HTTP Working Group
Internet-Draft
Intended status: Informational
Expires: April 30, 2018

S. Sahib

October 27, 2017

New protocol elements for HTTP Status Code 451
draft-451-new-protocol-elements-01

Abstract

This draft recommends protocol updates to Hypertext Transfer Protocol (HTTP) status code 451 (defined by RFC7725) based on an examination of how the new status code is being used by parties involved in denial of Internet resources because of legal demands.

Discussion of this draft is at https://www.irtf.org/mailman/listinfo/hrpc and https://lists.ghserv.net/mailman/listinfo/statuscode451.

Table of Contents

1. Introduction

[RFC7725] was standardized by the IETF in February 2016. It defined HTTP status code 451 - to be used when a "a server operator has received a legal demand to deny access to a resource or to a set of resources that includes the requested resource". The intention was to provide a uniform mechanism to indicate online censorship.

Subsequently, an effort was made to investigate usage of 451 status code and evaluate if it fulfills its mandate of providing "transparency in circumstances where issues of law or public policy affect server operations" [IMPL_REPORT_DRAFT]. This draft attempts to explicate the protocol recommendations arising out of that investigation.

2. Requirements

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].

3. Existing Protocol Elements

The status code as standardized by the IETF specifies the following elements [RFC7725] -

- A server can return status code 451 to indicate that it is denying access to a resource or multiple resources on account of a legal demand.

- Responses using the status code SHOULD include an explanation in the response body of the details of the legal demand.

- Responses SHOULD include a "Link" HTTP header field [RFC8288] whose value is a URI reference [RFC3986] identifying itself. The "Link" header field MUST have a "rel" parameter whose value is "blocked-by". The intent is that the header be used to identify the entity actually implementing blockage, not any other entity mandating it.

4. Recommendations

- In addition to the "blocked-by" header, an HTTP response with status code 451 SHOULD include another "Link" HTTP header field which has a "rel" parameter whose value is "blocking-authority". It’s important to distinguish between the implementer of the block, and the authority that mandated the block in the first place. This is because these two organizations might not be the same - a government (the blocking authority) could force an Internet Service Provider (the implementer of the block) to deny access to a certain resource.

- HTTP status code 451 is increasingly being used to deny access to resources based on geographical IP. The scope of this denial is sometimes as finely scoped as a city or a province. The response SHOULD contain a provisional header with geographical scope of block.

5. Security Considerations

This document does not add additional security considerations to [RFC7725].

6. IANA Considerations

The Link Relation Type Registry should be updated with the following entry [TBD]:

- Relation Name: blocking-authority

- Description: Identifies the authority that has issued the block.

- Reference: This document

In addition, IANA should be updated with the following provisional header [TBD]:

- Header field name: geo-scope-block

- Applicable protocol: http

- Status: provisional

- Specification document(s): this document

7. Normative References

[IMPL_REPORT_DRAFT]
Abraham, S., Canales, MP., Hall, J., Khrustaleva, O., ten Oever, N., Runnegar, C., and S. Sahib, "Implementation Report for HTTP Status Code 451", 2017, <https://tools.ietf.org/html/draft-451-imp-report-00>.

[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>.

[RFC7725] Bray, T., "An HTTP Status Code to Report Legal Obstacles", RFC 7725, DOI 10.17487/RFC7725, February 2016, <https://www.rfc-editor.org/info/rfc7725>.

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

Author’s Address

Shivan Kaul Sahib

EMail: shivankaulsahib@gmail.com

HTTP
Internet-Draft
Intended status: Standards Track
Expires: May 3, 2018

M. Bishop

N. Sullivan
Cloudflare
M. Thomson
Mozilla

October 30, 2017

Secondary Certificate Authentication in HTTP/2
draft-bishop-httpbis-http2-additional-certs-05

Abstract

TLS provides fundamental mutual authentication services for HTTP, supporting up to one server certificate and up to one client certificate associated to the session to prove client and server identities as necessary. This draft provides mechanisms for providing additional such certificates at the HTTP layer when these constraints are not sufficient.

Many HTTP servers host content from several origins. HTTP/2 [RFC7540] permits clients to reuse an existing HTTP connection to a server provided that the secondary origin is also in the certificate provided during the TLS [I-D.ietf-tls-tls13] handshake.

