent of TLS version. This mechanism can be implemented at the HTTP layer without breaking the existing interface between HTTP and applications above it.

This could be done in a naive manner by replicating the TLS messages as HTTP/2 frames on each stream. However, this would create needless redundancy between streams and require frequent expensive signing operations. Instead, TLS Exported Authenticators [I-D.ietf-tls-exported-authenticator] are exchanged on stream zero and other frames incorporate them to particular requests by reference as needed.

TLS Exported Authenticators are structured messages that can be exported by either party of a TLS connection and validated by the other party. Given an established TLS connection, a request can be constructed which describes the desired certificate and an authenticator message can be constructed proving possession of a certificate and a corresponding private key. Both requests and authenticators can be generated by either the client or the server. Exported Authenticators use the message structures from Sections 4.3.2 and 4.4 of [I-D.ietf-tls-tls13], but different parameters.

Each Authenticator is computed using a Handshake Context and Finished MAC Key derived from the TLS session. The Handshake Context is identical for both parties of the TLS connection, while the Finished MAC Key is dependent on whether the Authenticator is created by the client or the server.

Successfully verified Authenticators result in certificate chains, with verified possession of the corresponding private key, which can be supplied into a collection of available certificates. Likewise, descriptions of desired certificates can be supplied into these collections.

Section 2 describes how the feature is employed, defining means to detect support in peers (Section 2.1), make certificates and requests available (Section 2.2), and indicate when streams are blocked waiting on an appropriate certificate (Section 2.3). Section 3
defines the required frame types, which parallel the TLS 1.3 message exchange. Finally, Section 4 defines new error types which can be used to notify peers when the exchange has not been successful.

1.4. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. Discovering Additional Certificates at the HTTP/2 Layer

A certificate chain with proof of possession of the private key corresponding to the end-entity certificate is sent as a sequence of "CERTIFICATE" frames (see Section 3.4) on stream zero. Once the holder of a certificate has sent the chain and proof, this certificate chain is cached by the recipient and available for future use. Clients can proactively indicate the certificate they intend to use on each request using an unsolicited "USE_CERTIFICATE" frame, if desired. The previously-supplied certificates are available for reference without having to resend them.

Otherwise, the server uses a "CERTIFICATE_REQUEST" frame to describe a class of certificates on stream zero, then uses "CERTIFICATE_NEEDED" frames to associate these with individual requests. The client responds with a "USE_CERTIFICATE" frame indicating the certificate which should be used to satisfy the request.

Data sent by each peer is correlated by the ID given in each frame. This ID is unrelated to values used by the other peer, even if each uses the same ID in certain cases. "USE_CERTIFICATE" frames indicate whether they are sent proactively or are in response to a "CERTIFICATE_NEEDED" frame.

2.1. Indicating Support for HTTP-Layer Certificate Authentication

Clients and servers that will accept requests for HTTP-layer certificate authentication indicate this using the HTTP/2 "SETTINGS_HTTP_CERT_AUTH" (0xSETTING-TBD) setting.

The initial value for the "SETTINGS_HTTP_CERT_AUTH" setting is 0, indicating that the peer does not support HTTP-layer certificate authentication. If a peer does support HTTP-layer certificate authentication, the value is non-zero.
In order to ensure that the TLS connection is direct to the server, rather than via a TLS-terminating proxy, each side will separately compute and confirm the value of this setting. The setting is derived from a TLS exporter (see Section 7.5 of [I-D.ietf-tls-tls13] and [RFC5705] for more details on exporters). Clients MUST NOT use an early exporter during their 0-RTT flight, but MUST send an updated SETTINGS frame using a regular exporter after the TLS handshake completes.

The exporter is constructed with the following input:

- **Label:**
  - "EXPORTER HTTP CERTIFICATE client" for clients
  - "EXPORTER HTTP CERTIFICATE server" for servers
- **Context:** Empty
- **Length:** Four bytes

The resulting exporter is converted to a setting value as:

```
(Exporter & 0x3fffffff) | 0x80000000
```

That is, the most significant bit will always be set, regardless of the value of the exporter. Each endpoint will compute the expected value from their peer. If the setting is not received, or if the value received is not the expected value, the frames defined in this document SHOULD NOT be sent.

### 2.2. Making Certificates or Requests Available

When both peers have advertised support for HTTP-layer certificates as in Section 2.1, either party can supply additional certificates into the connection at any time. This means that clients or servers which predict a certificate will be required could supply the certificate before being asked. These certificates are available for reference by future "USE_CERTIFICATE" frames.

Certificates supplied by servers can be considered by clients without further action by the server. A server SHOULD NOT send certificates which do not cover origins which it is prepared to service on the current connection, but MAY use the ORIGIN frame [RFC8336] to indicate that not all covered origins will be served.
Likewise, either party can supply a "CERTIFICATE_REQUEST" that outlines parameters of a certificate they might request in the future. Upon receipt of a "CERTIFICATE_REQUEST", endpoints SHOULD provide a corresponding certificate in anticipation of a request shortly being blocked. Clients MAY wait for a "CERTIFICATE_NEEDED" frame to assist in associating the certificate request with a particular HTTP transaction.

2.3. Requiring Certificate Authentication

2.3.1. Requiring Additional Server Certificates

As defined in [RFC7540], when a client finds that a https:// origin (or Alternative Service [RFC7838]) to which it needs to make a request has the same IP address as a server to which it is already connected, it MAY check whether the TLS certificate provided contains the new origin as well, and if so, reuse the connection.

If the TLS certificate does not contain the new origin, but the server has claimed support for that origin (with an ORIGIN frame, see [RFC8336]) and advertised support for HTTP-layer certificates (see Section 2.1), the client MAY send a "CERTIFICATE_REQUEST" frame describing the desired origin. The client then sends a "CERTIFICATE_NEEDED" frame for stream zero referencing the request, indicating that the connection cannot be used for that origin until the certificate is provided.

If the server does not have the desired certificate, it MUST send an Empty Authenticator, as described in Section 5 of
[I-D.ietf-tls-exported-authenticator], in a "CERTIFICATE" frame in response to the request, followed by a "USE_CERTIFICATE" frame for stream zero which references the Empty Authenticator. In this case, or if the server has not advertised support for HTTP-layer certificates, the client MUST NOT send any requests for resources in that origin on the current connection.

Client                                      Server
<----------------------- (stream 0) ORIGIN --
-- (stream 0) CERTIFICATE_REQUEST ---------->
-- (stream 0) CERTIFICATE_NEEDED (S=0) ------>
<----------------------- (stream 0) CERTIFICATE --
-------- (stream 0) USE_CERTIFICATE (S=0) --
-- (stream N) GET /from-new-origin -------->
<----------------------- (stream N) 200 OK --

Figure 5: Client-requested certificate

If a client receives a "PUSH_PROMISE" referencing an origin for which it has not yet received the server’s certificate, this is a fatal connection error (see section 8.2 of [RFC7540]). To avoid this, servers MUST supply the associated certificates before pushing resources from a different origin.

2.3.2. Requiring Additional Client Certificates

Likewise, the server sends a "CERTIFICATE_NEEDED" frame for each stream where certificate authentication is required. The client answers with a "USE_CERTIFICATE" frame indicating the certificate to use on that stream. If the request parameters or the responding certificate are not already available, they will need to be sent as described in Section 2.2 as part of this exchange.

Client                                      Server
<------- (stream 0) CERTIFICATE_REQUEST --
...  
-- (stream N) GET /protected ------------------>
<------- (stream 0) CERTIFICATE_NEEDED (S=N) --
-- (stream 0) CERTIFICATE ------------------->
-- (stream 0) USE_CERTIFICATE (S=N) ------->
<----------------------- (stream N) 200 OK --

Figure 6: Reactive certificate authentication

If the client does not have the desired certificate, it instead sends an Empty Authenticator, as described in Section 5 of [I-D.ietf-tls-exported-authenticator], in a "CERTIFICATE" frame in response to the request, followed by a "USE_CERTIFICATE" frame which
3. Certificates Frames for HTTP/2

The "CERTIFICATE_REQUEST" and "CERTIFICATE_NEEDED" frames are correlated by their "Request-ID" field. Subsequent "CERTIFICATE_NEEDED" frames with the same "Request-ID" value MAY be sent for other streams where the sender is expecting a certificate with the same parameters.

The "CERTIFICATE", and "USE_CERTIFICATE" frames are correlated by their "Cert-ID" field. Subsequent "USE_CERTIFICATE" frames with the same "Cert-ID" MAY be sent in response to other "CERTIFICATE_NEEDED" frames and refer to the same certificate.

"CERTIFICATE_NEEDED" and "USE_CERTIFICATE" frames are correlated by the Stream ID they reference. Unsolicited "USE_CERTIFICATE" frames are not responses to "CERTIFICATE_NEEDED" frames; otherwise, each "USE_CERTIFICATE" frame for a stream is considered to respond to a "CERTIFICATE_NEEDED" frame for the same stream in sequence.

```
+---------+           +---------+
| REQUEST |           | CERT    |
+---------+           +---------+
    | Request-ID          | Cert-ID |
       v                     v
+---------+ Stream ID +---------+
| NEEDED  | ------------->| USE     |
+---------+           +---------+
```

Figure 7: Frame correlation

"Request-ID" and "Cert-ID" are independent and sender-local. The use of the same value by the other peer or in the other context does not imply any correlation between these frames. These values MUST be unique per sender for each space over the lifetime of the connection.

3.1. The CERTIFICATE_NEEDED Frame

The "CERTIFICATE_NEEDED" frame (0xFRAME-TBD1) is sent on stream zero to indicate that the HTTP request on the indicated stream is blocked pending certificate authentication. The frame includes stream ID and
a request identifier which can be used to correlate the stream with a previous "CERTIFICATE_REQUEST" frame sent on stream zero. The "CERTIFICATE_REQUEST" describes the certificate the sender requires to make progress on the stream in question.

![Frame Payload Diagram]

Figure 8: CERTIFICATE_NEEDED frame payload

The "CERTIFICATE_NEEDED" frame contains 6 octets. The first four octets indicate the Stream ID of the affected stream. The following two octets are the authentication request identifier, "Request-ID". A peer that receives a "CERTIFICATE_NEEDED" of any other length MUST treat this as a stream error of type "PROTOCOL_ERROR". Frames with identical request identifiers refer to the same "CERTIFICATE_REQUEST".

A server MAY send multiple "CERTIFICATE_NEEDED" frames for the same stream. If a server requires that a client provide multiple certificates before authorizing a single request, each required certificate MUST be indicated with a separate "CERTIFICATE_NEEDED" frame, each of which MUST have a different request identifier (referencing different "CERTIFICATE_REQUEST" frames describing each required certificate). To reduce the risk of client confusion, servers SHOULD NOT have multiple outstanding "CERTIFICATE_NEEDED" frames for the same stream at any given time.

Clients MUST only send multiple "CERTIFICATE_NEEDED" frames for stream zero. Multiple "CERTIFICATE_NEEDED" frames on any other stream MUST be considered a stream error of type "PROTOCOL_ERROR".

The "CERTIFICATE_NEEDED" frame MUST NOT be sent to a peer which has not advertised support for HTTP-layer certificate authentication.

The "CERTIFICATE_NEEDED" frame MUST NOT reference a stream in the "half-closed (local)" or "closed" states [RFC7540]. A client that receives a "CERTIFICATE_NEEDED" frame for a stream which is not in a valid state SHOULD treat this as a stream error of type "PROTOCOL_ERROR".
3.2. The USE_CERTIFICATE Frame

The "USE_CERTIFICATE" frame (0xFRAME-TBD4) is sent on stream zero to indicate which certificate is being used on a particular request stream.

The "USE_CERTIFICATE" frame defines a single flag:

UNSOLICITED (0x01): Indicates that no "CERTIFICATE_NEEDED" frame has yet been received for this stream.

The payload of the "USE_CERTIFICATE" frame is as follows:

```
0                   1                   2                   3
+---------------------------------------------------------------+
|R|                        Stream ID (31)                       |
+-------------------------------+-------------------------------+
|         [Cert-ID (16)]        |
+-------------------------------+-----------------------------+
```

Figure 9: USE_CERTIFICATE frame payload

The first four octets indicate the Stream ID of the affected stream. The following two octets, if present, contain the two-octet "Cert-ID" of the certificate the sender wishes to use. This MUST be the ID of a certificate for which proof of possession has been presented in a "CERTIFICATE" frame. Recipients of a "USE_CERTIFICATE" frame of any other length MUST treat this as a stream error of type "PROTOCOL_ERROR". Frames with identical certificate identifiers refer to the same certificate chain.

A "USE_CERTIFICATE" frame which omits the Cert-ID refers to the certificate provided at the TLS layer, if any. If no certificate was provided at the TLS layer, the stream should be processed with no authentication, likely returning an authentication-related error at the HTTP level (e.g. 403) for servers or routing the request to a new connection for clients.

The "UNSOLICITED" flag MAY be set by clients on the first "USE_CERTIFICATE" frame referring to a given stream. This permits a client to proactively indicate which certificate should be used when processing a new request. When such an unsolicited indication refers to a request that has not yet been received, servers SHOULD cache the indication briefly in anticipation of the request.

Receipt of more than one unsolicited "USE_CERTIFICATE" frames or an unsolicited "USE_CERTIFICATE" frame which is not the first in
reference to a given stream MUST be treated as a stream error of type "CERTIFICATE_OVERUSED".

Each "USE_CERTIFICATE" frame which is not marked as unsolicited is considered to respond in order to the "CERTIFICATE_NEEDED" frames for the same stream. If a "USE_CERTIFICATE" frame is received for which a "CERTIFICATE_NEEDED" frame has not been sent, this MUST be treated as a stream error of type "CERTIFICATE_OVERUSED".

Receipt of a "USE_CERTIFICATE" frame with an unknown "Cert-ID" MUST result in a stream error of type "PROTOCOL_ERROR".

The referenced certificate chain needs to conform to the requirements expressed in the "CERTIFICATE_REQUEST" to the best of the sender’s ability, or the recipient is likely to reject it as unsuitable despite properly validating the authenticator. If the recipient considers the certificate unsuitable, it MAY at its discretion either return an error at the HTTP semantic layer, or respond with a stream error [RFC7540] on any stream where the certificate is used. Section 4 defines certificate-related error codes which might be applicable.

3.3. The CERTIFICATE_REQUEST Frame

The "CERTIFICATE_REQUEST" frame (id=0xFRAME-TBD2) provides a exported authenticator request message from the TLS layer that specifies a desired certificate. This describes the certificate the sender wishes to have presented.

The "CERTIFICATE_REQUEST" frame SHOULD NOT be sent to a peer which has not advertised support for HTTP-layer certificate authentication.

The "CERTIFICATE_REQUEST" frame MUST be sent on stream zero. A "CERTIFICATE_REQUEST" frame received on any other stream MUST be rejected with a stream error of type "PROTOCOL_ERROR".

```
+-------------------------------+-------------------------------+
|        Request-ID (16)        |          Request (?)        ...
|-------------------------------|-------------------------------|
```

Figure 10: CERTIFICATE_REQUEST frame payload

The frame contains the following fields:
Request-ID: "Request-ID" is a 16-bit opaque identifier used to correlate subsequent certificate-related frames with this request. The identifier MUST be unique in the session for the sender.

Request: An exported authenticator request, generated using the "request" API described in [I-D.ietf-tls-exported-authenticator]. See Section 3.4.1 for more details on the input to this API.

3.3.1. Exported Authenticator Request Characteristics

The Exported Authenticator "request" API defined in [I-D.ietf-tls-exported-authenticator] takes as input a set of desired certificate characteristics and a "certificate_request_context", which needs to be unpredictable. When generating exported authenticators for use with this extension, the "certificate_request_context" MUST contain both the two-octet Request-ID as well as at least 96 bits of additional entropy.

The TLS library on the authenticating peer will provide mechanisms to select an appropriate certificate to respond to the transported request. TLS libraries on servers MUST be able to recognize the "server_name" extension ([RFC6066]) at a minimum. Clients MUST always specify the desired origin using this extension, though other extensions MAY also be included.

3.4. The CERTIFICATE Frame

The "CERTIFICATE" frame (id=0xFRAME-TBD3) provides a exported authenticator message from the TLS layer that provides a chain of certificates, associated extensions and proves possession of the private key corresponding to the end-entity certificate.

The "CERTIFICATE" frame defines two flags:

TO_BE_CONTINUED (0x01): Indicates that the exported authenticator spans more than one frame.

0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-------------------------------+-------------------------------+
|     Cert-ID (16)               | Authenticator Fragment (*)... |
+-------------------------------+-------------------------------+

Figure 11: CERTIFICATE frame payload

The "Exported Authenticator Fragment" field contains a portion of the opaque data returned from the TLS connection exported authenticator
"authenticate" API. See Section 3.4.1 for more details on the input to this API.

This opaque data is transported in zero or more "CERTIFICATE" frames with the "TO_BE_CONTINUED" flag set, followed by one "CERTIFICATE" frame with the "TO_BE_CONTINUED" flag unset. Each of these frames contains the same "Cert-ID" field, permitting them to be associated with each other. Receipt of any "CERTIFICATE" frame with the same "Cert-ID" following the receipt of a "CERTIFICATE" frame with "TO_BE_CONTINUED" unset MUST be treated as a connection error of type "PROTOCOL_ERROR".

Upon receiving a complete series of "CERTIFICATE" frames, the receiver may validate the Exported Authenticator value by using the exported authenticator API. This returns either an error indicating that the message was invalid, or the certificate chain and extensions used to create the message.

The "CERTIFICATE" frame MUST be sent on stream zero. A "CERTIFICATE" frame received on any other stream MUST be rejected with a stream error of type "PROTOCOL_ERROR".

3.4.1. Exported Authenticator Characteristics

The Exported Authenticator API defined in [I-D.ietf-tls-exported-authenticator] takes as input a request, a set of certificates, and supporting information about the certificate (OCSP, SCT, etc.). The result is an opaque token which is used when generating the "CERTIFICATE" frame.

Upon receipt of a "CERTIFICATE" frame, an endpoint MUST perform the following steps to validate the token it contains: - Using the "get context" API, retrieve the "certificate_request_context" used to generate the authenticator, if any. - Verify that the "certificate_request_context" is either empty (clients only) or contains the Request-ID of a previously-sent "CERTIFICATE_REQUEST" frame. - Use the "validate" API to confirm the validity of the authenticator with regard to the generated request (if any).

Once the authenticator is accepted, the endpoint can perform any other checks for the acceptability of the certificate itself.

4. Indicating Failures During HTTP-Layer Certificate Authentication

Because this draft permits certificates to be exchanged at the HTTP framing layer instead of the TLS layer, several certificate-related errors which are defined at the TLS layer might now occur at the HTTP
framing layer. In this section, those errors are restated and added to the HTTP/2 error code registry.

BAD_CERTIFICATE (0xERROR-TBD1): A certificate was corrupt, contained signatures that did not verify correctly, etc.

UNSUPPORTED_CERTIFICATE (0xERROR-TBD2): A certificate was of an unsupported type or did not contain required extensions

CERTIFICATE_REVOKED (0xERROR-TBD3): A certificate was revoked by its signer

CERTIFICATE_EXPIRED (0xERROR-TBD4): A certificate has expired or is not currently valid

CERTIFICATE_GENERAL (0xERROR-TBD5): Any other certificate-related error

CERTIFICATE_OVERUSED (0xERROR-TBD6): More certificates were used on a request than were requested

As described in [RFC7540], implementations MAY choose to treat a stream error as a connection error at any time. Of particular note, a stream error cannot occur on stream 0, which means that implementations cannot send non-session errors in response to "CERTIFICATE_REQUEST", and "CERTIFICATE" frames. Implementations which do not wish to terminate the connection MAY either send relevant errors on any stream which references the failing certificate in question or process the requests as unauthenticated and provide error information at the HTTP semantic layer.

5. Security Considerations

This mechanism defines an alternate way to obtain server and client certificates other than in the initial TLS handshake. While the signature of exported authenticator values is expected to be equally secure, it is important to recognize that a vulnerability in this code path is at least equal to a vulnerability in the TLS handshake.

5.1. Impersonation

This mechanism could increase the impact of a key compromise. Rather than needing to subvert DNS or IP routing in order to use a compromised certificate, a malicious server now only needs a client to connect to _some_ HTTPS site under its control in order to present the compromised certificate. As recommended in [RFC8336], clients opting not to consult DNS ought to employ some alternative means to increase confidence that the certificate is legitimate.
As noted in the Security Considerations of [I-D.ietf-tls-exported-authenticator], it difficult to formally prove that an endpoint is jointly authoritative over multiple certificates, rather than individually authoritative on each certificate. As a result, clients MUST NOT assume that because one origin was previously colocated with another, those origins will be reachable via the same endpoints in the future. Clients MUST NOT consider previous secondary certificates to be validated after TLS session resumption. However, clients MAY proactively query for previously-presented secondary certificates.

5.2. Fingerprinting

This draft defines a mechanism which could be used to probe servers for origins they support, but opens no new attack versus making repeat TLS connections with different SNI values. Servers SHOULD impose similar denial-of-service mitigations (e.g. request rate limits) to "CERTIFICATE_REQUEST" frames as to new TLS connections.

While the extensions in the "CERTIFICATE_REQUEST" frame permit the sender to enumerate the acceptable Certificate Authorities for the requested certificate, it might not be prudent (either for security or data consumption) to include the full list of trusted Certificate Authorities in every request. Senders, particularly clients, SHOULD send only the extensions that narrowly specify which certificates would be acceptable.

5.3. Denial of Service

Failure to provide a certificate on a stream after receiving "CERTIFICATE_NEEDED" blocks processing, and SHOULD be subject to standard timeouts used to guard against unresponsive peers.

Validating a multitude of signatures can be computationally expensive, while generating an invalid signature is computationally cheap. Implementations will require checks for attacks from this direction. Invalid exported authenticators SHOULD be treated as a session error, to avoid further attacks from the peer, though an implementation MAY instead disable HTTP-layer certificates for the current connection instead.

5.4. Confusion About State

Implementations need to be aware of the potential for confusion about the state of a connection. The presence or absence of a validated certificate can change during the processing of a request, potentially multiple times, as "USE_CERTIFICATE" frames are received. A server that uses certificate authentication needs to be prepared to
reevaluate the authorization state of a request as the set of certificates changes.

Client implementations need to carefully consider the impact of setting the "AUTOMATIC_USE" flag. This flag is a performance optimization, permitting the client to avoid a round-trip on each request where the server checks for certificate authentication. However, once this flag has been sent, the client has zero knowledge about whether the server will use the referenced cert for any future request, or even for an existing request which has not yet completed. Clients MUST NOT set this flag on any certificate which is not appropriate for currently-in-flight requests, and MUST NOT make any future requests on the same connection which they are not willing to have associated with the provided certificate.

6. IANA Considerations

This draft adds entries in three registries.

The HTTP/2 "SETTINGS_HTTP_CERT_AUTH" setting is registered in Section 6.1. Four frame types are registered in Section 6.2. Six error codes are registered in Section 6.3.

6.1. HTTP/2 SETTINGS_HTTP_CERT_AUTH Setting

The SETTINGS_HTTP_CERT_AUTH setting is registered in the "HTTP/2 Settings" registry established in [RFC7540].

Name: SETTINGS_HTTP_CERT_AUTH

Code: 0xSETTING-TBD

Initial Value: 0

Specification: This document.

6.2. New HTTP/2 Frames

Four new frame types are registered in the "HTTP/2 Frame Types" registry established in [RFC7540]. The entries in the following table are registered by this document.
<table>
<thead>
<tr>
<th>Frame Type</th>
<th>Code</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>CERTIFICATE_NEEDED</td>
<td>0xFRAME-TBD1</td>
<td>Section 3.1</td>
</tr>
<tr>
<td>CERTIFICATE_REQUEST</td>
<td>0xFRAME-TBD2</td>
<td>Section 3.3</td>
</tr>
<tr>
<td>CERTIFICATE</td>
<td>0xFRAME-TBD3</td>
<td>Section 3.4</td>
</tr>
<tr>
<td>USE_CERTIFICATE</td>
<td>0xFRAME-TBD4</td>
<td>Section 3.2</td>
</tr>
</tbody>
</table>

### 6.3. New HTTP/2 Error Codes

Six new error codes are registered in the "HTTP/2 Error Code" registry established in [RFC7540]. The entries in the following table are registered by this document.

