HTTP Working Group
Internet-Draft
Intended status: Experimental
Expires: October 8, 2018

K. Oku
Fastly
Y. Weiss
Akamai

April 6, 2018

Cache Digests for HTTP/2
draft-ietf-httpbis-cache-digest-04

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 October 8, 2018.

Copyright Notice

Table of Contents

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.

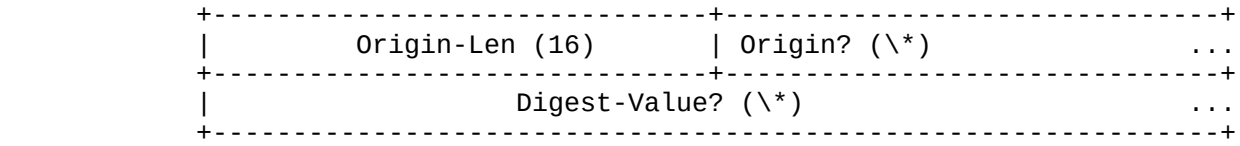
This specification defines a HTTP/2 frame type to allow clients to inform the server of their cache's contents using a Cuckoo-filter [Cuckoo] based digest. Servers can then use this to inform their choices of what to push to clients.

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).



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.

- *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, for the type of cached response indicated by the "STALE" flag.

- *VALIDATORS* (0x4): When set, indicates that the "validators" boolean in Section 2.1.5 is true.

- *STALE* (0x8): When set, indicates that all cached responses represented in the digest-value are stale [RFC7234] at the point in them that the digest was generated; otherwise, all are fresh.

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 cached 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.

CACHE_DIGEST allows the client to indicate whether the set of URLs used to compute the digest represent fresh or stale stored responses, using the STALE flag. Clients MAY decide whether to only send CACHE_DIGEST frames representing their fresh stored responses, their stale stored responses, or both.

Clients can choose to only send a subset of the suitable stored responses of each type (fresh or stale). However, when the CACHE_DIGEST frames sent represent the complete set of stored responses of a given type, the last such frame SHOULD have a COMPLETE flag set, to indicate to the server that it has all relevant state of that type. 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 can be computed to include cached responses' ETags, as indicated by the VALIDATORS flag. This information can be used by servers to decide what kinds of responses to push to clients; for example, a stale response that hasn't changed could be refreshed with a 304 (Not Modified) response; one that has changed can be replaced with a 200 (OK) response, whether the cached response was fresh or stale.

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]
- "ETag" a string corresponding to the entity-tag [RFC7232] of a cached response [RFC7234] (if the ETag is available; otherwise, null);
- "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" and "ETag" as inputs.
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:

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

- "ETag" a string corresponding to the entity-tag [RFC7232] of a cached response [RFC7234] (if the ETag is available; otherwise, null);

- "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" and "ETag" as inputs.

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:

- "key", an array of characters

- "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 infitisimaly 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:

- "URL", an array of characters
- "ETag", an array of characters
- "validators", a boolean indicating whether validators ([RFC7232]) are to be included in the digest

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

2. If "validators" is true and "ETag" is not null:

    1. Append "ETag" to "key" as an ASCII string, including both the "weak" indicator (if present) and double quotes, as per [RFC7232], Section 2.3.

3. Return "key"

TODO: Add an example of the ETag and the key calcuations.

2.1.6. Computing a Hash Value

Given the following inputs:

- "key", an array of characters.
- "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:

- "hash1", an integer indicating the previous hash.
- "fingerprint", an integer indicating the fingerprint value.
- "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 and ETag combination 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 and ETag 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].
- "ETag" a string corresponding to the entity-tag [RFC7232] of a cached response [RFC7234] (if the ETag is available; otherwise, null).
- "validators", a boolean
- "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" and "ETag" as inputs.
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 SENDING_CACHE_DIGEST SETTINGS Parameter

A Client SHOULD notify its support for CACHE_DIGEST frames by sending the 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 ACCEPT_CACHE_DIGEST SETTINGS Parameter

A server can notify its support for CACHE_DIGEST frame by sending the 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 it's 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 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]:

- Frame Type: CACHE_DIGEST
- Code: 0xd
- Specification: [this document]

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

- Code: 0x7
- Name: 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

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <https://www.rfc-editor.org/info/rfc6234>.

[RFC6454] Barth, A., "The Web Origin Concept", RFC 6454, DOI 10.17487/RFC6454, December 2011, <https://www.rfc-editor.org/info/rfc6454>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <https://www.rfc-editor.org/info/rfc7230>.

[RFC7232] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests", RFC 7232, DOI 10.17487/RFC7232, June 2014, <https://www.rfc-editor.org/info/rfc7232>.

[RFC7234] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching", RFC 7234, DOI 10.17487/RFC7234, June 2014, <https://www.rfc-editor.org/info/rfc7234>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <https://www.rfc-editor.org/info/rfc7540>.

7.2. Informative References

[Cuckoo] "Cuckoo Filter: Practically Better Than Bloom", n.d., <https://www.cs.cmu.edu/~dga/papers/cuckoo-conext2014.pdf>.

[Fetch] "Fetch Standard", n.d., <https://fetch.spec.whatwg.org/>.

[I-D.ietf-tls-tls13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-28 (work in progress), March 2018.

[RFC3864] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, DOI 10.17487/RFC3864, September 2004, <https://www.rfc-editor.org/info/rfc3864>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <https://www.rfc-editor.org/info/rfc6265>.

[Service-Workers] Russell, A., Song, J., Archibald, J., and M. Kruisselbrink, "Service Workers 1", W3C Working Draft WD-service-workers-1-20161011, October 2016, <https://www.w3.org/TR/2016/WD-service-workers-1-20161011/>.

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.

```
Cache-Digest  = 1#digest-entity
digest-entity = digest-value *(OWS ";" OWS digest-flag)
digest-value  = <Digest-Value encoded using base64url>
digest-flag   = 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 one digest of fresh 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 respresents 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. Acknowledgements

Thanks to Stefan Eissing for his suggestions.

Appendix C. Changes

C.1. Since draft-ietf-httpbis-cache-digest-03

- Yoav becomes an author; Mark steps down.

C.2. Since draft-ietf-httpbis-cache-digest-02

- Switch to Cuckoo Filter.

C.3. Since draft-ietf-httpbis-cache-digest-01

- Added definition of the Cache-Digest header.
- Introduce ACCEPT_CACHE_DIGEST SETTINGS parameter.
- Change intended status from Standard to Experimental.

C.4. Since draft-ietf-httpbis-cache-digest-00

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

Authors' Addresses

Kazuho Oku
Fastly

Email: kazuhooku@gmail.com

Yoav Weiss
Akamai

Email: yoav@yoav.ws
URI: https://blog.yoav.ws/

HTTP Working Group
Internet-Draft
Intended status: Experimental
Expires: May 1, 2018

K. Oku
Fastly

October 28, 2017

An HTTP Status Code for Indicating Hints
draft-ietf-httpbis-early-hints-05

Abstract

This memo introduces an informational HTTP status code that can be used to convey hints that help a client make preparations for processing the final response.

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 https://httpwg.github.io/ ; source code and issues list for this draft can be found at https://github.com/httpwg/http-extensions/labels/early-hints .

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 May 1, 2018.

Copyright Notice

Table of Contents

1. Introduction

It is common for HTTP responses to contain links to external resources that need to be fetched prior to their use; for example, rendering HTML by a Web browser. Having such links available to the client as early as possible helps to minimize perceived latency.

The "preload" ([Preload]) link relation can be used to convey such links in the Link header field of an HTTP response. However, it is not always possible for an origin server to generate the header block of a final response immediately after receiving a request. For example, the origin server might delegate a request to an upstream HTTP server running at a distant location, or the status code might depend on the result of a database query.

The dilemma here is that even though it is preferable for an origin server to send some header fields as soon as it receives a request, it cannot do so until the status code and the full header fields of the final HTTP response are determined.

HTTP/2 ([RFC7540]) server push can accelerate the delivery of resources, but only resources for which the server is authoritative. The other limitation of server push is that the response will be transmitted regardless of whether the client has the response cached. At the cost of spending one extra round-trip compared to server push in the worst case, delivering Link header fields in a timely fashion is more flexible and might consume less bandwidth.

This memo defines a status code for sending an informational response ([RFC7231], Section 6.2) that contains header fields that are likely to be included in the final response. A server can send the informational response containing some of the header fields to help the client start making preparations for processing the final response, and then run time-consuming operations to generate the final response. The informational response can also be used by an origin server to trigger HTTP/2 server push at a caching intermediary.

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. 103 Early Hints

The 103 (Early Hints) informational status code indicates to the client that the server is likely to send a final response with the header fields included in the informational response.

Typically, a server will include the header fields sent in a 103 (Early Hints) response in the final response as well. However, there might be cases when this is not desirable, such as when the server learns that they are not correct before the final response is sent.

A client can speculatively evaluate the header fields included in a 103 (Early Hints) response while waiting for the final response. For example, a client might recognize a Link header field value containing the relation type "preload" and start fetching the target resource. However, these header fields only provide hints to the client; they do not replace the header fields on the final response.

Aside from performance optimizations, such evaluation of the 103 (Early Hints) response’s header fields MUST NOT affect how the final response is processed. A client MUST NOT interpret the 103 (Early Hints) response header fields as if they applied to the informational response itself (e.g., as metadata about the 103 (Early Hints) response).

A server MAY use a 103 (Early Hints) response to indicate only some of the header fields that are expected to be found in the final response. A client SHOULD NOT interpret the nonexistence of a header field in a 103 (Early Hints) response as a speculation that the header field is unlikely to be part of the final response.

The following example illustrates a typical message exchange that involves a 103 (Early Hints) response.

Client request:

```
GET / HTTP/1.1
Host: example.com
```

Server response:

```
HTTP/1.1 103 Early Hints
Link: </style.css>; rel=preload; as=style
Link: </script.js>; rel=preload; as=script

HTTP/1.1 200 OK
Date: Fri, 26 May 2017 10:02:11 GMT
Content-Length: 1234
Content-Type: text/html; charset=utf-8
Link: </style.css>; rel=preload; as=style
Link: </script.js>; rel=preload; as=script

<!doctype html>
[... rest of the response body is omitted from the example ...]
```

As is the case with any informational response, a server might emit more than one 103 (Early Hints) response prior to sending a final response. This can happen for example when a caching intermediary generates a 103 (Early Hints) response based on the header fields of a stale-cached response, then forwards a 103 (Early Hints) response and a final response that were sent from the origin server in response to a revalidation request.

A server MAY emit multiple 103 (Early Hints) responses with additional header fields as new information becomes available while the request is being processed. It does not need to repeat the fields that were already emitted, though it doesn’t have to exclude them either. The client can consider any combination of header fields received in multiple 103 (Early Hints) responses when anticipating the list of header fields expected in the final response.

The following example illustrates a series of responses that a server might emit. In the example, the server uses two 103 (Early Hints) responses to notify the client that it is likely to send three Link header fields in the final response. Two of the three expected header fields are found in the final response. The other header field is replaced by another Link header field that contains a different value.

```
HTTP/1.1 103 Early Hints
Link: </main.css>; rel=preload; as=style

HTTP/1.1 103 Early Hints
Link: </style.css>; rel=preload; as=style
Link: </script.js>; rel=preload; as=script

HTTP/1.1 200 OK
Date: Fri, 26 May 2017 10:02:11 GMT
Content-Length: 1234
Content-Type: text/html; charset=utf-8
Link: </main.css>; rel=preload; as=style
Link: </newstyle.css>; rel=preload; as=style
Link: </script.js>; rel=preload; as=script

<!doctype html>
[... rest of the response body is omitted from the example ...]
```

3. Security Considerations

Some clients might have issues handling 103 (Early Hints), since informational responses are rarely used in reply to requests not including an Expect header field ([RFC7231], Section 5.1.1).

In particular, an HTTP/1.1 client that mishandles an informational response as a final response is likely to consider all responses to the succeeding requests sent over the same connection to be part of the final response. Such behavior might constitute a cross-origin information disclosure vulnerability in case the client multiplexes requests to different origins onto a single persistent connection.

Therefore, a server might refrain from sending Early Hints over HTTP/1.1 unless the client is known to handle informational responses correctly.

HTTP/2 clients are less likely to suffer from incorrect framing since handling of the response header fields does not affect how the end of the response body is determined.

4. IANA Considerations

The HTTP Status Codes Registry will be updated with the following entry:

- Code: 103
- Description: Early Hints
- Specification: [this document]

5. References

5.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <https://www.rfc-editor.org/info/rfc7230>.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <https://www.rfc-editor.org/info/rfc7231>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <https://www.rfc-editor.org/info/rfc7540>.

5.2. Informative References

[Preload] Grigorik, I., "Preload", n.d., <https://w3c.github.io/preload/>.

Appendix A. Changes

A.1. Since draft-ietf-httpbis-early-hints-04

- Clarified that the server is allowed to add headers not found in a 103 response to the final response.

- Clarify client's behavior when it receives more than one 103 response.

A.2. Since draft-ietf-httpbis-early-hints-03

- Removed statements that were either redundant or contradictory to RFC7230-7234.
- Clarified what the server's expected behavior is.
- Explain that a server might want to send more than one 103 response.
- Editorial Changes.

A.3. Since draft-ietf-httpbis-early-hints-02

- Editorial changes.
- Added an example.

A.4. Since draft-ietf-httpbis-early-hints-01

- Editorial changes.

A.5. Since draft-ietf-httpbis-early-hints-00

- Forbid processing the headers of a 103 response as part of the informational response.

Appendix B. Acknowledgements

Thanks to Tatsuhiro Tsujikawa for coming up with the idea of sending the Link header fields using an informational response.

Author's Address

Kazuho Oku
Fastly

Email: kazuhooku@gmail.com

HTTP
Internet-Draft
Intended status: Experimental
Expires: August 30, 2018

E. Stark
Google

February 26, 2018

Expect-CT Extension for HTTP
draft-ietf-httpbis-expect-ct-03

Abstract

This document defines a new HTTP header, 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. When configured in enforcement mode, user agents (UAs) will remember that hosts expect SCTs and will refuse connections that do not conform to the UA's Certificate Transparency policy. When configured in report-only mode, UAs will report the lack of valid SCTs to a URI configured by the host, but will allow the connection. By turning on Expect-CT, web host operators can 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 will expire on August 30, 2018.

Table of Contents

1. Introduction

This document defines a new HTTP header 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 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 See Certificate Transparency Qualified.

CT Policy See Certificate Transparency Policy.

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

Expect-CT Host See HTTP Expect-CT Host.

HTTP Expect-CT is the overall name for the combined UA- and server-side security policy defined by this specification.

HTTP Expect-CT Host is a conformant host implementing the HTTP server aspects of HTTP 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.

Known Expect-CT Host is an Expect-CT Host that the UA has noted as such. See Section 2.3.3 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" header field is a new response header defined in this specification. It is used by a server to indicate that UAs should evaluate connections to the host emitting the header 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 examples demonstrate valid Expect-CT response header fields:

```
Expect-CT: max-age=86400,enforce

Expect-CT: max-age=86400, enforce, 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

When replying to an HTTP request that was conveyed over a secure transport, an Expect-CT Host SHOULD include in its response exactly one Expect-CT header field. 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) at least one 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 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].

2.3.1. 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 was received for compliance with the UA's CT Policy, and then process the Expect-CT header field as follows.

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 2.3.3), 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.

If the connection does not comply with the UA's CT Policy (i.e. is not CT-qualified), then the UA MUST NOT note this host as a Known Expect-CT Host.

If the header field includes 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.

The UA MUST ignore any Expect-CT header field not conforming to the grammar specified in Section 2.1.

2.3.2. HTTP-Equiv <meta> Element Attribute

UAs MUST NOT heed "http-equiv="Expect-CT"" attribute settings on "<meta>" elements [W3C.REC-html51-20161101] in received content.

2.3.3. 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. The domain name and associated Expect-CT directives are collectively known as "Expect-CT metadata".

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, as specified in Section 2.3.4.

For forward compatibility, the UA MUST ignore any unrecognized Expect-CT header 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.

2.3.4. Storage Model

Known Expect-CT Hosts are identified only by domain names, and never IP addresses. If the substring matching the host production from the Request-URI (of the message to which the host responded) syntactically matches the IP-literal or IPv4address productions from

Section 3.2.2 of [RFC3986], then the UA MUST NOT note this host as a Known Expect-CT Host.

Otherwise, if the substring 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 value of the "report-uri" directive, if present.

If any other metadata from optional or future Expect-CT header directives are present in the Expect-CT header, 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, 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.)

2.4. Evaluating Expect-CT Connections for CT Compliance

When a UA connects to a Known Expect-CT Host using a TLS connection, if the TLS connection has errors, the UA MUST terminate the connection without allowing the user to proceed anyway. (This behavior is the same as that required by [RFC6797].)

If the 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. It is acceptable to skip this CT compliance check 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).

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.

If a connection to a Known 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 a non-recoverable error.

If a connection to a Known 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).

A UA that has previously noted a host as a Known Expect-CT Host MUST evaluate CT compliance when setting up the TLS session, before beginning an HTTP conversation over the TLS channel.

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 object with the following keys and values:

- "date-time": the value for this key indicates the time 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 2.3.4) for the Expect-CT Host that failed the CT compliance check. 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 of [RFC6962] and "2" if it is in the format defined in Section 4.6 of [I-D.ietf-trans-rfc6962-bis].

  * 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 or [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 4.6 of [I-D.ietf-trans-rfc6962-bis].

- "failure-mode": the value indicates whether the Expect-CT report was triggered by an Expect-CT policy in enforce or report-only mode. The value is provided as a string. The UA MUST set this value to "enforce" if the Expect-CT metadata indicates an "enforce" configuration, and "report-only" otherwise.

- "test-report": the value is set to true if the report is being sent by a testing client to verify that the reporting server behaves correctly. The value is provided as a boolean, and MUST be set to true if the report serves to test the server's behavior and can be discarded.