In many cases, servers will wish to maintain separate certificates for different origins but still desire the benefits of a shared HTTP connection. Similarly, servers may require clients to present authentication, but have different requirements based on the content the client is attempting to access.

This document describes how TLS exported authenticators [I-D.ietf-tls-exported-authenticator] can be used to provide proof of ownership of additional certificates to the HTTP layer to support both scenarios.

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/.

Copyright Notice

Table of Contents

1. Introduction

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

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

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

1.1. Server Certificate Authentication

Section 9.1.1 of [RFC7540] describes how connections may be used to make requests from multiple origins as long as the server is authoritative for both. A server is considered authoritative for an origin if DNS resolves the origin to the IP address of the server and (for TLS) if the certificate presented by the server contains the origin in the Subject Alternative Names field.

[RFC7838] enables a step of abstraction from the DNS resolution. If both hosts have provided an Alternative Service at hostnames which resolve to the IP address of the server, they are considered authoritative just as if DNS resolved the origin itself to that address. However, the server's one TLS certificate is still required to contain the name of each origin in question.

[I-D.ietf-httpbis-origin-frame] relaxes the requirement to perform the DNS lookup if already connected to a server with an appropriate certificate which claims support for a particular origin.

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

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

1.2. Client Certificate Authentication

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

The TLS 1.3 CertificateRequest can be used by servers to give clients hints about which certificate to offer. Servers that rely on certificate-based authentication might request different certificates for different resources. Such a server cannot use contextual information about the resource to construct an appropriate TLS CertificateRequest message during the initial handshake.

Consequently, client certificates are requested at connection establishment time only in cases where all clients are expected or required to have a single certificate that is used for all resources. Many other uses for client certificates are reactive, that is, certificates are requested in response to the client making a request.

1.2.1. HTTP/1.1 using TLS 1.2 and previous

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

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

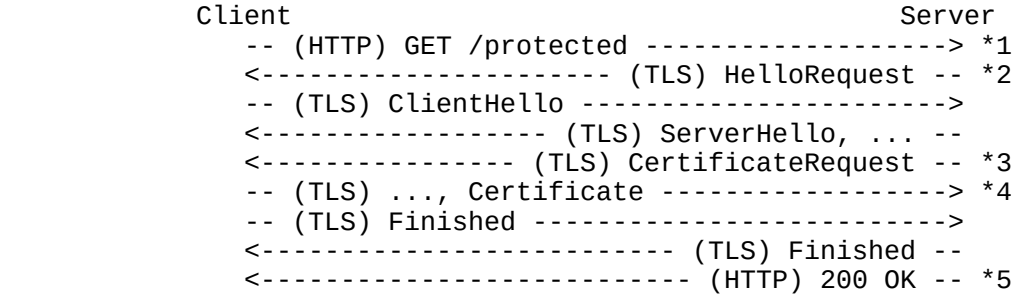


Figure 1: HTTP/1.1 Reactive Certificate Authentication with TLS 1.2

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

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

The critical messages for this example are the server requesting a certificate with a TLS CertificateRequest (*3); this request might use information about the request or resource. The client then provides a certificate and proof of possession of the private key in Certificate and CertificateVerify messages (*4).

When the handshake completes, the server performs any authorization checks a second time. With the client certificate available, it then authorizes the request and provides a response (*5).

1.2.2. HTTP/1.1 using TLS 1.3

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

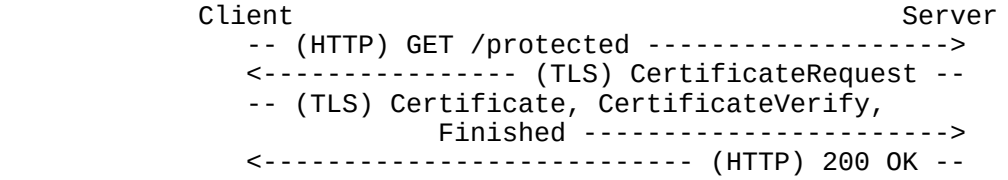


Figure 2: HTTP/1.1 Reactive Certificate Authentication with TLS 1.3

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

1.2.3. HTTP/2

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

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

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

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

renegotiation. This is problematic for a multiplexed protocol like HTTP/2.