<table>
<thead>
<tr>
<th>Name</th>
<th>Code</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAD_CERTIFICATE</td>
<td>0xERROR-TBD1</td>
<td>Section 4</td>
</tr>
<tr>
<td>UNSUPPORTED_CERTIFICATE</td>
<td>0xERROR-TBD2</td>
<td>Section 4</td>
</tr>
<tr>
<td>CERTIFICATE_REVOKED</td>
<td>0xERROR-TBD3</td>
<td>Section 4</td>
</tr>
<tr>
<td>CERTIFICATE_EXPIRED</td>
<td>0xERROR-TBD4</td>
<td>Section 4</td>
</tr>
<tr>
<td>CERTIFICATE_GENERAL</td>
<td>0xERROR-TBD5</td>
<td>Section 4</td>
</tr>
<tr>
<td>CERTIFICATE_OVERUSED</td>
<td>0xERROR-TBD6</td>
<td>Section 4</td>
</tr>
</tbody>
</table>

### 7. References

#### 7.1. Normative References

- [I-D.ietf-tls-exported-authenticator]

- [I-D.ietf-tls-tls13]
7.2. Informative References


7.3. URIs

[1] https://lists.w3.org/Archives/Public/ietf-http-wg/

Appendix A. Change Log

*RFC Editor’s Note:* Please remove this section prior to publication of a final version of this document.

A.1. Since draft-ietf-httpbis-http2-secondary-certs-01:

- Clients can send "CERTIFICATE_NEEDED" for stream 0 rather than speculatively reserving a stream for an origin.
- Use SETTINGS to disable when a TLS-terminating proxy is present (#617, #651)

A.2. Since draft-ietf-httpbis-http2-secondary-certs-00:

- All frames sent on stream zero; replaced "AUTOMATIC_USE" on "CERTIFICATE" with "UNSOLICITED" on "USE_CERTIFICATE". (#482, #566)
- Use Exported Requests from the TLS Exported Authenticators draft; eliminate facilities for expressing certificate requirements in "CERTIFICATE_REQUEST" frame. (#481)

A.3. Since draft-bishop-httpbis-http2-additional-certs-05:

- Adopted as draft-ietf-httpbis-http2-secondary-certs

Acknowledgements

Eric Rescorla pointed out several failings in an earlier revision. Andrei Popov contributed to the TLS considerations.

A substantial portion of Mike’s work on this draft was supported by Microsoft during his employment there.

Authors’ Addresses

Mike Bishop
Akamai
Email: mbishop@evequefou.be

Nick Sullivan
Cloudflare
Email: nick@cloudflare.com
HTTP Random Access and Live Content
draft-ietf-httpbis-rand-access-live-03

Abstract

To accommodate byte range requests for content that has data appended over time, this document defines semantics that allow a HTTP client and server to perform byte-range GET and HEAD requests that start at an arbitrary byte offset within the representation and ends at an indeterminate offset.

Editorial Note (To be removed by RFC Editor before publication)

Discussion of this draft takes place on the HTTPBIS working group mailing list (ietf-http-wg@w3.org), which is archived at <https://lists.w3.org/Archives/Public/ietf-http-wg/>.

Working Group information can be found at <http://httpwg.github.io/>; source code and issues list for this draft can be found at <https://github.com/httpwg/http-extensions/labels/rand-access-live>.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 21, 2018.
1. Introduction

Some Hypertext Transfer Protocol (HTTP) clients use byte-range requests (Range requests using the "bytes" Range Unit) to transfer select portions of large representations ([RFC7233]). And in some cases large representations require content to be continuously or periodically appended - such as representations consisting of live audio or video sources, blockchain databases, and log files. Clients cannot access the appended/live content using a Range request with the bytes range unit using the currently defined byte-range semantics without accepting performance or behavior sacrifices which are not acceptable for many applications.
For instance, HTTP clients have the ability to access appended content on an indeterminate-length resource by transferring the entire representation from the beginning and continuing to read the appended content as it’s made available. Obviously, this is highly inefficient for cases where the representation is large and only the most recently appended content is needed by the client.

Alternatively, clients can also access appended content by sending periodic open-ended bytes Range requests using the last-known end byte position as the range start. Performing low-frequency periodic bytes Range requests in this fashion (polling) introduces latency since the client will necessarily be somewhat behind the aggregated content — mimicking the behavior (and latency) of segmented content representations such as "HTTP Live Streaming" (HLS, [RFC8216]) or "Dynamic Adaptive Streaming over HTTP" (MPEG-DASH, [DASH]). And while performing these Range requests at higher frequency can reduce this latency, it also incurs more processing overhead and HTTP exchanges as many of the requests will return no content — since content is usually aggregated in groups of bytes (e.g. a video frame, audio sample, block, or log entry).

This document describes a usage model for range requests which enables efficient retrieval of representations that are appended to over time by using large values and associated semantics for communicating range end positions. This model allows representations to be progressively delivered by servers as new content is added. It also ensures compatibility with servers and intermediaries that don’t support this technique.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

1.2. Notational Conventions

This document cites productions in Augmented Backus-Naur Form (ABNF) productions from [RFC7233], using the notation defined in [RFC5234].

2. Performing Range requests on Random-Access Aggregating ("live") Content

This document recommends a two-step process for accessing resources that have indeterminate length representations.

Two steps are necessary because of limitations with the Range request header fields and the Content-Range response header fields. A server
cannot know from a range request that a client wishes to receive a response that does not have a definite end. More critically, the header fields do not allow the server to signal that a resource has indeterminate length without also providing a fixed portion of the resource.

A client first learns that the resource has a representation of indeterminate length by requesting a range of the resource. The server responds with the range that is available, but indicates that the length of the representation is unknown using the existing Content-Range syntax. See Section 2.1 for details and examples.

Once the client knows the resource has indeterminate length, it can request a range with a very large end position from the resource. The client chooses an explicit end value larger than can be transferred in the foreseeable term. A server which supports range requests of indeterminate length signals its understanding of the client’s indeterminate range request by indicating that the range it is providing has a range end that exactly matches the client’s requested range end rather than a range that is bounded by what is currently available. See Section 2.2 for details.

2.1. Establishing the Randomly Accessible Byte Range

Establishing if a representation is continuously aggregating ("live") and determining the randomly-accessible byte range can both be determined using the existing definition for an open-ended byte-range request. Specifically, Section 2.1 of [RFC7233] defines a byte-range request of the form:

\[
\text{byte-range-spec} = \text{first-byte-pos} \ "-" \ [ \ \text{last-byte-pos} \ ]
\]

which allows a client to send a HEAD request with a first-byte-pos and leave last-byte-pos absent. A server that receives a satisfiable byte-range request (with first-byte-pos smaller than the current representation length) may respond with a 206 status code (Partial Content) with a Content-Range header field indicating the currently satisfiable byte range. For example:

```
HEAD /resource HTTP/1.1
Host: example.com
Range: bytes=0-
```

returns a response of the form:

```
HTTP/1.1 206 Partial Content
Content-Range: bytes 0-1234567/*
```
from the server indicating that (1) the complete representation length is unknown (via the "*" in place of the complete-length field) and (2) that only bytes 0-1234567 were accessible at the time the request was processed by the server. The client can infer from this response that bytes 0-1234567 of the representation can be requested and returned in a timely fashion (the bytes are immediately available).

2.2. Byte-Range Requests Beyond the Randomly Accessible Byte Range

Once a client has determined that a representation has an indeterminate length and established the byte range that can be accessed, it may want to perform a request with a start position within the randomly-accessible content range and an end position at an indefinite "live" point - a point where the byte-range GET request is fulfilled on-demand as the content is aggregated.

For example, for a large video asset, a client may wish to start a content transfer from the video "key" frame immediately before the point of aggregation and continue the content transfer indefinitely as content is aggregated - in order to support low-latency startup of a live video stream.

Unlike a byte-range Range request, a byte-range Content-Range response header field cannot be "open ended", per Section 4.2 of [RFC7233]:

```
byte-content-range = bytes-unit SP
                   ( byte-range-resp / unsatisfied-range )
byte-range-resp   = byte-range "/" ( complete-length / "*" )
byte-range        = first-byte-pos "-" last-byte-pos
unsatisfied-range = "/" complete-length
complete-length   = 1*DIGIT
```

Specifically, last-byte-pos is required in byte-range. So in order to preserve interoperability with existing HTTP clients, servers, proxies, and caches, this document proposes a mechanism for a client to indicate support for handling an indeterminate-length byte-range response, and a mechanism for a server to indicate if/when it’s providing a indeterminate-length response.

A client can indicate support for handling indeterminate-length byte-range responses by providing a Very Large Value for the last-byte-pos in the byte-range request. For example, a client can perform a byte-range GET request of the form:
GET /resource HTTP/1.1
Host: example.com
Range: bytes=1230000-999999999999

where the last-byte-pos in the Request is much larger than the last-byte-pos returned in response to an open-ended byte-range HEAD request, as described above.

In response, a server may indicate that it is supplying a continuously aggregating ("live") response by supplying the client request’s last-byte-pos in the Content-Range response header field.

For example:

GET /resource HTTP/1.1
Host: example.com
Range: bytes=1230000-999999999999

returns

HTTP/1.1 206 Partial Content
Content-Range: bytes 1230000-999999999999/*

from the server to indicate that the response will start at byte 1230000 and continues indefinitely to include all aggregated content, as it becomes available.

A server that doesn’t support or supply a continuously aggregating ("live") response will supply the currently satisfiable byte range, as it would with an open-ended byte request.

For example:

GET /resource HTTP/1.1
Host: example.com
Range: bytes=1230000-999999999999

will return

HTTP/1.1 206 Partial Content
Content-Range: bytes 1230000-1234567/*

from the server to indicate that the response will start at byte 1230000 and end at byte 1234567 and will not include any aggregated...
content. This is the response expected from a typical HTTP server – one that doesn’t support byte-range requests on aggregating content.

A client that doesn’t receive a response indicating it is continuously aggregating must use other means to access aggregated content (e.g. periodic byte-range polling).

A server that does return a continuously aggregating ("live") response should return data using chunked transfer coding and not provide a Content-Length header field. A 0-length chunk indicates the end of the transfer, per Section 4.1 of [RFC7230].

3. Other Applications of Random-Access Aggregating Content

3.1. Requests Starting at the Aggregation ("Live") Point

A client that wishes to only receive newly-aggregated portions of a resource (i.e., start at the "live" point), can use a HEAD request to learn what range the server has currently available and initiate an indeterminate-length transfer. For example:

```
HEAD /resource HTTP/1.1
Host: example.com
Range: bytes=0-
```

With the Content-Range response header field indicating the range (or ranges) available. For example:

```
206 Partial Content
Content-Range: bytes 0-1234567/*
```

The client can then issue a request for a range starting at the end value (using a very large value for the end of a range) and receive only new content.

```
GET /resource HTTP/1.1
Host: example.com
Range: bytes=1234567-9999999999999
```

with a server returning a Content-Range response indicating that an indeterminate-length response body will be provided

```
206 Partial Content
Content-Range: bytes 1234567-9999999999999/*
```
3.2. Shift Buffer Representations

Some representations lend themselves to front-end content removal in addition to aggregation. While still supporting random access, representations of this type have a portion at the beginning (the "0" end) of the randomly-accessible region that become inaccessible over time. Examples of this kind of representation would be an audio-video time-shift buffer or a rolling log file.

For example a Range request containing:

```plaintext
HEAD /resource HTTP/1.1
Host: example.com
Range: bytes=0-
```

returns

```plaintext
206 Partial Content
Content-Range: bytes 1000000-1234567/*
```

indicating that the first 1000000 bytes were not accessible at the time the HEAD request was processed. Subsequent HEAD requests could return:

```plaintext
Content-Range: bytes 1000000-1234567/*
Content-Range: bytes 1010000-1244567/*
Content-Range: bytes 1020000-1254567/*
```

Note though that the difference between the first-byte-pos and last-byte-pos need not be constant.

The client could then follow-up with a GET Range request containing

```plaintext
GET /resource HTTP/1.1
Host: example.com
Range: bytes=1020000-999999999999
```

with the server returning

```plaintext
206 Partial Content
Content-Range: bytes 1020000-999999999999/*
```
with the response body returning bytes 1020000-1254567 immediately and aggregated ("live") data being returned as the content is aggregated.

A server that doesn’t support or supply a continuously aggregating ("live") response will supply the currently satisfiable byte range, as it would with an open-ended byte request.

For example:

```
GET /resource HTTP/1.1
Host: example.com
Range: bytes=0-999999999999
```

will return

```
HTTP/1.1 206 Partial Content
Content-Range: bytes 1020000-1254567/*
```

from the server to indicate that the response will start at byte 1020000, end at byte 1254567, and will not include any aggregated content. This is the response expected from a typical HTTP server—one that doesn’t support byte-range requests on aggregating content.

Note that responses to GET requests against shift-buffer representations using Range can be cached by intermediaries, since the Content-Range response header indicates which portion of the representation is being returned in the response body. However GET requests without a Range header cannot be cached since the first byte of the response body can vary from request to request. To ensure Range-less GET requests against shift-buffer representations are not cached, servers hosting a shift-buffer representation should either not return a 200-level response (e.g. sending a 300-level redirect response with a URI that represents the current start of the shift-buffer) or indicate the response is non-cacheable. See HTTP Caching ([RFC7234]) for details on HTTP cache control.

4. IANA Considerations

This document has no actions for IANA.

5. Security Considerations

One potential issue with this recommendation is related to the use of very-large last-byte-pos values. Some client and server implementations may not be prepared to deal with byte position values of 2^^63 and beyond. So in applications where there’s no expectation
that the representation will ever exceed $2^{^63}$, a value smaller than this value should be used as the Very Large last-byte-pos in a byte-seek request or content-range response. Also, some implementations (e.g. JavaScript-based clients and servers) are not able to represent all values beyond $2^{^53}$. So similarly, if there’s no expectation that a representation will ever exceed $2^{^53}$ bytes, values smaller than this limit should be used for the last-byte-pos in byte-range requests.

6. References

6.1. Normative References


6.2. Informative References


Acknowledgements


Authors’ Addresses

Craig Pratt
Portland, OR  97229
US

Email: pratt@acm.org

Darshak Thakore
CableLabs
858 Coal Creek Circle
Louisville, CO  80027
US

Email: d.thakore@cablelabs.com

Barbara Stark
AT&T
Atlanta, GA
US

Email: barbara.stark@att.com
Abstract

This document defines the HTTP Cookie and Set-Cookie header fields. These header fields can be used by HTTP servers to store state (called cookies) at HTTP user agents, letting the servers maintain a stateful session over the mostly stateless HTTP protocol. Although cookies have many historical infelicities that degrade their security and privacy, the Cookie and Set-Cookie header fields are widely used on the Internet. This document obsoletes RFC 6265.

Note to Readers

Discussion of this draft takes place on the HTTP working group mailing list (ietf-http-wg@w3.org), which is archived at https://lists.w3.org/Archives/Public/ietf-http-wg/.

Working Group information can be found at http://httpwg.github.io/; source code and issues list for this draft can be found at https://github.com/httpwg/http-extensions/labels/6265bis.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on February 8, 2018.
Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November 10, 2008. The person(s) controlling the copyright in some of this material may not have granted the IETF Trust the right to allow modifications of such material outside the IETF Standards Process. Without obtaining an adequate license from the person(s) controlling the copyright in such materials, this document may not be modified outside the IETF Standards Process, and derivative works of it may not be created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.

Table of Contents

1. Introduction ........................................ 4
2. Conventions ........................................... 5
   2.1. Conformance Criteria .................................. 5
   2.2. Syntax Notation ..................................... 5
   2.3. Terminology ....................................... 6
3. Overview .............................................. 7
   3.1. Examples .......................................... 7
4. Server Requirements ................................. 9
   4.1. Set-Cookie ........................................ 9
      4.1.1. Syntax ......................................... 9
      4.1.2. Semantics (Non-Normative) ..................... 11
      4.1.3. Cookie Name Prefixes ............................ 14
   4.2. Cookie ........................................... 15
      4.2.1. Syntax ......................................... 15
      4.2.2. Semantics ..................................... 16
5. User Agent Requirements ........................... 16
6.1. Subcomponent Algorithms ........................ 16
   6.1.1. Dates ........................................... 16
   6.1.2. Canonicalized Host Names ....................... 18
1. Introduction

This document defines the HTTP Cookie and Set-Cookie header fields. Using the Set-Cookie header field, an HTTP server can pass name/value pairs and associated metadata (called cookies) to a user agent. When the user agent makes subsequent requests to the server, the user agent uses the metadata and other information to determine whether to return the name/value pairs in the Cookie header.

Although simple on their surface, cookies have a number of complexities. For example, the server indicates a scope for each cookie when sending it to the user agent. The scope indicates the maximum amount of time in which the user agent should return the cookie, the servers to which the user agent should return the cookie, and the URI schemes for which the cookie is applicable.

For historical reasons, cookies contain a number of security and privacy infelicities. For example, a server can indicate that a given cookie is intended for "secure" connections, but the Secure attribute does not provide integrity in the presence of an active network attacker. Similarly, cookies for a given host are shared across all the ports on that host, even though the usual "same-origin policy" used by web browsers isolates content retrieved via different ports.

There are two audiences for this specification: developers of cookie-generating servers and developers of cookie-consuming user agents.

To maximize interoperability with user agents, servers SHOULD limit themselves to the well-behaved profile defined in Section 4 when generating cookies.

User agents MUST implement the more liberal processing rules defined in Section 5, in order to maximize interoperability with existing servers that do not conform to the well-behaved profile defined in Section 4.

This document specifies the syntax and semantics of these headers as they are actually used on the Internet. In particular, this document does not create new syntax or semantics beyond those in use today. The recommendations for cookie generation provided in Section 4 represent a preferred subset of current server behavior, and even the more liberal cookie processing algorithm provided in Section 5 does not recommend all of the syntactic and semantic variations in use today. Where some existing software differs from the recommended protocol in significant ways, the document contains a note explaining the difference.
This document obsoletes [RFC6265].

2. Conventions

2.1. Conformance Criteria

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

Requirements phrased in the imperative as part of algorithms (such as "strip any leading space characters" or "return false and abort these steps") are to be interpreted with the meaning of the key word ("MUST", "SHOULD", "MAY", etc.) used in introducing the algorithm.

Conformance requirements phrased as algorithms or specific steps can be implemented in any manner, so long as the end result is equivalent. In particular, the algorithms defined in this specification are intended to be easy to understand and are not intended to be performant.

2.2. Syntax Notation

This specification uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234].

The following core rules are included by reference, as defined in [RFC5234], Appendix B.1: ALPHA (letters), CR (carriage return), CRLF (CR LF), CTLs (controls), DIGIT (decimal 0-9), DQUOTE (double quote), HEXDIG (hexadecimal 0-9/A-F/a-f), LF (line feed), NUL (null octet), OCTET (any 8-bit sequence of data except NUL), SP (space), HTAB (horizontal tab), CHAR (any [USASCII] character), VCHAR (any visible [USASCII] character), and WSP (whitespace).

The OWS (optional whitespace) rule is used where zero or more linear whitespace characters MAY appear:

\[
\text{OWS} = \ast ( \text{obs-fold \ WSP } )
\]

\[
\text{obs-fold} = \text{CRLF}
\]

OWS SHOULD either not be produced or be produced as a single SP character.
2.3. Terminology

The terms "user agent", "client", "server", "proxy", and "origin server" have the same meaning as in the HTTP/1.1 specification ([RFC2616], Section 1.3).

The request-host is the name of the host, as known by the user agent, to which the user agent is sending an HTTP request or from which it is receiving an HTTP response (i.e., the name of the host to which it sent the corresponding HTTP request).

The term request-uri is defined in Section 5.1.2 of [RFC2616].

Two sequences of octets are said to case-insensitively match each other if and only if they are equivalent under the i;ascii-casemap collation defined in [RFC4790].

The term string means a sequence of non-NUL octets.

The terms "active document", "ancestor browsing context", "browsing context", "dedicated worker", "Document", "WorkerGlobalScope", "sandboxed origin browsing context flag", "parent browsing context", "shared worker", "the worker’s Documents", "nested browsing context", and "top-level browsing context" are defined in [HTML].

"Service Workers" are defined in the Service Workers specification [SERVICE-WORKERS].

The term "origin", the mechanism of deriving an origin from a URI, and the "the same" matching algorithm for origins are defined in [RFC6454].

"Safe" HTTP methods include "GET", "HEAD", "OPTIONS", and "TRACE", as defined in Section 4.2.1 of [RFC7231].

The term "public suffix" is defined in a note in Section 5.3 of [RFC6265] as "a domain that is controlled by a public registry", and are also known as "effective top-level domains" (eTLDs). For example, "example.com"’s public suffix is "com". User agents SHOULD use an up-to-date public suffix list, such as the one maintained by Mozilla at [PSL].

An origin’s "registered domain" is the origin’s host’s public suffix plus the label to its left. That is, for "https://www.example.com", the public suffix is "com", and the registered domain is "example.com". This concept is defined more rigorously in [PSL], and is also known as "effective top-level domain plus one" (eTLD+1).
The term "request", as well as a request’s "client", "current url", "method", and "target browsing context", are defined in [FETCH].

3. Overview

This section outlines a way for an origin server to send state information to a user agent and for the user agent to return the state information to the origin server.

To store state, the origin server includes a Set-Cookie header in an HTTP response. In subsequent requests, the user agent returns a Cookie request header to the origin server. The Cookie header contains cookies the user agent received in previous Set-Cookie headers. The origin server is free to ignore the Cookie header or use its contents for an application-defined purpose.

Origin servers MAY send a Set-Cookie response header with any response. User agents MAY ignore Set-Cookie headers contained in responses with 100-level status codes but MUST process Set-Cookie headers contained in other responses (including responses with 400- and 500-level status codes). An origin server can include multiple Set-Cookie header fields in a single response. The presence of a Cookie or a Set-Cookie header field does not preclude HTTP caches from storing and reusing a response.

Origin servers SHOULD NOT fold multiple Set-Cookie header fields into a single header field. The usual mechanism for folding HTTP headers fields (i.e., as defined in [RFC2616]) might change the semantics of the Set-Cookie header field because the %x2C ("," character is used by Set-Cookie in a way that conflicts with such folding.

3.1. Examples

Using the Set-Cookie header, a server can send the user agent a short string in an HTTP response that the user agent will return in future HTTP requests that are within the scope of the cookie. For example, the server can send the user agent a "session identifier" named SID with the value 31d4d96e407aad42. The user agent then returns the session identifier in subsequent requests.

== Server -> User Agent ==

Set-Cookie: SID=31d4d96e407aad42

== User Agent -> Server ==

Cookie: SID=31d4d96e407aad42
The server can alter the default scope of the cookie using the Path and Domain attributes. For example, the server can instruct the user agent to return the cookie to every path and every subdomain of example.com.

== Server -> User Agent ==
Set-Cookie: SID=31d4d96e407aad42; Path=/; Domain=example.com

== User Agent -> Server ==
Cookie: SID=31d4d96e407aad42

As shown in the next example, the server can store multiple cookies at the user agent. For example, the server can store a session identifier as well as the user’s preferred language by returning two Set-Cookie header fields. Notice that the server uses the Secure and HttpOnly attributes to provide additional security protections for the more sensitive session identifier (see Section 4.1.2).

== Server -> User Agent ==
Set-Cookie: SID=31d4d96e407aad42; Path=/; Secure; HttpOnly
Set-Cookie: lang=en-US; Path=/; Domain=example.com

== User Agent -> Server ==
Cookie: SID=31d4d96e407aad42; lang=en-US

Notice that the Cookie header above contains two cookies, one named SID and one named lang. If the server wishes the user agent to persist the cookie over multiple "sessions" (e.g., user agent restarts), the server can specify an expiration date in the Expires attribute. Note that the user agent might delete the cookie before the expiration date if the user agent’s cookie store exceeds its quota or if the user manually deletes the server’s cookie.

== Server -> User Agent ==

== User Agent -> Server ==
Cookie: SID=31d4d96e407aad42; lang=en-US

Finally, to remove a cookie, the server returns a Set-Cookie header with an expiration date in the past. The server will be successful in removing the cookie only if the Path and the Domain attribute in
the Set-Cookie header match the values used when the cookie was created.

== Server -> User Agent ==
Set-Cookie: lang=; Expires=Sun, 06 Nov 1994 08:49:37 GMT

== User Agent -> Server ==
Cookie: SID=31d4d96e407aad42

4. Server Requirements

This section describes the syntax and semantics of a well-behaved profile of the Cookie and Set-Cookie headers.

4.1. Set-Cookie

The Set-Cookie HTTP response header is used to send cookies from the server to the user agent.