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, 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 in enforcing mode with a long "max-age". Implementing user agents would note this as an Expect-CT Host (see Section 2.3.3). 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 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

TBD

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

8.1. HTTP Header

Expect-CT could be specified as a TLS extension or X.509 certificate extension instead of an HTTP response header. Using an HTTP header 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 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. Changes

9.1. 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.

9.2. Since -01

- Change SCT reporting format to support both RFC 6962 and 6962-bis SCTs.

9.3. 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

10. References

10.1. Normative References

[FETCH] van Kesteren, A., "Fetch", n.d., <https://fetch.spec.whatwg.org/>.

[HTML] Hickson, I., Pieters, S., van Kesteren, A., Jaegenstedt, P., and D. Denicola, "HTML", n.d., <https://html.spec.whatwg.org/>.

[I-D.ietf-trans-rfc6962-bis] Laurie, B., Langley, A., Kasper, E., Messeri, E., and R. Stradling, "Certificate Transparency Version 2.0", draft-ietf-trans-rfc6962-bis-27 (work in progress), October 2017.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3339] Klyne, G. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, DOI 10.17487/RFC3339, July 2002, <https://www.rfc-editor.org/info/rfc3339>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

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

[RFC6797] Hodges, J., Jackson, C., and A. Barth, "HTTP Strict Transport Security (HSTS)", RFC 6797, DOI 10.17487/RFC6797, November 2012, <https://www.rfc-editor.org/info/rfc6797>.

[RFC6962] Laurie, B., Langley, A., and E. Kasper, "Certificate Transparency", RFC 6962, DOI 10.17487/RFC6962, June 2013, <https://www.rfc-editor.org/info/rfc6962>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <https://www.rfc-editor.org/info/rfc7230>.

[RFC7234] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching", RFC 7234, DOI 10.17487/RFC7234, June 2014, <https://www.rfc-editor.org/info/rfc7234>.

[RFC7468] Josefsson, S. and S. Leonard, "Textual Encodings of PKIX, PKCS, and CMS Structures", RFC 7468, DOI 10.17487/RFC7468, April 2015, <https://www.rfc-editor.org/info/rfc7468>.

[W3C.REC-html51-20161101] Faulkner, S., Eicholz, A., Leithead, T., and A. Danilo, "HTML 5.1", World Wide Web Consortium Recommendation REC-html51-20161101, November 2016, <https://www.w3.org/TR/2016/REC-html51-20161101>.

10.2. URIs

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

[2] http://httpwg.github.io/

[3] https://github.com/httpwg/http-extensions/labels/expect-ct

Author's Address

Emily Stark
Google

Email: estark@google.com

HTTP
Internet-Draft
Intended status: Standards Track
Expires: September 5, 2018

M. Nottingham
Fastly
P-H. Kamp
The Varnish Cache Project

March 4, 2018

Structured Headers for HTTP
draft-ietf-httpbis-header-structure-04

Abstract

This document describes a set of data types and parsing 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

RFC 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].

Table of Contents

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 for it often need to be written, because each header has slightly different handling of what looks like common syntax.

This document introduces structured HTTP header field values (hereafter, Structured Headers) to address these problems. Structured Headers define a generic, abstract model for header field values, along with a concrete serialisation for expressing that model in textual HTTP headers, as used by HTTP/1 [RFC7230] and HTTP/2 [RFC7540].

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 and parsing.

Additionally, future versions of HTTP can define alternative serialisations of the abstract model of Structured Headers, 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 uses Structured Headers, see Section 2.

Section 4 defines a number of abstract data types that can be used in Structured Headers. Dictionaries and lists are only usable at the "top" level, while the remaining types can be specified appear at the top level or inside those structures.

Those abstract types can be serialised into textual headers - such as those used in HTTP/1 and HTTP/2 - using the algorithms described in Section 3.

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 DIGIT, ALPHA and DQUOTE rules from that document. It also includes the OWS rule from [RFC7230].

2. Specifying Structured Headers

A HTTP header that uses Structured Headers need to be defined to do so explicitly; recipients and generators need to know that the requirements of this document are in effect. The simplest way to do that is by referencing this document in its definition.

The field’s definition will also need to specify the field-value’s allowed syntax, in terms of the types described in Section 4, along with their associated semantics.

A header field definition cannot relax or otherwise modify the requirements of this specification; doing so would preclude handling by generic software.

However, header field authors are encouraged to clearly state additional constraints upon the syntax, as well as the consequences when those constraints are violated. Such additional constraints could include additional structure (e.g., a list of URLs [RFC3986] inside a string) that cannot be expressed using the primitives defined here.

For example:

```
# FooExample Header

The FooExample HTTP header field conveys a list of integers about how
much Foo the sender has.

FooExample is a Structured header [RFCxxxx]. Its value MUST be a
dictionary ([RFCxxxx], Section Y.Y).

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 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 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.

   Note that empty header field values are not allowed by the syntax,
   and therefore parsing for them will fail.
```

3. Parsing Text 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, return the parsed header value. 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.

1. Discard any leading OWS from input_string.

2. If the field-value is defined to be a dictionary, let output be the result of Parsing a Dictionary from Text (Section 4.1.1).

3. If the field-value is defined to be a list, let output be the result of Parsing a List from Text (Section 4.2.1).

4. If the field-value is defined to be a parameterised list, let output be the result of Parsing a Parameterised List from Text (Section 4.3.1).

5. Otherwise, let output be the result of Parsing an Item from Text (Section 4.4.1).

6. Discard any leading OWS from input_string.

7. If input_string is not empty, fail parsing.

8. Otherwise, return output.

Note that in the case of lists, parameterised lists and dictionaries, this has the effect of coalescing all of the values for that field. However, for singular items, parsing will fail if more than instance of that header field is present.

If parsing fails, the entire header field’s value MUST be discarded. This is intentionally strict, to improve interoperability and safety, and specifications referencing this document MUST NOT loosen this requirement.

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

4. Structured Header Data Types

This section defines the abstract value types that can be composed into Structured Headers, along with the textual HTTP serialisations of them.

4.1. Dictionaries

Dictionaries are unordered maps of key-value pairs, where the keys are identifiers (Section 4.8) and the values are items (Section 4.4). There can be between 1 and 1024 members, and keys are required to be unique.

In the textual HTTP serialisation, keys and values are separated by "=" (without whitespace), and key/value pairs are separated by a comma with optional whitespace. Duplicate keys MUST cause parsing to fail.

```
dictionary        = dictionary_member *1023( OWS "," OWS dictionary_member )
dictionary_member = identifier "=" item
```

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

ExampleDictHeader: foo=1.23, en="Applepie", da=*w4ZibGV0w6ZydGUK

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

4.1.1. 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 4.8.1) with input_string.
    2. If dictionary already contains this_key, fail parsing.
    3. Consume a "=" from input_string; if none is present, fail parsing.
    4. Let this_value be the result of running Parse Item from Text (Section 4.4.1) with input_string.
    5. Add key this_key with value this_value to dictionary.
    6. If dictionary has more than 1024 members, fail parsing.
    7. Discard any leading OWS from input_string.
    8. If input_string is empty, return dictionary.
    9. Consume a COMMA from input_string; if no comma is present, fail parsing.

10. Discard any leading OWS from input_string.

11. If input_string is empty, fail parsing.

3. If dictionary is empty, fail parsing.

4. Return dictionary.

4.2. Lists

Lists are arrays of items (Section 4.4) with one to 1024 members.

In the textual HTTP serialisation, each member is separated by a comma and optional whitespace.

```
list = list_member 0*1023( OWS "," OWS list_member )
list_member = item
```

For example, a header field whose value is defined as a list of identifiers could look like:

```
ExampleIdListHeader: foo, bar, baz_45
```

4.2.1. 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 4.4.1) with input_string.

    2. Append item to items.

    3. If items has more than 1024 members, fail parsing.

    4. Discard any leading OWS from input_string.

    5. If input_string is empty, return items.

    6. Consume a COMMA from input_string; if no comma is present, fail parsing.

    7. Discard any leading OWS from input_string.

8. If input_string is empty, fail parsing.

3. If items is empty, fail parsing.

4. Return items.

4.3. Parameterised Lists

Parameterised Lists are arrays of a parameterised identifiers with 1 to 256 members.

A parameterised identifier is an identifier (Section 4.8) with up to 256 parameters, each parameter having a identifier and an optional value that is an item (Section 4.4). Ordering between parameters is not significant, and duplicate parameters MUST cause parsing to fail.

In the textual HTTP serialisation, each parameterised identifier is separated by a comma and optional whitespace. Parameters are delimited from each other using semicolons (";"), and equals ("=") delimits the parameter name from its value.

```
param_list = param_id 0*255( OWS "," OWS param_id )
param_id   = identifier 0*256( OWS ";" OWS identifier [ "=" item ] )
```

For example,

```
ExampleParamListHeader: abc_123;a=1;b=2; c, def_456, ghi;q="19";r=foo
```

4.3.1. 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 4.3.2) with input_string.

    2. Append item to items.

    3. If items has more than 256 members, fail parsing.

    4. Discard any leading OWS from input_string.

    5. If input_string is empty, return items.

6. Consume a COMMA from input_string; if no comma is present, fail parsing.

7. Discard any leading OWS from input_string.

8. If input_string is empty, fail parsing.

3. If items is empty, fail parsing.

4. Return items.

4.3.2. Parsing a Parameterised Identifier from Text

Given an ASCII string input_string, return a identifier with an 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 4.8.1) 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 4.8.1) 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 4.4.1) from input_string.

9. If parameters has more than 255 members, fail parsing.

10. Add param_name to parameters with the value param_value.

4. Return the tuple (primary_identifier, parameters).

4.4. Items

An item is can be a integer (Section 4.5), float (Section 4.6), string (Section 4.7), identifier (Section 4.8) or binary content (Section 4.9).

```
item = integer / float / string / identifier / binary
```

4.4.1. 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 4.5.1) and return the result.

3. If the first character of input_string is a DQUOTE, process input_string as a string (Section 4.7.1) and return the result.

4. If the first character of input_string is "*", process input_string as binary content (Section 4.9.1) and return the result.

5. If the first character of input_string is an lcalpha, process input_string as a identifier (Section 4.8.1) and return the result.

6. Otherwise, fail parsing.

4.5. Integers

Abstractly, 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).

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

Parsers that encounter an integer outside the range defined above MUST fail parsing. Therefore, the value "9223372036854775808" would

be invalid. Likewise, values that do not conform to the ABNF above are invalid, and MUST fail parsing.

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

ExampleIntegerHeader: 42

4.5.1. Parsing a Number from Text

NOTE: This algorithm parses both Integers and Floats Section 4.6, and returns the corresponding structure.

1. If the first character of input_string is not "-" or a DIGIT, fail parsing.

2. Let input_number be the result of consuming input_string up to (but not including) the first character that is not in DIGIT, "-", and ".".

3. If input_number contains ".", parse it as a floating point number and let output_number be the result.

4. Otherwise, parse input_number as an integer and let output_number be the result.

5. Return output_number.

4.6. Floats

Abstractly, floats are integers with a fractional part. They have a maximum of fifteen digits available to be used in both of the parts, as reflected in the ABNF below; this allows them to be stored as IEEE 754 double precision numbers (binary64) ([IEEE754]).

The textual HTTP serialisation of floats allows a maximum of fifteen digits between the integer and fractional part, with at least one required on each side, along with an optional "-" indicating negative numbers.

```
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 )
```

Values that do not conform to the ABNF above are invalid, and MUST fail parsing.

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

```
ExampleFloatHeader: 4.5
```

See Section 4.5.1 for the parsing algorithm for floats.

4.7. Strings

Abstractly, strings are up to 1024 printable ASCII [RFC0020] characters (i.e., the range 0x20 to 0x7E). Note that this excludes tabs, newlines and carriage returns.

The textual HTTP serialisation of strings uses a backslash ("\") to escape double quotes and backslashes in strings.

```
string    = DQUOTE 0*1024(char) DQUOTE
char      = unescaped / escape ( DQUOTE / "\" )
unescaped = %x20-21 / %x23-5B / %x5D-7E
escape    = "\"
```

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

```
ExampleStringHeader: "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 Structured Headers, 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 4.9) SHOULD be specified, along with a character encoding (preferably, UTF-8).

4.7.1. 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, append char to output_string.
    5. If output_string contains more than 1024 characters, fail parsing.
5. Otherwise, fail parsing.

4.8. Identifiers

Identifiers are short (up to 256 characters) textual identifiers; their abstract model is identical to their expression in the textual HTTP serialisation.

```
identifier = lcalpha *255( lcalpha / DIGIT / "_" / "-"/ "*" / "/" )
lcalpha    = %x61-7A ; a-z
```

Note that identifiers can only contain lowercase letters.

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

```
ExampleIdHeader: foo/bar
```

4.8.1. Parsing a 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. If output_string contains more than 256 characters, fail parsing.
4. Return output_string.

4.9. Binary Content

Arbitrary binary content up to 16384 bytes in size can be conveyed in Structured Headers.

The textual HTTP serialisation encodes the data using Base 64 Encoding [RFC4648], Section 4, and surrounds it with a pair of asterisks ("*") to delimit from other content.

The encoded data is required to be padded with "=", 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 with some base64 implementations.

Likewise, encoded data is required to have pad bits set to zero, as per [RFC4648], Section 3.5. It is RECOMMENDED that parsers fail on encoded data that has non-zero pad bits, although this might not be possible with some base64 implementations.

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.

```
binary = "*" 0*21846(base64) "*"
base64 = ALPHA / DIGIT / "+" / "/" / "="
```

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