1.3. HTTP-Layer Certificate Authentication

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

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

TLS Exported Authenticators are structured messages that can be exported by either party of a TLS connection and validated by the other party. An authenticator message can be constructed by either the client or the server given an established TLS connection, a certificate, and a corresponding private key. Exported Authenticators use the message structures from section 4.4 of [I-D.ietf-tls-tls13], but different parameters.

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

Successfully verified Authenticators result in certificate chains, with verified possession of the corresponding private key, which can be supplied into a collection of available certificates. Likewise, descriptions of desired certificates can be supplied into these collections. These pre-supplied elements are then available for automatic use (in some situations) or for reference by individual streams.

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

1.4. Terminology

RFC 2119 [RFC2119] defines the terms "MUST", "MUST NOT", "SHOULD" and "MAY".

2. Discovering Additional Certificates at the HTTP/2 Layer

A certificate chain with proof of possession of the private key corresponding to the end-entity certificate is sent as a single "CERTIFICATE" frame (see Section 3.4) on stream zero. Once the holder of a certificate has sent the chain and proof, this certificate chain is cached by the recipient and available for future use. If the certificate is marked as "AUTOMATIC_USE", the certificate may be used by the recipient to authorize any current or future request. Otherwise, the recipient requests the required certificate on each stream, but the previously-supplied certificates are available for reference without having to resend them.

Likewise, the details of a request are sent on stream zero and stored by the recipient. These details will be referenced by subsequent "CERTIFICATE_NEEDED" frames.

Data sent by each peer is correlated by the ID given in each frame. This ID is unrelated to values used by the other peer, even if each uses the same ID in certain cases.

2.1. Indicating support for HTTP-layer certificate authentication

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

The initial value for the "SETTINGS_HTTP_CERT_AUTH" setting is 0, indicating that the peer does not support HTTP-layer certificate authentication. If a peer does support HTTP-layer certificate authentication, the value is 1.

2.2. Making certificates or requests available

When a peer has advertised support for HTTP-layer certificates as in Section 2.1, either party can supply additional certificates into the connection at any time. These certificates then become available for the peer to consider when deciding whether a connection is suitable to transport a particular request.

Available certificates which have the "AUTOMATIC_USE" flag set MAY be used by the recipient without further notice. This means that clients or servers which predict a certificate will be required could

pre-supply the certificate without being asked. Regardless of whether "AUTOMATIC_USE" is set, these certificates are available for reference by future "USE_CERTIFICATE" frames.

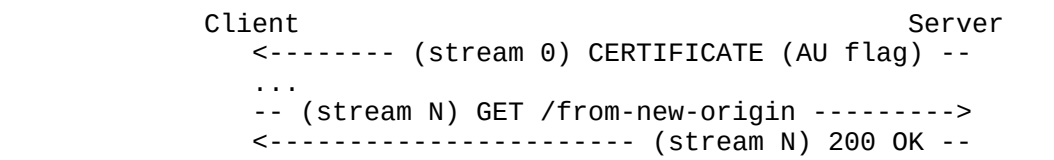


Figure 3: Proactive Server Certificate

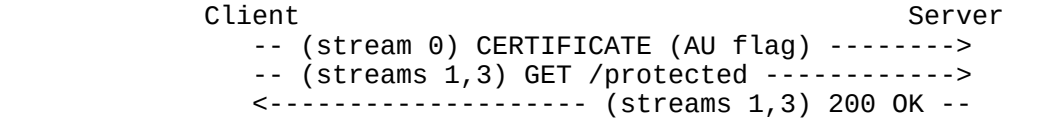


Figure 4: Proactive Client Certificate

Likewise, either party can supply a "CERTIFICATE_REQUEST" that outlines parameters of a certificate they might request in the future. It is important to note that this does not currently request such a certificate, but makes the contents of the request available for reference by a future "CERTIFICATE_NEEDED" frame.

2.3. Requiring certificate authentication

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

If the TLS certificate does not contain the new origin, but the server has claimed support for that origin (with an ORIGIN frame, see [I-D.ietf-httpbis-origin-frame]) and advertised support for HTTP-layer certificates (see Section 2.1), it MAY send a "CERTIFICATE_NEEDED" frame on the stream it will use to make the request. (If the request parameters have not already been made available using a "CERTIFICATE_REQUEST" frame, the client will need to send the "CERTIFICATE_REQUEST" in order to generate the "CERTIFICATE_NEEDED" frame.) The stream represents a pending request to that origin which is blocked until a valid certificate is processed.