4.1.1. Syntax

Informally, the Set-Cookie response header contains the header name "Set-Cookie" followed by a ":" and a cookie. Each cookie begins with a name-value-pair, followed by zero or more attribute-value pairs. Servers SHOULD NOT send Set-Cookie headers that fail to conform to the following grammar:
set-cookie-header = "Set-Cookie:" SP set-cookie-string
set-cookie-string = cookie-pair *( ";" SP cookie-av )
cookie-pair = cookie-name "=" cookie-value
cookie-name = token
cookie-value = *cookie-octet / ( DQUOTE *cookie-octet DQUOTE )
cookie-octet = %x21 / %x23-2B / %x2D-3A / %x3C-5B / %x5D-7E
; US-ASCII characters excluding CTLs,
; whitespace DQUOTE, comma, semicolon,
; and backslash
token = <token, defined in [RFC2616], Section 2.2>
cookie-av = expires-av / max-age-av / domain-av /
            path-av / secure-av / httponly-av /
            samesite-av / extension-av
expires-av = "Expires=" sane-cookie-date
sane-cookie-date = <rfc1123-date, defined in [RFC2616], Section 3.3.1>
max-age-av = "Max-Age=" non-zero-digit *DIGIT
; In practice, both expires-av and max-age-av
; are limited to dates representable by the
; user agent.
non-zero-digit = %x31-39
; digits 1 through 9
domain-av = "Domain=" domain-value
domain-value = <subdomain>
; defined in [RFC1034], Section 3.5, as
; enhanced by [RFC1123], Section 2.1
path-av = "Path=" path-value
path-value = *av-octet
secure-av = "Secure"
httponly-av = "HttpOnly"
samesite-av = "SameSite=" samesite-value
samesite-value = "Strict" / "Lax"
extension-av = *av-octet
av-octet = %x20-3A / %x3C-7E
; any CHAR except CTLs or ";

Note that some of the grammatical terms above reference documents
that use different grammatical notations than this document (which
uses ABNF from [RFC5234]).

The semantics of the cookie-value are not defined by this document.

To maximize compatibility with user agents, servers that wish to
store arbitrary data in a cookie-value SHOULD encode that data, for
example, using Base64 [RFC4648].
Per the grammar above, the cookie-value MAY be wrapped in DQUOTE characters. Note that in this case, the initial and trailing DQUOTE characters are not stripped. They are part of the cookie-value, and will be included in Cookie headers sent to the server.

The portions of the set-cookie-string produced by the cookie-av term are known as attributes. To maximize compatibility with user agents, servers SHOULD NOT produce two attributes with the same name in the same set-cookie-string. (See Section 5.4 for how user agents handle this case.)

Servers SHOULD NOT include more than one Set-Cookie header field in the same response with the same cookie-name. (See Section 5.3 for how user agents handle this case.)

If a server sends multiple responses containing Set-Cookie headers concurrently to the user agent (e.g., when communicating with the user agent over multiple sockets), these responses create a "race condition" that can lead to unpredictable behavior.

NOTE: Some existing user agents differ in their interpretation of two-digit years. To avoid compatibility issues, servers SHOULD use the rfc1123-date format, which requires a four-digit year.

NOTE: Some user agents store and process dates in cookies as 32-bit UNIX time_t values. Implementation bugs in the libraries supporting time_t processing on some systems might cause such user agents to process dates after the year 2038 incorrectly.

4.1.2. Semantics (Non-Normative)

This section describes simplified semantics of the Set-Cookie header. These semantics are detailed enough to be useful for understanding the most common uses of cookies by servers. The full semantics are described in Section 5.

When the user agent receives a Set-Cookie header, the user agent stores the cookie together with its attributes. Subsequently, when the user agent makes an HTTP request, the user agent includes the applicable, non-expired cookies in the Cookie header.

If the user agent receives a new cookie with the same cookie-name, domain-value, and path-value as a cookie that it has already stored, the existing cookie is evicted and replaced with the new cookie. Notice that servers can delete cookies by sending the user agent a new cookie with an Expires attribute with a value in the past.
Unless the cookie’s attributes indicate otherwise, the cookie is returned only to the origin server (and not, for example, to any subdomains), and it expires at the end of the current session (as defined by the user agent). User agents ignore unrecognized cookie attributes (but not the entire cookie).

4.1.2.1. The Expires Attribute

The Expires attribute indicates the maximum lifetime of the cookie, represented as the date and time at which the cookie expires. The user agent is not required to retain the cookie until the specified date has passed. In fact, user agents often evict cookies due to memory pressure or privacy concerns.

4.1.2.2. The Max-Age Attribute

The Max-Age attribute indicates the maximum lifetime of the cookie, represented as the number of seconds until the cookie expires. The user agent is not required to retain the cookie for the specified duration. In fact, user agents often evict cookies due to memory pressure or privacy concerns.

NOTE: Some existing user agents do not support the Max-Age attribute. User agents that do not support the Max-Age attribute ignore the attribute.

If a cookie has both the Max-Age and the Expires attribute, the Max-Age attribute has precedence and controls the expiration date of the cookie. If a cookie has neither the Max-Age nor the Expires attribute, the user agent will retain the cookie until "the current session is over" (as defined by the user agent).

4.1.2.3. The Domain Attribute

The Domain attribute specifies those hosts to which the cookie will be sent. For example, if the value of the Domain attribute is "example.com", the user agent will include the cookie in the Cookie header when making HTTP requests to example.com, www.example.com, and www.corp.example.com. (Note that a leading %x2E ("."), if present, is ignored even though that character is not permitted, but a trailing %x2E ("."), if present, will cause the user agent to ignore the attribute.) If the server omits the Domain attribute, the user agent will return the cookie only to the origin server.

WARNING: Some existing user agents treat an absent Domain attribute as if the Domain attribute were present and contained the current host name. For example, if example.com returns a Set-Cookie header
without a Domain attribute, these user agents will erroneously send
the cookie to www.example.com as well.

The user agent will reject cookies unless the Domain attribute
specifies a scope for the cookie that would include the origin
server. For example, the user agent will accept a cookie with a
Domain attribute of "example.com" or of "foo.example.com" from
foo.example.com, but the user agent will not accept a cookie with a
Domain attribute of "bar.example.com" or of "baz.foo.example.com".

NOTE: For security reasons, many user agents are configured to reject
Domain attributes that correspond to "public suffixes". For example,
some user agents will reject Domain attributes of "com" or "co.uk".
(See Section 5.4 for more information.)

4.1.2.4. The Path Attribute

The scope of each cookie is limited to a set of paths, controlled by
the Path attribute. If the server omits the Path attribute, the user
agent will use the "directory" of the request-uri’s path component as
the default value. (See Section 5.1.4 for more details.)

The user agent will include the cookie in an HTTP request only if the
path portion of the request-uri matches (or is a subdirectory of) the
cookie’s Path attribute, where the %x2F ("/") character is
interpreted as a directory separator.

Although seemingly useful for isolating cookies between different
paths within a given host, the Path attribute cannot be relied upon
for security (see Section 8).

4.1.2.5. The Secure Attribute

The Secure attribute limits the scope of the cookie to "secure"
channels (where "secure" is defined by the user agent). When a
cookie has the Secure attribute, the user agent will include the
cookie in an HTTP request only if the request is transmitted over a
secure channel (typically HTTP over Transport Layer Security (TLS)
[RFC2818]).

Although seemingly useful for protecting cookies from active network
attackers, the Secure attribute protects only the cookie’s
confidentiality. An active network attacker can overwrite Secure
cookies from an insecure channel, disrupting their integrity (see
Section 8.6 for more details).
4.1.2.6. The HttpOnly Attribute

The HttpOnly attribute limits the scope of the cookie to HTTP requests. In particular, the attribute instructs the user agent to omit the cookie when providing access to cookies via "non-HTTP" APIs (such as a web browser API that exposes cookies to scripts).

Note that the HttpOnly attribute is independent of the Secure attribute: a cookie can have both the HttpOnly and the Secure attribute.

4.1.2.7. The SameSite Attribute

The "SameSite" attribute limits the scope of the cookie such that it will only be attached to requests if those requests are same-site, as defined by the algorithm in Section 5.2. For example, requests for "https://example.com/sekrit-image" will attach same-site cookies if and only if initiated from a context whose "site for cookies" is "example.com".

If the "SameSite" attribute’s value is "Strict", the cookie will only be sent along with "same-site" requests. If the value is "Lax", the cookie will be sent with same-site requests, and with "cross-site" top-level navigations, as described in Section 5.3.7.1. If the "SameSite" attribute’s value is neither of these, the cookie will be ignored.

4.1.3. Cookie Name Prefixes

Section 8.5 and Section 8.6 of this document spell out some of the drawbacks of cookies’ historical implementation. In particular, it is impossible for a server to have confidence that a given cookie was set with a particular set of attributes. In order to provide such confidence in a backwards-compatible way, two common sets of requirements can be inferred from the first few characters of the cookie’s name.

The normative requirements for the prefixes described below are detailed in the storage model algorithm defined in Section 5.4.

4.1.3.1. The "__Secure-" Prefix

If a cookie’s name begins with a case-sensitive match for the string "__Secure-", then the cookie will have been set with a "Secure" attribute.

For example, the following "Set-Cookie" header would be rejected by a conformant user agent, as it does not have a "Secure" attribute.
Set-Cookie: __Secure-SID=12345; Domain=example.com

Whereas the following "Set-Cookie" header would be accepted:
Set-Cookie: __Secure-SID=12345; Domain=example.com; Secure

4.1.3.2. The "__Host-" Prefix

If a cookie’s name begins with a case-sensitive match for the string "__Host-", then the cookie will have been set with a "Secure" attribute, a "Path" attribute with a value of "/", and no "Domain" attribute.

This combination yields a cookie that hews as closely as a cookie can to treating the origin as a security boundary. The lack of a "Domain" attribute ensures that the cookie’s "host-only-flag" is true, locking the cookie to a particular host, rather than allowing it to span subdomains. Setting the "Path" to "/" means that the cookie is effective for the entire host, and won’t be overridden for specific paths. The "Secure" attribute ensures that the cookie is unaltered by non-secure origins, and won’t span protocols.

Ports are the only piece of the origin model that "__Host-" cookies continue to ignore.

For example, the following cookies would always be rejected:
Set-Cookie: __Host-SID=12345
Set-Cookie: __Host-SID=12345; Secure
Set-Cookie: __Host-SID=12345; Domain=example.com
Set-Cookie: __Host-SID=12345; Domain=example.com; Path=/
Set-Cookie: __Host-SID=12345; Secure; Domain=example.com; Path=/

While the would be accepted if set from a secure origin (e.g. "https://example.com/"), and rejected otherwise:
Set-Cookie: __Host-SID=12345; Secure; Path=/

4.2. Cookie

4.2.1. Syntax

The user agent sends stored cookies to the origin server in the Cookie header. If the server conforms to the requirements in Section 4.1 (and the user agent conforms to the requirements in Section 5), the user agent will send a Cookie header that conforms to the following grammar:
4.2.2. Semantics

Each cookie-pair represents a cookie stored by the user agent. The cookie-pair contains the cookie-name and cookie-value the user agent received in the Set-Cookie header.

Notice that the cookie attributes are not returned. In particular, the server cannot determine from the Cookie header alone when a cookie will expire, for which hosts the cookie is valid, for which paths the cookie is valid, or whether the cookie was set with the Secure or HttpOnly attributes.

The semantics of individual cookies in the Cookie header are not defined by this document. Servers are expected to imbue these cookies with application-specific semantics.

Although cookies are serialized linearly in the Cookie header, servers SHOULD NOT rely upon the serialization order. In particular, if the Cookie header contains two cookies with the same name (e.g., that were set with different Path or Domain attributes), servers SHOULD NOT rely upon the order in which these cookies appear in the header.

5. User Agent Requirements

This section specifies the Cookie and Set-Cookie headers in sufficient detail that a user agent implementing these requirements precisely can interoperate with existing servers (even those that do not conform to the well-behaved profile described in Section 4).

A user agent could enforce more restrictions than those specified herein (e.g., for the sake of improved security); however, experiments have shown that such strictness reduces the likelihood that a user agent will be able to interoperate with existing servers.

5.1. Subcomponent Algorithms

This section defines some algorithms used by user agents to process specific subcomponents of the Cookie and Set-Cookie headers.

5.1.1. Dates

The user agent MUST use an algorithm equivalent to the following algorithm to parse a cookie-date. Note that the various boolean
flags defined as a part of the algorithm (i.e., found-time, found-day-of-month, found-month, found-year) are initially "not set".

1. Using the grammar below, divide the cookie-date into date-tokens.

```
cookie-date     = *delimiter date-token-list *delimiter
date-token-list = date-token *( 1*delimiter date-token )
date-token      = 1*non-delimiter
delimiter       = %x09 / %x20-2F / %x3B-40 / %x5B-60 / %x7B-7E
non-delimiter   = %x00-08 / %x0A-1F / DIGIT / ":" / ALPHA / %x7F-FF
non-digit       = %x00-2F / %x3A-FF
day-of-month    = 1*2DIGIT [ non-digit *OCTET ]
month           = ( "jan" / "feb" / "mar" / "apr" / "may" / "jun" / "jul" / "aug" / "sep" / "oct" / "nov" / "dec" ) *OCTET
year            = 2*4DIGIT [ non-digit *OCTET ]
time            = hms-time [ non-digit *OCTET ]
hms-time        = time-field ":" time-field ":" time-field
time-field      = 1*2DIGIT
```

2. Process each date-token sequentially in the order the date-tokens appear in the cookie-date:

1. If the found-time flag is not set and the token matches the time production, set the found-time flag and set the hour-value, minute-value, and second-value to the numbers denoted by the digits in the date-token, respectively. Skip the remaining sub-steps and continue to the next date-token.

2. If the found-day-of-month flag is not set and the date-token matches the day-of-month production, set the found-day-of-month flag and set the day-of-month-value to the number denoted by the date-token. Skip the remaining sub-steps and continue to the next date-token.

3. If the found-month flag is not set and the date-token matches the month production, set the found-month flag and set the month-value to the month denoted by the date-token. Skip the remaining sub-steps and continue to the next date-token.

4. If the found-year flag is not set and the date-token matches the year production, set the found-year flag and set the year-value to the number denoted by the date-token. Skip the remaining sub-steps and continue to the next date-token.
3. If the year-value is greater than or equal to 70 and less than or equal to 99, increment the year-value by 1900.

4. If the year-value is greater than or equal to 0 and less than or equal to 69, increment the year-value by 2000.

1. NOTE: Some existing user agents interpret two-digit years differently.

5. Abort these steps and fail to parse the cookie-date if:
   * at least one of the found-day-of-month, found-month, found-year, or found-time flags is not set,
   * the day-of-month-value is less than 1 or greater than 31,
   * the year-value is less than 1601,
   * the hour-value is greater than 23,
   * the minute-value is greater than 59, or
   * the second-value is greater than 59.

   (Note that leap seconds cannot be represented in this syntax.)

6. Let the parsed-cookie-date be the date whose day-of-month, month, year, hour, minute, and second (in UTC) are the day-of-month-value, the month-value, the year-value, the hour-value, the minute-value, and the second-value, respectively. If no such date exists, abort these steps and fail to parse the cookie-date.

7. Return the parsed-cookie-date as the result of this algorithm.

5.1.2. Canonicalized Host Names

A canonicalized host name is the string generated by the following algorithm:

1. Convert the host name to a sequence of individual domain name labels.

2. Convert each label that is not a Non-Reserved LDH (NR-LDH) label, to an A-label (see Section 2.3.2.1 of [RFC5890] for the former and latter), or to a "punycode label" (a label resulting from the "ToASCII" conversion in Section 4 of [RFC3490]), as appropriate (see Section 6.3 of this specification).
3. Concatenate the resulting labels, separated by a %x2E (".")

character.

5.1.3. Domain Matching

A string domain-matches a given domain string if at least one of the
following conditions hold:

- The domain string and the string are identical. (Note that both
  the domain string and the string will have been canonicalized to
  lower case at this point.)

- All of the following conditions hold:
  * The domain string is a suffix of the string.
  * The last character of the string that is not included in the
    domain string is a %x2E ("."), character.
  * The string is a host name (i.e., not an IP address).

5.1.4. Paths and Path-Match

The user agent MUST use an algorithm equivalent to the following
algorithm to compute the default-path of a cookie:

1. Let uri-path be the path portion of the request-uri if such a
   portion exists (and empty otherwise). For example, if the
   request-uri contains just a path (and optional query string),
   then the uri-path is that path (without the %x3F ("?") character
   or query string), and if the request-uri contains a full
   absoluteURI, the uri-path is the path component of that URI.

2. If the uri-path is empty or if the first character of the uri-
   path is not a %x2F ("/") character, output %x2F ("/") and skip
   the remaining steps.

3. If the uri-path contains no more than one %x2F ("/") character,
   output %x2F ("/") and skip the remaining step.

4. Output the characters of the uri-path from the first character up
   to, but not including, the right-most %x2F ("/").

A request-path path-matches a given cookie-path if at least one of
the following conditions holds:

- The cookie-path and the request-path are identical.
Note that this differs from the rules in [RFC3986] for equivalence of the path component, and hence two equivalent paths can have different cookies.

- The cookie-path is a prefix of the request-path, and the last character of the cookie-path is %x2F ("/").
- The cookie-path is a prefix of the request-path, and the first character of the request-path that is not included in the cookie-path is a %x2F ("/") character.

5.2. "Same-site" and "cross-site" Requests

A request is "same-site" if its target’s URI’s origin’s registered domain is an exact match for the request’s client’s "site for cookies", or if the request has no client. The request is otherwise "cross-site".

For a given request ("request"), the following algorithm returns "same-site" or "cross-site":

1. If "request"’s client is "null", return "same-site".
   Note that this is the case for navigation triggered by the user directly (e.g. by typing directly into a user agent’s address bar).
2. Let "site" be "request"’s client’s "site for cookies" (as defined in the following sections).
3. Let "target" be the registered domain of "request"’s current url.
4. If "site" is an exact match for "target", return "same-site".
5. Return "cross-site".

The request’s client’s "site for cookies" is calculated depending upon its client’s type, as described in the following subsections:

5.2.1. Document-based requests

The URI displayed in a user agent’s address bar is the only security context directly exposed to users, and therefore the only signal users can reasonably rely upon to determine whether or not they trust a particular website. The registered domain of that URI’s origin represents the context in which a user most likely believes themselves to be interacting. We’ll label this domain the "top-level site".
For a document displayed in a top-level browsing context, we can stop here; the document’s "site for cookies" is the top-level site.

For documents which are displayed in nested browsing contexts, we need to audit the origins of each of a document’s ancestor browsing contexts’ active documents in order to account for the "multiple-nested scenarios" described in Section 4 of [RFC7034]. These document’s "site for cookies" is the top-level site if and only if the document and each of its ancestor documents’ origins have the same registered domain as the top-level site. Otherwise its "site for cookies" is the empty string.

Given a Document ("document"), the following algorithm returns its "site for cookies" (either a registered domain, or the empty string):

1. Let "top-document" be the active document in "document"’s browsing context’s top-level browsing context.

2. Let "top-origin" be the origin of "top-document"’s URI if "top-document"’s sandboxed origin browsing context flag is set, and "top-document"’s origin otherwise.

3. Let "documents" be a list containing "document" and each of "document"’s ancestor browsing contexts’ active documents.

4. For each "item" in "documents":
   1. Let "origin" be the origin of "item"’s URI if "item"’s sandboxed origin browsing context flag is set, and "item"’s origin otherwise.
   2. If "origin"’s host’s registered domain is not an exact match for "top-origin"’s host’s registered domain, return the empty string.

5. Return "top-origin"’s host’s registered domain.

5.2.2. Worker-based requests

Worker-driven requests aren't as clear-cut as document-driven requests, as there isn’t a clear link between a top-level browsing context and a worker. This is especially true for Service Workers [SERVICE-WORKERS], which may execute code in the background, without any document visible at all.

Note: The descriptions below assume that workers must be same-origin with the documents that instantiate them. If this invariant changes,
we’ll need to take the worker’s script’s URI into account when determining their status.

5.2.2.1. Dedicated and Shared Workers

Dedicated workers are simple, as each dedicated worker is bound to one and only one document. Requests generated from a dedicated worker (via "importScripts", "XMLHttpRequest", "fetch()", etc) define their "site for cookies" as that document’s "site for cookies".

Shared workers may be bound to multiple documents at once. As it is quite possible for those documents to have distinct "site for cookie" values, the worker’s "site for cookies" will be the empty string in cases where the values diverge, and the shared value in cases where the values agree.

Given a WorkerGlobalScope ("worker"), the following algorithm returns its "site for cookies" (either a registered domain, or the empty string):

1. Let "site" be "worker"’s origin’s host’s registered domain.
2. For each "document" in "worker"’s Documents:
   1. Let "document-site" be "document"’s "site for cookies" (as defined in Section 5.2.1).
   2. If "document-site" is not an exact match for "site", return the empty string.
3. Return "site".

5.2.2.2. Service Workers

Service Workers are more complicated, as they act as a completely separate execution context with only tangential relationship to the Document which registered them.

Requests which simply pass through a service worker will be handled as described above: the request’s client will be the Document or Worker which initiated the request, and its "site for cookies" will be those defined in Section 5.2.1 and Section 5.2.2.1

Requests which are initiated by the Service Worker itself (via a direct call to "fetch()", for instance), on the other hand, will have a client which is a ServiceWorkerGlobalScope. Its "site for cookies" will be the registered domain of the Service Worker’s URI.
Given a ServiceWorkerGlobalScope ("worker"), the following algorithm returns its "site for cookies" (either a registered domain, or the empty string):

1. Return "worker"’s origin’s host’s registered domain.

5.3. The Set-Cookie Header

When a user agent receives a Set-Cookie header field in an HTTP response, the user agent MAY ignore the Set-Cookie header field in its entirety. For example, the user agent might wish to block responses to "third-party" requests from setting cookies (see Section 7.1).

If the user agent does not ignore the Set-Cookie header field in its entirety, the user agent MUST parse the field-value of the Set-Cookie header field as a set-cookie-string (defined below).

NOTE: The algorithm below is more permissive than the grammar in Section 4.1. For example, the algorithm strips leading and trailing whitespace from the cookie name and value (but maintains internal whitespace), whereas the grammar in Section 4.1 forbids whitespace in these positions. User agents use this algorithm so as to interoperate with servers that do not follow the recommendations in Section 4.

A user agent MUST use an algorithm equivalent to the following algorithm to parse a set-cookie-string:

1. If the set-cookie-string contains a %x3B (";") character:
   1. The name-value-pair string consists of the characters up to, but not including, the first %x3B (";"), and the unparsed-attributes consist of the remainder of the set-cookie-string (including the %x3B (";") in question).
   2. If the name-value-pair string lacks a %x3D ("=") character, ignore the set-cookie-string entirely.
   3. The (possibly empty) name string consists of the characters up to, but not including, the first %x3D ("=") character, and the...
(possibly empty) value string consists of the characters after the first %x3D ("=") character.

4. Remove any leading or trailing WSP characters from the name string and the value string.

5. If the name string is empty, ignore the set-cookie-string entirely.

6. The cookie-name is the name string, and the cookie-value is the value string.

The user agent MUST use an algorithm equivalent to the following algorithm to parse the unparsed-attributes:

1. If the unparsed-attributes string is empty, skip the rest of these steps.

2. Discard the first character of the unparsed-attributes (which will be a %x3B (";") character).

3. If the remaining unparsed-attributes contains a %x3B (";") character:
   1. Consume the characters of the unparsed-attributes up to, but not including, the first %x3B (";") character.

   Otherwise:

   1. Consume the remainder of the unparsed-attributes.

   Let the cookie-av string be the characters consumed in this step.

4. If the cookie-av string contains a %x3D ("=") character:
   1. The (possibly empty) attribute-name string consists of the characters up to, but not including, the first %x3D ("=") character, and the (possibly empty) attribute-value string consists of the characters after the first %x3D ("=") character.

   Otherwise:

   1. The attribute-name string consists of the entire cookie-av string, and the attribute-value string is empty.

5. Remove any leading or trailing WSP characters from the attribute-name string and the attribute-value string.
6. Process the attribute-name and attribute-value according to the requirements in the following subsections. (Notice that attributes with unrecognized attribute-names are ignored.)

7. Return to Step 1 of this algorithm.

When the user agent finishes parsing the set-cookie-string, the user agent is said to "receive a cookie" from the request-uri with name cookie-name, value cookie-value, and attributes cookie-attribute-list. (See Section 5.4 for additional requirements triggered by receiving a cookie.)

5.3.1. The Expires Attribute

If the attribute-name case-insensitively matches the string "Expires", the user agent MUST process the cookie-av as follows.