```
ExampleBinaryHeader: *cHJldGVuZCB0aGlzIGlzIGJpbmFyeSBjb250ZW50Lg*
```

4.9.1. 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 is has more than 21846 characters, 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 in Section 4.9).

7. Return binary_content.

5. IANA Considerations

This draft has no actions for IANA.

6. Security Considerations

TBD

7. References

7.1. Normative References

[RFC0020] Cerf, V., "ASCII format for network interchange", STD 80, RFC 20, DOI 10.17487/RFC0020, October 1969, <https://www.rfc-editor.org/info/rfc20>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <https://www.rfc-editor.org/info/rfc7230>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

7.2. Informative References

[IEEE754] IEEE, "IEEE Standard for Floating-Point Arithmetic", 2008, <http://grouper.ieee.org/groups/754/>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <https://www.rfc-editor.org/info/rfc7231>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <https://www.rfc-editor.org/info/rfc7540>.

7.3. URIs

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

[2] https://httpwg.github.io/

[3] https://github.com/httpwg/http-extensions/labels/header-structure

Appendix A. Changes

A.1. Since draft-ietf-httpbis-header-structure-03

- Strengthen language around failure handling.

A.2. 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.

A.3. Since draft-ietf-httpbis-header-structure-01

- Replaced with draft-nottingham-structured-headers.

A.4. 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

HTTP Working Group
Internet-Draft
Intended status: Standards Track
Expires: January 4, 2018

P. McManus
Mozilla

July 3, 2017

HTTP Immutable Responses
draft-ietf-httpbis-immutable-03

Abstract

The immutable HTTP response Cache-Control extension allows servers to identify resources that will not be updated during their freshness lifetime. This assures that a client never needs to revalidate a cached fresh resource to be certain it has not been modified.

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/immutable .

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 4, 2018.

Copyright Notice

1. Introduction

HTTP's freshness lifetime mechanism [RFC7234] allows a client to safely reuse a stored response to satisfy future requests for a specified period of time. However, it is still possible that the resource will be modified during that period.

For instance, a front page newspaper photo with a freshness lifetime of one hour would mean that no user would see a cached photo more than one hour old. However, the photo could be updated at any time resulting in different users seeing different photos depending on the contents of their caches for up to one hour. This is compliant with the caching mechanism defined in [RFC7234].

Users that need to confirm there have been no updates to their cached responses typically use the reload (or refresh) mechanism in their user agents. This in turn generates a conditional request [RFC7232] and either a new representation or, if unmodified, a 304 (Not Modified) response [RFC7232] is returned. A user agent that understands HTML and fetches its dependent sub-resources might issue hundreds of conditional requests to refresh all portions of a common page [REQPERPAGE].

However some content providers never create more than one variant of a sub-resource, because they use "versioned" URLs. When these resources need an update they are simply published under a new URL, typically embedding an identifier unique to that version of the resource in the path, and references to the sub-resource are updated with the new path information.

For example, "https://www.example.com/101016/main.css" might be updated and republished as "https://www.example.com/102026/main.css", with any links that reference it being changed at the same time. This design pattern allows a very large freshness lifetime to be used for the sub-resource without guessing when it will be updated in the future.

Unfortunately, the user agent does not know when this versioned URL design pattern is used. As a result, user-driven refreshes still translate into wasted conditional requests for each sub-resource as each will return 304 responses.

The "immutable" HTTP response Cache-Control extension allows servers to identify responses that will not be updated during their freshness lifetimes.

This effectively informs clients that any conditional request for that response can be safely skipped without worrying that it has been updated.

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 immutable Cache-Control extension

When present in an HTTP response, the "immutable" Cache-Control extension indicates that the origin server will not update the representation of that resource during the freshness lifetime of the response.

Clients SHOULD NOT issue a conditional request during the response’s freshness lifetime (e.g. upon a reload) unless explicitly overridden by the user (e.g. a force reload).

The immutable extension only applies during the freshness lifetime of the stored response. Stale responses SHOULD be revalidated as they normally would be in the absence of immutable.

The immutable extension takes no arguments. If any arguments are present, they have no meaning, and MUST be ignored. Multiple instances of the immutable extension are equivalent to one instance. The presence of an immutable Cache-Control extension in a request has no effect.

2.1. About Intermediaries

An immutable response has the same semantic meaning when received by proxy clients as it does when received by User-Agent based clients. Therefore proxies SHOULD skip conditionally revalidating fresh responses containing the immutable extension unless there is a signal from the client that a validation is necessary (e.g. a no-cache

Cache-Control request directive defined by Section 5.2.1.4 of [RFC7234]).

A proxy that uses immutable to bypass a conditional revalidation can choose whether to reply with a 304 or 200 to its requesting client based on the request headers the proxy received.

2.2. Example

Cache-Control: max-age=31536000, immutable

3. Security Considerations

The immutable mechanism acts as form of soft pinning and, as with all pinning mechanisms, creates a vector for amplification of cache corruption incidents. These incidents include cache poisoning attacks. Three mechanisms are suggested for mitigation of this risk:

- Clients SHOULD ignore immutable from resources that are not part of an authenticated context such as HTTPS. Authenticated resources are less vulnerable to cache poisoning.

- User-Agents often provide two different refresh mechanisms: reload and some form of force-reload. The latter is used to rectify interrupted loads and other corruption. These reloads, typically indicated through no-cache request attributes, SHOULD ignore immutable as well.

- Clients SHOULD ignore immutable for resources that do not provide a strong indication that the stored response size is the correct response size such as responses delimited by connection close.

4. IANA Considerations

Section 7.1 of [RFC7234] requires registration of the immutable extension in the "Hypertext Transfer Protocol (HTTP) Cache Directive Registry" with IETF Review.

- Cache-Directive: immutable

- Pointer to specification text: [this document]

5. Acknowledgments

Thank you to Ben Maurer for partnership in developing and testing this idea. Thank you to Amos Jeffries for help with proxy interactions and to Mark Nottingham for help with the documentation.

6. References

6.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC7232] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests", RFC 7232, DOI 10.17487/RFC7232, June 2014, <http://www.rfc-editor.org/info/rfc7232>.

[RFC7234] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching", RFC 7234, DOI 10.17487/RFC7234, June 2014, <http://www.rfc-editor.org/info/rfc7234>.

6.2. Informative References

[REQPERPAGE] "HTTP Archive", n.d., <http://httparchive.org/interesting.php#reqTotal>.

Author’s Address

Patrick McManus
Mozilla

Email: pmcmanus@mozilla.com

HTTP
Internet-Draft
Intended status: Standards Track
Expires: July 17, 2018

M. Nottingham

E. Nygren
Akamai

January 13, 2018

The ORIGIN HTTP/2 Frame
draft-ietf-httpbis-origin-frame-06

Abstract

This document specifies the ORIGIN frame for HTTP/2, to indicate what origins are available on a given connection.

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/origin-frame [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 July 17, 2018.

Table of Contents

1. Introduction

HTTP/2 [RFC7540] allows clients to coalesce different origins [RFC6454] onto the same connection when certain conditions are met. However, in certain cases, a connection is not usable for a coalesced origin, so the 421 (Misdirected Request) status code ([RFC7540], Section 9.1.2) was defined.

Using a status code in this manner allows clients to recover from misdirected requests, but at the penalty of adding latency. To address that, this specification defines a new HTTP/2 frame type, "ORIGIN", to allow servers to indicate what origins a connection is usable for.

Additionally, experience has shown that HTTP/2's requirement to establish server authority using both DNS and the server's certificate is onerous. This specification relaxes the requirement to check DNS when the ORIGIN frame is in use. Doing so has

additional benefits, such as removing the latency associated with some DNS lookups.

1.1. Notational Conventions

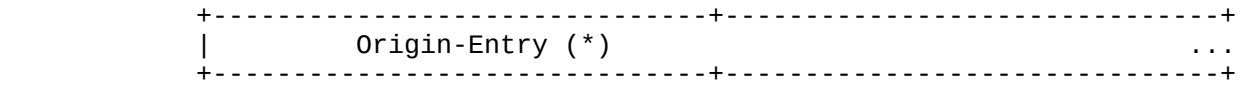
The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. The ORIGIN HTTP/2 Frame

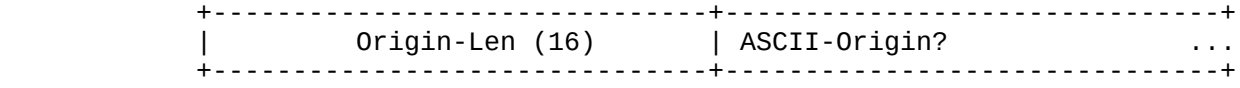
This document defines a new HTTP/2 frame type ([RFC7540], Section 4) called ORIGIN, that allows a server to indicate what origin(s) [RFC6454] the server would like the client to consider as members of the Origin Set (Section 2.3) for the connection it occurs within.

2.1. Syntax

The ORIGIN frame type is 0xc (decimal 12), and contains zero or more instances of the Origin-Entry field.



An Origin-Entry is a length-delimited string:



Specifically:

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

Origin: An OPTIONAL sequence of characters containing the ASCII serialization of an origin ([RFC6454], Section 6.2) that the sender asserts this connection is or could be authoritative for.

The ORIGIN frame does not define any flags. However, future updates to this specification MAY define flags. See Section 2.2.

2.2. Processing ORIGIN Frames

The ORIGIN frame is a non-critical extension to HTTP/2. Endpoints that do not support this frame can safely ignore it upon receipt.

When received by an implementing client, it is used to initialise and manipulate the Origin Set (see Section 2.3), thereby changing how the client establishes authority for origin servers (see Section 2.4).

The ORIGIN frame MUST be sent on stream 0; an ORIGIN frame on any other stream is invalid and MUST be ignored.

Likewise, the ORIGIN frame is only valid on connections with the "h2" protocol identifier, or when specifically nominated by the protocol's definition; it MUST be ignored when received on a connection with the "h2c" protocol identifier.

This specification does not define any flags for the ORIGIN frame, but future updates to this specification (through IETF consensus) might use them to change its semantics. The first four flags (0x1, 0x2, 0x4 and 0x8) are reserved for backwards-incompatible changes, and therefore when any of them are set, the ORIGIN frame containing them MUST be ignored by clients conforming to this specification, unless the flag's semantics are understood. The remaining flags are reserved for backwards-compatible changes, and do not affect processing by clients conformant to this specification.

The ORIGIN frame describes a property of the connection, and therefore is processed hop-by-hop. An intermediary MUST NOT forward ORIGIN frames. Clients configured to use a proxy MUST ignore any ORIGIN frames received from it.

Each ASCII-Origin field in the frame's payload MUST be parsed as an ASCII serialisation of an origin ([RFC6454], Section 6.2). If parsing fails, the field MUST be ignored.

Note that the ORIGIN frame does not support wildcard names (e.g., "*.example.com") in Origin-Entry. As a result, sending ORIGIN when a wildcard certificate is in use effectively disables any origins that are not explicitly listed in the ORIGIN frame(s) (when the client understands ORIGIN).

See Appendix A for an illustrative algorithm for processing ORIGIN frames.

2.3. The Origin Set

The set of origins (as per [RFC6454]) that a given connection might be used for is known in this specification as the Origin Set.

By default, the Origin Set for a connection is uninitialised. An uninitialized Origin Set means that clients apply the coalescing rules from Section 9.1.1 of [RFC7540].

When an ORIGIN frame is first received and successfully processed by a client, the connection's Origin Set is defined to contain an initial origin. The initial origin is composed from:

- Scheme: "https"
- Host: the value sent in Server Name Indication (SNI, [RFC6066], Section 3), converted to lower case; if SNI is not present, the remote address of the connection (i.e., the server's IP address)
- Port: the remote port of the connection (i.e., the server's port)

The contents of that ORIGIN frame (and subsequent ones) allows the server to incrementally add new origins to the Origin Set, as described in Section 2.2.

The Origin Set is also affected by the 421 (Misdirected Request) response status code, defined in [RFC7540], Section 9.1.2. Upon receipt of a response with this status code, implementing clients MUST create the ASCII serialisation of the corresponding request's origin (as per [RFC6454], Section 6.2) and remove it from the connection's Origin Set, if present.

Note: When sending an ORIGIN frame to a connection that is initialised as an Alternative Service [RFC7838], the initial origin set (Section 2.3) will contain an origin with the appropriate scheme and hostname (since Alternative Services specifies that the origin's hostname be sent in SNI). However, it is possible that the port will be different than that of the intended origin, since the initial origin set is calculated using the actual port in use, which can be different for the alternative service. In this case, the intended origin needs to be sent in the ORIGIN frame explicitly.

For example, a client making requests for "https://example.com" is directed to an alternative service at ("h2", "x.example.net", "8443"). If this alternative service sends an ORIGIN frame, the initial origin will be "https://example.com:8443". The client will not be able to use the alternative service to make requests

for "https://example.com" unless that origin is explicitly included in the ORIGIN frame.

2.4. Authority, Push and Coalescing with ORIGIN

Section 10.1 of [RFC7540] uses both DNS and the presented TLS certificate to establish the origin server(s) that a connection is authoritative for, just as HTTP/1.1 does in [RFC7230].

Furthermore, Section 9.1.1 of [RFC7540] explicitly allows a connection to be used for more than one origin server, if it is authoritative. This affects what responses can be considered authoritative, both for direct responses to requests and for server push (see [RFC7540], Section 8.2.2). Indirectly, it also affects what requests will be sent on a connection, since clients will generally only send requests on connections that they believe to be authoritative for the origin in question.

Once an Origin Set has been initialised for a connection, clients that implement this specification use it to help determine what the connection is authoritative for. Specifically, such clients MUST NOT consider a connection to be authoritative for an origin not present in the Origin Set, and SHOULD use the connection for all requests to origins in the Origin Set for which the connection is authoritative, unless there are operational reasons for opening a new connection.

Note that for a connection to be considered authoritative for a given origin, the server is still required to authenticate with certificate that passes suitable checks; see Section 9.1.1 of [RFC7540] for more information. This includes verifying that the host matches a "dNSName" value from the certificate "subjectAltName" field (using the rules defined in [RFC2818]; see also [RFC5280], Section 4.2.1.6).

Additionally, clients MAY avoid consulting DNS to establish the connection’s authority for new requests to origins in the Origin Set; however, those that do so face new risks, as explained in Section 4.

Because ORIGIN can change the set of origins a connection is used for over time, it is possible that a client might have more than one viable connection to an origin open at any time. When this occurs, clients SHOULD NOT emit new requests on any connection whose Origin Set is a proper subset of another connection’s Origin Set, and SHOULD close it once all outstanding requests are satisfied.

The Origin Set is unaffected by any alternative services [RFC7838] advertisements made by the server. Advertising an alternative service does not affect whether a server is authoritative.

3. IANA Considerations

This specification adds an entry to the "HTTP/2 Frame Type" registry.

- Frame Type: ORIGIN
- Code: 0xc
- Specification: [this document]

4. Security Considerations

Clients that blindly trust the ORIGIN frame's contents will be vulnerable to a large number of attacks. See Section 2.4 for mitigations.

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 coalesce connections to the target onto their existing connection.

As a result, clients opting not to consult DNS ought to employ some alternative means to establish a high degree of confidence that the certificate is legitimate. For example, clients might skip consulting DNS only if they receive proof of inclusion in a Certificate Transparency log [RFC6962] or they have a recent OCSP response [RFC6960] (possibly using the "status_request" TLS extension [RFC6066]) showing that the certificate was not revoked.

The Origin Set's size is unbounded by this specification, and thus could be used by attackers to exhaust client resources. To mitigate this risk, clients can monitor their state commitment and close the connection if it is too high.

5. References

5.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, DOI 10.17487/RFC2818, May 2000, <https://www.rfc-editor.org/info/rfc2818>.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <https://www.rfc-editor.org/info/rfc5280>.

[RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <https://www.rfc-editor.org/info/rfc6066>.

[RFC6454] Barth, A., "The Web Origin Concept", RFC 6454, DOI 10.17487/RFC6454, December 2011, <https://www.rfc-editor.org/info/rfc6454>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <https://www.rfc-editor.org/info/rfc7540>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

5.2. Informative References

[RFC6960] Santesson, S., Myers, M., Ankney, R., Malpani, A., Galperin, S., and C. Adams, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP", RFC 6960, DOI 10.17487/RFC6960, June 2013, <https://www.rfc-editor.org/info/rfc6960>.

[RFC6962] Laurie, B., Langley, A., and E. Kasper, "Certificate Transparency", RFC 6962, DOI 10.17487/RFC6962, June 2013, <https://www.rfc-editor.org/info/rfc6962>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <https://www.rfc-editor.org/info/rfc7230>.

[RFC7838] Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", RFC 7838, DOI 10.17487/RFC7838, April 2016, <https://www.rfc-editor.org/info/rfc7838>.

[RFC8288] Nottingham, M., "Web Linking", RFC 8288, DOI 10.17487/RFC8288, October 2017, <https://www.rfc-editor.org/info/rfc8288>.

5.3. URIs

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

[2] http://httpwg.github.io/

[3] https://github.com/httpwg/http-extensions/labels/origin-frame

Appendix A. Non-Normative Processing Algorithm

The following algorithm illustrates how a client could handle received ORIGIN frames:

1. If the client is configured to use a proxy for the connection, ignore the frame and stop processing.

2. If the connection is not identified with the "h2" protocol identifier or another protocol that has explicitly opted into this specification, ignore the frame and stop processing.

3. If the frame occurs upon any stream except stream 0, ignore the frame and stop processing.

4. If any of the flags 0x1, 0x2, 0x4 or 0x8 are set, ignore the frame and stop processing.

5. If no previous ORIGIN frame on the connection has reached this step, initialise the Origin Set as per Section 2.3.

6. For each "Origin-Entry" in the frame payload:

    1. Parse "ASCII-Origin" as an ASCII serialization of an origin ([RFC6454], Section 6.2) and let the result be "parsed_origin". If parsing fails, skip to the next "Origin-Entry".

    2. Add "parsed_origin" to the Origin Set.

Appendix B. Operational Considerations for Servers

The ORIGIN frame allows a server to indicate for which origins a given connection ought be used. The set of origins advertised using this mechanism is under control of the server; servers are not obligated to use it, or to advertise all origins which they might be able to answer a request for.

For example, it can be used to inform the client that the connection is to only be used for the SNI-based origin, by sending an empty

ORIGIN frame. Or, a larger number of origins can be indicated by including a payload.

Generally, this information is most useful to send before sending any part of a response that might initiate a new connection; for example, "Link" header fields [RFC8288] in a response HEADERS, or links in the response body.

Therefore, the ORIGIN frame ought be sent as soon as possible on a connection, ideally before any HEADERS or PUSH_PROMISE frames.

However, if it’s desirable to associate a large number of origins with a connection, doing so might introduce end-user perceived latency, due to their size. As a result, it might be necessary to select a "core" set of origins to send initially, expanding the set of origins the connection is used for with subsequent ORIGIN frames later (e.g., when the connection is idle).

That said, senders are encouraged to include as many origins as practical within a single ORIGIN frame; clients need to make decisions about creating connections on the fly, and if the origin set is split across many frames, their behaviour might be suboptimal.

Senders take note that, as per Section 4, Step 5 of [RFC6454], the values in an ORIGIN header need to be case-normalised before serialisation.

Finally, servers that host alternative services [RFC7838] will need to explicitly advertise their origins when sending ORIGIN, because the default contents of the Origin Set (as per Section 2.3) do not contain any Alternative Services’ origins, even if they have been used previously on the connection.

Authors’ Addresses

Mark Nottingham

Email: mnot@mnot.net
URI: https://www.mnot.net/

Erik Nygren
Akamai

Email: nygren@akamai.com

HTTP Working Group
Internet-Draft
Intended status: Experimental
Expires: September 21, 2018

C. Pratt
D. Thakore
CableLabs
B. Stark
AT&T
March 20, 2018

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.

Table of Contents

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:

```
byte-range-spec = first-byte-pos "-" [ 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 accessable 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=1234567-999999999999
```

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

```
206 Partial Content
Content-Range: bytes 1234567-999999999999/*
```

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:

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

returns

```
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:

```
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

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

with the server returning

```
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

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <https://www.rfc-editor.org/info/rfc7230>.

[RFC7233] Fielding, R., Ed., Lafon, Y., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Range Requests", RFC 7233, DOI 10.17487/RFC7233, June 2014, <https://www.rfc-editor.org/info/rfc7233>.

[RFC7234] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching", RFC 7234, DOI 10.17487/RFC7234, June 2014, <https://www.rfc-editor.org/info/rfc7234>.

6.2. Informative References

[DASH] ISO, "Information technology -- Dynamic adaptive streaming over HTTP (DASH) -- Part 1: Media presentation description and segment formats", ISO/IEC 23009-1:2014, May 2014, <http://standards.iso.org/ittf/PubliclyAvailableStandards/c065274_ISO_IEC_23009-1_2014.zip>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[RFC8216] Pantos, R., Ed. and W. May, "HTTP Live Streaming", RFC 8216, DOI 10.17487/RFC8216, August 2017, <https://www.rfc-editor.org/info/rfc8216>.

Acknowledgements

Mark Nottingham, Patrick McManus, Julian Reschke, Remy Lebeau, Rodger Combs, Thorsten Lohmar, Martin Thompson, Adrien de Croy, K. Morgan, Roy T. Fielding, Jeremy Poulter.

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

HTTP Working Group
Internet-Draft
Obsoletes: 6265 (if approved)
Intended status: Standards Track
Expires: February 8, 2018

A. Barth
M. West
Google, Inc

August 7, 2017

Cookies: HTTP State Management Mechanism
draft-ietf-httpbis-rfc6265bis-02

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.