The request is blocked until the server has responded with a "USE_CERTIFICATE" frame pointing to a certificate for that origin. If the certificate is already available, the server SHOULD immediately respond with the appropriate "USE_CERTIFICATE" frame. (If the certificate has not already been transmitted, the server will need to make the certificate available as described in Section 2.2 before completing the exchange.)

If the server does not have the desired certificate, it MUST respond with an empty "USE_CERTIFICATE" frame. In this case, or if the server has not advertised support for HTTP-layer certificates, the client MUST NOT send any requests for resources in that origin on the current connection.

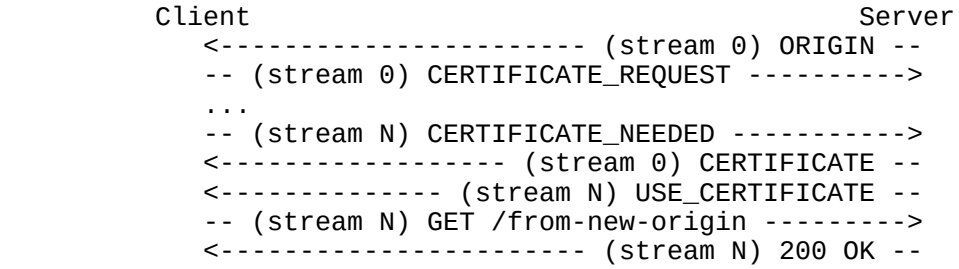


Figure 5: Client-Requested Certificate

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

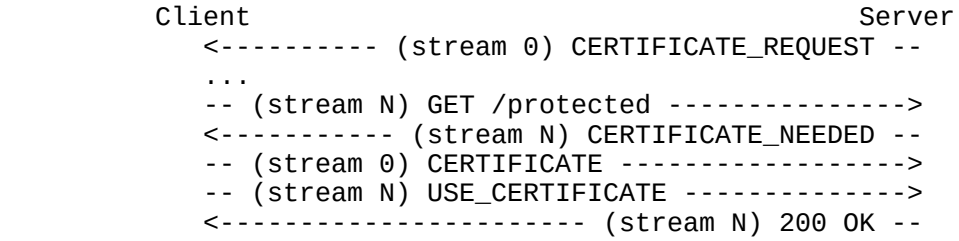


Figure 6: Reactive Certificate Authentication

A server SHOULD provide certificates for an origin before pushing resources from it or supplying content referencing the origin. If a

client receives a "PUSH_PROMISE" referencing an origin for which it has not yet received the server's certificate, the client MUST verify the server's possession of an appropriate certificate by sending a "CERTIFICATE_NEEDED" frame on the pushed stream to inform the server that progress is blocked until the request is satisfied. The client MUST NOT use the pushed resource until an appropriate certificate has been received and validated.

3. Certificates Frames for HTTP/2

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

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

"Request-ID" and "Cert-ID" are sender-local, and the use of the same value by the other peer does not imply any correlation between their frames. These values MUST be unique per sender over the lifetime of the connection.

3.1. The CERTIFICATE_NEEDED frame

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

The "CERTIFICATE_NEEDED" frame contains 2 octets, which is the authentication request identifier, "Request-ID". A peer that receives a "CERTIFICATE_NEEDED" of any other length MUST treat this as a stream error of type "PROTOCOL_ERROR". Frames with identical request identifiers refer to the same "CERTIFICATE_REQUEST".

A server MAY send multiple "CERTIFICATE_NEEDED" frames on the same stream. If a server requires that a client provide multiple certificates before authorizing a single request, each required certificate MUST be indicated with a separate "CERTIFICATE_NEEDED" frame, each of which MUST have a different request identifier (referencing different "CERTIFICATE_REQUEST" frames describing each

required certificate). To reduce the risk of client confusion, servers SHOULD NOT have multiple outstanding "CERTIFICATE_NEEDED" frames on the same stream at any given time.

Clients MUST NOT send multiple "CERTIFICATE_NEEDED" frames on the same stream.

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

The "CERTIFICATE_NEEDED" frame MUST NOT be sent on stream zero, and MUST NOT be sent on a stream in the "half-closed (local)" state [RFC7540]. A client that receives a "CERTIFICATE_NEEDED" frame on a stream which is not in a valid state SHOULD treat this as a stream error of type "PROTOCOL_ERROR".