1. Let the expiry-time be the result of parsing the attribute-value as cookie-date (see Section 5.1.1).

2. If the attribute-value failed to parse as a cookie date, ignore the cookie-av.

3. If the expiry-time is later than the last date the user agent can represent, the user agent MAY replace the expiry-time with the last representable date.

4. If the expiry-time is earlier than the earliest date the user agent can represent, the user agent MAY replace the expiry-time with the earliest representable date.

5. Append an attribute to the cookie-attribute-list with an attribute-name of Expires and an attribute-value of expiry-time.

5.3.2. The Max-Age Attribute

If the attribute-name case-insensitively matches the string "Max-Age", the user agent MUST process the cookie-av as follows.

1. If the first character of the attribute-value is not a DIGIT or a "-" character, ignore the cookie-av.

2. If the remainder of attribute-value contains a non-DIGIT character, ignore the cookie-av.

3. Let delta-seconds be the attribute-value converted to an integer.
4. If delta-seconds is less than or equal to zero (0), let expiry-time be the earliest representable date and time. Otherwise, let the expiry-time be the current date and time plus delta-seconds seconds.

5. Append an attribute to the cookie-attribute-list with an attribute-name of Max-Age and an attribute-value of expiry-time.

5.3.3. The Domain Attribute

If the attribute-name case-insensitively matches the string "Domain", the user agent MUST process the cookie-av as follows.

1. If the attribute-value is empty, the behavior is undefined. However, the user agent SHOULD ignore the cookie-av entirely.

2. If the first character of the attribute-value string is %x2E ("."):
   1. Let cookie-domain be the attribute-value without the leading %x2E (".") character.

   Otherwise:
   1. Let cookie-domain be the entire attribute-value.

3. Convert the cookie-domain to lower case.

4. Append an attribute to the cookie-attribute-list with an attribute-name of Domain and an attribute-value of cookie-domain.

5.3.4. The Path Attribute

If the attribute-name case-insensitively matches the string "Path", the user agent MUST process the cookie-av as follows.

1. If the attribute-value is empty or if the first character of the attribute-value is not %x2F ("/"):
   1. Let cookie-path be the default-path.

   Otherwise:
   1. Let cookie-path be the attribute-value.

2. Append an attribute to the cookie-attribute-list with an attribute-name of Path and an attribute-value of cookie-path.
5.3.5. The Secure Attribute

If the attribute-name case-insensitively matches the string "Secure", the user agent MUST append an attribute to the cookie-attribute-list with an attribute-name of Secure and an empty attribute-value.

5.3.6. The HttpOnly Attribute

If the attribute-name case-insensitively matches the string "HttpOnly", the user agent MUST append an attribute to the cookie-attribute-list with an attribute-name of HttpOnly and an empty attribute-value.

5.3.7. The SameSite Attribute

If the attribute-name case-insensitively matches the string "SameSite", the user agent MUST process the cookie-av as follows:

1. If cookie-av’s attribute-value is not a case-insensitive match for "Strict" or "Lax", ignore the "cookie-av".

2. Let "enforcement" be "Lax" if cookie-av’s attribute-value is a case-insensitive match for "Lax", and "Strict" otherwise.

3. Append an attribute to the cookie-attribute-list with an attribute-name of "SameSite" and an attribute-value of "enforcement".

5.3.7.1. "Strict" and "Lax" enforcement

Same-site cookies in "Strict" enforcement mode will not be sent along with top-level navigations which are triggered from a cross-site document context. As discussed in Section 8.8.2, this might or might not be compatible with existing session management systems. In the interests of providing a drop-in mechanism that mitigates the risk of CSRF attacks, developers may set the "SameSite" attribute in a "Lax" enforcement mode that carves out an exception which sends same-site cookies along with cross-site requests if and only if they are top-level navigations which use a "safe" (in the [RFC7231] sense) HTTP method.

Lax enforcement provides reasonable defense in depth against CSRF attacks that rely on unsafe HTTP methods (like "POST"), but does not offer a robust defense against CSRF as a general category of attack:

1. Attackers can still pop up new windows or trigger top-level navigations in order to create a "same-site" request (as
2. Features like "<link rel='prerender'>" [prerendering] can be exploited to create "same-site" requests without the risk of user detection.

When possible, developers should use a session management mechanism such as that described in Section 8.8.2 to mitigate the risk of CSRF more completely.

5.4. Storage Model

The user agent stores the following fields about each cookie: name, value, expiry-time, domain, path, creation-time, last-access-time, persistent-flag, host-only-flag, secure-only-flag, http-only-flag, and same-site-flag.

When the user agent "receives a cookie" from a request-uri with name cookie-name, value cookie-value, and attributes cookie-attribute-list, the user agent MUST process the cookie as follows:

1. A user agent MAY ignore a received cookie in its entirety. For example, the user agent might wish to block receiving cookies from "third-party" responses or the user agent might not wish to store cookies that exceed some size.

2. Create a new cookie with name cookie-name, value cookie-value. Set the creation-time and the last-access-time to the current date and time.

3. If the cookie-attribute-list contains an attribute with an attribute-name of "Max-Age":
   1. Set the cookie’s persistent-flag to true.
   2. Set the cookie’s expiry-time to attribute-value of the last attribute in the cookie-attribute-list with an attribute-name of "Max-Age".

   Otherwise, if the cookie-attribute-list contains an attribute with an attribute-name of "Expires" (and does not contain an attribute with an attribute-name of "Max-Age"):
   1. Set the cookie’s persistent-flag to true.
2. Set the cookie’s expiry-time to attribute-value of the last attribute in the cookie-attribute-list with an attribute-name of "Expires".

Otherwise:

1. Set the cookie’s persistent-flag to false.

2. Set the cookie’s expiry-time to the latest representable date.

4. If the cookie-attribute-list contains an attribute with an attribute-name of "Domain":

   1. Let the domain-attribute be the attribute-value of the last attribute in the cookie-attribute-list with an attribute-name of "Domain".

Otherwise:

1. Let the domain-attribute be the empty string.

5. If the user agent is configured to reject "public suffixes" and the domain-attribute is a public suffix:

   1. If the domain-attribute is identical to the canonicalized request-host:

      1. Let the domain-attribute be the empty string.

   Otherwise:

      1. Ignore the cookie entirely and abort these steps.

NOTE: A "public suffix" is a domain that is controlled by a public registry, such as "com", "co.uk", and "pvt.k12.wy.us". This step is essential for preventing attacker.com from disrupting the integrity of example.com by setting a cookie with a Domain attribute of "com". Unfortunately, the set of public suffixes (also known as "registry controlled domains") changes over time. If feasible, user agents SHOULD use an up-to-date public suffix list, such as the one maintained by the Mozilla project at http://publicsuffix.org/.

6. If the domain-attribute is non-empty:

   1. If the canonicalized request-host does not domain-match the domain-attribute:
1. Ignore the cookie entirely and abort these steps.

Otherwise:

1. Set the cookie’s host-only-flag to false.

2. Set the cookie’s domain to the domain-attribute.

Otherwise:

1. Set the cookie’s host-only-flag to true.

2. Set the cookie’s domain to the canonicalized request-host.

7. If the cookie-attribute-list contains an attribute with an attribute-name of "Path", set the cookie’s path to attribute-value of the last attribute in the cookie-attribute-list with an attribute-name of "Path". Otherwise, set the cookie’s path to the default-path of the request-uri.

8. If the cookie-attribute-list contains an attribute with an attribute-name of "Secure", set the cookie’s secure-only-flag to true. Otherwise, set the cookie’s secure-only-flag to false.

9. If the scheme component of the request-uri does not denote a "secure" protocol (as defined by the user agent), and the cookie’s secure-only-flag is true, then abort these steps and ignore the cookie entirely.

10. If the cookie-attribute-list contains an attribute with an attribute-name of "HttpOnly", set the cookie’s http-only-flag to true. Otherwise, set the cookie’s http-only-flag to false.

11. If the cookie was received from a "non-HTTP" API and the cookie’s http-only-flag is true, abort these steps and ignore the cookie entirely.

12. If the cookie’s secure-only-flag is not set, and the scheme component of request-uri does not denote a "secure" protocol, then abort these steps and ignore the cookie entirely if the cookie store contains one or more cookies that meet all of the following criteria:

1. Their name matches the name of the newly-created cookie.

2. Their secure-only-flag is true.
3. Their domain domain-matches the domain of the newly-created cookie, or vice-versa.

4. The path of the newly-created cookie path-matches the path of the existing cookie.

Note: The path comparison is not symmetric, ensuring only that a newly-created, non-secure cookie does not overlay an existing secure cookie, providing some mitigation against cookie-fixing attacks. That is, given an existing secure cookie named 'a' with a path of '/login', a non-secure cookie named 'a' could be set for a path of '/' or '/foo', but not for a path of '/login' or '/login/en'.

13. If the cookie-attribute-list contains an attribute with an attribute-name of "SameSite", set the cookie’s same-site-flag to attribute-value (i.e. either "Strict" or "Lax"). Otherwise, set the cookie’s same-site-flag to "None".

14. If the cookie’s "same-site-flag" is not "None", and the cookie is being set from a context whose "site for cookies" is not an exact match for request-uri’s host’s registered domain, then abort these steps and ignore the newly created cookie entirely.

15. If the cookie-name begins with a case-sensitive match for the string "__Secure-", abort these steps and ignore the cookie entirely unless the cookie’s secure-only-flag is true.

16. If the cookie-name begins with a case-sensitive match for the string "__Host-", abort these steps and ignore the cookie entirely unless the cookie meets all the following criteria:
   1. The cookie’s secure-only-flag is true.
   2. The cookie’s host-only-flag is true.
   3. The cookie-attribute-list contains an attribute with an attribute-name of "Path", and the cookie’s path is "/".

17. If the cookie store contains a cookie with the same name, domain, and path as the newly-created cookie:
   1. Let old-cookie be the existing cookie with the same name, domain, and path as the newly-created cookie. (Notice that this algorithm maintains the invariant that there is at most one such cookie.)
2. If the newly-created cookie was received from a "non-HTTP" API and the old-cookie’s http-only-flag is true, abort these steps and ignore the newly created cookie entirely.

3. Update the creation-time of the newly-created cookie to match the creation-time of the old-cookie.

4. Remove the old-cookie from the cookie store.

18. Insert the newly-created cookie into the cookie store.

A cookie is "expired" if the cookie has an expiry date in the past.

The user agent MUST evict all expired cookies from the cookie store if, at any time, an expired cookie exists in the cookie store.

At any time, the user agent MAY "remove excess cookies" from the cookie store if the number of cookies sharing a domain field exceeds some implementation-defined upper bound (such as 50 cookies).

At any time, the user agent MAY "remove excess cookies" from the cookie store if the cookie store exceeds some predetermined upper bound (such as 3000 cookies).

When the user agent removes excess cookies from the cookie store, the user agent MUST evict cookies in the following priority order:

1. Expired cookies.

2. Cookies whose secure-only-flag is not set, and which share a domain field with more than a predetermined number of other cookies.

3. Cookies that share a domain field with more than a predetermined number of other cookies.

4. All cookies.

If two cookies have the same removal priority, the user agent MUST evict the cookie with the earliest last-access date first.

When "the current session is over" (as defined by the user agent), the user agent MUST remove from the cookie store all cookies with the persistent-flag set to false.
5.5. The Cookie Header

The user agent includes stored cookies in the Cookie HTTP request header.

When the user agent generates an HTTP request, the user agent MUST NOT attach more than one Cookie header field.

A user agent MAY omit the Cookie header in its entirety. For example, the user agent might wish to block sending cookies during "third-party" requests from setting cookies (see Section 7.1).

If the user agent does attach a Cookie header field to an HTTP request, the user agent MUST send the cookie-string (defined below) as the value of the header field.

The user agent MUST use an algorithm equivalent to the following algorithm to compute the cookie-string from a cookie store and a request-uri:

1. Let cookie-list be the set of cookies from the cookie store that meets all of the following requirements:

   * Either:
     + The cookie’s host-only-flag is true and the canonicalized request-host is identical to the cookie’s domain.
     
     Or:
     + The cookie’s host-only-flag is false and the canonicalized request-host domain-matches the cookie’s domain.

   * The request-uri’s path path-matches the cookie’s path.

   * If the cookie’s secure-only-flag is true, then the request-uri’s scheme must denote a "secure" protocol (as defined by the user agent).

     NOTE: The notion of a "secure" protocol is not defined by this document. Typically, user agents consider a protocol secure if the protocol makes use of transport-layer security, such as SSL or TLS. For example, most user agents consider "https" to be a scheme that denotes a secure protocol.

   * If the cookie’s http-only-flag is true, then exclude the cookie if the cookie-string is being generated for a "non-HTTP" API (as defined by the user agent).
If the cookie’s same-site-flag is not "None", and the HTTP request is cross-site (as defined in Section 5.2) then exclude the cookie unless all of the following statements hold:

1. The same-site-flag is "Lax"
2. The HTTP request’s method is "safe".
3. The HTTP request’s target browsing context is a top-level browsing context.

2. The user agent SHOULD sort the cookie-list in the following order:

* Cookies with longer paths are listed before cookies with shorter paths.
* Among cookies that have equal-length path fields, cookies with earlier creation-times are listed before cookies with later creation-times.

NOTE: Not all user agents sort the cookie-list in this order, but this order reflects common practice when this document was written, and, historically, there have been servers that (erroneously) depended on this order.

3. Update the last-access-time of each cookie in the cookie-list to the current date and time.

4. Serialize the cookie-list into a cookie-string by processing each cookie in the cookie-list in order:

1. Output the cookie’s name, the %x3D ("=") character, and the cookie’s value.
2. If there is an unprocessed cookie in the cookie-list, output the characters %x3B and %x20 ("; ").

NOTE: Despite its name, the cookie-string is actually a sequence of octets, not a sequence of characters. To convert the cookie-string (or components thereof) into a sequence of characters (e.g., for presentation to the user), the user agent might wish to try using the UTF-8 character encoding [RFC3629] to decode the octet sequence. This decoding might fail, however, because not every sequence of octets is valid UTF-8.
6. Implementation Considerations

6.1. Limits

Practical user agent implementations have limits on the number and size of cookies that they can store. General-use user agents SHOULD provide each of the following minimum capabilities:

- At least 4096 bytes per cookie (as measured by the sum of the length of the cookie’s name, value, and attributes).
- At least 50 cookies per domain.
- At least 3000 cookies total.

Servers SHOULD use as few and as small cookies as possible to avoid reaching these implementation limits and to minimize network bandwidth due to the Cookie header being included in every request.

Servers SHOULD gracefully degrade if the user agent fails to return one or more cookies in the Cookie header because the user agent might evict any cookie at any time on orders from the user.

6.2. Application Programming Interfaces

One reason the Cookie and Set-Cookie headers use such esoteric syntax is that many platforms (both in servers and user agents) provide a string-based application programming interface (API) to cookies, requiring application-layer programmers to generate and parse the syntax used by the Cookie and Set-Cookie headers, which many programmers have done incorrectly, resulting in interoperability problems.

Instead of providing string-based APIs to cookies, platforms would be well-served by providing more semantic APIs. It is beyond the scope of this document to recommend specific API designs, but there are clear benefits to accepting an abstract "Date" object instead of a serialized date string.

6.3. IDNA Dependency and Migration

IDNA2008 [RFC5890] supersedes IDNA2003 [RFC3490]. However, there are differences between the two specifications, and thus there can be differences in processing (e.g., converting) domain name labels that have been registered under one from those registered under the other. There will be a transition period of some time during which IDNA2003-based domain name labels will exist in the wild. User agents SHOULD implement IDNA2008 [RFC5890] and MAY implement [UTS46]
or [RFC5895] in order to facilitate their IDNA transition. If a user agent does not implement IDNA2008, the user agent MUST implement IDNA2003 [RFC3490].

7. Privacy Considerations

Cookies are often criticized for letting servers track users. For example, a number of "web analytics" companies use cookies to recognize when a user returns to a web site or visits another web site. Although cookies are not the only mechanism servers can use to track users across HTTP requests, cookies facilitate tracking because they are persistent across user agent sessions and can be shared between hosts.

7.1. Third-Party Cookies

Particularly worrisome are so-called "third-party" cookies. In rendering an HTML document, a user agent often requests resources from other servers (such as advertising networks). These third-party servers can use cookies to track the user even if the user never visits the server directly. For example, if a user visits a site that contains content from a third party and then later visits another site that contains content from the same third party, the third party can track the user between the two sites.

Given this risk to user privacy, some user agents restrict how third-party cookies behave, and those restrictions vary widely. For instance, user agents might block third-party cookies entirely by refusing to send Cookie headers or process Set-Cookie headers during third-party requests. They might take a less draconian approach by partitioning cookies based on the first-party context, sending one set of cookies to a given third party in one first-party context, and another to the same third party in another.

This document grants user agents wide latitude to experiment with third-party cookie policies that balance the privacy and compatibility needs of their users. However, this document does not endorse any particular third-party cookie policy.

Third-party cookie blocking policies are often ineffective at achieving their privacy goals if servers attempt to work around their restrictions to track users. In particular, two collaborating servers can often track users without using cookies at all by injecting identifying information into dynamic URLs.
7.2. User Controls

User agents SHOULD provide users with a mechanism for managing the cookies stored in the cookie store. For example, a user agent might let users delete all cookies received during a specified time period or all the cookies related to a particular domain. In addition, many user agents include a user interface element that lets users examine the cookies stored in their cookie store.

User agents SHOULD provide users with a mechanism for disabling cookies. When cookies are disabled, the user agent MUST NOT include a Cookie header in outbound HTTP requests and the user agent MUST NOT process Set-Cookie headers in inbound HTTP responses.

Some user agents provide users the option of preventing persistent storage of cookies across sessions. When configured thusly, user agents MUST treat all received cookies as if the persistent-flag were set to false. Some popular user agents expose this functionality via "private browsing" mode [Aggarwal2010].

Some user agents provide users with the ability to approve individual writes to the cookie store. In many common usage scenarios, these controls generate a large number of prompts. However, some privacy-conscious users find these controls useful nonetheless.

7.3. Expiration Dates

Although servers can set the expiration date for cookies to the distant future, most user agents do not actually retain cookies for multiple decades. Rather than choosing gratuitously long expiration periods, servers SHOULD promote user privacy by selecting reasonable cookie expiration periods based on the purpose of the cookie. For example, a typical session identifier might reasonably be set to expire in two weeks.

8. Security Considerations

8.1. Overview

Cookies have a number of security pitfalls. This section overviews a few of the more salient issues.

In particular, cookies encourage developers to rely on ambient authority for authentication, often becoming vulnerable to attacks such as cross-site request forgery [CSRF]. Also, when storing session identifiers in cookies, developers often create session fixation vulnerabilities.
Transport-layer encryption, such as that employed in HTTPS, is insufficient to prevent a network attacker from obtaining or altering a victim’s cookies because the cookie protocol itself has various vulnerabilities (see "Weak Confidentiality" and "Weak Integrity", below). In addition, by default, cookies do not provide confidentiality or integrity from network attackers, even when used in conjunction with HTTPS.

8.2. Ambient Authority

A server that uses cookies to authenticate users can suffer security vulnerabilities because some user agents let remote parties issue HTTP requests from the user agent (e.g., via HTTP redirects or HTML forms). When issuing those requests, user agents attach cookies even if the remote party does not know the contents of the cookies, potentially letting the remote party exercise authority at an unwary server.

Although this security concern goes by a number of names (e.g., cross-site request forgery, confused deputy), the issue stems from cookies being a form of ambient authority. Cookies encourage server operators to separate designation (in the form of URLs) from authorization (in the form of cookies). Consequently, the user agent might supply the authorization for a resource designated by the attacker, possibly causing the server or its clients to undertake actions designated by the attacker as though they were authorized by the user.

Instead of using cookies for authorization, server operators might wish to consider entangling designation and authorization by treating URLs as capabilities. Instead of storing secrets in cookies, this approach stores secrets in URLs, requiring the remote entity to supply the secret itself. Although this approach is not a panacea, judicious application of these principles can lead to more robust security.

8.3. Clear Text

Unless sent over a secure channel (such as TLS), the information in the Cookie and Set-Cookie headers is transmitted in the clear.

1. All sensitive information conveyed in these headers is exposed to an eavesdropper.

2. A malicious intermediary could alter the headers as they travel in either direction, with unpredictable results.
3. A malicious client could alter the Cookie header before transmission, with unpredictable results.

Servers SHOULD encrypt and sign the contents of cookies (using whatever format the server desires) when transmitting them to the user agent (even when sending the cookies over a secure channel). However, encrypting and signing cookie contents does not prevent an attacker from transplanting a cookie from one user agent to another or from replaying the cookie at a later time.

In addition to encrypting and signing the contents of every cookie, servers that require a higher level of security SHOULD use the Cookie and Set-Cookie headers only over a secure channel. When using cookies over a secure channel, servers SHOULD set the Secure attribute (see Section 4.1.2.5) for every cookie. If a server does not set the Secure attribute, the protection provided by the secure channel will be largely moot.

For example, consider a webmail server that stores a session identifier in a cookie and is typically accessed over HTTPS. If the server does not set the Secure attribute on its cookies, an active network attacker can intercept any outbound HTTP request from the user agent and redirect that request to the webmail server over HTTP. Even if the webmail server is not listening for HTTP connections, the user agent will still include cookies in the request. The active network attacker can intercept these cookies, replay them against the server, and learn the contents of the user’s email. If, instead, the server had set the Secure attribute on its cookies, the user agent would not have included the cookies in the clear-text request.

8.4. Session Identifiers

Instead of storing session information directly in a cookie (where it might be exposed to or replayed by an attacker), servers commonly store a nonce (or "session identifier") in a cookie. When the server receives an HTTP request with a nonce, the server can look up state information associated with the cookie using the nonce as a key.

Using session identifier cookies limits the damage an attacker can cause if the attacker learns the contents of a cookie because the nonce is useful only for interacting with the server (unlike non-nonce cookie content, which might itself be sensitive). Furthermore, using a single nonce prevents an attacker from "splicing" together cookie content from two interactions with the server, which could cause the server to behave unexpectedly.

Using session identifiers is not without risk. For example, the server SHOULD take care to avoid "session fixation" vulnerabilities.
8.5. Weak Confidentiality

Cookies do not provide isolation by port. If a cookie is readable by a service running on one port, the cookie is also readable by a service running on another port of the same server. If a cookie is writable by a service on one port, the cookie is also writable by a service running on another port of the same server. For this reason, servers SHOULD NOT both run mutually distrusting services on different ports of the same host and use cookies to store security-sensitive information.

Cookies do not provide isolation by scheme. Although most commonly used with the http and https schemes, the cookies for a given host might also be available to other schemes, such as ftp and gopher. Although this lack of isolation by scheme is most apparent in non-HTTP APIs that permit access to cookies (e.g., HTML’s document.cookie API), the lack of isolation by scheme is actually present in requirements for processing cookies themselves (e.g., consider retrieving a URI with the gopher scheme via HTTP).

Cookies do not always provide isolation by path. Although the network-level protocol does not send cookies stored for one path to another, some user agents expose cookies via non-HTTP APIs, such as HTML’s document.cookie API. Because some of these user agents (e.g., web browsers) do not isolate resources received from different paths, a resource retrieved from one path might be able to access cookies stored for another path.

8.6. Weak Integrity

Cookies do not provide integrity guarantees for sibling domains (and their subdomains). For example, consider foo.example.com and bar.example.com. The foo.example.com server can set a cookie with a Domain attribute of "example.com" (possibly overwriting an existing "example.com" cookie set by bar.example.com), and the user agent will include that cookie in HTTP requests to bar.example.com. In the worst case, bar.example.com will be unable to distinguish this cookie from a cookie it set itself. The foo.example.com server might be
able to leverage this ability to mount an attack against bar.example.com.

Even though the Set-Cookie header supports the Path attribute, the Path attribute does not provide any integrity protection because the user agent will accept an arbitrary Path attribute in a Set-Cookie header. For example, an HTTP response to a request for http://example.com/foo/bar can set a cookie with a Path attribute of "/qux". Consequently, servers SHOULD NOT both run mutually distrusting services on different paths of the same host and use cookies to store security-sensitive information.

An active network attacker can also inject cookies into the Cookie header sent to https://example.com/ by impersonating a response from http://example.com/ and injecting a Set-Cookie header. The HTTPS server at example.com will be unable to distinguish these cookies from cookies that it set itself in an HTTPS response. An active network attacker might be able to leverage this ability to mount an attack against example.com even if example.com uses HTTPS exclusively.

Servers can partially mitigate these attacks by encrypting and signing the contents of their cookies. However, using cryptography does not mitigate the issue completely because an attacker can replay a cookie he or she received from the authentic example.com server in the user’s session, with unpredictable results.