Table of Contents

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:

```
OWS            = *( [ obs-fold ] WSP )
                   ; "optional" whitespace
obs-fold       = 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 know 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 know 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 ==

Set-Cookie: lang=en-US; Expires=Wed, 09 Jun 2021 10:18:14 GMT

== 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:

```
cookie-header = "Cookie:" OWS cookie-string OWS
cookie-string = cookie-pair *( ";" SP cookie-pair )
```

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).

    Otherwise:

    1. The name-value-pair string consists of all the characters contained in the set-cookie-string, and the unparsed-attributes is the empty string.

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

described in section 2.1), which is only a speedbump along the road to exploitation.

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 widly. 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.

A session fixation attack proceeds in three steps. First, the attacker transplants a session identifier from his or her user agent to the victim's user agent. Second, the victim uses that session identifier to interact with the server, possibly imbuing the session identifier with the user's credentials or confidential information. Third, the attacker uses the session identifier to interact with server directly, possibly obtaining the user's authority or confidential information.

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

[FETCH] van Kesteren, A., "Fetch", n.d., <https://fetch.spec.whatwg.org/>.

[HTML] Hickson, I., Pieters, S., van Kesteren, A., Jaegenstedt, P., and D. Denicola, "HTML", n.d., <https://html.spec.whatwg.org/>.

[PSL] "Public Suffix List", n.d., <https://publicsuffix.org/list/>.

[RFC1034] Mockapetris, P., "Domain names - concepts and facilities", STD 13, RFC 1034, DOI 10.17487/RFC1034, November 1987, <http://www.rfc-editor.org/info/rfc1034>.

[RFC1123] Braden, R., Ed., "Requirements for Internet Hosts - Application and Support", STD 3, RFC 1123, DOI 10.17487/RFC1123, October 1989, <http://www.rfc-editor.org/info/rfc1123>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, DOI 10.17487/RFC2616, June 1999, <http://www.rfc-editor.org/info/rfc2616>.

[RFC3490] Costello, A., "Internationalizing Domain Names in Applications (IDNA)", RFC 3490, DOI 10.17487/RFC3490, March 2003, <http://www.rfc-editor.org/info/rfc3490>.

See Section 6.3 for an explanation why the normative reference to an obsoleted specification is needed.

[RFC4790] Newman, C., Duerst, M., and A. Gulbrandsen, "Internet Application Protocol Collation Registry", RFC 4790, DOI 10.17487/RFC4790, March 2007, <http://www.rfc-editor.org/info/rfc4790>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <http://www.rfc-editor.org/info/rfc5234>.

[RFC5890] Klensin, J., "Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework", RFC 5890, DOI 10.17487/RFC5890, August 2010, <http://www.rfc-editor.org/info/rfc5890>.

[RFC6454] Barth, A., "The Web Origin Concept", RFC 6454, DOI 10.17487/RFC6454, December 2011, <http://www.rfc-editor.org/info/rfc6454>.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <http://www.rfc-editor.org/info/rfc7231>.

[SERVICE-WORKERS] Russell, A., Song, J., and J. Archibald, "Service Workers", n.d., <http://www.w3.org/TR/service-workers/>.

[USASCII] American National Standards Institute, "Coded Character Set -- 7-bit American Standard Code for Information Interchange", ANSI X3.4, 1986.

10.2. Informative References

[Aggarwal2010] Aggarwal, G., Burzstein, E., Jackson, C., and D. Boneh, "An Analysis of Private Browsing Modes in Modern Browsers", 2010, <http://www.usenix.org/events/sec10/tech/full_papers/Aggarwal.pdf>.

[app-isolation] Chen, E., Bau, J., Reis, C., Barth, A., and C. Jackson, "App Isolation - Get the Security of Multiple Browsers with Just One", 2011, <http://www.collinjackson.com/research/papers/appisolation.pdf>.

[CSRF] Barth, A., Jackson, C., and J. Mitchell, "Robust Defenses for Cross-Site Request Forgery", DOI 10.1145/1455770.1455782, ISBN 978-1-59593-810-7, ACM CCS '08: Proceedings of the 15th ACM conference on Computer and communications security (pages 75-88), October 2008, <http://portal.acm.org/citation.cfm?id=1455770.1455782>.

[I-D.ietf-httpbis-cookie-alone] West, M., "Deprecate modification of 'secure' cookies from non-secure origins", draft-ietf-httpbis-cookie-alone-01 (work in progress), September 2016.

[I-D.ietf-httpbis-cookie-prefixes] West, M., "Cookie Prefixes", draft-ietf-httpbis-cookie-prefixes-00 (work in progress), February 2016.

[I-D.ietf-httpbis-cookie-same-site] West, M. and M. Goodwin, "Same-Site Cookies", draft-ietf-httpbis-cookie-same-site-00 (work in progress), June 2016.

[prerendering] Bentzel, C., "Chrome Prerendering", n.d., <https://www.chromium.org/developers/design-documents/prerender>.

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, DOI 10.17487/RFC2818, May 2000, <http://www.rfc-editor.org/info/rfc2818>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <http://www.rfc-editor.org/info/rfc3629>.

[RFC3864] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, DOI 10.17487/RFC3864, September 2004, <http://www.rfc-editor.org/info/rfc3864>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <http://www.rfc-editor.org/info/rfc3986>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <http://www.rfc-editor.org/info/rfc4648>.

[RFC5895] Resnick, P. and P. Hoffman, "Mapping Characters for Internationalized Domain Names in Applications (IDNA) 2008", RFC 5895, DOI 10.17487/RFC5895, September 2010, <http://www.rfc-editor.org/info/rfc5895>.

[RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <http://www.rfc-editor.org/info/rfc6265>.

[RFC7034] Ross, D. and T. Gondrom, "HTTP Header Field X-Frame-Options", RFC 7034, DOI 10.17487/RFC7034, October 2013, <http://www.rfc-editor.org/info/rfc7034>.

[UTS46] Davis, M. and M. Suignard, "Unicode IDNA Compatibility Processing", UNICODE Unicode Technical Standards # 46, June 2016, <http://unicode.org/reports/tr46/>.

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. https://www.rfc-editor.org/errata_search.php?rfc=6265
- 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/

QUIC
Internet-Draft
Intended status: Standards Track
Expires: October 19, 2018

M. Bishop, Ed.
Akamai

April 17, 2018

Hypertext Transfer Protocol (HTTP) over QUIC
draft-ietf-quic-http-11

Abstract

The QUIC transport protocol has several features that are desirable in a transport for HTTP, such as stream multiplexing, per-stream flow control, and low-latency connection establishment. This document describes a mapping of HTTP semantics over QUIC. This document also identifies HTTP/2 features that are subsumed by QUIC, and describes how HTTP/2 extensions can be ported to QUIC.

Note to Readers

Discussion of this draft takes place on the QUIC working group mailing list (quic@ietf.org), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=quic [1].

Working Group information can be found at https://github.com/quicwg [2]; source code and issues list for this draft can be found at https://github.com/quicwg/base-drafts/labels/-http [3].

Status of This Memo

Table of Contents

1. Introduction

The QUIC transport protocol has several features that are desirable in a transport for HTTP, such as stream multiplexing, per-stream flow control, and low-latency connection establishment. This document describes a mapping of HTTP semantics over QUIC, drawing heavily on the existing TCP mapping, HTTP/2. Specifically, this document identifies HTTP/2 features that are subsumed by QUIC, and describes how the other features can be implemented atop QUIC.

QUIC is described in [QUIC-TRANSPORT]. For a full description of HTTP/2, see [RFC7540].

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.

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

This document uses the variable-length integer encoding from [QUIC-TRANSPORT].

Protocol elements called "frames" exist in both this document and [QUIC-TRANSPORT]. Where frames from [QUIC-TRANSPORT] are referenced, the frame name will be prefaced with "QUIC." For example, "QUIC APPLICATION_CLOSE frames." References without this preface refer to frames defined in Section 4.2.

2. Connection Setup and Management

2.1. Discovering an HTTP/QUIC Endpoint

An HTTP origin advertises the availability of an equivalent HTTP/QUIC endpoint via the Alt-Svc HTTP response header or the HTTP/2 ALTSVC frame ([RFC7838]), using the ALPN token defined in Section 2.2.

For example, an origin could indicate in an HTTP/1.1 or HTTP/2 response that HTTP/QUIC was available on UDP port 50781 at the same hostname by including the following header in any response:

```
Alt-Svc: hq=":50781"
```

On receipt of an Alt-Svc record indicating HTTP/QUIC support, a client MAY attempt to establish a QUIC connection to the indicated host and port and, if successful, send HTTP requests using the mapping described in this document.

Connectivity problems (e.g. firewall blocking UDP) can result in QUIC connection establishment failure, in which case the client SHOULD continue using the existing connection or try another alternative endpoint offered by the origin.

Servers MAY serve HTTP/QUIC on any UDP port, since an alternative always includes an explicit port.

2.1.1. QUIC Version Hints

This document defines the "quic" parameter for Alt-Svc, which MAY be used to provide version-negotiation hints to HTTP/QUIC clients. QUIC versions are four-octet sequences with no additional constraints on format. Leading zeros SHOULD be omitted for brevity.

Syntax:

```
quic = DQUOTE version-number [ "," version-number ] * DQUOTE
version-number = 1*8HEXDIG; hex-encoded QUIC version
```

Where multiple versions are listed, the order of the values reflects the server's preference (with the first value being the most preferred version). Reserved versions MAY be listed, but unreserved versions which are not supported by the alternative SHOULD NOT be present in the list. Origins MAY omit supported versions for any reason.

Clients MUST ignore any included versions which they do not support. The "quic" parameter MUST NOT occur more than once; clients SHOULD process only the first occurrence.

For example, suppose a server supported both version 0x00000001 and the version rendered in ASCII as "Q034". If it opted to include the reserved versions (from Section 4 of [QUIC-TRANSPORT]) 0x0 and 0x1abadaba, it could specify the following header:

```
Alt-Svc: hq=":49288";quic="1,1abadaba,51303334,0"
```

A client acting on this header would drop the reserved versions (because it does not support them), then attempt to connect to the alternative using the first version in the list which it does support.

2.2. Connection Establishment

HTTP/QUIC relies on QUIC as the underlying transport. The QUIC version being used MUST use TLS version 1.3 or greater as its handshake protocol. The Server Name Indication (SNI) extension [RFC6066] MUST be included in the TLS handshake.

QUIC connections are established as described in [QUIC-TRANSPORT]. During connection establishment, HTTP/QUIC support is indicated by selecting the ALPN token "hq" in the TLS handshake. Support for other application-layer protocols MAY be offered in the same handshake.

While connection-level options pertaining to the core QUIC protocol are set in the initial crypto handshake, HTTP-specific settings are conveyed in the SETTINGS frame. After the QUIC connection is established, a SETTINGS frame (Section 4.2.5) MUST be sent by each endpoint as the initial frame of their respective HTTP control stream (Stream ID 2 or 3, see Section 3). The server MUST NOT send data on any other stream until the client's SETTINGS frame has been received.

2.2.1. Draft Version Identification

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

Only implementations of the final, published RFC can identify themselves as "hq". Until such an RFC exists, implementations MUST NOT identify themselves using this string.

Implementations of draft versions of the protocol MUST add the string "-" and the corresponding draft number to the identifier. For example, draft-ietf-quic-http-01 is identified using the string "hq-01".

Non-compatible experiments that are based on these draft versions MUST append the string "-" and an experiment name to the identifier. For example, an experimental implementation based on draft-ietf-quic-http-09 which reserves an extra stream for unsolicited transmission of 1980s pop music might identify itself as "hq-09-rickroll". Note that any label MUST conform to the "token" syntax defined in Section 3.2.6 of [RFC7230]. Experimenters are encouraged to coordinate their experiments on the quic@ietf.org mailing list.

2.3. Connection Reuse

Once a connection exists to a server endpoint, this connection MAY be reused for requests with multiple different URI authority components. The client MAY send any requests for which the client considers the server authoritative.

An authoritative HTTP/QUIC endpoint is typically discovered because the client has received an Alt-Svc record from the request's origin which nominates the endpoint as a valid HTTP Alternative Service for that origin. As required by [RFC7838], clients MUST check that the nominated server can present a valid certificate for the origin before considering it authoritative. Clients MUST NOT assume that an HTTP/QUIC endpoint is authoritative for other origins without an explicit signal.

A server that does not wish clients to reuse connections for a particular origin can indicate that it is not authoritative for a request by sending a 421 (Misdirected Request) status code in response to the request (see Section 9.1.2 of [RFC7540]).

3. Stream Mapping and Usage

A QUIC stream provides reliable in-order delivery of bytes, but makes no guarantees about order of delivery with regard to bytes on other streams. On the wire, data is framed into QUIC STREAM frames, but this framing is invisible to the HTTP framing layer. A QUIC receiver buffers and orders received STREAM frames, exposing the data contained within as a reliable byte stream to the application.

QUIC reserves the first client-initiated, bidirectional stream (Stream 0) for cryptographic operations. HTTP over QUIC reserves the first unidirectional stream sent by either peer (Streams 2 and 3) for sending and receiving HTTP control frames. This pair of unidirectional streams is analogous to HTTP/2's Stream 0. The data sent on these streams is critical to the HTTP connection. If either control stream is closed for any reason, this MUST be treated as a connection error of type QUIC_CLOSED_CRITICAL_STREAM.

When HTTP headers and data are sent over QUIC, the QUIC layer handles most of the stream management.

An HTTP request/response consumes a single client-initiated, bidirectional stream. A bidirectional stream ensures that the response can be readily correlated with the request. This means that the client's first request occurs on QUIC stream 4, with subsequent requests on stream 8, 12, and so on.

Server push uses server-initiated, unidirectional streams. Thus, the server's first push consumes stream 7 and subsequent pushes use stream 11, 15, and so on.

These streams carry frames related to the request/response (see Section 4.2). When a stream terminates cleanly, if the last frame on the stream was truncated, this MUST be treated as a connection error (see HTTP_MALFORMED_FRAME in Section 6.1). Streams which terminate abruptly may be reset at any point in the frame.

Streams SHOULD be used sequentially, with no gaps.

HTTP does not need to do any separate multiplexing when using QUIC - data sent over a QUIC stream always maps to a particular HTTP transaction. Requests and responses are considered complete when the corresponding QUIC stream is closed in the appropriate direction.

3.1. Control Streams

Since most connection-level concerns will be managed by QUIC, the primary use of Streams 2 and 3 will be for the SETTINGS frame when the connection opens and for PRIORITY frames subsequently.

A pair of unidirectional streams is used rather than a single bidirectional stream. This allows either peer to send data as soon they are able. Depending on whether 0-RTT is enabled on the connection, either client or server might be able to send stream data first after the cryptographic handshake completes.

3.2. HTTP Message Exchanges

A client sends an HTTP request on a client-initiated, bidirectional QUIC stream. A server sends an HTTP response on the same stream as the request.

An HTTP message (request or response) consists of:

1. one header block (see Section 4.2.2) containing the message headers (see [RFC7230], Section 3.2),

2. the payload body (see [RFC7230], Section 3.3), sent as a series of DATA frames (see Section 4.2.1),

3. optionally, one header block containing the trailer-part, if present (see [RFC7230], Section 4.1.2).

In addition, prior to sending the message header block indicated above, a response may contain zero or more header blocks containing the message headers of informational (1xx) HTTP responses (see [RFC7230], Section 3.2 and [RFC7231], Section 6.2).

PUSH_PROMISE frames MAY be interleaved with the frames of a response message indicating a pushed resource related to the response. These PUSH_PROMISE frames are not part of the response, but carry the headers of a separate HTTP request message. See Section 3.4 for more details.

The "chunked" transfer encoding defined in Section 4.1 of [RFC7230] MUST NOT be used.

Trailing header fields are carried in an additional header block following the body. Such a header block is a sequence of HEADERS frames with End Header Block set on the last frame. Senders MUST send only one header block in the trailers section; receivers MUST discard any subsequent header blocks.

An HTTP request/response exchange fully consumes a QUIC stream. After sending a request, a client closes the stream for sending; after sending a response, the server closes the stream for sending and the QUIC stream is fully closed.

A server can send a complete response prior to the client sending an entire request if the response does not depend on any portion of the request that has not been sent and received. When this is true, a server MAY request that the client abort transmission of a request without error by triggering a QUIC STOP_SENDING with error code HTTP_EARLY_RESPONSE, sending a complete response, and cleanly closing its streams. Clients MUST NOT discard complete responses as a result of having their request terminated abruptly, though clients can always discard responses at their discretion for other reasons. Servers MUST NOT abort a response in progress as a result of receiving a solicited RST_STREAM.

3.2.1. Header Compression

HTTP/QUIC uses QPACK header compression as described in [QPACK], a variation of HPACK which allows the flexibility to avoid header-compression-induced head-of-line blocking. See that document for additional details.

3.2.2. The CONNECT Method

The pseudo-method CONNECT ([RFC7231], Section 4.3.6) is primarily used with HTTP proxies to establish a TLS session with an origin server for the purposes of interacting with "https" resources. In HTTP/1.x, CONNECT is used to convert an entire HTTP connection into a tunnel to a remote host. In HTTP/2, the CONNECT method is used to establish a tunnel over a single HTTP/2 stream to a remote host for similar purposes.

A CONNECT request in HTTP/QUIC functions in the same manner as in HTTP/2. The request MUST be formatted as described in [RFC7540], Section 8.3. A CONNECT request that does not conform to these restrictions is malformed. The request stream MUST NOT be half-closed at the end of the request.

A proxy that supports CONNECT establishes a TCP connection ([RFC0793]) to the server identified in the ":authority" pseudo-header field. Once this connection is successfully established, the proxy sends a HEADERS frame containing a 2xx series status code to the client, as defined in [RFC7231], Section 4.3.6.

All DATA frames on the request stream correspond to data sent on the TCP connection. Any DATA frame sent by the client is transmitted by

the proxy to the TCP server; data received from the TCP server is packaged into DATA frames by the proxy. Note that the size and number of TCP segments is not guaranteed to map predictably to the size and number of HTTP DATA or QUIC STREAM frames.

The TCP connection can be closed by either peer. When the client ends the request stream (that is, the receive stream at the proxy enters the "Data Recvd" state), the proxy will set the FIN bit on its connection to the TCP server. When the proxy receives a packet with the FIN bit set, it will terminate the send stream that it sends to client. TCP connections which remain half-closed in a single direction are not invalid, but are often handled poorly by servers, so clients SHOULD NOT cause send a STREAM frame with a FIN bit for connections on which they are still expecting data.

A TCP connection error is signaled with RST_STREAM. A proxy treats any error in the TCP connection, which includes receiving a TCP segment with the RST bit set, as a stream error of type HTTP_CONNECT_ERROR (Section 6.1). Correspondingly, a proxy MUST send a TCP segment with the RST bit set if it detects an error with the stream or the QUIC connection.

3.2.3. Request Cancellation

Either client or server can cancel requests by closing the stream (QUIC RST_STREAM or STOP_SENDING frames, as appropriate) with an error type of HTTP_REQUEST_CANCELLED (Section 6.1). When the client cancels a request or response, it indicates that the response is no longer of interest.

When the server cancels either direction of the request stream using HTTP_REQUEST_CANCELLED, it indicates that no application processing was performed. The client can treat requests cancelled by the server as though they had never been sent at all, thereby allowing them to be retried later on a new connection. Servers MUST NOT use the HTTP_REQUEST_CANCELLED status for requests which were partially or fully processed.

Note: In this context, "processed" means that some data from the stream was passed to some higher layer of software that might have taken some action as a result.

If a stream is cancelled after receiving a complete response, the client MAY ignore the cancellation and use the response. However, if a stream is cancelled after receiving a partial response, the response SHOULD NOT be used. Automatically retrying such requests is not possible, unless this is otherwise permitted (e.g., idempotent actions like GET, PUT, or DELETE).

3.3. Request Prioritization

HTTP/QUIC uses the priority scheme described in [RFC7540], Section 5.3. In this priority scheme, a given request can be designated as dependent upon another request, which expresses the preference that the latter stream (the "parent" request) be allocated resources before the former stream (the "dependent" request). Taken together, the dependencies across all requests in a connection form a dependency tree. The structure of the dependency tree changes as PRIORITY frames add, remove, or change the dependency links between requests.

The PRIORITY frame Section 4.2.3 identifies a request either by identifying the stream that carries a request or by using a Push ID (Section 4.2.6).

Only a client can send PRIORITY frames. A server MUST NOT send a PRIORITY frame.

3.4. Server Push

HTTP/QUIC supports server push in a similar manner to [RFC7540], but uses different mechanisms. During connection establishment, the client enables server push by sending a MAX_PUSH_ID frame (see Section 4.2.9). A server cannot use server push until it receives a MAX_PUSH_ID frame.

As with server push for HTTP/2, the server initiates a server push by sending a PUSH_PROMISE frame (see Section 4.2.6) that includes request headers for the promised request. Promised requests MUST conform to the requirements in Section 8.2 of [RFC7540].

The PUSH_PROMISE frame is sent on the client-initiated, bidirectional stream that carried the request that generated the push. This allows the server push to be associated with a request. Ordering of a PUSH_PROMISE in relation to certain parts of the response is important (see Section 8.2.1 of [RFC7540]).

Unlike HTTP/2, the PUSH_PROMISE does not reference a stream; it contains a Push ID. The Push ID uniquely identifies a server push. This allows a server to fulfill promises in the order that best suits its needs.

When a server later fulfills a promise, the server push response is conveyed on a push stream. A push stream is a server-initiated, unidirectional stream. A push stream identifies the Push ID of the promise that it fulfills, encoded as a variable-length integer.

A server SHOULD use Push IDs sequentially, starting at 0. A client uses the MAX_PUSH_ID frame (Section 4.2.9) to limit the number of pushes that a server can promise. A client MUST treat receipt of a push stream with a Push ID that is greater than the maximum Push ID as a connection error of type HTTP_PUSH_LIMIT_EXCEEDED.

If a promised server push is not needed by the client, the client SHOULD send a CANCEL_PUSH frame; if the push stream is already open, a QUIC STOP_SENDING frame with an appropriate error code can be used instead (e.g., HTTP_PUSH_REFUSED, HTTP_PUSH_ALREADY_IN_CACHE; see Section 6). This asks the server not to transfer the data and indicates that it will be discarded upon receipt.

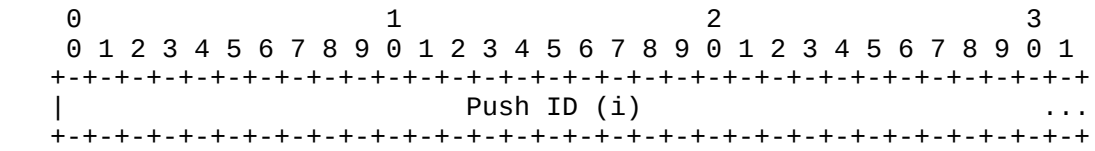


Figure 1: Push Stream Header

Push streams always begin with a header containing the Push ID. Each Push ID MUST only be used once in a push stream header. If a push stream header includes a Push ID that was used in another push stream header, the client MUST treat this as a connection error of type HTTP_DUPLICATE_PUSH. The same Push ID can be used in multiple PUSH_PROMISE frames (see Section 4.2.6).

After the header, a push stream contains a response (Section 3.2), with response headers, a response body (if any) carried by DATA frames, then trailers (if any) carried by HEADERS frames.

4. HTTP Framing Layer

Frames are used on each stream. This section describes HTTP framing in QUIC and highlights some differences from HTTP/2 framing. For more detail on differences from HTTP/2, see Section 7.2.

4.1. Frame Layout

All frames have the following format:

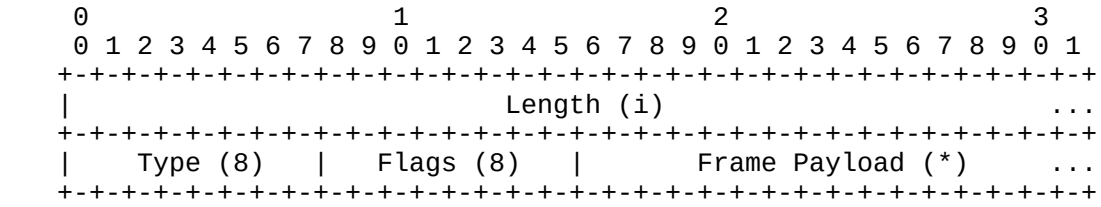


Figure 2: HTTP/QUIC frame format

A frame includes the following fields:

Length: A variable-length integer that describes the length of the Frame Payload. This length does not include the frame header.

Type: An 8-bit type for the frame.

Flags: An 8-bit field containing flags. The Type field determines the semantics of flags.

Frame Payload: A payload, the semantics of which are determined by the Type field.

4.2. Frame Definitions

4.2.1. DATA

DATA frames (type=0x0) convey arbitrary, variable-length sequences of octets associated with an HTTP request or response payload.

The DATA frame defines no flags.

DATA frames MUST be associated with an HTTP request or response. If a DATA frame is received on either control stream, the recipient MUST respond with a connection error (Section 6) of type HTTP_WRONG_STREAM.

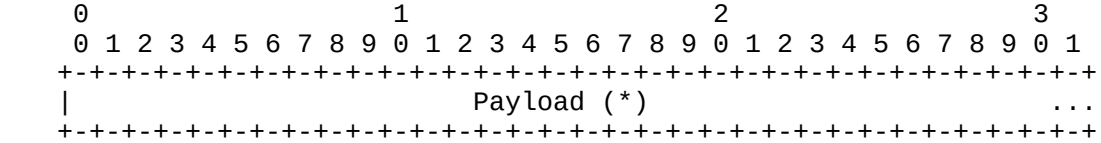


Figure 3: DATA frame payload

DATA frames MUST contain a non-zero-length payload. If a DATA frame is received with a payload length of zero, the recipient MUST respond with a stream error (Section 6) of type HTTP_MALFORMED_FRAME.

4.2.2. HEADERS

The HEADERS frame (type=0x1) is used to carry a header block, compressed using QPACK. See [QPACK] for more details.

The HEADERS frame defines a single flag:

BLOCKING (0x01): Indicates the stream might need to wait for dependent headers before processing. If 0, the frame can be processed immediately upon receipt.

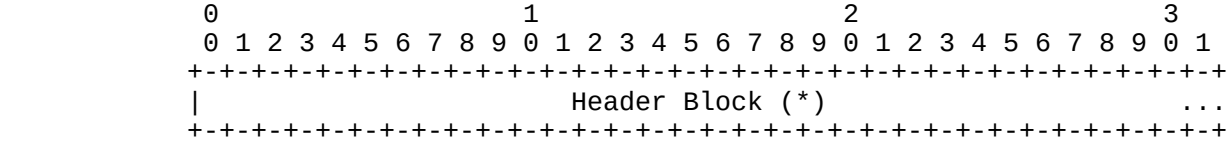


Figure 4: HEADERS frame payload

HEADERS frames can be sent on the Control Stream as well as on request / push streams. The value of BLOCKING MUST be 0 for HEADERS frames on the Control Stream, since they can only depend on previous HEADERS on the same stream.

4.2.3. PRIORITY

The PRIORITY (type=0x02) frame specifies the sender-advised priority of a stream and is substantially different in format from [RFC7540]. In order to ensure that prioritization is processed in a consistent order, PRIORITY frames MUST be sent on the control stream. A PRIORITY frame sent on any other stream MUST be treated as a HTTP_WRONG_STREAM error.

The format has been modified to accommodate not being sent on a request stream, to allow for identification of server pushes, and the larger stream ID space of QUIC. The semantics of the Stream Dependency, Weight, and E flag are otherwise the same as in HTTP/2.

The flags defined are:

PUSH_PRIORITIZED (0x04): Indicates that the Prioritized Stream is a server push rather than a request.

PUSH_DEPENDENT (0x02): Indicates a dependency on a server push.

E (0x01): Indicates that the stream dependency is exclusive (see [RFC7540], Section 5.3).

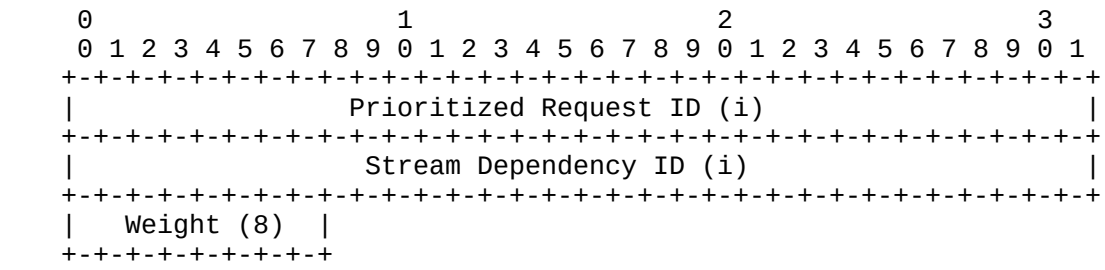


Figure 5: PRIORITY frame payload

The PRIORITY frame payload has the following fields:

Prioritized Request ID: A variable-length integer that identifies a request. This contains the Stream ID of a request stream when the PUSH_PRIORITIZED flag is clear, or a Push ID when the PUSH_PRIORITIZED flag is set.

Stream Dependency ID: A variable-length integer that identifies a dependent request. This contains the Stream ID of a request stream when the PUSH_DEPENDENT flag is clear, or a Push ID when the PUSH_DEPENDENT flag is set. A request Stream ID of 0 indicates a dependency on the root stream. For details of dependencies, see Section 3.3 and [RFC7540], Section 5.3.

Weight: An unsigned 8-bit integer representing a priority weight for the stream (see [RFC7540], Section 5.3). Add one to the value to obtain a weight between 1 and 256.

A PRIORITY frame identifies a request to prioritize, and a request upon which that request is dependent. A Prioritized Request ID or Stream Dependency ID identifies a client-initiated request using the corresponding stream ID when the corresponding PUSH_PRIORITIZED or PUSH_DEPENDENT flag is not set. Setting the PUSH_PRIORITIZED or PUSH_DEPENDENT flag causes the Prioritized Request ID or Stream Dependency ID (respectively) to identify a server push using a Push ID (see Section 4.2.6 for details).

A PRIORITY frame MAY identify a Stream Dependency ID using a Stream ID of 0; as in [RFC7540], this makes the request dependent on the root of the dependency tree.

A PRIORITY frame MUST identify a client-initiated, bidirectional stream. A server MUST treat receipt of PRIORITY frame with a Stream ID of any other type as a connection error of type HTTP_MALFORMED_FRAME.

Stream ID 0 cannot be reprioritized. A Prioritized Request ID that identifies Stream 0 MUST be treated as a connection error of type HTTP_MALFORMED_FRAME.

A PRIORITY frame that does not reference a request MUST be treated as a HTTP_MALFORMED_FRAME error, unless it references Stream ID 0. A PRIORITY that sets a PUSH_PRIORITIZED or PUSH_DEPENDENT flag, but then references a non-existent Push ID MUST be treated as a HTTP_MALFORMED_FRAME error.

A PRIORITY frame MUST contain only the identified fields. A PRIORITY frame that contains more or fewer fields, or a PRIORITY frame that includes a truncated integer encoding MUST be treated as a connection error of type HTTP_MALFORMED_FRAME.

4.2.4. CANCEL_PUSH

The CANCEL_PUSH frame (type=0x3) is used to request cancellation of server push prior to the push stream being created. The CANCEL_PUSH frame identifies a server push request by Push ID (see Section 4.2.6) using a variable-length integer.

When a server receives this frame, it aborts sending the response for the identified server push. If the server has not yet started to send the server push, it can use the receipt of a CANCEL_PUSH frame to avoid opening a stream. If the push stream has been opened by the server, the server SHOULD sent a QUIC RST_STREAM frame on those streams and cease transmission of the response.

A server can send this frame to indicate that it won't be sending a response prior to creation of a push stream. Once the push stream has been created, sending CANCEL_PUSH has no effect on the state of the push stream. A QUIC RST_STREAM frame SHOULD be used instead to cancel transmission of the server push response.

A CANCEL_PUSH frame is sent on the control stream. Sending a CANCEL_PUSH frame on a stream other than the control stream MUST be treated as a stream error of type HTTP_WRONG_STREAM.

The CANCEL_PUSH frame has no defined flags.

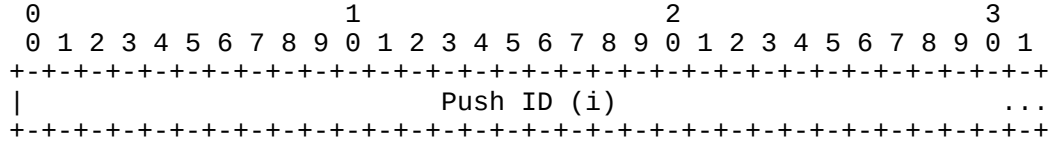


Figure 6: CANCEL_PUSH frame payload

The CANCEL_PUSH frame carries a Push ID encoded as a variable-length integer. The Push ID identifies the server push that is being cancelled (see Section 4.2.6).

If the client receives a CANCEL_PUSH frame, that frame might identify a Push ID that has not yet been mentioned by a PUSH_PROMISE frame.

An endpoint MUST treat a CANCEL_PUSH frame which does not contain exactly one properly-formatted variable-length integer as a connection error of type HTTP_MALFORMED_FRAME.

4.2.5. SETTINGS

The SETTINGS frame (type=0x4) conveys configuration parameters that affect how endpoints communicate, such as preferences and constraints on peer behavior, and is different from [RFC7540]. Individually, a SETTINGS parameter can also be referred to as a "setting".

SETTINGS parameters are not negotiated; they describe characteristics of the sending peer, which can be used by the receiving peer. However, a negotiation can be implied by the use of SETTINGS - a peer uses SETTINGS to advertise a set of supported values. The recipient can then choose which entries from this list are also acceptable and proceed with the value it has chosen. (This choice could be announced in a field of an extension frame, or in its own value in SETTINGS.)

Different values for the same parameter can be advertised by each peer. For example, a client might be willing to consume very large response headers, while servers are more cautious about request size.

Parameters MUST NOT occur more than once. A receiver MAY treat the presence of the same parameter more than once as a connection error of type HTTP_MALFORMED_FRAME.

The SETTINGS frame defines no flags.

The payload of a SETTINGS frame consists of zero or more parameters, each consisting of an unsigned 16-bit setting identifier and a length-prefixed binary value.

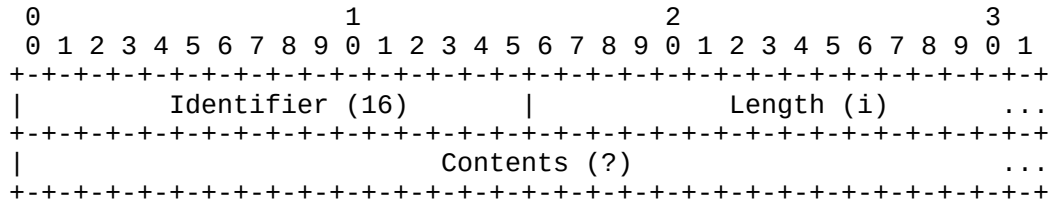


Figure 7: SETTINGS value format

A zero-length content indicates that the setting value is a Boolean and true. False is indicated by the absence of the setting.

Non-zero-length values MUST be compared against the remaining length of the SETTINGS frame. Any value which purports to cross the end of the frame MUST cause the SETTINGS frame to be considered malformed and trigger a connection error of type HTTP_MALFORMED_FRAME.

An implementation MUST ignore the contents for any SETTINGS identifier it does not understand.