Finally, an attacker might be able to force the user agent to delete cookies by storing a large number of cookies. Once the user agent reaches its storage limit, the user agent will be forced to evict some cookies. Servers SHOULD NOT rely upon user agents retaining cookies.

8.7. Reliance on DNS

Cookies rely upon the Domain Name System (DNS) for security. If the DNS is partially or fully compromised, the cookie protocol might fail to provide the security properties required by applications.

8.8. SameSite Cookies

8.8.1. Defense in depth

"SameSite" cookies offer a robust defense against CSRF attack when deployed in strict mode, and when supported by the client. It is, however, prudent to ensure that this designation is not the extent of a site's defense against CSRF, as same-site navigations and
submissions can certainly be executed in conjunction with other attack vectors such as cross-site scripting.

Developers are strongly encouraged to deploy the usual server-side defenses (CSRF tokens, ensuring that "safe" HTTP methods are idempotent, etc) to mitigate the risk more fully.

Additionally, client-side techniques such as those described in [app-isolation] may also prove effective against CSRF, and are certainly worth exploring in combination with "SameSite" cookies.

8.8.2. Top-level Navigations

Setting the "SameSite" attribute in "strict" mode provides robust defense in depth against CSRF attacks, but has the potential to confuse users unless sites’ developers carefully ensure that their cookie-based session management systems deal reasonably well with top-level navigations.

Consider the scenario in which a user reads their email at MegaCorp Inc’s webmail provider "https://example.com/". They might expect that clicking on an emailed link to "https://projects.com/secret/project" would show them the secret project that they’re authorized to see, but if "projects.com" has marked their session cookies as "SameSite", then this cross-site navigation won’t send them along with the request. "projects.com" will render a 404 error to avoid leaking secret information, and the user will be quite confused.

Developers can avoid this confusion by adopting a session management system that relies on not one, but two cookies: one conceptually granting "read" access, another granting "write" access. The latter could be marked as "SameSite", and its absence would prompt a reauthentication step before executing any non-idempotent action. The former could drop the "SameSite" attribute entirely, or choose the "Lax" version of enforcement, in order to allow users access to data via top-level navigation.

8.8.3. Mashups and Widgets

The "SameSite" attribute is inappropriate for some important use-cases. In particular, note that content intended for embedding in a cross-site contexts (social networking widgets or commenting services, for instance) will not have access to same-site cookies. Cookies may be required for requests triggered in these cross-site contexts in order to provide seamless functionality that relies on a user’s state.
Likewise, some forms of Single-Sign-On might require cookie-based authentication in a cross-site context; these mechanisms will not function as intended with same-site cookies.

8.8.4. Server-controlled

SameSite cookies in and of themselves don’t do anything to address the general privacy concerns outlined in Section 7.1 of [RFC6265]. The "SameSite" attribute is set by the server, and serves to mitigate the risk of certain kinds of attacks that the server is worried about. The user is not involved in this decision. Moreover, a number of side-channels exist which could allow a server to link distinct requests even in the absence of cookies. Connection and/or socket pooling, Token Binding, and Channel ID all offer explicit methods of identification that servers could take advantage of.

9. IANA Considerations

The permanent message header field registry (see [RFC3864]) needs to be updated with the following registrations.

9.1. Cookie

Header field name: Cookie
Applicable protocol: http
Status: standard
Author/Change controller: IETF
Specification document: this specification (Section 5.5)

9.2. Set-Cookie

Header field name: Set-Cookie
Applicable protocol: http
Status: standard
Author/Change controller: IETF
Specification document: this specification (Section 5.3)
10. References

10.1. Normative References


See Section 6.3 for an explanation why the normative reference to an obsoleted specification is needed.


[SERVICE-WORKERS]


10.2. Informative References

[Aggarwal2010]

[app-isolation]


Appendix A. Changes

A.1. draft-ietf-httpbis-rfc6265bis-00

- Port [RFC6265] to Markdown. No (intentional) normative changes.

A.2. draft-ietf-httpbis-rfc6265bis-01

- Fixes to formatting caused by mistakes in the initial port to Markdown:
  - https://github.com/httpwg/http-extensions/issues/243
  - https://github.com/httpwg/http-extensions/issues/246

- Addresses errata 3444 by updating the "path-value" and "extension-av" grammar, errata 4148 by updating the "day-of-month", "year", and "time" grammar, and errata 3663 by adding the requested note. 

- Dropped "Cookie2" and "Set-Cookie2" from the IANA Considerations section: https://github.com/httpwg/http-extensions/issues/247

- Merged the recommendations from [I-D.ietf-httpbis-cookie-alone], removing the ability for a non-secure origin to set cookies with a 'secure' flag, and to overwrite cookies whose 'secure' flag is true.

- Merged the recommendations from [I-D.ietf-httpbis-cookie-prefixes], adding "__Secure-" and "__Host-" cookie name prefix processing instructions.
A.3. draft-ietf-httpbis-rfc6265bis-02

- Merged the recommendations from [I-D.ietf-httpbis-cookie-same-site], adding support for the "SameSite" attribute.
- Closed a number of editorial bugs:
  * Clarified address bar behavior for SameSite cookies: https://github.com/httpwg/http-extensions/issues/201
  * Added the word "Cookies" to the document’s name: https://github.com/httpwg/http-extensions/issues/204
  * Clarified that the "__Host-" prefix requires an explicit "Path" attribute: https://github.com/httpwg/http-extensions/issues/222
  * Expanded the options for dealing with third-party cookies to include a brief mention of partitioning based on first-party: https://github.com/httpwg/http-extensions/issues/248
  * Noted that double-quotes in cookie values are part of the value, and are not stripped: https://github.com/httpwg/http-extensions/issues/295
  * Fixed the "site for cookies" algorithm to return something that makes sense: https://github.com/httpwg/http-extensions/issues/302

Appendix B. Acknowledgements

This document is a minor update of RFC 6265, adding small features, and aligning the specification with the reality of today’s deployments. Here, we’re standing upon the shoulders of giants.

Authors’ Addresses

Adam Barth
Google, Inc

URI: https://www.adambarth.com/

Mike West
Google, Inc

Email: mkwst@google.com
URI: https://mikewest.org/
On the use of HTTP as a Substrate
draft-nottingham-bcp56bis-00

Abstract

HTTP is often used as a substrate for other application protocols. This document specifies best practices for these protocols’ use of HTTP.

Note to Readers

The issues list for this draft can be found at https://github.com/mnot/I-D/labels/bcp56bis.

The most recent (often, unpublished) draft is at https://mnot.github.io/I-D/bcp56bis/.

Recent changes are listed at https://github.com/mnot/I-D/commits/gh-pages/bcp56bis.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on November 12, 2017.
1. Introduction

HTTP [RFC7230] is often used as a substrate for other application protocols. This is done for a variety of reasons, including:
familiarity by implementers, specifiers, administrators, developers and users,
availability of a variety of client, server and proxy implementations,
ease of use,
ubiquity of Web browsers,
reuse of existing mechanisms like authentication and encryption,
presence of HTTP servers and clients in target deployments, and
its ability to traverse firewalls.

The Internet community has a long tradition of protocol reuse, dating back to the use of Telnet [RFC0854] as a substrate for FTP [RFC0959] and SMTP [RFC2821]. However, layering new protocols over HTTP brings its own set of issues:

Should an application using HTTP define a new URL scheme? Use new ports?
Should it use standard HTTP methods and status codes, or define new ones?
How can the maximum value be extracted from the use of HTTP?
How does it coexist with other uses of HTTP — especially Web browsing?
How can interoperability problems and "protocol dead ends" be avoided?

This document contains best current practices regarding the use of HTTP by applications other than Web browsing. Section 2 defines what applications it applies to; Section 3 surveys the properties of HTTP that are important to preserve, and Section 4 conveys best practices for those applications that do use HTTP.

It is written primarily to guide IETF efforts, but might be applicable in other situations. Note that the requirements herein do not necessarily apply to the development of generic HTTP extensions.
1.1. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. Is HTTP Being Used?

Different applications have different goals when using HTTP. In this document, we say an application is _using HTTP_ when any of the following conditions are true:

- The transport port in use is 80 or 443,
- The URL scheme "http" or "https" is used,
- The ALPN protocol ID [RFC7301] "http/1.1", "h2" or "h2c" is used, or
- The message formats described in [RFC7320] and/or [RFC7540] are used in conjunction with the IANA registries defined for HTTP.

When an application is using HTTP, all of the requirements of the HTTP protocol suite (including but not limited to [RFC7320], [RFC7321], [RFC7322], [RFC7233], [RFC7234], [RFC7325] and [RFC7540]) are in force.

An application might not be _using HTTP_ according to this definition, but still relying upon the HTTP specifications in some manner. For example, an application might wish to avoid re-specifying parts of the message format, but change others; or, it might want to use a different set of methods.

Such applications are referred to as _protocols based upon HTTP_ in this document. These have more freedom to modify protocol operation, but are also likely to lose at least a portion of the benefits outlined above, as most HTTP implementations won’t be easily adaptable to these changes, and as the protocol diverges from HTTP, the benefit of mindshare will be lost.

Protocols that are based upon HTTP MUST NOT reuse HTTP’s URL schemes, transport ports, ALPN protocol IDs or IANA registries; rather, they are encouraged to establish their own.
3. What’s Important About HTTP

There are many ways that HTTP applications are defined and deployed, and sometimes they are brought to the IETF for standardisation. In that process, what might be workable for deployment in a limited fashion isn’t appropriate for standardisation and the corresponding broader deployment.

This section examines the facets of the protocol that are important to preserve in these situations.

3.1. Generic Semantics

When writing an application’s specification, it’s often tempting to specify exactly how HTTP is to be implemented, supported and used.

However, this can easily lead to an unintended profile of HTTP’s behaviour. For example, it’s common to see specifications with language like this:

A ‘200 OK’ response means that the widget has successfully been updated.

This sort of specification is bad practice, because it is adding new semantics to HTTP’s status codes and methods, respectively; a recipient - whether it’s an origin server, client library, intermediary or cache - now has to know these extra semantics to understand the message.

Some applications even require specific behaviours, such as:

A 'POST' request MUST result in a '201 Created' response.

This forms an expectation in the client that the response will always be "201 Created", when in fact there are a number of reasons why the status code might differ in a real deployment. If the client does not anticipate this, the application’s deployment is brittle.

Much of the value of HTTP is in its _generic semantics_ - that is, the protocol elements defined by HTTP are potentially applicable to every resource, not specific to a particular context. Application-specific semantics are expressed in the payload; mostly, in the body, but also in header fields.

This allows a HTTP message to be examined by generic HTTP software (e.g., HTTP servers, intermediaries, client implementations), and its handling to be correctly determined. It also allows people to leverage their knowledge of HTTP semantics without special-casing them for a particular application.
Therefore, applications that use HTTP MUST NOT re-define, refine or overlay the semantics of defined protocol elements. Instead, they SHOULD focus their specifications on protocol elements that are specific to them; namely their HTTP resources.

See Section 4.2 for details.

3.2. Links

Another common practice is assuming that the HTTP server’s name space (or a portion thereof) is exclusively for the use of a single application. This effectively overlays special, application-specific semantics onto that space, precludes other applications from using it.

As explained in [RFC7320], such "squatting" on a part of the URL space by a standard usurps the server’s authority over its own resources, can cause deployment issues, and is therefore bad practice in standards.

Instead of statically defining URL paths, it is RECOMMENDED that applications using HTTP define links in payloads, to allow flexibility in deployment.

Using runtime links in this fashion has a number of other benefits. For example, navigating with a link allows a request to be routed to a different server without the overhead of a redirection, thereby supporting deployment across machines well. It becomes possible to "mix" different applications on the same server, and offers a natural path for extensibility, versioning and capability management.

3.3. Getting Value from HTTP

The simplest possible use of HTTP is to POST data to a single URL, thereby effectively tunnelling through the protocol.

This "RPC" style of communication does get some benefit from using HTTP – namely, message framing and the availability of implementations – but fails to realise many others:

- Caching for server scalability, latency and bandwidth reduction, and reliability;
- Authentication and access control;
- Automatic redirection;
- Partial content to selectively request part of a response;
Natural support for extensions and versioning through protocol extension; and

The ability to interact with the application easily using a Web browser.

Using such a high-level protocol to tunnel simple semantics has downsides too; because of its more advanced capabilities, breadth of deployment and age, HTTP’s complexity can cause interoperability problems that could be avoided by using a simpler substrate (e.g., WebSockets [RFC6455], if browser support is necessary, or TCP [RFC0793] if not), or making the application be based upon HTTP_, instead of using it (as defined in Section 2).

Applications that use HTTP are encouraged to accommodate the various features that the protocol offers, so that their users receive the maximum benefit from it. This document does not require specific features to be used, since the appropriate design tradeoffs are highly specific to a given situation. However, following the practices in Section 4 will help make them available.

4. Best Practices for Using HTTP

This section contains best practices regarding the use of HTTP by applications, including practices for specific HTTP protocol elements.

4.1. Specifying the Use of HTTP

When specifying the use of HTTP, an application SHOULD use [RFC7230] as the primary reference; it is not necessary to reference all of the specifications in the HTTP suite unless there are specific reasons to do so (e.g., a particular feature is called out).

Applications using HTTP MAY specify a minimum version to be supported (HTTP/1.1 is suggested), and MUST NOT specify a maximum version.

Likewise, applications need not specify what HTTP mechanisms - such as redirection, caching, authentication, proxy authentication, and so on - are to be supported. Full featured support for HTTP SHOULD be taken for granted in servers and clients, and the application’s function SHOULD degrade gracefully if they are not (although this might be achieved by informing the user that their task cannot be completed).

For example, an application can specify that it uses HTTP like this:
Foo Application uses HTTP {{RFC7230}}. Implementations MUST support HTTP/1.1, and MAY support later versions. Support for common HTTP mechanisms such as redirection and caching are assumed.

4.2. Defining HTTP Resources

HTTP Applications SHOULD focus on defining the following application-specific protocol elements:

- Media types [RFC6838], often based upon a format convention such as JSON [RFC7159],
- HTTP header fields, as per Section 4.7, and
- The behaviour of resources, as identified by link relations [RFC5988].

By composing these protocol elements, an application can define a set of resources, identified by link relations, that implement specified behaviours, including:

- Retrieval of their state using GET, in one or more formats identified by media type;
- Resource creation or update using POST or PUT, with an appropriately identified request body format;
- Data processing using POST and identified request and response body format(s); and
- Resource deletion using DELETE.

For example, an application might specify:

Resources linked to with the "example-widget" link relation type are Widgets. The state of a Widget can be fetched in the "application/example-widget+json" format, and can be updated by PUT to the same link. Widget resources can be deleted.

The "Example-Count" response header field on Widget representations indicates how many Widgets are held by the sender.

The "application/example-widget+json" format is a JSON {{RFC7159}} format representing the state of a Widget. It contains links to related information in the link indicated by the Link header field value with the "example-other-info" link relation type.
4.3. HTTP URLs

In HTTP, URLs are opaque identifiers under the control of the server. As outlined in [RFC7320], standards cannot usurp this space, since it might conflict with existing resources, and constrain implementation and deployment.

In other words, applications that use HTTP MUST NOT associate application semantics with specific URL paths. For example, specifying that a "GET to the URL /foo retrieves a bar document" is bad practice. Likewise, specifying "The widget API is at the path /bar" violates [RFC7320].

Instead, applications that use HTTP are encouraged to use typed links [RFC5988] to convey the URIs that are in use, as well as the semantics of the resources that they identify. See Section 4.2 for details.

4.3.1. Initial URL Discovery

Generally, a client with begin interacting with a given application server by requesting an initial document that contains information about that particular deployment, potentially including links to other relevant resources.

Applications that use HTTP SHOULD allow an arbitrary URL to be used as that entry point. For example, rather than specifying "the initial document is at "/foo/v1", they should allow a deployment to use any URL as the entry point for the application.

In cases where doing so is impractical (e.g., it is not possible to convey a whole URL, but only a hostname) applications that use HTTP MAY define a well-known URL [RFC5785] as an entry point.

4.3.2. URL Schemes

Applications that use HTTP MUST allow use of the "https" URL scheme, and SHOULD NOT allow use of the "http" URL scheme, unless interoperability considerations with existing deployments require it. They MUST NOT use other URL schemes.

"https" is preferred to mitigate pervasive monitoring attacks [RFC7258].

Using other schemes to denote an application using HTTP makes it more difficult to use with existing implementations (e.g., Web browsers), and is likely to fail to meet the requirements of [RFC7595].
If it is necessary to advertise the application in use, this SHOULD be done in message payloads, not the URL scheme.

4.3.3. Transport Ports

Applications that use HTTP SHOULD use the default port for the URL scheme in use. If it is felt that networks might need to distinguish the application’s traffic for operational reasons, it MAY register a separate port, but be aware that this has privacy implications for that protocol’s users. The impact of doing so MUST be documented in Security Considerations.

4.4. Authentication and Application State

Applications that use HTTP MAY use stateful cookies [RFC6265] to identify a client and/or store client-specific data to contextualise requests.

If it is only necessary to identify clients, applications that use HTTP MAY use HTTP authentication [RFC7235]; if the Basic authentication scheme [RFC7617] is used, it MUST NOT be used with the ‘http’ URL scheme.

In either case, it is important to carefully specify the scoping and use of these mechanisms; if they expose sensitive data or capabilities (e.g., by acting as an ambient authority), exploits are possible. Mitigations include using a request-specific token to assure the intent of the client.

4.5. HTTP Methods

Applications that use HTTP MUST confine themselves to using registered HTTP methods such as GET, POST, PUT, DELETE, and PATCH.

New HTTP methods are rare; they are required to be registered with IETF Review (see [RFC7232]), and are also required to be _generic_. That means that they need to be potentially applicable to all resources, not just those of one application.

While historically some applications (e.g., [RFC6352] and [RFC4791]) have defined non-generic methods, [RFC7231] now forbids this.

When it is believed that a new method is required, authors are encouraged to engage with the HTTP community early, and document their proposal as a separate HTTP extension, rather than as part of an application’s specification.
4.6. HTTP Status Codes

Applications that use HTTP MUST only use registered HTTP status codes.

As with methods, new HTTP status codes are rare, and required (by [RFC7231]) to be registered with IETF review. Similarly, HTTP status codes are generic; they are required (by [RFC7231]) to be potentially applicable to all resources, not just to those of one application.

When it is believed that a new status code is required, authors are encouraged to engage with the HTTP community early, and document their proposal as a separate HTTP extension, rather than as part of an application’s specification.

Status codes’ primary function is to convey HTTP semantics for the benefit of generic HTTP software, not application-specific semantics. Therefore, applications MUST NOT specify additional semantics or refine existing semantics for status codes.

In particular, specifying that a particular status code has a specific meaning in the context of an application is harmful, as these are not generic semantics, since the consumer needs to be in the context of the application to understand them.

Furthermore, applications using HTTP MUST NOT re-specify the semantics of HTTP status codes, even if it is only by copying their definition. They MUST NOT require specific status phrases to be used; the status phrase has no function in HTTP, and is not guaranteed to be preserved by implementations.

Typically, applications using HTTP will convey application-specific information in the message body and/or HTTP header fields, not the status code.

Specifications sometimes also create a "laundry list" of potential status codes, in an effort to be helpful. The problem with doing so is that such a list is never complete; for example, if a network proxy is interposed, the client might encounter a "407 Proxy Authentication Required" response; or, if the server is rate limiting the client, it might receive a "429 Too Many Requests" response.

Since the list of HTTP status codes can be added to, it’s safer to refer to it directly, and point out that clients SHOULD be able to handle all applicable protocol elements gracefully (i.e., falling back to the generic "n00" semantics of a given status code; e.g., "499" can be safely handled as "400" by clients that don’t recognise it).
4.7. HTTP Header Fields

Applications that use HTTP MAY define new HTTP header fields, following the advice in [RFC7321], Section 8.3.1.

Typically, using HTTP header fields is appropriate in a few different situations:

- Their content is useful to intermediaries (who often wish to avoid parsing the body), and/or
- Their content is useful to generic HTTP software (e.g., clients, servers), and/or
- It is not possible to include their content in the message body (usually because a format does not allow it).

If none of these motivations apply, using a header field is NOT RECOMMENDED.

New header fields MUST be registered, as per [RFC7231] and [RFC3864].

It is RECOMMENDED that header field names be short (even when HTTP/2 header compression is in effect, there is an overhead) but appropriately specific. In particular, if a header field is specific to an application, an identifier for that application SHOULD form a prefix to the header field name, separated by a "-".

The semantics of existing HTTP header fields MUST NOT be re-defined without updating their registration or defining an extension to them (if allowed). For example, an application using HTTP cannot specify that the "Location" header has a special meaning in a certain context.

See Section 4.4 for requirements regarding header fields that carry application state (e.g., Cookie).

5. IANA Considerations

This document has no requirements for IANA.

6. Security Considerations

Section 4.4 discusses the impact of using stateful mechanisms in the protocol as ambient authority, and suggests a mitigation.

Section 4.3.2 requires support for 'https' URLs, and discourages the use of 'http' URLs, to mitigate pervasive monitoring attacks.
7. References

7.1. Normative References


7.2. Informative References


Author’s Address

Mark Nottingham

Email: mnot@mnot.net
URI: https://www.mnot.net/
Abstract

The HTTP Alternative Services (Alt-Svc) mechanism allows an HTTP origin to be served from multiple network endpoints, and over multiple protocols. However, the client must first contact the origin server, in order to learn of the alternative services. This draft proposes a straightforward mapping of Alt-Svc into DNS, allowing clients to learn of these services before their first contact with the origin. This arrangement offers potential benefits to both performance and privacy.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on October 25, 2018.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect
to this document. Code Components extracted from this document must 
include Simplified BSD License text as described in Section 4.e of 
the Trust Legal Provisions and are provided without warranty as 
described in the Simplified BSD License.

Table of Contents

1. Introduction .................................................. 2
   1.1. Terminology ............................................. 3
2. The ALTSVC record type ....................................... 3
   2.1. Comparison with alternatives ........................... 4
      2.1.1. Differences from the SRV RTYPE .................... 4
      2.1.2. Differences from the TXT RTYPE .................... 4
3. Differences from Alt-Svc as transmitted over HTTP .......... 5
   3.1. Omitting Max Age ...................................... 5
   3.2. Interaction with other standards ........................ 5
   3.3. Granularity and lifetime control ........................ 5
4. Client behaviors ............................................. 6
   4.1. Cache interaction ...................................... 6
   4.2. Optimizing for performance ............................. 6
   4.3. Optimizing for privacy .................................. 7
5. Security Considerations ....................................... 7
6. IANA Considerations .......................................... 8
7. References .................................................... 8
   7.1. Normative References .................................. 8
   7.2. Informative References ................................ 9
Authors’ Addresses ............................................... 9

1. Introduction

The HTTP Alternative Services standard [AltSvc] defines

- an extensible data model for describing alternative network endpoints that are authoritative for an origin
- the "Alt-Svc Field Value", a text format for representing this information
- standards for sending information in this format from a server to a client over HTTP/1.1 and HTTP/2.

Together, these components provide a toolkit that has proven useful and effective for informing a client of alternative services for an origin. However, making use of an alternative service requires contacting the origin server first. This creates an obvious performance cost: users wait for a full HTTP connection initiation (multiple roundtrips) before learning of an alternative service that is preferred by the origin. The first connection also publicly
reveals the user’s intended destination to all entities along the network path.

This draft proposes a straightforward mechanism to distribute the Alt-Svc Field Value, in its standard text format, through the DNS. If a client receives this information during DNS resolution, it can skip the initial connection and proceed directly to an alternative service.

1.1. Terminology

For consistency with [AltSvc], we adopt the following definitions

- An "origin" is an information source as in [RFC6454].
- The "origin server" is the server that the client would reach when accessing the origin in the absence of Alt-Svc.
- An "alternative service" is a different server that can serve the origin.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. The ALTSVC record type

The ALTSVC DNS resource record (RR) type (RRTYPE ???) is used to associate an Alternative Service Field Value with an origin. Abstractly, the origin consists of a scheme (typically "https"), a host name, and a port (typically "443").

In the case of the ALTSVC RR, the origin is represented by prefixing the port and scheme with ",", then concatenating them with the host, resulting in a domain name like ",443.,https,.www.example.com,".

The RDATA portion of an ALTSVC resource record contains an Alt-Svc Field Value, exactly as defined in Section 4 of [AltSvc].

For example, if the operator of https://www.example.com intends to include an HTTP response header like

Alt-Svc: h2=:8000; ma=60

They would also publish an ALTSVC DNS record like

Schwartz & Bishop       Expires October 25, 2018                [Page 3]
This data type can be represented as an Unknown RR as described in [RFC3597]:

```
_443._https.www.example.com. 60S IN TYPE ??? # 10
68323D223A3830303022
```

This construction is intended to be extensible in two ways. First, any extensions that are made to the Alt-Svc format for transmission over HTTPS are also applicable here, unless expressly mentioned otherwise. Second, including the scheme in the DNS name allows for ALTSVC to serve schemes other than HTTPS, such as HTTP with Opportunistic Security [RFC8164] and any future schemes for which Alt-Svc may be defined.

### 2.1. Comparison with alternatives

The ALTSVC record type closely resembles some existing record types.

#### 2.1.1. Differences from the SRV RRTYPE

An SRV record can perform a similar function to the ALTSVC record, informing a client to look in a different location for a service. However, there are several differences:

- SRV records are typically mandatory, whereas clients will always continue to function correctly without making use of Alt-Svc.

- SRV records cannot instruct the client to switch or upgrade protocols, whereas Alt-Svc can signal such an upgrade (e.g. to HTTP/2).

- SRV records are not extensible, whereas Alt-Svc can be extended with new parameters. For example, this is what allows the privacy improvements related to SNI selection in [AltSvcSNI].

- Using SRV records would not allow a client to skip processing of the Alt-Svc information in a subsequent connection, so it does not confer a performance advantage.

#### 2.1.2. Differences from the TXT RRTYPE

The ALTSVC record uses an identical format to a TXT record, and could be implemented as such. However, we define a new record type for clarity, and to respect the use of TXT for human-readable notes as recommended in [RFC3507].
3. Differences from Alt-Svc as transmitted over HTTP

Publishing an ALTSVC record in DNS is intended to be equivalent to transmitting this field value over HTTP, and receiving an ALTSVC record is intended to be equivalent to receiving this field value over HTTP. However, there are some small differences in the intended client and server behavior.

3.1. Omitting Max Age

When publishing an ALTSVC record in DNS, server operators MUST omit the "ma" parameter, which encodes the "max age" (i.e. expiration time) of an Alt-Svc Field Value. Instead, server operators SHOULD encode the expiration time in the DNS TTL, and MUST NOT set a TTL longer than the intended "max age".

Server operators MAY publish multiple ALTSVC records as an RRSET, with semantics equivalent to other mechanisms of providing multiple Alt-Svc values to the client. When publishing an RRSET with multiple ALTSVC records, the server operator MUST set the overall TTL to the minimum of the "max age" values (following Section 5.2 of [RFC2181]).

When receiving an ALTSVC record, clients MAY synthesize a new "ma" parameter from the DNS TTL, in order to interoperate with Alt-Svc processing subsystems.

3.2. Interaction with other standards

The purpose of this standard is to reduce connection latency and improve user privacy. Server operators implementing this standard SHOULD also implement TLS 1.3 [I-D.ietf-tls-tls13] and OCSP Stapling [RFC6066], both of which confer substantial performance and privacy benefits when used in combination with ALTSVC records.

To realize the greatest privacy benefits, this proposal is intended for use with a privacy-preserving DNS transport (like DNS over TLS [RFC7858] or DNS over HTTPS [DOH]), and with the "SNI" Alt-Svc Parameter [AltSvcSNI]. However, performance improvements, and some modest privacy improvements, are possible without the use of those standards.

3.3. Granularity and lifetime control

Sending Alt-Svc over HTTP allows the server to tailor the Alt-Svc Field Value specifically to the client. When using an ALTSVC DNS record, groups of clients will necessarily receive the same Alt-Svc Field Value. Therefore, this standard is not suitable for servers that require single-client granularity in Alt-Svc. Server operators
that want to serve different Alt-Svc Field Values to different geographic or network regions SHOULD configure their authoritative DNS server to respect the EDNS0 Client Subnet extension [RFC7871].

Some DNS caching systems incorrectly extend the lifetime of DNS records beyond the stated TTL. Server operators MUST NOT rely on ALTSVC records expiring on time, and MAY shorten the TTL to compensate.

4. Client behaviors

4.1. Cache interaction

If the client has an Alt-Svc cache, and a usable Alt-Svc value is present in that cache, then the client SHOULD NOT issue an ALTSVC DNS query. Instead, the client SHOULD proceed with alternative service connection as usual.

If the client has a cached Alt-Svc entry that is expiring, the client MAY perform an ALTSVC query to refresh the entry.

4.2. Optimizing for performance

Clients that are optimizing for performance (i.e. minimum connection setup time) SHOULD implement the following connection sequence:

1. Issue address (AAAA and/or A) queries, immediately followed by the ALTSVC query.

2. If an ALTSVC response is received first, proceed with alternative service connection and ignore the address responses if they are no longer relevant.

3. Otherwise, initiate connection to the origin server.

4. As soon as an Alt-Svc field value is received, through the DNS or over HTTP, proceed with alternative service connection. Do not abort this connection if an Alt-Svc field value is received from the other source later.

If the ALTSVC and address queries return approximately simultaneously, this process typically saves three roundtrips on a fresh connection that uses Alt-Svc: one each for TCP, TLS 1.3, and HTTP. (On subsequent connections, the Alt-Svc information is expected to be cached, so this procedure does not apply.)

If a client can cache Alt-Svc entries that were received over both HTTP and DNS, the client MAY prefer entries that were received over
HTTP. These records may be more narrowly targeted for the specific client.

As an additional optimization, when choosing among multiple Alt-Svc values, clients MAY prefer those that will not require an address query, either because the corresponding address record is already in cache or because the host is an IP address.

Note that this procedure does not rely on recursive resolvers handling the ALTSVC record type correctly. If ALTSVC queries receive spurious NXDOMAIN responses, or even no response at all, connections will proceed as usual without any delay.

4.3. Optimizing for privacy

Clients that are optimizing for privacy SHOULD implement [AltSvcSNI] and DNS over a secure transport (e.g. [RFC7858] or [DOH]). Use of a secure transport is important not only for privacy protection, but also to ensure that queries for the new ALTSVC RRTYPE are handled correctly. Additionally, these clients SHOULD implement the following connection sequence:

1. Issue the ALTSVC DNS query first, immediately followed by the address queries.
2. Wait for the ALTSVC record response.
3. If the response is nonempty, proceed with alternative service connection and ignore the address query responses.
4. Otherwise, wait for the address queries and connect as usual.

Note that this process is also expected to be faster than Alt-Svc over HTTP in the case of HTTP Opportunistic Upgrade Probing (Section 2 of [RFC8164]).

5. Security Considerations

Alt-Svc Field Values are intended for distribution over untrusted channels, and clients are REQUIRED to verify that the alternative service is authoritative for the origin (Section 2.1 of [AltSvc]). Therefore, DNSSEC signing and validation are OPTIONAL for publishing and using ALTSVC records.
6. IANA Considerations

This draft requires assignment of a new DNS RRTYPE value.

7. References

7.1. Normative References


7.2. Informative References


Authors' Addresses

Ben Schwartz
Google
Email: bemasc@google.com

Mike Bishop
Akamai Technologies
Email: mbishop@evequefou.be
Signed HTTP Exchanges
draft-yasskin-http-origin-signed-responses-04

Abstract

This document specifies how a server can send an HTTP request/response pair, known as an exchange, with signatures that vouch for that exchange’s authenticity. These signatures can be verified against an origin’s certificate to establish that the exchange is authoritative for an origin even if it was transferred over a connection that isn’t. The signatures can also be used in other ways described in the appendices.

These signatures contain countermeasures against downgrade and protocol-confusion attacks.

Note to Readers

Discussion of this draft takes place on the HTTP working group mailing list (ietf-http-wg@w3.org), which is archived at https://lists.w3.org/Archives/Public/ietf-http-wg/ [1].

The source code and issues list for this draft can be found in https://github.com/WICG/webpackage [2].

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on December 16, 2018.
1. Introduction ........................................ 4
2. Terminology ....................................... 4
3. Signing an exchange ................................ 5
   3.1. The Signature Header .......................... 5
       3.1.1. Examples .................................. 6
       3.1.2. Open Questions ............................ 8
   3.2. CBOR representation of exchange headers ..... 8
       3.2.1. Example ................................... 9
   3.3. Loading a certificate chain .................. 10
   3.4. Canonical CBOR serialization ................. 11
   3.5. Signature validity ............................ 12
       3.5.1. Open Questions ............................ 15
   3.6. Updating signature validity .................. 15
       3.6.1. Examples ................................... 16
   3.7. The Accept-Signature header .................. 18
       3.7.1. Integrity identifiers ....................... 19
       3.7.2. Key type identifiers ....................... 19
       3.7.3. Key value identifiers ....................... 19
       3.7.4. Examples ................................... 20
       3.7.5. Open Questions ............................ 20
4. Cross-origin trust .................................. 21
   4.1. Stateful header fields ......................... 22
   4.2. Certificate Requirements ........................ 23
5. Transferring a signed exchange .................... 24
   5.1. Same-origin response .......................... 24
       5.1.1. Significant headers for a same-origin response ... 24
       5.1.2. The Signed-Headers Header .................. 25
   5.2. HTTP/2 extension for cross-origin Server Push . 26
       5.2.1. Indicating support for cross-origin Server Push . 26
       5.2.2. NO_TRUSTED_EXCHANGE_SIGNATURE error code .......... 26
       5.2.3. Validating a cross-origin Push ................ 26
5.3. application/signed-exchange format ................. 27
  5.3.1. Cross-origin trust in application/signed-exchange . 28
  5.3.2. Example .................................. 28
  5.3.3. Open Questions ............................ 29
6. Security considerations ............................. 29
  6.1. Over-signing ................................ 29
    6.1.1. Session fixation ........................ 30
    6.1.2. Misleading content ...................... 30
  6.2. Off-path attackers ............................ 30
  6.3. Downgrades .................................. 30
  6.4. Signing oracles are permanent ................... 31
  6.5. Unsigned headers ............................. 31
  6.6. application/signed-exchange ...................... 31
7. Privacy considerations ............................. 32
8. IANA considerations ............................... 32
  8.1. Signature Header Field Registration ................ 32
  8.2. Accept-Signature Header Field Registration ......... 33
  8.3. Signed-Headers Header Field Registration .......... 33
  8.4. HTTP/2 Settings ................................ 33
  8.5. HTTP/2 Error code ............................ 33
  8.6. Internet Media Type application/signed-exchange .... 34
  8.7. Internet Media Type application/cert-chain+cbor    35
9. References ....................................... 36
  9.1. Normative References .......................... 36
  9.2. Informative References ......................... 38
  9.3. URIs ........................................ 40
Appendix A. Use cases ................................. 41
  A.1. PUSHed subresources ........................... 41
  A.2. Explicit use of a content distributor for subresources 41
  A.3. Subresource Integrity .......................... 42
  A.4. Binary Transparency ........................... 42
  A.5. Static Analysis ............................... 43
  A.6. Offline websites .............................. 43
Appendix B. Requirements .............................. 43
  B.1. Proof of origin ................................ 43
    B.1.1. Certificate constraints .................... 43
    B.1.2. Signature constraints ...................... 44
    B.1.3. Retrieving the certificate .................. 44
    B.2. How much to sign ............................ 45
    B.2.1. Conveying the signed headers ............... 45
    B.3. Response lifespan ............................ 46
    B.3.1. Certificate revocation ...................... 46
    B.3.2. Response downgrade attacks ................. 46
    B.4. Low implementation complexity ................. 47
    B.4.1. Limited choices ........................... 47
    B.4.2. Bounded-buffering integrity checking ......... 47
Appendix C. Determining validity using cache control ....... 48
  C.1. Example of updating cache control ............... 48
Signed HTTP exchanges provide a way to prove the authenticity of a resource in cases where the transport layer isn’t sufficient. This can be used in several ways:

- When signed by a certificate ([RFC5280]) that’s trusted for an origin, an exchange can be treated as authoritative for that origin, even if it was transferred over a connection that isn’t authoritative (Section 9.1 of [RFC7230]) for that origin. See Appendix A.1 and Appendix A.2.

- A top-level resource can use a public key to identify an expected publisher for particular subresources, a system known as Subresource Integrity ([SRI]). An exchange’s signature provides the matching proof of authorship. See Appendix A.3.

- A signature can vouch for the exchange in some way, for example that it appears in a transparency log or that static analysis indicates that it omits certain attacks. See Appendix A.4 and Appendix A.5.

Subsequent work toward the use cases in [I-D.yasskin-webpackage-use-cases] will provide a way to group signed exchanges into bundles that can be transmitted and stored together, but single signed exchanges are useful enough to standardize on their own.

2. Terminology

Author  The entity that wrote the content in a particular resource. This specification deals with publishers rather than authors.

Publisher  The entity that controls the server for a particular origin ([RFC6454]). The publisher can get a CA to issue certificates for their private keys and can run a TLS server for their origin.

Exchange  An HTTP request/response pair. This can either be a request from a client and the matching response from a server or the request in a PUSH_PROMISE and its matching response stream. Defined by Section 8 of [RFC7540].
Intermediate An entity that fetches signed HTTP exchanges from a publisher or another intermediate and forwards them to another intermediate or a client.

Client An entity that uses a signed HTTP exchange and needs to be able to prove that the publisher vouched for it as coming from its claimed origin.

Unix time Defined by [POSIX] section 4.16 [3].

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Signing an exchange

In the response of an HTTP exchange the server MAY include a "Signature" header field (Section 3.1) holding a list of one or more parameterised signatures that vouch for the content of the exchange. Exactly which content the signature vouches for can depend on how the exchange is transferred (Section 5).

The client categorizes each signature as "valid" or "invalid" by validating that signature with its certificate or public key and other metadata against the exchange’s headers and content (Section 3.5). This validity then informs higher-level protocols.

Each signature is parameterised with information to let a client fetch assurance that a signed exchange is still valid, in the face of revoked certificates and newly-discovered vulnerabilities. This assurance can be bundled back into the signed exchange and forwarded to another client, which won’t have to re-fetch this validity information for some period of time.

3.1. The Signature Header

The "Signature" header field conveys a list of signatures for an exchange, each one accompanied by information about how to determine the authority of and refresh that signature. Each signature directly signs the exchange’s headers and identifies one of those headers that enforces the integrity of the exchange’s payload.

The "Signature" header is a Structured Header as defined by [I-D.ietf-httpbis-header-structure]. Its value MUST be a parameterised list (Section 3.3 of [I-D.ietf-httpbis-header-structure]). Its ABNF is:
Signature = sh-param-list

Each parameterised identifier in the list MUST have parameters named "sig", "integrity", "validity-url", "date", and "expires". Each parameterised identifier MUST also have either "cert-url" and "cert-sha256" parameters or an "ed25519key" parameter. This specification gives no meaning to the identifier itself, which can be used as a human-readable identifier for the signature (see Section 3.1.2, Paragraph 1). The present parameters MUST have the following values:

"sig" Binary content (Section 3.9 of [I-D.ietf-httpbis-header-structure]) holding the signature of most of these parameters and the exchange’s headers.

"integrity" A string (Section 3.7 of [I-D.ietf-httpbis-header-structure]) containing the lowercase name of the response header field that guards the response payload’s integrity.

"cert-url" A string (Section 3.7 of [I-D.ietf-httpbis-header-structure]) containing an absolute-URL string [4] ([URL]).

"cert-sha256" Binary content (Section 3.9 of [I-D.ietf-httpbis-header-structure]) holding the SHA-256 hash of the first certificate found at "cert-url".

"ed25519key" Binary content (Section 3.9 of [I-D.ietf-httpbis-header-structure]) holding an Ed25519 public key ([RFC8032]).

"validity-url" A string (Section 3.7 of [I-D.ietf-httpbis-header-structure]) containing an absolute-URL string [5] ([URL]).

"date" and "expires" An integer (Section 3.5 of [I-D.ietf-httpbis-header-structure]) representing a Unix time.

The "cert-url" parameter is _not_ signed, so intermediates can update it with a pointer to a cached version.

3.1.1. Examples

The following header is included in the response for an exchange with effective request URI "https://example.com/resource.html". Newlines are added for readability.
There are 4 signatures: 2 from different secp256r1 certificates within "https://example.com/", one using a raw ed25519 public key that's also controlled by "example.com", and a fourth using a secp256r1 certificate owned by "thirdparty.example.com".

All 4 signatures rely on the "MI" response header to guard the integrity of the response payload.

The signatures include a "validity-url" that includes the first time the resource was seen. This allows multiple versions of a resource at the same URL to be updated with new signatures, which allows clients to avoid transferring extra data while the old versions don't have known security bugs.

The certificates at "https://example.com/oldcerts" and "https://example.com/newcerts" have "subjectAltName"s of "example.com", meaning that if they and their signatures validate, the exchange can be trusted as having an origin of "https://example.com/". The publisher might be using two
certificates because their readers have disjoint sets of roots in their trust stores.

The publisher signed with all three certificates at the same time, so they share a validity range: 7 days starting at 2017-11-19 21:53 UTC.

The publisher then requested an additional signature from "thirdparty.example.com", which did some validation or processing and then signed the resource at 2017-11-19 23:11 UTC. "thirdparty.example.com" only grants 4-day signatures, so clients will need to re-validate more often.

3.1.2. Open Questions

[I-D.ietf-httpbis-header-structure] provides a way to parameterise identifiers but not other supported types like binary content. If the "Signature" header field is notionally a list of parameterised signatures, maybe we should add a "parameterised binary content" type.

Should the cert-url and validity-url be lists so that intermediates can offer a cache without losing the original URLs? Putting lists in dictionary fields is more complex than [I-D.ietf-httpbis-header-structure] allows, so they’re single items for now.

3.2. CBOR representation of exchange headers

To sign an exchange’s headers, they need to be serialized into a byte string. Since intermediaries and distributors (Appendix A.2) might rearrange, add, or just reserialize headers, we can’t use the literal bytes of the headers as this serialization. Instead, this section defines a CBOR representation that can be embedded into other CBOR, canonically serialized (Section 3.4), and then signed.

The CBOR representation of an exchange "exchange"’s headers is the CBOR ([RFC7049]) array with the following content:

1. The map mapping:

* The byte string ’:method’ to the byte string containing "exchange"’s request’s method.

* The byte string ’:url’ to the byte string containing "exchange"’s request’s effective request URI, which MUST be an absolute-URL string [6] ([URL]).
* For each request header field in "exchange" except for the "Host" header field, the header field’s lowercase name as a byte string to the header field’s value as a byte string.

Note: "Host" is excluded because it is already part of the effective request URI.

2. The map mapping:

* the byte string ':status' to the byte string containing "exchange"’s response’s 3-digit status code, and

* for each response header field in "exchange", the header field’s lowercase name as a byte string to the header field’s value as a byte string.

3.2.1. Example

Given the HTTP exchange:

GET / HTTP/1.1
Host: example.com
Accept: */*

HTTP/1.1 200
Content-Type: text/html
MI: mi-sha256=20addcf7368837f616d549f035bf6784ea6d4bf4817a3736cd2fc7a763897fe3
Signed-Headers: "content-type", "mi"

<!doctype html>
<html>
...

The cbor representation consists of the following item, represented using the extended diagnostic notation from [I-D.ietf-cbor-cddl] appendix G:
3.3.  Loading a certificate chain

The resource at a signature’s "cert-url" MUST have the "application/cert-chain+cbor" content type, MUST be canonically-encoded CBOR (Section 3.4), and MUST match the following CDDL:

cert-chain = [
   "📜⛓", ; U+1F4DC U+26D3
   + {
      cert: bytes,
      ? ocsp: bytes,
      ? sct: bytes,
      * tstr => any,
   }
]

The first item in the CBOR array is treated as the end-entity certificate, and the client will attempt to build a path (RFC5280) to it from a trusted root using the other certificates in the chain.

1. Each "cert" value MUST be a DER-encoded X.509v3 certificate (RFC5280). Other key/value pairs in the same array item define properties of this certificate.

2. The first certificate’s "ocsp" value if any MUST be a complete, DER-encoded OCSP response for that certificate (using the ASN.1 type "OCSPResponse" defined in RFC6960). Subsequent certificates MUST NOT have an "ocsp" value.

3. Each certificate’s "sct" value MUST be a "SignedCertificateTimestampList" for that certificate as defined by Section 3.3 of RFC6962.

Loading a "cert-url" takes a "forceFetch" flag. The client MUST:
1. Let "raw-chain" be the result of fetching ([FETCH]) "cert-url". If "forceFetch" is _not_ set, the fetch can be fulfilled from a cache using normal HTTP semantics [RFC7234]. If this fetch fails, return "invalid".

2. Let "certificate-chain" be the array of certificates and properties produced by parsing "raw-chain" using the CDDL above. If any of the requirements above aren’t satisfied, return "invalid". Note that this validation requirement might be impractical to completely achieve due to certificate validation implementations that don’t enforce DER encoding or other standard constraints.

3. Return "certificate-chain".

3.4. Canonical CBOR serialization

Within this specification, the canonical serialization of a CBOR item uses the following rules derived from Section 3.9 of [RFC7049] with erratum 4964 applied:

- Integers and the lengths of arrays, maps, and strings MUST use the smallest possible encoding.
- Items MUST NOT be encoded with indefinite length.
- The keys in every map MUST be sorted in the bytewise lexicographic order of their canonical encodings. For example, the following keys are correctly sorted:
  1. 10, encoded as 0A.
  2. 100, encoded as 18 64.
  3. -1, encoded as 20.
  4. "z", encoded as 61 7A.
  5. "aa", encoded as 62 61 61.
  6. [100], encoded as 81 18 64.
  7. [-1], encoded as 81 20.
  8. false, encoded as F4.
Note: this specification does not use floating point, tags, or other more complex data types, so it doesn’t need rules to canonicalize those.

3.5. Signature validity

The client MUST parse the "Signature" header field as the parameterised list (Section 4.2.3 of [I-D.ietf-httpbis-header-structure]) described in Section 3.1. If an error is thrown during this parsing or any of the requirements described there aren’t satisfied, the exchange has no valid signatures. Otherwise, each member of this list represents a signature with parameters.

The client MUST use the following algorithm to determine whether each signature with parameters is invalid or potentially-valid for an "exchange". Potentially-valid results include:

- The signed headers of the exchange so that higher-level protocols can avoid relying on unsigned headers, and
- Either a certificate chain or a public key so that a higher-level protocol can determine whether it’s actually valid.

This algorithm accepts a "forceFetch" flag that avoids the cache when fetching URLs. A client that determines that a potentially-valid certificate chain is actually invalid due to an expired OCSP response MAY retry with "forceFetch" set to retrieve an updated OCSP from the original server.

1. Let "payload" be the payload body (Section 3.3 of [RFC7230]) of "exchange". Note that the payload body is the message body with any transfer encodings removed.

2. Let:

   - "signature" be the signature (binary content in the parameterised identifier’s "sig" parameter).
   - "integrity" be the signature’s "integrity" parameter.
   - "validity-url" be the signature’s "validity-url" parameter.
   - "cert-url" be the signature’s "cert-url" parameter, if any.
   - "cert-sha256" be the signature’s "cert-sha256" parameter, if any.
* "ed25519key" be the signature’s "ed25519key" parameter, if any.
* "date" be the signature’s "date" parameter, interpreted as a Unix time.
* "expires" be the signature’s "expires" parameter, interpreted as a Unix time.

3. If "integrity" names a header field that is not present in "exchange"’s response headers or which the client cannot use to check the integrity of "payload" (for example, the header field is new and hasn’t been implemented yet), then return "invalid". If the selected header field provides integrity guarantees weaker than SHA-256, return "invalid". If validating integrity using the selected header field requires the client to process records larger than TBD bytes, return "invalid". Clients MUST implement at least the "MI" ([I-D.thomson-http-mice]) header field with its "mi-sha256" content encoding.

4. Set "publicKey" and "signing-alg" depending on which key fields are present:

1. If "cert-url" is present:
   1. Let "certificate-chain" be the result of loading the certificate chain at "cert-url" passing the "forceFetch" flag (Section 3.3). If this returns "invalid", return "invalid".
   2. Let "main-certificate" be the first certificate in "certificate-chain".
   3. Set "publicKey" to "main-certificate"’s public key.
   4. If "publicKey" is an RSA key, return "invalid".
   5. If "publicKey" is a key using the secp256r1 elliptic curve, set "signing-alg" to ecdsa_secp256r1_sha256 as defined in Section 4.2.3 of [I-D.ietf-tls-tls13].
   6. Otherwise, either return "invalid" or set "signing-alg" to a non-legacy signing algorithm defined by TLS 1.3 or later ([I-D.ietf-tls-tls13]). This choice MUST depend only on "publicKey"’s type and not on any other context.