SETTINGS frames always apply to a connection, never a single stream. A SETTINGS frame MUST be sent as the first frame of either control stream (see Section 3) by each peer, and MUST NOT be sent subsequently or on any other stream. If an endpoint receives an SETTINGS frame on a different stream, the endpoint MUST respond with a connection error of type HTTP_WRONG_STREAM. If an endpoint receives a second SETTINGS frame, the endpoint MUST respond with a connection error of type HTTP_MALFORMED_FRAME.

The SETTINGS frame affects connection state. A badly formed or incomplete SETTINGS frame MUST be treated as a connection error (Section 6) of type HTTP_MALFORMED_FRAME.

4.2.5.1. Integer encoding

Settings which are integers use the QUIC variable-length integer encoding.

4.2.5.2. Defined SETTINGS Parameters

The following settings are defined in HTTP/QUIC:

SETTINGS_HEADER_TABLE_SIZE (0x1): An integer with a maximum value of 2^30 - 1.

SETTINGS_MAX_HEADER_LIST_SIZE (0x6): An integer with a maximum value of 2^30 - 1

4.2.5.3. Initial SETTINGS Values

When a 0-RTT QUIC connection is being used, the client's initial requests will be sent before the arrival of the server's SETTINGS frame. Clients MUST store the settings the server provided in the session being resumed and MUST comply with stored settings until the server's current settings are received.

Servers MAY continue processing data from clients which exceed its current configuration during the initial flight. In this case, the client MUST apply the new settings immediately upon receipt.

When a 1-RTT QUIC connection is being used, the client MUST NOT send requests prior to receiving and processing the server's SETTINGS frame.

4.2.6. PUSH_PROMISE

The PUSH_PROMISE frame (type=0x05) is used to carry a request header set from server to client, as in HTTP/2. The PUSH_PROMISE frame defines a single flag:

BLOCKING (0x01): Indicates the stream might need to wait for dependent headers before processing. If 0, the frame can be processed immediately upon receipt.

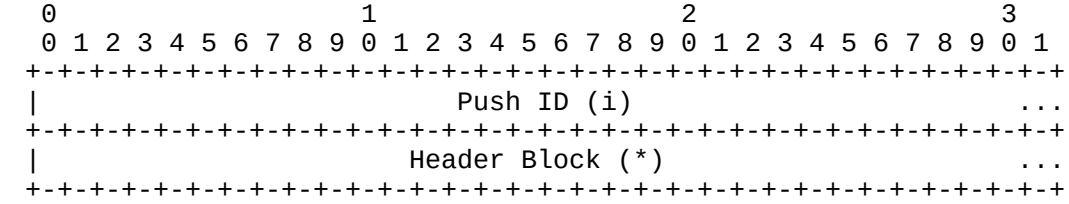


Figure 8: PUSH_PROMISE frame payload

The payload consists of:

Push ID: A variable-length integer that identifies the server push request. A push ID is used in push stream header (Section 3.4), CANCEL_PUSH frames (Section 4.2.4), and PRIORITY frames (Section 4.2.3).

Header Block: QPACK-compressed request headers for the promised response. See [QPACK] for more details.

A server MUST NOT use a Push ID that is larger than the client has provided in a MAX_PUSH_ID frame (Section 4.2.9). A client MUST treat

receipt of a PUSH_PROMISE that contains a larger Push ID than the client has advertised as a connection error of type HTTP_MALFORMED_FRAME.

A server MAY use the same Push ID in multiple PUSH_PROMISE frames. This allows the server to use the same server push in response to multiple concurrent requests. Referencing the same server push ensures that a PUSH_PROMISE can be made in relation to every response in which server push might be needed without duplicating pushes.

A server that uses the same Push ID in multiple PUSH_PROMISE frames MUST include the same header fields each time. The octets of the header block MAY be different due to differing encoding, but the header fields and their values MUST be identical. Note that ordering of header fields is significant. A client MUST treat receipt of a PUSH_PROMISE with conflicting header field values for the same Push ID as a connection error of type HTTP_MALFORMED_FRAME.

Allowing duplicate references to the same Push ID is primarily to reduce duplication caused by concurrent requests. A server SHOULD avoid reusing a Push ID over a long period. Clients are likely to consume server push responses and not retain them for reuse over time. Clients that see a PUSH_PROMISE that uses a Push ID that they have since consumed and discarded are forced to ignore the PUSH_PROMISE.

4.2.7. GOAWAY

The GOAWAY frame (type=0x7) is used to initiate graceful shutdown of a connection by a server. GOAWAY allows a server to stop accepting new requests while still finishing processing of previously received requests. This enables administrative actions, like server maintenance. GOAWAY by itself does not close a connection.

The GOAWAY frame does not define any flags.

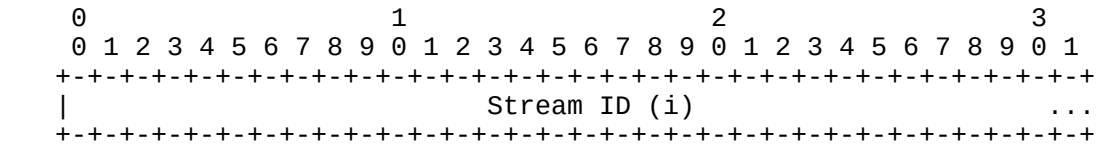


Figure 9: GOAWAY frame payload

The GOAWAY frame carries a QUIC Stream ID for a client-initiated, bidirectional stream encoded as a variable-length integer. A client MUST treat receipt of a GOAWAY frame containing a Stream ID of any other type as a connection error of type HTTP_MALFORMED_FRAME.

Clients do not need to send GOAWAY to initiate a graceful shutdown; they simply stop making new requests. A server MUST treat receipt of a GOAWAY frame as a connection error (Section 6) of type HTTP_UNEXPECTED_GOAWAY.

The GOAWAY frame applies to the connection, not a specific stream. An endpoint MUST treat a GOAWAY frame on a stream other than the control stream as a connection error (Section 6) of type HTTP_WRONG_STREAM.

New client requests might already have been sent before the client receives the server's GOAWAY frame. The GOAWAY frame contains the Stream ID of the last client-initiated request that was or might be processed in this connection, which enables client and server to agree on which requests were accepted prior to the connection shutdown. This identifier MAY be lower than the stream limit identified by a QUIC MAX_STREAM_ID frame, and MAY be zero if no requests were processed. Servers SHOULD NOT increase the MAX_STREAM_ID limit after sending a GOAWAY frame.

Once sent, the server MUST cancel requests sent on streams with an identifier higher than the included last Stream ID. Clients MUST NOT send new requests on the connection after receiving GOAWAY, although requests might already be in transit. A new connection can be established for new requests.

If the client has sent requests on streams with a higher Stream ID than indicated in the GOAWAY frame, those requests are considered cancelled (Section 3.2.3). Clients SHOULD reset any streams above this ID with the error code HTTP_REQUEST_CANCELLED. Servers MAY also cancel requests on streams below the indicated ID if these requests were not processed.

Requests on Stream IDs less than or equal to the Stream ID in the GOAWAY frame might have been processed; their status cannot be known until they are completed successfully, reset individually, or the connection terminates.

Servers SHOULD send a GOAWAY frame when the closing of a connection is known in advance, even if the advance notice is small, so that the remote peer can know whether a stream has been partially processed or not. For example, if an HTTP client sends a POST at the same time that a server closes a QUIC connection, the client cannot know if the server started to process that POST request if the server does not send a GOAWAY frame to indicate what streams it might have acted on.

For unexpected closures caused by error conditions, a QUIC CONNECTION_CLOSE or APPLICATION_CLOSE frame MUST be used. However, a

GOAWAY MAY be sent first to provide additional detail to clients and to allow the client to retry requests. Including the GOAWAY frame in the same packet as the QUIC CONNECTION_CLOSE or APPLICATION_CLOSE frame improves the chances of the frame being received by clients.

If a connection terminates without a GOAWAY frame, the last Stream ID is effectively the highest possible Stream ID (as determined by QUIC's MAX_STREAM_ID).

An endpoint MAY send multiple GOAWAY frames if circumstances change. For instance, an endpoint that sends GOAWAY without an error code during graceful shutdown could subsequently encounter an error condition. The last stream identifier from the last GOAWAY frame received indicates which streams could have been acted upon. A server MUST NOT increase the value they send in the last Stream ID, since clients might already have retried unprocessed requests on another connection.

A client that is unable to retry requests loses all requests that are in flight when the server closes the connection. A server that is attempting to gracefully shut down a connection SHOULD send an initial GOAWAY frame with the last Stream ID set to the current value of QUIC's MAX_STREAM_ID and SHOULD NOT increase the MAX_STREAM_ID thereafter. This signals to the client that a shutdown is imminent and that initiating further requests is prohibited. After allowing time for any in-flight requests (at least one round-trip time), the server MAY send another GOAWAY frame with an updated last Stream ID. This ensures that a connection can be cleanly shut down without losing requests.

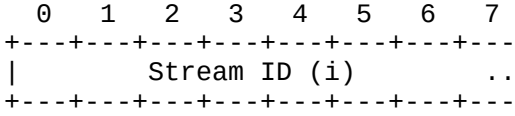
Once all requests on streams at or below the identified stream number have been completed or cancelled, and all promised server push responses associated with those requests have been completed or cancelled, the connection can be closed using an Immediate Close (see [QUIC-TRANSPORT]). An endpoint that completes a graceful shutdown SHOULD use the QUIC APPLICATION_CLOSE frame with the HTTP_NO_ERROR code.

4.2.8. HEADER_ACK

The HEADER_ACK frame (type=0x8) is used by header compression to ensure consistency. The frames are sent from the QPACK decoder to the QPACK encoder; that is, the server sends them to the client to acknowledge processing of the client's header blocks, and the client sends them to the server to acknowledge processing of the server's header blocks.

The HEADER_ACK frame is sent on the Control Stream when the QPACK decoder has fully processed a header block. It is used by the peer's QPACK encoder to determine whether subsequent indexed representations that might reference that block are vulnerable to head-of-line blocking, and to prevent eviction races. See [QPACK] for more details on the use of this information.

The HEADER_ACK frame indicates the stream on which the header block was processed by encoding the Stream ID as a variable-length integer. The same Stream ID can be identified multiple times, as multiple header-containing blocks can be sent on a single stream in the case of intermediate responses, trailers, pushed requests, etc. as well as on the Control Streams. Since header frames on each stream are received and processed in order, this gives the encoder precise feedback on which header blocks within a stream have been fully processed.



HEADER_ACK frame

The HEADER_ACK frame does not define any flags.

4.2.9. MAX_PUSH_ID

The MAX_PUSH_ID frame (type=0xD) is used by clients to control the number of server pushes that the server can initiate. This sets the maximum value for a Push ID that the server can use in a PUSH_PROMISE frame. Consequently, this also limits the number of push streams that the server can initiate in addition to the limit set by the QUIC MAX_STREAM_ID frame.

The MAX_PUSH_ID frame is always sent on a control stream. Receipt of a MAX_PUSH_ID frame on any other stream MUST be treated as a connection error of type HTTP_WRONG_STREAM.

A server MUST NOT send a MAX_PUSH_ID frame. A client MUST treat the receipt of a MAX_PUSH_ID frame as a connection error of type HTTP_MALFORMED_FRAME.

The maximum Push ID is unset when a connection is created, meaning that a server cannot push until it receives a MAX_PUSH_ID frame. A client that wishes to manage the number of promised server pushes can increase the maximum Push ID by sending a MAX_PUSH_ID frame as the server fulfills or cancels server pushes.

The MAX_PUSH_ID frame has no defined flags.

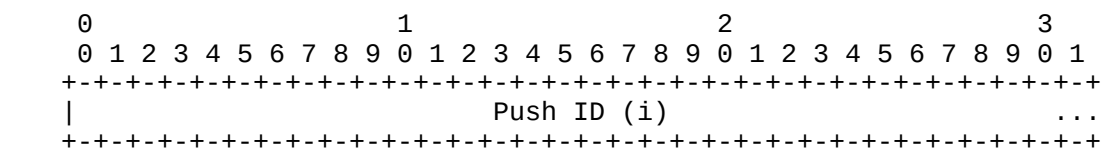


Figure 10: MAX_PUSH_ID frame payload

The MAX_PUSH_ID frame carries a single variable-length integer that identifies the maximum value for a Push ID that the server can use (see Section 4.2.6). A MAX_PUSH_ID frame cannot reduce the maximum Push ID; receipt of a MAX_PUSH_ID that contains a smaller value than previously received MUST be treated as a connection error of type HTTP_MALFORMED_FRAME.

A server MUST treat a MAX_PUSH_ID frame payload that does not contain a single variable-length integer as a connection error of type HTTP_MALFORMED_FRAME.

5. Connection Management

QUIC connections are persistent. All of the considerations in Section 9.1 of [RFC7540] apply to the management of QUIC connections.

HTTP clients are expected to use QUIC PING frames to keep connections open. Servers SHOULD NOT use PING frames to keep a connection open. A client SHOULD NOT use PING frames for this purpose unless there are responses outstanding for requests or server pushes. If the client is not expecting a response from the server, allowing an idle connection to time out (based on the idle_timeout transport parameter) is preferred over expending effort maintaining a connection that might not be needed. A gateway MAY use PING to maintain connections in anticipation of need rather than incur the latency cost of connection establishment to servers.

6. Error Handling

QUIC allows the application to abruptly terminate (reset) individual streams or the entire connection when an error is encountered. These are referred to as "stream errors" or "connection errors" and are described in more detail in [QUIC-TRANSPORT].

This section describes HTTP-specific error codes which can be used to express the cause of a connection or stream error.

6.1. HTTP/QUIC Error Codes

The following error codes are defined for use in QUIC RST_STREAM, STOP_SENDING, and CONNECTION_CLOSE frames when using HTTP/QUIC.

STOPPING (0x00): This value is reserved by the transport to be used in response to QUIC STOP_SENDING frames.

HTTP_NO_ERROR (0x01): No error. This is used when the connection or stream needs to be closed, but there is no error to signal.

HTTP_PUSH_REFUSED (0x02): The server has attempted to push content which the client will not accept on this connection.

HTTP_INTERNAL_ERROR (0x03): An internal error has occurred in the HTTP stack.

HTTP_PUSH_ALREADY_IN_CACHE (0x04): The server has attempted to push content which the client has cached.

HTTP_REQUEST_CANCELLED (0x05): The client no longer needs the requested data.

HTTP_HPACK_DECOMPRESSION_FAILED (0x06): HPACK failed to decompress a frame and cannot continue.

HTTP_CONNECT_ERROR (0x07): The connection established in response to a CONNECT request was reset or abnormally closed.

HTTP_EXCESSIVE_LOAD (0x08): The endpoint detected that its peer is exhibiting a behavior that might be generating excessive load.

HTTP_VERSION_FALLBACK (0x09): The requested operation cannot be served over HTTP/QUIC. The peer should retry over HTTP/2.

HTTP_WRONG_STREAM (0x0A): A frame was received on stream where it is not permitted.

HTTP_PUSH_LIMIT_EXCEEDED (0x0B): A Push ID greater than the current maximum Push ID was referenced.

HTTP_DUPLICATE_PUSH (0x0C): A Push ID was referenced in two different stream headers.

HTTP_MALFORMED_FRAME (0x01XX): An error in a specific frame type. The frame type is included as the last octet of the error code. For example, an error in a MAX_PUSH_ID frame would be indicated with the code (0x10D).

7. Considerations for Transitioning from HTTP/2

HTTP/QUIC is strongly informed by HTTP/2, and bears many similarities. This section describes the approach taken to design HTTP/QUIC, points out important differences from HTTP/2, and describes how to map HTTP/2 extensions into HTTP/QUIC.

HTTP/QUIC begins from the premise that HTTP/2 code reuse is a useful feature, but not a hard requirement. HTTP/QUIC departs from HTTP/2 primarily where necessary to accommodate the differences in behavior between QUIC and TCP (lack of ordering, support for streams). We intend to avoid gratuitous changes which make it difficult or impossible to build extensions with the same semantics applicable to both protocols at once.

These departures are noted in this section.

7.1. Streams

HTTP/QUIC permits use of a larger number of streams (2^62-1) than HTTP/2. The considerations about exhaustion of stream identifier space apply, though the space is significantly larger such that it is likely that other limits in QUIC are reached first, such as the limit on the connection flow control window.

7.2. HTTP Frame Types

Many framing concepts from HTTP/2 can be elided away on QUIC, because the transport deals with them. Because frames are already on a stream, they can omit the stream number. Because frames do not block multiplexing (QUIC's multiplexing occurs below this layer), the support for variable-maximum-length packets can be removed. Because stream termination is handled by QUIC, an END_STREAM flag is not required.

Frame payloads are largely drawn from [RFC7540]. However, QUIC includes many features (e.g. flow control) which are also present in HTTP/2. In these cases, the HTTP mapping does not re-implement them. As a result, several HTTP/2 frame types are not required in HTTP/QUIC. Where an HTTP/2-defined frame is no longer used, the frame ID has been reserved in order to maximize portability between HTTP/2 and HTTP/QUIC implementations. However, even equivalent frames between the two mappings are not identical.

Many of the differences arise from the fact that HTTP/2 provides an absolute ordering between frames across all streams, while QUIC provides this guarantee on each stream only. As a result, if a frame

type makes assumptions that frames from different streams will still be received in the order sent, HTTP/QUIC will break them.

For example, implicit in the HTTP/2 prioritization scheme is the notion of in-order delivery of priority changes (i.e., dependency tree mutations): since operations on the dependency tree such as reparenting a subtree are not commutative, both sender and receiver must apply them in the same order to ensure that both sides have a consistent view of the stream dependency tree. HTTP/2 specifies priority assignments in PRIORITY frames and (optionally) in HEADERS frames. To achieve in-order delivery of priority changes in HTTP/QUIC, PRIORITY frames are sent on the control stream and the PRIORITY section is removed from the HEADERS frame.

Likewise, HPACK was designed with the assumption of in-order delivery. A sequence of encoded header blocks must arrive (and be decoded) at an endpoint in the same order in which they were encoded. This ensures that the dynamic state at the two endpoints remains in sync. As a result, HTTP/QUIC uses a modified version of HPACK, described in [QPACK].

Frame type definitions in HTTP/QUIC often use the QUIC variable-length integer encoding. In particular, Stream IDs use this encoding, which allow for a larger range of possible values than the encoding used in HTTP/2. Some frames in HTTP/QUIC use an identifier rather than a Stream ID (e.g. Push IDs in PRIORITY frames). Redefinition of the encoding of extension frame types might be necessary if the encoding includes a Stream ID.

Other than this issue, frame type HTTP/2 extensions are typically portable to QUIC simply by replacing Stream 0 in HTTP/2 with Stream 2 or 3 in HTTP/QUIC. HTTP/QUIC extensions will not assume ordering, but would not be harmed by ordering, and would be portable to HTTP/2 in the same manner.

Below is a listing of how each HTTP/2 frame type is mapped:

DATA (0x0): Padding is not defined in HTTP/QUIC frames. See Section 4.2.1.

HEADERS (0x1): As described above, the PRIORITY region of HEADERS is not supported. A separate PRIORITY frame MUST be used. Padding is not defined in HTTP/QUIC frames. See Section 4.2.2.

PRIORITY (0x2): As described above, the PRIORITY frame is sent on the control stream and can reference either a Stream ID or a Push ID. See Section 4.2.3.

RST_STREAM (0x3): RST_STREAM frames do not exist, since QUIC provides stream lifecycle management. The same code point is used for the CANCEL_PUSH frame (Section 4.2.4).

SETTINGS (0x4): SETTINGS frames are sent only at the beginning of the connection. See Section 4.2.5 and Section 7.3.

PUSH_PROMISE (0x5): The PUSH_PROMISE does not reference a stream; instead the push stream references the PUSH_PROMISE frame using a Push ID. See Section 4.2.6.

PING (0x6): PING frames do not exist, since QUIC provides equivalent functionality.

GOAWAY (0x7): GOAWAY is sent only from server to client and does not contain an error code. See Section 4.2.7.

WINDOW_UPDATE (0x8): WINDOW_UPDATE frames do not exist, since QUIC provides flow control.

CONTINUATION (0x9): CONTINUATION frames do not exist; instead, larger HEADERS/PUSH_PROMISE frames than HTTP/2 are permitted, and HEADERS frames can be used in series.

Frame types defined by extensions to HTTP/2 need to be separately registered for HTTP/QUIC if still applicable. The IDs of frames defined in [RFC7540] have been reserved for simplicity. See Section 9.3.

7.3. HTTP/2 SETTINGS Parameters

An important difference from HTTP/2 is that settings are sent once, at the beginning of the connection, and thereafter cannot change. This eliminates many corner cases around synchronization of changes.

Some transport-level options that HTTP/2 specifies via the SETTINGS frame are superseded by QUIC transport parameters in HTTP/QUIC. The HTTP-level options that are retained in HTTP/QUIC have the same value as in HTTP/2.

Below is a listing of how each HTTP/2 SETTINGS parameter is mapped:

SETTINGS_HEADER_TABLE_SIZE: See Section 4.2.5.2.

SETTINGS_ENABLE_PUSH: This is removed in favor of the MAX_PUSH_ID which provides a more granular control over server push.

SETTINGS_MAX_CONCURRENT_STREAMS: QUIC controls the largest open Stream ID as part of its flow control logic. Specifying SETTINGS_MAX_CONCURRENT_STREAMS in the SETTINGS frame is an error.

SETTINGS_INITIAL_WINDOW_SIZE: QUIC requires both stream and connection flow control window sizes to be specified in the initial transport handshake. Specifying SETTINGS_INITIAL_WINDOW_SIZE in the SETTINGS frame is an error.

SETTINGS_MAX_FRAME_SIZE: This setting has no equivalent in HTTP/QUIC. Specifying it in the SETTINGS frame is an error.

SETTINGS_MAX_HEADER_LIST_SIZE: See Section 4.2.5.2.

Settings need to be defined separately for HTTP/2 and HTTP/QUIC. The IDs of settings defined in [RFC7540] have been reserved for simplicity. See Section 9.4.

7.4. HTTP/2 Error Codes

QUIC has the same concepts of "stream" and "connection" errors that HTTP/2 provides. However, because the error code space is shared between multiple components, there is no direct portability of HTTP/2 error codes.

The HTTP/2 error codes defined in Section 7 of [RFC7540] map to the HTTP over QUIC error codes as follows:

NO_ERROR (0x0): HTTP_NO_ERROR in Section 6.1.

PROTOCOL_ERROR (0x1): No single mapping. See new HTTP_MALFORMED_FRAME error codes defined in Section 6.1.

INTERNAL_ERROR (0x2): HTTP_INTERNAL_ERROR in Section 6.1.

FLOW_CONTROL_ERROR (0x3): Not applicable, since QUIC handles flow control. Would provoke a QUIC_FLOW_CONTROL_RECEIVED_TOO_MUCH_DATA from the QUIC layer.

SETTINGS_TIMEOUT (0x4): Not applicable, since no acknowledgement of SETTINGS is defined.

STREAM_CLOSED (0x5): Not applicable, since QUIC handles stream management. Would provoke a QUIC_STREAM_DATA_AFTER_TERMINATION from the QUIC layer.

FRAME_SIZE_ERROR (0x6) No single mapping. See new error codes defined in Section 6.1.

REFUSED_STREAM (0x7): Not applicable, since QUIC handles stream management. Would provoke a QUIC_TOO_MANY_OPEN_STREAMS from the QUIC layer.

CANCEL (0x8): HTTP_REQUEST_CANCELLED in Section 6.1.

COMPRESSION_ERROR (0x9): HTTP_HPACK_DECOMPRESSION_FAILED in Section 6.1.

CONNECT_ERROR (0xa): HTTP_CONNECT_ERROR in Section 6.1.

ENHANCE_YOUR_CALM (0xb): HTTP_EXCESSIVE_LOAD in Section 6.1.

INADEQUATE_SECURITY (0xc): Not applicable, since QUIC is assumed to provide sufficient security on all connections.

HTTP_1_1_REQUIRED (0xd): HTTP_VERSION_FALLBACK in Section 6.1.

Error codes need to be defined for HTTP/2 and HTTP/QUIC separately. See Section 9.5.

8. Security Considerations

The security considerations of HTTP over QUIC should be comparable to those of HTTP/2.

The modified SETTINGS format contains nested length elements, which could pose a security risk to an uncautious implementer. A SETTINGS frame parser MUST ensure that the length of the frame exactly matches the length of the settings it contains.

9. IANA Considerations

9.1. Registration of HTTP/QUIC Identification String

This document creates a new registration for the identification of HTTP/QUIC in the "Application Layer Protocol Negotiation (ALPN) Protocol IDs" registry established in [RFC7301].

The "hq" string identifies HTTP/QUIC:

Protocol: HTTP over QUIC

Identification Sequence: 0x68 0x71 ("hq")

Specification: This document

9.2. Registration of QUIC Version Hint Alt-Svc Parameter

This document creates a new registration for version-negotiation hints in the "Hypertext Transfer Protocol (HTTP) Alt-Svc Parameter" registry established in [RFC7838].

Parameter: "quic"

Specification: This document, Section 2.1.1

9.3. Frame Types