2. If "ed25519key" is present, set "publicKey" to "ed25519key" and "signing-alg" to ed25519, as defined by [RFC8032]
5. If "expires" is more than 7 days (604800 seconds) after "date", return "invalid".

6. If the current time is before "date" or after "expires", return "invalid".

7. Let "message" be the concatenation of the following byte strings. This matches the [I-D.ietf-tls-tls13] format to avoid cross-protocol attacks if anyone uses the same key in a TLS certificate and an exchange-signing certificate.
   1. A string that consists of octet 32 (0x20) repeated 64 times.
   2. A context string: the ASCII encoding of "HTTP Exchange 1".
      Note: RFC EDITOR PLEASE DELETE THIS NOTE; The implementation of the final RFC MUST use this context string, but implementations of drafts MUST NOT use it and MUST use another draft-specific string beginning with "HTTP Exchange 1" instead. This ensures that signers can predict how their signatures will be used.
   3. A single 0 byte which serves as a separator.
   4. The bytes of the canonical CBOR serialization (Section 3.4) of a CBOR map mapping:
      1. If "cert-sha256" is set:
         1. The text string "cert-sha256" to the byte string value of "cert-sha256".
         2. The text string "validity-url" to the byte string value of "validity-url".
         3. The text string "date" to the integer value of "date".
         4. The text string "expires" to the integer value of "expires".
         5. The text string "headers" to the CBOR representation (Section 3.2) of "exchange"’s headers.
      8. If "cert-url" is present and the SHA-256 hash of "main-certificate"’s "cert_data" is not equal to "cert-sha256" (whose presence was checked when the "Signature" header field was parsed), return "invalid".
Note that this intentionally differs from TLS 1.3, which signs the entire certificate chain in its Certificate Verify (Section 4.4.3 of [I-D.ietf-tls-tls13]), in order to allow updating the stapled OCSP response without updating signatures at the same time.

9. If "signature" is a valid signature of "message" by "publicKey" using "signing-alg", return "potentially-valid" with whichever is present of "certificate-chain" or "ed25519key". Otherwise, return "invalid".

Note that the above algorithm can determine that an exchange’s headers are potentially-valid before the exchange’s payload is received. Similarly, if "integrity" identifies a header field like "MI" ([I-D.thomson-http-mice]) that can incrementally validate the payload, early parts of the payload can be determined to be potentially-valid before later parts of the payload. Higher-level protocols MAY process parts of the exchange that have been determined to be potentially-valid as soon as that determination is made but MUST NOT process parts of the exchange that are not yet potentially-valid. Similarly, as the higher-level protocol determines that parts of the exchange are actually valid, the client MAY process those parts of the exchange and MUST wait to process other parts of the exchange until they too are determined to be valid.

3.5.1. Open Questions

Should the signed message use the TLS format (with an initial 64 spaces) even though these certificates can’t be used in TLS servers?

3.6. Updating signature validity

Both OCSP responses and signatures are designed to expire a short time after they’re signed, so that revoked certificates and signed exchanges with known vulnerabilities are distrusted promptly.

This specification provides no way to update OCSP responses by themselves. Instead, clients need to re-fetch the "cert-url" (Section 3.5, Paragraph 4) to get a chain including a newer OCSP response.

The "validity-url" parameter (Paragraph 6) of the signatures provides a way to fetch new signatures or learn where to fetch a complete updated exchange.

Each version of a signed exchange SHOULD have its own validity URLs, since each version needs different signatures and becomes obsolete at different times.
The resource at a "validity-url" is "validity data", a CBOR map matching the following CDDL ([I-D.ietf-cbor-cddl]):

```
validity = {
  ? signatures: [ + bytes ]
  ? update: {
    ? size: uint,
  }
}
```

The elements of the "signatures" array are parameterised identifiers (Section 4.2.4 of [I-D.ietf-httpbis-header-structure]) meant to replace the signatures within the "Signature" header field pointing to this validity data. If the signed exchange contains a bug severe enough that clients need to stop using the content, the "signatures" array MUST NOT be present.

If the the "update" map is present, that indicates that a new version of the signed exchange is available at its effective request URI (Section 5.5 of [RFC7230]) and can give an estimate of the size of the updated exchange ("update.size"). If the signed exchange is currently the most recent version, the "update" SHOULD NOT be present.

If both the "signatures" and "update" fields are present, clients can use the estimated size to decide whether to update the whole resource or just its signatures.

3.6.1. Examples

For example, say a signed exchange whose URL is "https://example.com/resource" has the following "Signature" header field (with line breaks included and irrelevant fields omitted for ease of reading).
Signature:
sig1;
sig=*MEUCIQ...*;
...
validity-url="https://example.com/resource.validity.1511157180";
cert-url="https://example.com/oldcerts";
date=1511128380; expires=1511733180,
sig2;
sig=*MEQCIG...*;
...
validity-url="https://example.com/resource.validity.1511157180";
cert-url="https://example.com/newcerts";
date=1511128380; expires=1511733180,
thirdpartysig;
sig=*MEYCIQ...*;
...
validity-url="https://thirdparty.example.com/resource.validity.1511161860";
cert-url="https://thirdparty.example.com/certs";
date=1511478660; expires=1511824260

At 2017-11-27 11:02 UTC, "sig1" and "sig2" have expired, but "thirdpartysig" doesn’t expire until 23:11 that night, so the client needs to fetch "https://example.com/resource.validity.1511157180" (the "validity-url" of "sig1" and "sig2") to update those signatures. This URL might contain:

```
{
  "signatures": [
    "sig1;",
    "sig=*MEQCIC/I9Q+7BZFP6cSDsWx43pBAL0ujTbON/+7RwKVk+ba5AiB3FSFLZqpzmDJ0NumNwN04pqgJZE99fcK86UjkPbj4jw=*;",
    "validity-url="https://example.com/resource.validity.1511157180";",
    "integrity="mi";
    "cert-url="https://example.com/newcerts";
    "cert-sha256=*J/lEm9kNRODdCmINbvitpvdYKNQ+YgBj99D1Yp4fEXw=*;",
    "date=1511733180; expires=1512337980"
  ],
  "update": {
    "size": 5557452
  }
}
```

This indicates that the client could fetch a newer version at "https://example.com/resource" (the original URL of the exchange), or that the validity period of the old version can be extended by replacing the first two of the original signatures (the ones with a validity-url of "https://example.com/resource.validity.1511157180") with the single new signature provided. (This might happen at the
end of a migration to a new root certificate.) The signatures of the updated signed exchange would be:

Signature:
  sig1;
  sig=*MEQCIC...*;
  ... 
  validity-url="https://example.com/resource.validity.1511157180";
  cert-url="https://example.com/newcerts";
  date=1511733180; expires=1512337980,
  thirdparty.sig;
  sig=*MEYCIQ...*;
  ... 
  validity-url="https://thirdparty.example.com/resource.validity.1511161860";
  cert-url="https://thirdparty.example.com/certs";
  date=1511478660; expires=1511824260

"https://example.com/resource.validity.1511157180" could also expand the set of signatures if its "signatures" array contained more than 2 elements.

3.7. The Accept-Signature header

"Signature" header fields cost on the order of 300 bytes for ECDSA signatures, so servers might prefer to avoid sending them to clients that don’t intend to use them. A client can send the "Accept-Signature" header field to indicate that it does intend to take advantage of any available signatures and to indicate what kinds of signatures it supports.

When a server receives an "Accept-Signature" header field in a client request, it SHOULD reply with any available "Signature" header fields for its response that the "Accept-Signature" header field indicates the client supports. However, if the "Accept-Signature" value violates a requirement in this section, the server MUST behave as if it hadn’t received any "Accept-Signature" header at all.

The "Accept-Signature" header field is a Structured Header as defined by [I-D.ietf-httpbis-header-structure]. Its value MUST be a parameterised list (Section 3.3 of [I-D.ietf-httpbis-header-structure]). Its ABNF is:

Accept-Signature = sh-param-list

The order of identifiers in the "Accept-Signature" list is not significant. Identifiers, ignoring any initial "-" character, MUST NOT be duplicated.
Each identifier in the "Accept-Signature" header field’s value indicates that a feature of the "Signature" header field (Section 3.1) is supported. If the identifier begins with a "-" character, it instead indicates that the feature named by the rest of the identifier is not supported. Unknown identifiers and parameters MUST be ignored because new identifiers and new parameters on existing identifiers may be defined by future specifications.

3.7.1. Integrity identifiers

Identifiers starting with "mi/" indicate that the client supports the "MI" header field ([I-D.thomson-http-mice]) with the parameter from the HTTP MI Parameter Registry registry named in lower-case by the rest of the identifier. For example, "mi/mi-blake2" indicates support for Merkle integrity with the as-yet-unspecified mi-blake2 parameter, and "-digest/mi-sha256" indicates non-support for Merkle integrity with the mi-sha256 content encoding.

If the "Accept-Signature" header field is present, servers SHOULD assume support for "mi/mi-sha256" unless the header field states otherwise.

3.7.2. Key type identifiers

Identifiers starting with "ecdsa/" indicate that the client supports certificates holding ECDSA public keys on the curve named in lower-case by the rest of the identifier.

If the "Accept-Signature" header field is present, servers SHOULD assume support for "ecdsa/secp256r1" unless the header field states otherwise.

3.7.3. Key value identifiers

The "ed25519key" identifier has parameters indicating the public keys that will be used to validate the returned signature. Each parameter’s name is re-interpreted as binary content (Section 3.9 of [I-D.ietf-httpbis-header-structure]) encoding a prefix of the public key. For example, if the client will validate signatures using the public key whose base64 encoding is "11qYAYxCrVS/7TyWQHOg7hcvPapiMlrwIaaPcHURo=", valid "Accept-Signature" header fields include:

Accept-Signature: ..., ed25519key; *11qYAYxCrVS/7TyWQHOg7hcvPapiMlrwIaaPcHURo=*
Accept-Signature: ..., ed25519key; *11qYAYxCrVS/7TyWQHOg==*
Accept-Signature: ..., ed25519key; *11qYAG==*
Accept-Signature: ..., ed25519key; **
but not

Accept-Signature: ..., ed25519key; *11qYA===*

because 5 bytes isn’t a valid length for encoded base64, and not

Accept-Signature: ..., ed25519key; 11qYAQ

because it doesn’t start or end with the "*"s that indicate binary content.

Note that "ed25519key; **" is an empty prefix, which matches all public keys, so it’s useful in subresource integrity (Appendix A.3) cases like "<link rel=preload as=script href="...">" where the public key isn’t known until the matching "<script src="..." integrity="...">" tag.

3.7.4. Examples

Accept-Signature: mi/mi-sha256

states that the client will accept signatures with payload integrity assured by the "MI" header and "mi-sha256" content encoding and implies that the client will accept integrity assured by the "Digest: SHA-256" header and signatures from ECDSA keys on the secp256r1 curve.

Accept-Signature: -ecdsa/secp256r1, ecdsa/secp384r1

states that the client will accept ECDSA keys on the secp384r1 curve but not the secp256r1 curve and payload integrity assured with the "MI: mi-sha256" header field.

3.7.5. Open Questions

Is an "Accept-Signature" header useful enough to pay for itself? If clients wind up sending it on most requests, that may cost more than the cost of sending "Signature"s unconditionally. On the other hand, it gives servers an indication of which kinds of signatures are supported, which can help us upgrade the ecosystem in the future.

Is "Accept-Signature" the right spelling, or do we want to imitate "Want-Digest" (Section 4.3.1 of [RFC3230]) instead?

Do I have the right structure for the identifiers indicating feature support?
4. Cross-origin trust

To determine whether to trust a cross-origin exchange, the client takes a "Signature" header field (Section 3.1) and the "exchange". The client MUST parse the "Signature" header into a list of signatures according to the instructions in Section 3.5, and run the following algorithm for each signature, stopping at the first one that returns "valid". If any signature returns "valid", return "valid". Otherwise, return "invalid".

1. If the signature’s "validity-url" parameter (Paragraph 6) is not same-origin [7] with "exchange"’s effective request URI (Section 5.5 of [RFC7230]), return "invalid".

2. Use Section 3.5 to determine the signature’s validity for "exchange", getting "certificate-chain" back. If this returned "invalid" or didn’t return a certificate chain, return "invalid".

3. If "exchange”’s request method is not safe (Section 4.2.1 of [RFC7231]) or not cacheable (Section 4.2.3 of [RFC7231]), return "invalid".

4. If "exchange"’s headers contain a stateful header field, as defined in Section 4.1, return "invalid".

5. Let "authority" be the host component of "exchange"’s effective request URI.

6. Validate the "certificate-chain" using the following substeps. If any of them fail, re-run Section 3.5 once over the signature with the "forceFetch" flag set, and restart from step 2. If a substep fails again, return "invalid".

   1. Use "certificate-chain" to validate that its first entry, "main-certificate" is trusted as "authority”’s server certificate ([RFC5280] and other undocumented conventions). Let "path" be the path that was used from the "main-certificate" to a trusted root, including the "main-certificate" but excluding the root.

   2. Validate that "main-certificate" has the CanSignHttpExchanges extension (Section 4.2).

   3. Validate that "main-certificate" has an "ocsp" property (Section 3.3) with a valid OCSP response whose lifetime ("nextUpdate - thisUpdate") is less than 7 days ([RFC6960]). Note that this does not check for revocation of intermediate
certificates, and clients SHOULD implement another mechanism for that.

4. Validate that valid SCTs from trusted logs are available from any of:

+ The "SignedCertificateTimestampList" in "main-certificate"’s "sct" property (Section 3.3),

+ An OCSP extension in the OCSP response in "main-certificate"’s "ocsp" property, or

+ An X.509 extension in the certificate in "main-certificate"’s "cert" property,

as described by Section 3.3 of [RFC6962].

7. Return "valid".

4.1. Stateful header fields

As described in Section 6.1, a publisher can cause problems if they sign an exchange that includes private information. There’s no way for a client to be sure an exchange does or does not include private information, but header fields that store or convey stored state in the client are a good sign.

A stateful request header field informs the server of per-client state. These include but are not limited to:

- "Authorization", [RFC7235]
- "Cookie", [RFC6265]
- "Cookie2", [RFC2965]
- "Proxy-Authorization", [RFC7235]
- "Sec-WebSocket-Key", [RFC6455]

A stateful response header field modifies state, including authentication status, in the client. The HTTP cache is not considered part of this state. These include but are not limited to:

- "Authentication-Control", [RFC8053]
- "Authentication-Info", [RFC7615]
o "Optional-WWW-Authenticate", [RFC8053]
o "Proxy-Authenticate", [RFC7235]
o "Proxy-Authentication-Info", [RFC7615]
o "Sec-WebSocket-Accept", [RFC6455]
o "Set-Cookie", [RFC6265]
o "Set-Cookie2", [RFC2965]
o "SetProfile", [W3C.NOTE-OPS-OverHTTP]
o "WWW-Authenticate", [RFC7235]

4.2. Certificate Requirements

We define a new X.509 extension, CanSignHttpExchanges to be used in the certificate when the certificate permits the usage of signed exchanges. When this extension is not present the client MUST NOT accept a signature from the certificate as proof that a signed exchange is authoritative for a domain covered by the certificate. When it is present, the client MUST follow the validation procedure in Section 4.

    id-ce-canSignHttpExchanges OBJECT IDENTIFIER ::= { TBD }

    CanSignHttpExchanges ::= NULL

Note that this extension contains an ASN.1 NULL (bytes "05 00") because some implementations have bugs with empty extensions.

Leaf certificates without this extension need to be revoked if the private key is exposed to an unauthorized entity, but they generally don’t need to be revoked if a signing oracle is exposed and then removed.

CA certificates, by contrast, need to be revoked if an unauthorized entity is able to make even one unauthorized signature.

Certificates with this extension MUST be revoked if an unauthorized entity is able to make even one unauthorized signature.

Conforming CAs MUST mark this extension as critical, and clients MUST NOT accept certificates with this extension in TLS connections (Section 4.4.2.2 of [I-D.ietf-tls-tls13]). This simplifies security analysis of this protocol and avoids encouraging server operators to
put exchange-signing keys on servers exposed directly to the internet.

RFC EDITOR PLEASE DELETE THE REST OF THE PARAGRAPHS IN THIS SECTION

id-ce-google OBJECT IDENTIFIER ::= { 1 3 6 1 4 1 11129 }
id-ce-testCanSignHttpExchanges OBJECT IDENTIFIER ::= { id-ce-google 2 1 22 }

Implementations of drafts of this specification MAY recognize the "id-ce-testCanSignHttpExchanges" OID as identifying the CanSignHttpExchanges extension. This OID might or might not be used as the final OID for the extension, so certificates including it might need to be reissued once the final RFC is published.

5. Transferring a signed exchange

A signed exchange can be transferred in several ways, of which three are described here.

5.1. Same-origin response

The signature for a signed exchange can be included in a normal HTTP response. Because different clients send different request header fields, and intermediate servers add response header fields, it can be impossible to have a signature for the exact request and response that the client sees. Therefore, when a client validates the "Signature" header field for an exchange represented as a normal HTTP request/response pair, it MUST pass only the subset of header fields defined by Section 5.1.1 to the validation procedure (Section 3.5).

If the client relies on signature validity for any aspect of its behavior, it MUST ignore any header fields that it didn’t pass to the validation procedure.

5.1.1. Significant headers for a same-origin response

The significant headers of an exchange represented as a normal HTTP request/response pair (Section 2.1 of [RFC7230] or Section 8.1 of [RFC7540]) are:

- The method (Section 4 of [RFC7231]) and effective request URI (Section 5.5 of [RFC7230]) of the request.
- The response status code (Section 6 of [RFC7231]) and the response header fields whose names are listed in that exchange’s "Signed-Headers" header field (Section 5.1.2), in the order they appear in that header field. If a response header field name from "Signed-
Headers" does not appear in the exchange’s response header fields, the exchange has no significant headers.

If the exchange’s "Signed-Headers" header field is not present, doesn’t parse as a Structured Header ([I-D.ietf-httpbis-header-structure]) or doesn’t follow the constraints on its value described in Section 5.1.2, the exchange has no significant headers.

5.1.1.1. Open Questions

Do the significant headers of an exchange need to include the "Signed-Headers" header field itself?

5.1.2. The Signed-Headers Header

The "Signed-Headers" header field identifies an ordered list of response header fields to include in a signature. The request URL and response status are included unconditionally. This allows a TLS-terminating intermediate to reorder headers without breaking the signature. This _can_ also allow the intermediate to add headers that will be ignored by some higher-level protocols, but Section 3.5 provides a hook to let other higher-level protocols reject such insecure headers.

This header field appears once instead of being incorporated into the signatures’ parameters because the signed header fields need to be consistent across all signatures of an exchange, to avoid forcing higher-level protocols to merge the header field lists of valid signatures.

See Appendix B.2 for a discussion of why only the URL from the request is included and not other request headers.

"Signed-Headers" is a Structured Header as defined by [I-D.ietf-httpbis-header-structure]. Its value MUST be a list (Section 3.2 of [I-D.ietf-httpbis-header-structure]). Its ABNF is:

Signed-Headers = sh-list

Each element of the "Signed-Headers" list must be a lowercase string (Section 3.7 of [I-D.ietf-httpbis-header-structure]) naming an HTTP response header field. Pseudo-header field names (Section 8.1.2.1 of [RFC7540]) MUST NOT appear in this list.

Higher-level protocols SHOULD place requirements on the minimum set of headers to include in the "Signed-Headers" header field.
5.2. HTTP/2 extension for cross-origin Server Push

To allow servers to Server-Push (Section 8.2 of [RFC7540]) signed exchanges (Section 3) signed by an authority for which the server is not authoritative (Section 9.1 of [RFC7230]), this section defines an HTTP/2 extension.

5.2.1. Indicating support for cross-origin Server Push

Clients that might accept signed Server Pushes with an authority for which the server is not authoritative indicate this using the HTTP/2 SETTINGS parameter ENABLE_CROSS_ORIGIN_PUSH (0xSETTING-TBD).

An ENABLE_CROSS_ORIGIN_PUSH value of 0 indicates that the client does not support cross-origin Push. A value of 1 indicates that the client does support cross-origin Push.

A client MUST NOT send a ENABLE_CROSS_ORIGIN_PUSH setting with a value other than 0 or 1 or a value of 0 after previously sending a value of 1. If a server receives a value that violates these rules, it MUST treat it as a connection error (Section 5.4.1 of [RFC7540]) of type PROTOCOL_ERROR.

The use of a SETTINGS parameter to opt-in to an otherwise incompatible protocol change is a use of "Extending HTTP/2" defined by Section 5.5 of [RFC7540]. If a server were to send a cross-origin Push without first receiving a ENABLE_CROSS_ORIGIN_PUSH setting with the value of 1 it would be a protocol violation.

5.2.2. NO_TRUSTED_EXCHANGE_SIGNATURE error code

The signatures on a Pushed cross-origin exchange may be untrusted for several reasons, for example that the certificate could not be fetched, that the certificate does not chain to a trusted root, that the signature itself doesn’t validate, that the signature is expired, etc. This draft conflates all of these possible failures into one error code, NO_TRUSTED_EXCHANGE_SIGNATURE (0xERROR-TBD).

5.2.2.1. Open Questions

How fine-grained should this specification’s error codes be?

5.2.3. Validating a cross-origin Push

If the client has set the ENABLE_CROSS_ORIGIN_PUSH setting to 1, the server MAY Push a signed exchange for which it is not authoritative, and the client MUST NOT treat a PUSH_PROMISE for which the server is
not authoritative as a stream error (Section 5.4.2 of [RFC7540]) of type PROTOCOL_ERROR, as described in Section 8.2 of [RFC7540].

Instead, the client MUST validate such a PUSH_PROMISE and its response by taking the "Signature" header field from the response, and the exchange consisting of the PUSH_PROMISE and the response without that "Signature" header field, and passing them to the algorithm in Section 4. If this returns "invalid", the client MUST treat the response as a stream error (Section 5.4.2 of [RFC7540]) of type NO_TRUSTED_EXCHANGE_SIGNATURE. Otherwise, the client MUST treat the pushed response as if the server were authoritative for the PUSH_PROMISE’s authority.

5.2.3.1. Open Questions

Is it right that "validity-url" is required to be same-origin with the exchange? This allows the mitigation against downgrades in Section 6.3, but prohibits intermediates from providing a cache of the validity information. We could do both with a list of URLs.

5.3. application/signed-exchange format

To allow signed exchanges to be the targets of "<link rel=prefetch>" tags, we define the "application/signed-exchange" content type that represents a signed HTTP exchange, including request metadata and header fields, response metadata and header fields, and a response payload.

This content type consists of the concatenation of the following items:

1. The ASCII characters "sxg1" followed by a 0 byte, to serve as a file signature. This is redundant with the MIME type, and recipients that receive both MUST check that they match and stop parsing if they don’t.

   Note: RFC EDITOR PLEASE DELETE THIS NOTE; The implementation of the final RFC MUST use this file signature, but implementations of drafts MUST NOT use it and MUST use another implementation-specific string beginning with "sxg1-" and ending with a 0 byte instead.

2. 3 bytes storing a big-endian integer "sigLength". If this is larger than TBD, parsing MUST fail.

3. 3 bytes storing a big-endian integer "headerLength". If this is larger than TBD, parsing MUST fail.
4. "sigLength" bytes holding the "Signature" header field’s value (Section 3.1).

5. "headerLength" bytes holding the signed headers, the canonical serialization (Section 3.4) of the CBOR representation of the request and response headers of the exchange represented by the "application/signed-exchange" resource (Section 3.2), excluding the "Signature" header field.

Note that this is exactly the bytes used when checking signature validity in Section 3.5.

6. The payload body (Section 3.3 of [RFC7230]) of the exchange represented by the "application/signed-exchange" resource.

Note that the use of the payload body here means that a "Transfer-Encoding" header field inside the "application/signed-exchange" header block has no effect. A "Transfer-Encoding" header field on the outer HTTP response that transfers this resource still has its normal effect.

5.3.1. Cross-origin trust in application/signed-exchange

To determine whether to trust a cross-origin exchange stored in an "application/signed-exchange" resource, pass the "Signature" header field from the non-signed header section and an exchange consisting of the headers from the signed headers section and the payload body, to the algorithm in Section 4.

5.3.2. Example

An example "application/signed-exchange" file representing a possible signed exchange with https://example.com/ [8] follows, with lengths represented by descriptions in "<>"s, CBOR represented in the extended diagnostic format defined in Appendix G of [I-D.ietf-cbor-cddl], and most of the "Signature" header field and payload elided with a ...:
sxgl\0<3-byte length of the following header
value>sig1; sig="...; integrity="mi"; ...<3-byte length of the encoding of the following array>
{
    ':method': 'GET',
    ':url': 'https://example.com/",
    'accept', '*/*'
},
{
    ':status': '200',
    'content-type': 'text/html'
}
<!doctype html>
<html>...

5.3.3. Open Questions

Should this be a CBOR format, or is the current mix of binary and CBOR better?

Are the mime type, extension, and magic number right?

6. Security considerations

6.1. Over-signing

If a publisher blindly signs all responses as their origin, they can cause at least two kinds of problems, described below. To avoid this, publishers SHOULD design their systems to opt particular public content that doesn’t depend on authentication status into signatures instead of signing by default.

Signing systems SHOULD also incorporate the following mitigations to reduce the risk that private responses are signed:

1. Strip the "Cookie" request header field and other identifying information like client authentication and TLS session IDs from requests whose exchange is destined to be signed, before forwarding the request to a backend.

2. Only sign exchanges where the response includes a "Cache-Control: public" header. Clients are not required to fail signature-checking for exchanges that omit this "Cache-Control" response header field to reduce the risk that naive signing systems blindly add it.
6.1.1. Session fixation

Blind signing can sign responses that create session cookies or otherwise change state on the client to identify a particular session. This breaks certain kinds of CSRF defense and can allow an attacker to force a user into the attacker’s account, where the user might unintentionally save private information, like credit card numbers or addresses.

This specification defends against cookie-based attacks by blocking the "Set-Cookie" response header, but it cannot prevent Javascript or other response content from changing state.

6.1.2. Misleading content

If a site signs private information, an attacker might set up their own account to show particular private information, forward that signed information to a victim, and use that victim’s confusion in a more sophisticated attack.

Stripping authentication information from requests before sending them to backends is likely to prevent the backend from showing attacker-specific information in the signed response. It does not prevent the attacker from showing their victim a signed-out page when the victim is actually signed in, but while this is still misleading, it seems less likely to be useful to the attacker.

6.2. Off-path attackers

Relaxing the requirement to consult DNS when determining authority for an origin means that an attacker who possesses a valid certificate no longer needs to be on-path to redirect traffic to them; instead of modifying DNS, they need only convince the user to visit another Web site in order to serve responses signed as the target. This consideration and mitigations for it are shared by the combination of [RFC8336] and [I-D.ietf-httpbis-http2-secondary-certs].

6.3. Downgrades

Signing a bad response can affect more users than simply serving a bad response, since a served response will only affect users who make a request while the bad version is live, while an attacker can forward a signed response until its signature expires. Publishers should consider shorter signature expiration times than they use for cache expiration times.
Clients MAY also check the "validity-url" (Paragraph 6) of an exchange more often than the signature’s expiration would require. Doing so for an exchange with an HTTPS request URI provides a TLS guarantee that the exchange isn’t out of date (as long as Section 5.2.3.1 is resolved to keep the same-origin requirement).

6.4. Signing oracles are permanent

An attacker with temporary access to a signing oracle can sign "still valid" assertions with arbitrary timestamps and expiration times. As a result, when a signing oracle is removed, the keys it provided access to MUST be revoked so that, even if the attacker used them to sign future-dated exchange validity assertions, the key’s OCSP assertion will expire, causing the exchange as a whole to become untrusted.

6.5. Unsigned headers

The use of a single "Signed-Headers" header field prevents us from signing aspects of the request other than its effective request URI (Section 5.5 of [RFC7230]). For example, if a publisher signs both "Content-Encoding: br" and "Content-Encoding: gzip" variants of a response, what’s the impact if an attacker serves the brotli one for a request with "Accept-Encoding: gzip"?

The simple form of "Signed-Headers" also prevents us from signing less than the full request URL. The SRI use case (Appendix A.3) may benefit from being able to leave the authority less constrained.

Section 3.5 can succeed when some delivered headers aren’t included in the signed set. This accommodates current TLS-terminating intermediates and may be useful for SRI (Appendix A.3), but is risky for trusting cross-origin responses (Appendix A.1, Appendix A.2, and Appendix A.6). Section 5.2 requires all headers to be included in the signature before trusting cross-origin pushed resources, at Ryan Sleevi’s recommendation.

6.6. application/signed-exchange

Clients MUST NOT trust an effective request URI claimed by an "application/signed-exchange" resource (Section 5.3) without either ensuring the resource was transferred from a server that was authoritative (Section 9.1 of [RFC7230]) for that URI’s origin, or calling the algorithm in Section 5.3.1 and getting "valid" back.
7. Privacy considerations

Normally, when a client fetches "https://o1.com/resource.js", "o1.com" learns that the client is interested in the resource. If "o1.com" signs "resource.js", "o2.com" serves it as "https://o2.com/o1resource.js", and the client fetches it from there, then "o2.com" learns that the client is interested, and if the client executes the Javascript, that could also report the client’s interest back to "o1.com".

Often, "o2.com" already knew about the client’s interest, because it’s the entity that directed the client to "o1resource.js", but there may be cases where this leaks extra information.

For non-executable resource types, a signed response can improve the privacy situation by hiding the client’s interest from the original publisher.

To prevent network operators other than "o1.com" or "o2.com" from learning which exchanges were read, clients SHOULD only load exchanges fetched over a transport that’s protected from eavesdroppers. This can be difficult to determine when the exchange is being loaded from local disk, but when the client itself requested the exchange over a network it SHOULD require TLS ([I-D.ietf-tls-tls13]) or a successor transport layer, and MUST NOT accept exchanges transferred over plain HTTP without TLS.

8. IANA considerations

TODO: possibly register the validity-url format.

8.1. Signature Header Field Registration

This section registers the "Signature" header field in the "Permanent Message Header Field Names" registry ([RFC3864]).

Header field name: "Signature"

Applicable protocol: http

Status: standard

Author/Change controller: IETF

Specification document(s): Section 3.1 of this document
8.2. Accept-Signature Header Field Registration

This section registers the "Accept-Signature" header field in the "Permanent Message Header Field Names" registry ([RFC3864]).

Header field name: "Accept-Signature"

Applicable protocol: http

Status: standard

Author/Change controller: IETF

Specification document(s): Section 3.7 of this document

8.3. Signed-Headers Header Field Registration

This section registers the "Signed-Headers" header field in the "Permanent Message Header Field Names" registry ([RFC3864]).

Header field name: "Signed-Headers"

Applicable protocol: http

Status: standard

Author/Change controller: IETF

Specification document(s): Section 5.1.2 of this document

8.4. HTTP/2 Settings

This section establishes an entry for the HTTP/2 Settings Registry that was established by Section 11.3 of [RFC7540]

Name: ENABLE_CROSS_ORIGIN_PUSH

Code: 0xSETTING-TBD

Initial Value: 0

Specification: This document

8.5. HTTP/2 Error code

This section establishes an entry for the HTTP/2 Error Code Registry that was established by Section 11.4 of [RFC7540]
Name: NO_TRUSTED_EXCHANGE_SIGNATURE

Code: 0xERROR-TBD

Description: The client does not trust the signature for a cross-origin Pushed signed exchange.

Specification: This document

8.6. Internet Media Type application/signed-exchange

  Type name: application

  Subtype name: signed-exchange

  Required parameters:

    o v: A string denoting the version of the file format. ([RFC5234] ABNF: "version = DIGIT/%x61-7A") The version defined in this specification is "1". When used with the "Accept" header field (Section 5.3.1 of [RFC7231]), this parameter can be a comma (,)-separated list of version strings. ([RFC5234] ABNF: "version-list = version *( , version )") The server is then expected to reply with a resource using a particular version from that list.

    Note: RFC EDITOR PLEASE DELETE THIS NOTE; Implementations of drafts of this specification MUST NOT use simple integers to describe their versions, and MUST instead define implementation-specific strings to identify which draft is implemented.

  Optional parameters: N/A

  Encoding considerations: binary

  Security considerations: see Section 6.6

  Interoperability considerations: N/A

  Published specification: This specification (see Section 5.3).

  Applications that use this media type: N/A

  Fragment identifier considerations: N/A

  Additional information:

  Deprecated alias names for this type: N/A
Magic number(s): 73 78 67 31 00
File extension(s): .sxg
Macintosh file type code(s): N/A

Person and email address to contact for further information: See Authors’ Addresses section.

Intended usage: COMMON
Restrictions on usage: N/A
Author: See Authors’ Addresses section.
Change controller: IESG

8.7. Internet Media Type application/cert-chain+cbor

Type name: application
Subtype name: cert-chain+cbor
Required parameters: N/A
Optional parameters: N/A
Encoding considerations: binary
Security considerations: N/A
Interoperability considerations: N/A
Published specification: This specification (see Section 3.3).
Applications that use this media type: N/A
Fragment identifier considerations: N/A
Additional information:
Deprecated alias names for this type: N/A
Magic number(s): 1*9(??) 67 F0 9F 93 9C E2 9B 93
File extension(s): N/A
Macintosh file type code(s): N/A
9. References

9.1. Normative References


9.2.  Informative References


Yasskin, J., "Use Cases and Requirements for Web Packages", draft-yasskin-webpackage-use-cases-01 (work in progress), March 2018.


9.3. URIs

[1] https://lists.w3.org/Archives/Public/ietf-http-wg/
[8] https://example.com/
Appendix A. Use cases

A.1. PUSHed subresources

To reduce round trips, a server might use HTTP/2 Push (Section 8.2 of [RFC7540]) to inject a subresource from another server into the client's cache. If anything about the subresource is expired or can't be verified, the client would fetch it from the original server.

For example, if "https://example.com/index.html" includes

```html
<script src="https://jquery.com/jquery-1.2.3.min.js">

Then to avoid the need to look up and connect to "jquery.com" in the critical path, "example.com" might push that resource signed by "jquery.com".

A.2. Explicit use of a content distributor for subresources

In order to speed up loading but still maintain control over its content, an HTML page in a particular origin "O.com" could tell clients to load its subresources from an intermediate content distributor that's not authoritative, but require that those resources be signed by "O.com" so that the distributor couldn’t modify the resources. This is more constrained than the common CDN case where "O.com" has a CNAME granting the CDN the right to serve arbitrary content as "O.com".

```html

To make it easier to configure the right distributor for a given request, computation of the "physicalsrc" could be encapsulated in a custom element:

```html
<dist-img src="https://O.com/img.png"></dist-img>

where the "<dist-img>" implementation generates an appropriate "<img>" based on, for example, a "<meta name="dist-base">" tag elsewhere in the page. However, this has the downside that the
preloader [9] can no longer see the physical source to download it. The resulting delay might cancel out the benefit of using a distributor.

This could be used for some of the same purposes as SRI (Appendix A.3).

To implement this with the current proposal, the distributor would respond to the physical request to "https://distributor.com/O.com/img.png" with first a signed PUSH_PROMISE for "https://O.com/img.png" and then a redirect to "https://O.com/img.png".

A.3. Subresource Integrity

The W3C WebAppSec group is investigating using signatures [10] in [SRI]. They need a way to transmit the signature with the response, which this proposal provides.

Their needs are simpler than most other use cases in that the "integrity="ed25519-[public-key]"" attribute and CSP-based ways of expressing a public key don’t need that key to be wrapped into a certificate.

The "ed25519key" signature parameter supports this simpler way of attaching a key.

The current proposal for signature-based SRI describes signing only the content of a resource, while this specification requires them to sign the request URI as well. This issue is tracked in https://github.com/mikewest/signature-based-sri/issues/5 [11]. The details of what they need to sign will affect whether and how they can use this proposal.

A.4. Binary Transparency

So-called "Binary Transparency" may eventually allow users to verify that a program they’ve been delivered is one that’s available to the public, and not a specially-built version intended to attack just them. Binary transparency systems don’t exist yet, but they’re likely to work similarly to the successful Certificate Transparency logs described by [RFC6962].

Certificate Transparency depends on Signed Certificate Timestamps that prove a log contained a particular certificate at a particular time. To build the same thing for Binary Transparency logs containing HTTP resources or full websites, we’ll need a way to provide signatures of those resources, which signed exchanges provides.
A.5. Static Analysis

Native app stores like the Apple App Store [12] and the Android Play Store [13] grant their contents powerful abilities, which they attempt to make safe by analyzing the applications before offering them to people. The web has no equivalent way for people to wait to run an update of a web application until a trusted authority has vouched for it.

While full application analysis probably needs to wait until the authority can sign bundles of exchanges, authorities may be able to guarantee certain properties by just checking a top-level resource and its (SRI)-constrained sub-resources.

A.6. Offline websites

Fully-offline websites can be represented as bundles of signed exchanges, although an optimization to reduce the number of signature verifications may be needed. Work on this is in progress in the https://github.com/WICG/webpackage [14] repository.

Appendix B. Requirements

B.1. Proof of origin

To verify that a thing came from a particular origin, for use in the same context as a TLS connection, we need someone to vouch for the signing key with as much verification as the signing keys used in TLS. The obvious way to do this is to re-use the web PKI and CA ecosystem.

B.1.1. Certificate constraints

If we re-use existing TLS server certificates, we incur the risks that:

1. TLS server certificates must be accessible from online servers, so they’re easier to steal or use as signing oracles than an offline key. An exchange’s signing key doesn’t need to be online.

2. A server using an origin-trusted key for one purpose (e.g. TLS) might accidentally sign something that looks like an exchange, or vice versa.

These risks are considered too high, so we define a new X.509 certificate extension in Section 4.2 that requires CAs to issue new
certificates for this purpose. We expect at least one low-cost CA to be willing to sign certificates with this extension.

B.1.2. Signature constraints

In order to prevent an attacker who can convince the server to sign some resource from causing those signed bytes to be interpreted as something else the new X.509 extension here is forbidden from being used in TLS servers. If Section 4.2 changes to allow re-use in TLS servers, we would need to:

1. Avoid key types that are used for non-TLS protocols whose output could be confused with a signature. That may be just the "rsaEncryption" OID from [RFC8017].

2. Use the same format as TLS’s signatures, specified in Section 4.4.3 of [I-D.ietf-tls-tls13], with a context string that’s specific to this use.

The specification also needs to define which signing algorithm to use. It currently specifies that as a function from the key type, instead of allowing attacker-controlled data to specify it.

B.1.3. Retrieving the certificate

The client needs to be able to find the certificate vouching for the signing key, a chain from that certificate to a trusted root, and possibly other trust information like SCTs ([RFC6962]). One approach would be to include the certificate and its chain in the signature metadata itself, but this wastes bytes when the same certificate is used for multiple HTTP responses. If we decide to put the signature in an HTTP header, certificates are also unusually large for that context.

Another option is to pass a URL that the client can fetch to retrieve the certificate and chain. To avoid extra round trips in fetching that URL, it could be bundled (Appendix A.6) with the signed content or PUSHed (Appendix A.1) with it. The risks from the "client_certificate_url" extension (Section 11.3 of [RFC6066]) don’t seem to apply here, since an attacker who can get a client to load an exchange and fetch the certificates it references, can also get the client to perform those fetches by loading other HTML.

To avoid using an unintended certificate with the same public key as the intended one, the content of the leaf certificate or the chain should be included in the signed data, like TLS does (Section 4.4.3 of [I-D.ietf-tls-tls13]).
B.2. How much to sign

The previous [I-D.thomson-http-content-signature] and [I-D.burke-content-signature] schemes signed just the content, while [I-D.cavage-http-signatures] could also sign the response headers and the request method and path. However, the same path, response headers, and content may mean something very different when retrieved from a different server. Section 5.1.1 currently includes the whole request URL in the signature, but it’s possible we need a more flexible scheme to allow some higher-level protocols to accept a less-signed URL.

The question of whether to include other request headers--primarily the "accept*" family--is still open. These headers need to be represented so that clients wanting a different language, say, can avoid using the wrong-language response, but it’s not obvious that there’s a security vulnerability if an attacker can spoof them. For now, the proposal (Section 3) omits other request headers.

In order to allow multiple clients to consume the same signed exchange, the exchange shouldn’t include the exact request headers that any particular client sends. For example, a Japanese resource wouldn’t include

```
accept-language: ja-JP, ja;q=0.9, en;q=0.8, zh;q=0.7, *;q=0.5
```

Instead, it would probably include just

```
accept-language: ja-JP, ja
```

and clients would use the same matching logic as for PUSH_PROMISE [15] frame headers.

B.2.1. Conveying the signed headers

HTTP headers are traditionally munged by proxies, making it impossible to guarantee that the client will see the same sequence of bytes as the publisher published. In the HTTPS world, we have more end-to-end header integrity, but it’s still likely that there are enough TLS-terminating proxies that the publisher’s signatures would tend to break before getting to the client.

There’s also no way in current HTTP for the response to a client-initiated request (Section 8.1 of [RFC7540]) to convey the request headers it expected to respond to. A PUSH_PROMISE (Section 8.2 of [RFC7540]) does not have this problem, and it would be possible to introduce a response header to convey the expected request headers.
Since proxies are unlikely to modify unknown content types, we can wrap the original exchange into an "application/signed-exchange" format (Section 5.3) and include the "Cache-Control: no-transform" header when sending it.

To reduce the likelihood of accidental modification by proxies, the "application/signed-exchange" format includes a file signature that doesn’t collide with other known signatures.

To help the PUSHed subresources use case (Appendix A.1), we might also want to extend the "PUSH_PROMISE" frame type to include a signature, and that could tell intermediates not to change the ensuing headers.

B.3. Response lifespan

A normal HTTPS response is authoritative only for one client, for as long as its cache headers say it should live. A signed exchange can be re-used for many clients, and if it was generated while a server was compromised, it can continue compromising clients even if their requests happen after the server recovers. This signing scheme needs to mitigate that risk.

B.3.1. Certificate revocation

Certificates are mis-issued and private keys are stolen, and in response clients need to be able to stop trusting these certificates as promptly as possible. Online revocation checks don’t work [16], so the industry has moved to pushed revocation lists and stapled OCSP responses [RFC6066].

Pushed revocation lists work as-is to block trust in the certificate signing an exchange, but the signatures need an explicit strategy to staple OCSP responses. One option is to extend the certificate download (Appendix B.1.3) to include the OCSP response too, perhaps in the TLS 1.3 CertificateEntry [17] format.

B.3.2. Response downgrade attacks

The signed content in a response might be vulnerable to attacks, such as XSS, or might simply be discovered to be incorrect after publication. Once the author fixes those vulnerabilities or mistakes, clients should stop trusting the old signed content in a reasonable amount of time. Similar to certificate revocation, I expect the best option to be stapled "this version is still valid" assertions with short expiration times.

These assertions could be structured as:
1. A signed minimum version number or timestamp for a set of request headers: This requires that signed responses need to include a version number or timestamp, but allows a server to provide a single signature covering all valid versions.

2. A replacement for the whole exchange’s signature. This requires the publisher to separately re-sign each valid version and requires each version to include a different update URL, but allows intermediates to serve less data. This is the approach taken in Section 3.

3. A replacement for the exchange’s signature and an update for the embedded "expires" and related cache-control HTTP headers [RFC7234]. This naturally extends publishers’ intuitions about cache expiration and the existing cache revalidation behavior to signed exchanges. This is sketched and its downsides explored in Appendix C.

The signature also needs to include instructions to intermediates for how to fetch updated validity assertions.

B.4. Low implementation complexity

Simpler implementations are, all things equal, less likely to include bugs. This section describes decisions that were made in the rest of the specification to reduce complexity.

B.4.1. Limited choices

In general, we’re trying to eliminate unnecessary choices in the specification. For example, instead of requiring clients to support two methods for verifying payload integrity, we only require one.

B.4.2. Bounded-buffering integrity checking

Clients can be designed with a more-trusted network layer that decides how to trust resources and then provides those resources to less-trusted rendering processes along with handles to the storage and other resources they’re allowed to access. If the network layer can enforce that it only operates on chunks of data up to a certain size, it can avoid the complexity of spooling large files to disk.

To allow the network layer to verify signed exchanges using a bounded amount of memory, Section 5.3 requires the signature and headers to be less than TBD bytes long, and Section 3.5 requires that the MI record size be less than TBD bytes. This allows the network layer to validate a bounded chunk at a time, and pass that chunk on to a
renderer, and then forget about that chunk before processing the next one.

The "Digest" header field from [RFC3230] requires the network layer to buffer the entire response body, so it’s disallowed.

Appendix C. Determining validity using cache control

This draft could expire signature validity using the normal HTTP cache control headers ([RFC7234]) instead of embedding an expiration date in the signature itself. This section specifies how that would work, and describes why I haven’t chosen that option.

The signatures in the "Signature" header field (Section 3.1) would no longer contain "date" or "expires" fields.

The validity-checking algorithm (Section 3.5) would initialize "date" from the resource’s "Date" header field (Section 7.1.1.2 of [RFC7231]) and initialize "expires" from either the "Expires" header field (Section 5.3 of [RFC7234]) or the "Cache-Control" header field’s "max-age" directive (Section 5.2.2.8 of [RFC7234]) (added to "date"), whichever is present, preferring "max-age" (or failing) if both are present.

Validity updates (Section 3.6) would include a list of replacement response header fields. For each header field name in this list, the client would remove matching header fields from the stored exchange’s response header fields. Then the client would append the replacement header fields to the stored exchange’s response header fields.

C.1. Example of updating cache control

For example, given a stored exchange of:

GET / HTTP/1.1
Host: example.com
Accept: */*

HTTP/1.1 200
Date: Mon, 20 Nov 2017 10:00:00 UTC
Content-Type: text/html
Date: Tue, 21 Nov 2017 10:00:00 UTC
Expires: Sun, 26 Nov 2017 10:00:00 UTC

<!doctype html>
<html>
...

<!doctype html>
<html>
...
And an update listing the following headers:

Expires: Fri, 1 Dec 2017 10:00:00 UTC
Date: Sat, 25 Nov 2017 10:00:00 UTC

The resulting stored exchange would be:

GET / HTTP/1.1
Host: example.com
Accept: */*

HTTP/1.1 200
Content-Type: text/html
Expires: Fri, 1 Dec 2017 10:00:00 UTC
Date: Sat, 25 Nov 2017 10:00:00 UTC

<!doctype html>
<html>
...

C.2. Downsides of updating cache control

In an exchange with multiple signatures, using cache control to expire signatures forces all signatures to initially live for the same period. Worse, the update from one signature’s "validity-url" might not match the update for another signature. Clients would need to maintain a current set of headers for each signature, and then decide which set to use when actually parsing the resource itself.

This need to store and reconcile multiple sets of headers for a single signed exchange argues for embedding a signature’s lifetime into the signature.

Appendix D. Change Log

RFC EDITOR PLEASE DELETE THIS SECTION.

draft-04

- Update to draft-ietf-httpbis-header-structure-06.

- Replace the application/http-exchange+cbor format with a simpler application/signed-exchange format that:

  - Doesn’t require a streaming CBOR parser parse it from a network stream.
* Doesn’t allow request payloads or response trailers, which
don’t fit into the signature model.

* Allows checking the signature before parsing the exchange
headers.

- Require absolute URLs.
- Make all identifiers in headers lower-case, as required by
Structured Headers.
- Switch back to the TLS 1.3 signature format.
- Include the version and draft number in the signature context
string.
- Remove support for integrity protection using the Digest header
field.
- Limit the record size in the mi-sha256 encoding.
- Forbid RSA keys, and only require clients to support secp256r1
keys.
- Add a test OID for the CanSignHttpExchanges X.509 extension.

draft-03

- Allow each method of transferring an exchange to define which
headers are signed, have the cross-origin methods use all headers,
and remove the "allResponseHeaders" flag.
- Describe footguns around signing private content, and block
certain headers to make it less likely.
- Define a CBOR structure to hold the certificate chain instead of
re-using the TLS1.3 message. The TLS 1.3 parser fails on
unexpected extensions while this format should ignore them, and
apparently TLS implementations don’t expose their message parsers
enough to allow passing a message to a certificate verifier.
- Require an X.509 extension for the signing certificate.

draft-02

- Signatures identify a header (e.g. Digest or MI) to guard the
payload’s integrity instead of directly signing over the payload.
The validityUrl is signed.

Use CBOR maps where appropriate, and define how they’re canonicalized.

Remove the update.url field from signature validity updates, in favor of just re-fetching the original request URL.

Define an HTTP/2 extension to use a setting to enable cross-origin Server Push.

Define an "Accept-Signature" header to negotiate whether to send Signatures and which ones.

Define an "application/http-exchange+cbor" format to fetch signed exchanges without HTTP/2 Push.

2 new use cases.

Appendix E. Acknowledgements

Thanks to Ilari Liusvaara, Justin Schuh, Mark Nottingham, Mike Bishop, Ryan Sleevi, and Yoav Weiss for comments that improved this draft.

Author’s Address

Jeffrey Yasskin
Google

Email: jyasskin@chromium.org