This document establishes a registry for HTTP/QUIC frame type codes. The "HTTP/QUIC Frame Type" registry manages an 8-bit space. The "HTTP/QUIC Frame Type" registry operates under either of the "IETF Review" or "IESG Approval" policies [RFC8126] for values between 0x00 and 0xef, with values between 0xf0 and 0xff being reserved for Experimental Use.

While this registry is separate from the "HTTP/2 Frame Type" registry defined in [RFC7540], it is preferable that the assignments parallel each other. If an entry is present in only one registry, every effort SHOULD be made to avoid assigning the corresponding value to an unrelated operation.

New entries in this registry require the following information:

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

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

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

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

Frame Type	Code	Specification
DATA	0x0	Section 4.2.1
HEADERS	0x1	Section 4.2.2
PRIORITY	0x2	Section 4.2.3
CANCEL_PUSH	0x3	Section 4.2.4
SETTINGS	0x4	Section 4.2.5
PUSH_PROMISE	0x5	Section 4.2.6
Reserved	0x6	N/A
GOAWAY	0x7	Section 4.2.7
HEADER_ACK	0x8	Section 4.2.8
Reserved	0x9	N/A
MAX_PUSH_ID	0xD	Section 4.2.9

9.4. Settings Parameters

This document establishes a registry for HTTP/QUIC settings. The "HTTP/QUIC Settings" registry manages a 16-bit space. The "HTTP/QUIC Settings" registry operates under the "Expert Review" policy [RFC8126] for values in the range from 0x0000 to 0xefff, with values between and 0xf000 and 0xffff being reserved for Experimental Use. The designated experts are the same as those for the "HTTP/2 Settings" registry defined in [RFC7540].

While this registry is separate from the "HTTP/2 Settings" registry defined in [RFC7540], it is preferable that the assignments parallel each other. If an entry is present in only one registry, every effort SHOULD be made to avoid assigning the corresponding value to an unrelated operation.

New registrations are advised to provide the following information:

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

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

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

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

Setting Name	Code	Specification
HEADER_TABLE_SIZE	0x1	Section 4.2.5.2
Reserved	0x2	N/A
Reserved	0x3	N/A
Reserved	0x4	N/A
Reserved	0x5	N/A
MAX_HEADER_LIST_SIZE	0x6	Section 4.2.5.2

9.5. Error Codes

This document establishes a registry for HTTP/QUIC error codes. The "HTTP/QUIC Error Code" registry manages a 16-bit space. The "HTTP/QUIC Error Code" registry operates under the "Expert Review" policy [RFC8126].

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

New registrations are advised to provide the following information:

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

Code: The 16-bit error code value.

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

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

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

Name	Code	Description	Specification
STOPPING	0x0000	Reserved by QUIC	[QUIC-TRANSPORT]
HTTP_NO_ERROR	0x0001	No error	Section 6.1
HTTP_PUSH_REFUSED	0x0002	Client refused pushed content	Section 6.1
HTTP_INTERNAL_ERROR	0x0003	Internal error	Section 6.1
HTTP_PUSH_ALREADY_IN_CACHE	0x0004	Pushed content already cached	Section 6.1
HTTP_REQUEST_CANCELLED	0x0005	Data no longer needed	Section 6.1
HTTP_HPACK_DECOMPRESSION_FAILED	0x0006	HPACK cannot continue	Section 6.1
HTTP_CONNECT_ERROR	0x0007	TCP reset or error on CONNECT request	Section 6.1
HTTP_EXCESSIVE_LOAD	0x0008	Peer generating excessive load	Section 6.1
HTTP_VERSION_FALLBACK	0x0009	Retry over HTTP/2	Section 6.1
HTTP_WRONG_STREAM	0x000A	A frame was sent on the wrong stream	Section 6.1

HTTP_PUSH_LIMIT_EXCEEDED	0x000B	Maximum Push ID exceeded	Section 6.1
HTTP_DUPLICATE_PUSH	0x000C	Push ID was fulfilled multiple times	Section 6.1
HTTP_MALFORMED_FRAME	0x01XX	Error in frame formatting or use	Section 6.1

10. References

10.1. Normative References

[QPACK] Krasic, C., Bishop, M., and A. Frindell, Ed., "QPACK: Header Compression for HTTP over QUIC", draft-ietf-quic-qpack-00 (work in progress), April 2018.

[QUIC-TRANSPORT] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-11 (work in progress), April 2018.

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <https://www.rfc-editor.org/info/rfc793>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <https://www.rfc-editor.org/info/rfc6066>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <https://www.rfc-editor.org/info/rfc7230>.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <https://www.rfc-editor.org/info/rfc7231>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <https://www.rfc-editor.org/info/rfc7540>.

[RFC7838] Nottingham, M., McManus, P., and J. Reschke, "HTTP Alternative Services", RFC 7838, DOI 10.17487/RFC7838, April 2016, <https://www.rfc-editor.org/info/rfc7838>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

10.2. Informative References

[RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <https://www.rfc-editor.org/info/rfc7301>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

10.3. URIs

[1] https://mailarchive.ietf.org/arch/search/?email_list=quic

[2] https://github.com/quicwg

[3] https://github.com/quicwg/base-drafts/labels/-http

Appendix A. Contributors

The original authors of this specification were Robbie Shade and Mike Warres.

A substantial portion of Mike's contribution was supported by Microsoft during his employment there.

Appendix B. Change Log

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

B.1. Since draft-ietf-quic-http-10

- Settings need to be remembered when attempting and accepting 0-RTT (#1157, #1207)

B.2. Since draft-ietf-quic-http-09

- Selected QCRAM for header compression (#228, #1117)
- The server_name TLS extension is now mandatory (#296, #495)
- Specified handling of unsupported versions in Alt-Svc (#1093, #1097)

B.3. Since draft-ietf-quic-http-08

- Clarified connection coalescing rules (#940, #1024)

B.4. Since draft-ietf-quic-http-07

- Changes for integer encodings in QUIC (#595,#905)
- Use unidirectional streams as appropriate (#515, #240, #281, #886)
- Improvement to the description of GOAWAY (#604, #898)
- Improve description of server push usage (#947, #950, #957)

B.5. Since draft-ietf-quic-http-06

- Track changes in QUIC error code usage (#485)

B.6. Since draft-ietf-quic-http-05

- Made push ID sequential, add MAX_PUSH_ID, remove SETTINGS_ENABLE_PUSH (#709)
- Guidance about keep-alive and QUIC PINGs (#729)
- Expanded text on GOAWAY and cancellation (#757)

B.7. Since draft-ietf-quic-http-04

- Cite RFC 5234 (#404)
- Return to a single stream per request (#245,#557)
- Use separate frame type and settings registries from HTTP/2 (#81)
- SETTINGS_ENABLE_PUSH instead of SETTINGS_DISABLE_PUSH (#477)
- Restored GOAWAY (#696)
- Identify server push using Push ID rather than a stream ID (#702,#281)
- DATA frames cannot be empty (#700)

B.8. Since draft-ietf-quic-http-03

None.

B.9. Since draft-ietf-quic-http-02

- Track changes in transport draft

B.10. Since draft-ietf-quic-http-01

- SETTINGS changes (#181):
  - SETTINGS can be sent only once at the start of a connection; no changes thereafter
  - SETTINGS_ACK removed
  - Settings can only occur in the SETTINGS frame a single time
  - Boolean format updated
- Alt-Svc parameter changed from "v" to "quic"; format updated (#229)
- Closing the connection control stream or any message control stream is a fatal error (#176)
- HPACK Sequence counter can wrap (#173)
- 0-RTT guidance added

- Guide to differences from HTTP/2 and porting HTTP/2 extensions added (#127,#242)

B.11. Since draft-ietf-quic-http-00

- Changed "HTTP/2-over-QUIC" to "HTTP/QUIC" throughout (#11,#29)
- Changed from using HTTP/2 framing within Stream 3 to new framing format and two-stream-per-request model (#71,#72,#73)
- Adopted SETTINGS format from draft-bishop-httpbis-extended-settings-01
- Reworked SETTINGS_ACK to account for indeterminate inter-stream order (#75)
- Described CONNECT pseudo-method (#95)
- Updated ALPN token and Alt-Svc guidance (#13,#87)
- Application-layer-defined error codes (#19,#74)

B.12. Since draft-shade-quic-http2-mapping-00

- Adopted as base for draft-ietf-quic-http
- Updated authors/editors list

Author’s Address

Mike Bishop (editor)
Akamai

Email: mbishop@evequefou.be

Network Working Group
Internet-Draft
Intended status: Informational
Expires: August 5, 2017

M. Nottingham
February 1, 2017

Retrying HTTP Requests
draft-nottingham-httpbis-retry-01

Abstract

HTTP allows requests to be automatically retried under certain circumstances. This draft explores how this is implemented, requirements for similar functionality from other parts of the stack, and potential future improvements.

Note to Readers

This draft is not intended to be published as an RFC.

The issues list for this draft can be found at https://github.com/mnot/I-D/labels/httpbis-retry .

The most recent (often, unpublished) draft is at https://mnot.github.io/I-D/httpbis-retry/ .

Recent changes are listed at https://github.com/mnot/I-D/commits/gh-pages/httpbis-retry .

Status of This Memo

Table of Contents

1. Introduction

One of the benefits of HTTP's well-defined method semantics is that they allow failed requests to be retried, under certain circumstances.

However, interest in extending, redefining or just clarifying HTTP's retry semantics is increasing, for a number of reasons:

- Since HTTP/1.1's requirements were written, there has been a substantial amount of experience deploying and using HTTP, leading implementations to refine their behaviour, often diverging from the specification.

- Likewise, changes such as HTTP/2 [RFC7540] might change the underlying assumptions that these requirements were based upon.

- Emerging lower-layer developments such as TCP Fast Open [RFC7413], TLS/1.3 [I-D.ietf-tls-tls13] and QUIC [I-D.ietf-quic-transport] introduce the possibility of replayed requests in the beginning of a connection, thanks to Zero Round Trip (0RT) modes. In some ways, these are similar to retries - but not completely.

- Applications sometimes want requests to be retried by infrastructure, but can't easily express them in a non-idempotent request (such as GET).

This draft gives some background in Section 2, discusses aspects of these issues in Section 3, suggesting possible areas of work in Section 4, and cataloguing current implementation behaviours in Appendix A.

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. Background

2.1. Retries and Replays: A Taxonomy of Repetition

In HTTP, there are three similar but separate phenomena that deserve consideration for the purposes of this document:

1. *User Retries* happen when a user initiates an action that results in a duplicate HTTP request message being emitted. For example, a user retry might occur when a "reload" button is

pressed, a URL is typed in again, "return" is pressed in the URL bar again, or a navigation link or form button is pressed twice while still on screen.

2. *Automatic Retries* happen when an HTTP client implementation resends a previous request message without user intervention or initiation. This might happen when a GET request fails to return a complete response, or when a connection drops before the request is sent. Note that automatic retries can (and are) performed both by user agents and intermediary clients.

3. *Replays* happen when the underlying transport units (e.g., TCP packets, QUIC frames) containing a HTTP request message are re-sent on the network *and* appear to be separate requests to the downstream server, either automatically as part of transport protocol operation, or by an attacker. The upstream HTTP client might not have any indication that a replay has occurred.

Note that retries initiated by code shipped to the client by the server (e.g., in JavaScript) occupy a grey area here. Because they are not initiated by the generic HTTP client implementation itself, we will consider them user retries for the time being.

Also, this document doesn't include transport layer loss recovery (e.g., TCP retransmission). This is distinguished from replays because the transport automatically suppresses duplicates.

2.2. What the Spec Says: Automatic Retries

[RFC7230], Section 6.3.1 allows HTTP requests to be retried in certain circumstances:

> When an inbound connection is closed prematurely, a client MAY open a new connection and automatically retransmit an aborted sequence of requests if all of those requests have idempotent methods (Section 4.2.2 of [RFC7231]). A proxy MUST NOT automatically retry non-idempotent requests.
>
> A user agent MUST NOT automatically retry a request with a non-idempotent method unless it has some means to know that the request semantics are actually idempotent, regardless of the method, or some means to detect that the original request was never applied. For example, a user agent that knows (through design or configuration) that a POST request to a given resource is safe can repeat that request automatically. Likewise, a user agent designed specifically to operate on a version control repository might be able to recover from partial failure conditions by checking the target resource revision(s) after a

> failed connection, reverting or fixing any changes that were partially applied, and then automatically retrying the requests that failed.
>
> A client SHOULD NOT automatically retry a failed automatic retry.

Note that the complete list of idempotent methods is maintained in the IANA HTTP Method Registry [4].

2.3. What the Specs Say: Replay

2.3.1. TCP Fast Open

[RFC7413], Section 6.3.1 addresses HTTP Request Replay with TCP Fast Open:

> While TFO is motivated by Web applications, the browser should not use TFO to send requests in SYNs if those requests cannot tolerate replays. One example is POST requests without application-layer transaction protection (e.g., a unique identifier in the request header).
>
> On the other hand, TFO is particularly useful for GET requests. GET request replay could happen across striped TCP connections: after a server receives an HTTP request but before the ACKs of the requests reach the browser, the browser may time out and retry the same request on another (possibly new) TCP connection. This differs from a TFO replay only in that the replay is initiated by the browser, not by the TCP stack.

The same specification addresses HTTP over TLS in Section 6.3.2:

> For Transport Layer Security (TLS) over TCP, it is safe and useful to include a TLS client_hello in the SYN packet to save one RTT in the TLS handshake. There is no concern about violating idempotency. In particular, it can be used alone with the speculative connection above.

2.3.2. TLS 1.3

[I-D.ietf-tls-tls13], Section 2.3 explains the properties of Zero-RTT Data in TLS 1.3:

> IMPORTANT NOTE: The security properties for 0-RTT data (regardless of the cipher suite) are weaker than those for other kinds of TLS data. Specifically:

1. This data is not forward secret, because it is encrypted solely with the PSK.

2. There are no guarantees of non-replay between connections. Unless the server takes special measures outside those provided by TLS, the server has no guarantee that the same 0-RTT data was not transmitted on multiple 0-RTT connections (See Section 4.2.6.2 for more details). This is especially relevant if the data is authenticated either with TLS client authentication or inside the application layer protocol. However, 0-RTT data cannot be duplicated within a connection (i.e., the server will not process the same data twice for the same connection) and an attacker will not be able to make 0-RTT data appear to be 1-RTT data (because it is protected with different keys.)

Section 4.2.6 defines a mechanism to limit the exposure to replay.

2.3.3. QUIC

[I-D.ietf-quic-tls] Section 7.2 says this about the risks of replay during the 0RTT handshake:

> If 0-RTT keys are available, the lack of replay protection means that restrictions on their use are necessary to avoid replay attacks on the protocol.
>
> A client MUST only use 0-RTT keys to protect data that is idempotent. A client MAY wish to apply additional restrictions on what data it sends prior to the completion of the TLS handshake. A client otherwise treats 0-RTT keys as equivalent to 1-RTT keys.
>
> A client that receives an indication that its 0-RTT data has been accepted by a server can send 0-RTT data until it receives all of the server's handshake messages. A client SHOULD stop sending 0-RTT data if it receives an indication that 0-RTT data has been rejected.
>
> A server MUST NOT use 0-RTT keys to protect packets.

3. Discussion

3.1. Automatic Retries In Practice

In practice, it has been observed (see Appendix A) that some client implementations (both user agent and intermediary) do automatically retry requests. However, they do not do so consistently, and

arguably not in the spirit of the specification, unless this vague catch-all:

> some means to detect that the original request was never applied

is interpreted very broadly.

On the server side, it has been widely observed that content on the Web doesn't always honour HTTP idemotency semantics, with many GET requests incurring side effects, and with some sites even requiring browsers to retry POST requests in order to properly interoperate.

Despite this situation, the Web seems to work reasonably well to date (with notable exceptions [5]).

The status quo, therefore, is that no Web application can read HTTP's retry requirements as a guarantee that any given request won't be retried, even for methods that are not idempotent. As a result, applications that care about avoiding duplicate requests need to build a way to detect not only user retries but also automatic retries into the application "above" HTTP itself.

3.2. Replays Are Different

TCP Fast Open [RFC7413], TLS/1.3 [I-D.ietf-tls-tls13] and QUIC [I-D.ietf-quic-transport] all have mechanisms to carry application data on the first packet sent by a client, to avoid the latency of connection setup.

The request(s) in this first packet might be _replayed_, either because the first packet (now carrying a HTTP request) is thought to be lost and retransmitted, or because an attacker observes the packet and sends a duplicate at some point in the future.

At first glance, it seems as if the idempotency semantics of HTTP request methods could be used to determine what requests are suitable for inclusion in the first packet of various 0RTT mechanisms being discussed (as suggested by TCP Fast Open). For example, we could disallow POST (and other non-idempotent methods) in 0RTT data.

Upon reflection, though, the observations above lead us to believe that since any request might be retried (automatically or by users), applications will still need to have a means of detecting duplicate requests, thereby preventing side effects from replays as well as retries. Thus, any HTTP request can be included in the first packet of a 0RTT, despite the risk of replay.

Two types of attack specific to replayed HTTP requests need to be taken into account, however:

1. A replay is a potential Denial of Service vector. An attacker that can replay a request many times can probe for weaknesses in retry protections, and can bring a server that needs to do any substantial processing down.

2. An attacker might use a replayed request to leak information about the response over time. If they can observe the encrypted payload on the wire, they can infer the size of the response (e.g., it might get bigger if the user's bank account has more in it).

The first attack cannot be mitigated by HTTP; the 0RT mechanism itself needs some transport-layer means of scoping the usability of the first packet on a connection so that it cannot be reused broadly. For example, this might be by time, or by network location.

The second attack is more difficult to mitigate; scoping the usability of the first packet helps, but does not completely prevent the attack. If the replayed request is state-changing, the application's retry detection should kick in and prevent information leakage (since the response will likely contain an error, instead of the desired information).

If it is not (e.g., a GET), the information being targeted is vulnerable as long as both the first packet and the credentials in the request (if any) are valid.

4. Possible Areas of Work

4.1. Updating HTTP's Requirements for Retries

The currently language in [RFC7230] about retries is vague about the conditions under which a request can be retried, leading to significant variance in implementation behaviour. For example, it's been observed that many automated clients fail under circumstances when browsers succeed, because they do not retry in the same way.

As a result, more carefully specifying the conditions under which a request can be retried would be helpful. Such work would need to take into account varying conditions, such as:

- Connection closes
- TCP RST

- Connection timeouts
- Whether or not any part of the response has been received
- Whether or not it is the first request on the connection
- Variance due to use of HTTP/2, TLS/1.3, TCP Fast Open and QUIC.

Furthermore, readers might mistake the language in RFC7230 as guaranteeing that some requests (e.g., POST) are never automatically retried; this should be clarified.

4.2. Protocol Extensions

A number of mechanisms have been mooted at various times, e.g.:

- Adding a header to automatically retried requests, to aid de-duplication by servers
- Defining a request header to by added by intermediaries when they have received a request in a way that could have been replayed
- Defining a status code to allow servers to indicate that the request needs to be sent in a way that can’t be replayed

4.3. Feedback to Transport 0RTT Efforts

If the observations above hold, we should disabuse any notion that HTTP method idempotency is a useful way to avoid problems with replay attacks. Instead, we should encourage development of mechanisms to mitigate the aspects of replay that are different than retries (e.g., potential for DOS attacks).

5. Security Considerations

Yep.

6. Acknowledgements

Thanks to Brad Fitzpatrick, Leif Hedstrom, Subodh Iyengar, Amos Jeffries, Patrick McManus, Matt Menke, Miroslav Ponec, Daniel Stenberg and Martin Thomson for their input and feedback.

Thanks also to the participants in the 2016 HTTP Workshop for their lively discussion of this topic.

7. References

7.1. Normative References

[I-D.ietf-quic-tls]
Thomson, M. and (. (Unknown), "Using Transport Layer Security (TLS) to Secure QUIC", draft-ietf-quic-tls-01 (work in progress), January 2017.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <http://www.rfc-editor.org/info/rfc7230>.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <http://www.rfc-editor.org/info/rfc7231>.

[RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <http://www.rfc-editor.org/info/rfc7413>.

7.2. Informative References

[I-D.ietf-quic-transport]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-01 (work in progress), January 2017.

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-18 (work in progress), October 2016.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <http://www.rfc-editor.org/info/rfc7540>.

7.3. URIs

[1] https://www.iana.org/assignments/http-methods/http-methods.xhtml

[2] https://signalvnoise.com/archives2/google_web_accelerator_hey_not_so_fast_an_alert_for_web_app_designers.php

[3] http://bazaar.launchpad.net/~squid/squid/trunk/view/head:/src/FwdState.cc#L594

[4] https://git-wip-us.apache.org/repos/asf?p=trafficserver.git;a=blob;f=proxy/http/HttpTransact.cc;h=8a1f5364d47654b118296a07a2a95284f119d84b;hb=HEAD#l6408

[5] https://git-wip-us.apache.org/repos/asf?p=trafficserver.git;a=blob;f=proxy/http/HttpTransact.cc;hb=48d7b25ba8a8229b0471d37cdaa6ef24cc634bb0#l3634

[6] http://mxr.mozilla.org/mozilla-release/source/netwerk/protocol/http/nsHttpTransaction.cpp#938

[7] http://mxr.mozilla.org/mozilla-release/source/netwerk/protocol/http/nsHttpRequestHead.cpp#67

[8] https://www.fxsitecompat.com/en-CA/docs/2016/post-request-fails-on-certain-sites-showing-connection-reset-page/

[9] https://chromium.googlesource.com/chromium/src.git/+/master/net/http/http_network_transaction.cc#1657

[10] https://github.com/curl/curl/blob/master/lib/transfer.c#L1892

Appendix A. When Clients Retry

In implementations, clients have been observed to retry requests in a number of circumstances.

Note: This section is intended to inform the discussion, not to be published as a standard. If you have relevant information about these or other implementations (open or closed), please get in touch.

A.1. Squid

Squid is a caching proxy server that retries requests that it considers safe *or* idempotent, as long as there is not a request body:

```
/// Whether we may try sending this request again after a failure.
bool
FwdState::checkRetriable()
{
    // Optimize: A compliant proxy may retry PUTs, but Squid lacks the [rather
    // complicated] code required to protect the PUT request body from being
    // nibbled during the first try. Thus, Squid cannot retry some PUTs today.
    if (request->body_pipe != NULL)
        return false;

    // RFC2616 9.1 Safe and Idempotent Methods
    return (request->method.isHttpSafe() || request->method.isIdempotent());
}
```

(source [6])

Currently, it considers GET, HEAD, OPTIONS, REPORT, PROPFIND, SEARCH and PRI to be safe, and GET, HEAD, PUT, DELETE, OPTIONS, TRACE, PROPFIND, PROPPATCH, MKCOL, COPY, MOVE, UNLOCK, and PRI to be idempotent.

A.2. Traffic Server

Apache Traffic Server, a caching proxy server, ties retry-ability to whether the request required a "tunnel" - i.e., forwarding the request body to the next server. This is indicated by "request_body_start", which is set when a POST tunnel is used.

```
// bool HttpTransact::is_request_retryable
//
//   If we started a POST/PUT tunnel then we can
//    not retry failed requests
//
bool
HttpTransact::is_request_retryable(State *s)
{
  if (s->hdr_info.request_body_start == true) {
    return false;
  }

  if (s->state_machine->plugin_tunnel_type != HTTP_NO_PLUGIN_TUNNEL) {
    // API can override
    if (s->state_machine->plugin_tunnel_type == HTTP_PLUGIN_AS_SERVER &&
        s->api_info.retry_intercept_failures == true) {
      // This used to be an == comparison, which made no sense. Changed
      // to be an assignment, hoping the state is correct.
      s->state_machine->plugin_tunnel_type = HTTP_NO_PLUGIN_TUNNEL;
    } else {
      return false;
    }
  }

  return true;
}
```

(source [7])

When connected to an origin server, Traffic Server attempts to retry under a number of failure conditions:

```
////////////////////////////////////////////////////////////////////////
// Name       : handle_response_from_server
// Description: response is from the origin server
//
// Details    :
//
//   response from the origin server. one of three things can happen now.
//   if the response is bad, then we can either retry (by first downgrading
//   the request, maybe making it non-keepalive, etc.), or we can give up.
//   the latter case is handled by handle_server_connection_not_open and
//   sends an error response back to the client. if the response is good
//   handle_forward_server_connection_open is called.
//
//
// Possible Next States From Here:
//
```

```
//////////////////////////////////////////////////////////////////////
void
HttpTransact::handle_response_from_server(State *s)
{

[...]

  switch (s->current.state) {
  case CONNECTION_ALIVE:
    DebugTxn("http_trans", "[hrfs] connection alive");
    SET_VIA_STRING(VIA_DETAIL_SERVER_CONNECT, VIA_DETAIL_SERVER_SUCCESS);
    s->current.server->clear_connect_fail();
    handle_forward_server_connection_open(s);
    break;

[...]

  case OPEN_RAW_ERROR:
  /* fall through */
  case CONNECTION_ERROR:
  /* fall through */
  case STATE_UNDEFINED:
  /* fall through */
  case INACTIVE_TIMEOUT:
    // Set to generic I/O error if not already set specifically.
    if (!s->current.server->had_connect_fail())
      s->current.server->set_connect_fail(EIO);

    if (is_server_negative_cached(s)) {
      max_connect_retries = s->txn_conf->connect_attempts_max_retries_dead_serve
r;
    } else {
      // server not yet negative cached - use default number of retries
      max_connect_retries = s->txn_conf->connect_attempts_max_retries;
    }
    if (s->pCongestionEntry != NULL)
      max_connect_retries = s->pCongestionEntry->connect_retries();

    if (is_request_retryable(s) && s->current.attempts < max_connect_retries) {
```

(source [8])

A.3. Firefox

Firefox is a Web browser that retries under the following conditions:

```
// if the connection was reset or closed before we wrote any part of the
// request or if we wrote the request but didn't receive any part of the
// response and the connection was being reused, then we can (and really
// should) assume that we wrote to a stale connection and we must therefore
// repeat the request over a new connection.
//
// We have decided to retry not only in case of the reused connections, but
// all safe methods(bug 1236277).
//
// NOTE: the conditions under which we will automatically retry the HTTP
// request have to be carefully selected to avoid duplication of the
// request from the point-of-view of the server.  such duplication could
// have dire consequences including repeated purchases, etc.
//
// NOTE: because of the way SSL proxy CONNECT is implemented, it is
// possible that the transaction may have received data without having
// sent any data.  for this reason, mSendData == FALSE does not imply
// mReceivedData == FALSE.  (see bug 203057 for more info.)
//

[...]

   if (!mReceivedData &&
       ((mRequestHead && mRequestHead->IsSafeMethod()) ||
        !reallySentData || connReused)) {
       // if restarting fails, then we must proceed to close the pipe,
       // which will notify the channel that the transaction failed.
```

(source [9])

... and it considers GET, HEAD, OPTIONS, TRACE, PROPFIND, REPORT, and SEARCH to be safe:

```
 bool
 nsHttpRequestHead::IsSafeMethod() const
 {
   // This code will need to be extended for new safe methods, otherwise
   // they'll default to "not safe".
     if (IsGet() || IsHead() || IsOptions() || IsTrace()) {
         return true;
     }

     if (mParsedMethod != kMethod_Custom) {
         return false;
     }

     return (!strcmp(mMethod.get(), "PROPFIND") ||
             !strcmp(mMethod.get(), "REPORT") ||
             !strcmp(mMethod.get(), "SEARCH"));
 }
```

(source [10])

Note that "connReused" is tested; if a connection has been used before, Firefox will retry _any_ request, safe or not. A recent change attempted to remove this behaviour, but it caused compatibility problems [11], and is being backed out.

A.4. Chromium

Chromium is a Web browser that appears to retry any request when a connection is broken, as long as it's successfully used the connection before, and hasn't received any response headers yet:

```
bool HttpNetworkTransaction::ShouldResendRequest() const {
  bool connection_is_proven = stream_->IsConnectionReused();
  bool has_received_headers = GetResponseHeaders() != NULL;

  // NOTE: we resend a request only if we reused a keep-alive connection.
  // This automatically prevents an infinite resend loop because we'll run
  // out of the cached keep-alive connections eventually.
  if (connection_is_proven && !has_received_headers)
    return true;
  return false;
}
```

(source [12])

A.5. Curl

Curl is both a command-line client and widely-used library for HTTP. Like Chromium, it will retry a request if the response hasn't started.

```
CURLcode Curl_retry_request(struct connectdata *conn,
                            char **url)
{
  struct Curl_easy *data = conn->data;

  *url = NULL;

  /* if we're talking upload, we can't do the checks below, unless the protocol
     is HTTP as when uploading over HTTP we will still get a response */
  if(data->set.upload &&
     !(conn->handler->protocol&(PROTO_FAMILY_HTTP|CURLPROTO_RTSP)))
    return CURLE_OK;

  if((data->req.bytecount + data->req.headerbytecount == 0) &&
     conn->bits.reuse &&
     (data->set.rtspreq != RTSPREQ_RECEIVE)) {
    /* We didn't get a single byte when we attempted to re-use a
       connection. This might happen if the connection was left alive when we
       were done using it before, but that was closed when we wanted to use it
       again. Bad luck. Retry the same request on a fresh connect! */
    infof(conn->data, "Connection died, retrying a fresh connect\n");
    *url = strdup(conn->data->change.url);
    if(!*url)
      return CURLE_OUT_OF_MEMORY;

    connclose(conn, "retry"); /* close this connection */
    conn->bits.retry = TRUE; /* mark this as a connection we're about
                                to retry. Marking it this way should
                                prevent i.e HTTP transfers to return
                                error just because nothing has been
                                transferred! */


    if(conn->handler->protocol&PROTO_FAMILY_HTTP) {
      struct HTTP *http = data->req.protop;
      if(http->writebytecount)
        return Curl_readrewind(conn);
    }
  }
  return CURLE_OK;
}
```

(source [13])

Author's Address

Mark Nottingham

Email: mnot@mnot.net
URI: https://www.mnot.net/