3.2. The USE_CERTIFICATE Frame

The "USE_CERTIFICATE" frame (0xFRAME-TBD4) is sent in response to a "CERTIFICATE_NEEDED" frame to indicate which certificate is being used to satisfy the requirement.

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

Otherwise, the "USE_CERTIFICATE" frame contains the two-octet "Cert-ID" of the certificate the sender wishes to use. This MUST be the ID of a certificate for which proof of possession has been presented in a "CERTIFICATE" frame. Recipients of a "USE_CERTIFICATE" frame of any other length MUST treat this as a stream error of type "PROTOCOL_ERROR". Frames with identical certificate identifiers refer to the same certificate chain.

The "USE_CERTIFICATE" frame MUST NOT be sent on stream zero or a stream on which a "CERTIFICATE_NEEDED" frame has not been received. Receipt of a "USE_CERTIFICATE" frame in these circumstances SHOULD be treated as a stream error of type "PROTOCOL_ERROR". Each "USE_CERTIFICATE" frame should reference a preceding "CERTIFICATE" frame. Receipt of a "USE_CERTIFICATE" frame before the necessary frames have been received on stream zero MUST also result in a stream error of type "PROTOCOL_ERROR".

The referenced certificate chain MUST conform to the requirements expressed in the "CERTIFICATE_REQUEST" to the best of the sender’s

ability. Specifically, if the "CERTIFICATE_REQUEST" contained a non-empty "Cert-Extensions" element, the end-entity certificate MUST match with regard to the extensions recognized by the sender.

If these requirements are not satisfied, the recipient MAY at its discretion either return an error at the HTTP semantic layer, or respond with a stream error [RFC7540] on any stream where the certificate is used. Section 4 defines certificate-related error codes which might be applicable.

3.3. The CERTIFICATE_REQUEST Frame

TLS 1.3 defines the "CertificateRequest" message, which prompts the client to provide a certificate which conforms to certain properties specified by the server. This draft defines the "CERTIFICATE_REQUEST" frame (0xFRAME-TBD2), which uses the same set of extensions to specify a desired certificate, but can be sent over any TLS version and can be sent by either peer.

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

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

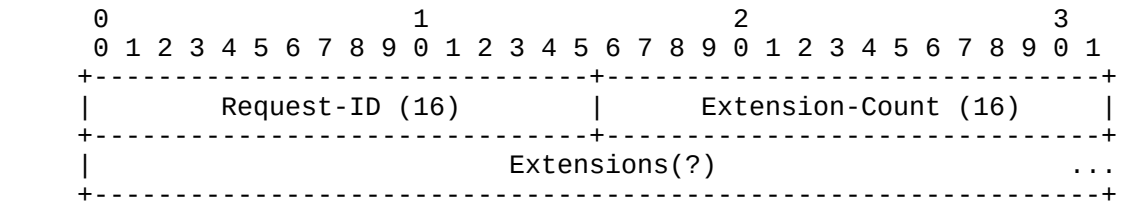


Figure 7: CERTIFICATE_REQUEST frame payload

The frame contains the following fields:

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

Extension-Count and Extensions: A list of certificate selection criteria, represented in a series of "Extension" structures (see [I-D.ietf-tls-tls13] section 4.2). This criteria MUST be used in certificate selection as described in [I-D.ietf-tls-tls13]. The number of "Extension" structures is given by the 16-bit "Extension-Count" field, which MAY be zero.

Some extensions used for certificate selection allow multiple values (e.g. oid_filters on Extended Key Usage). If the sender has included a non-empty Extensions list, the certificate MUST match all criteria specified by extensions the recipient recognizes. However, the recipient MUST ignore and skip any unrecognized certificate selection extensions.

Servers MUST be able to recognize the "server_name" extension ([RFC6066]) at a minimum. Clients MUST always specify the desired origin using this extension, though other extensions MAY also be included.

3.4. The CERTIFICATE Frame

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

The "CERTIFICATE" frame defines two flags:

AUTOMATIC_USE (0x01): Indicates that the certificate can be used automatically on future requests.

TO_BE_CONTINUED (0x02): Indicates that the exported authenticator spans more than one frame.

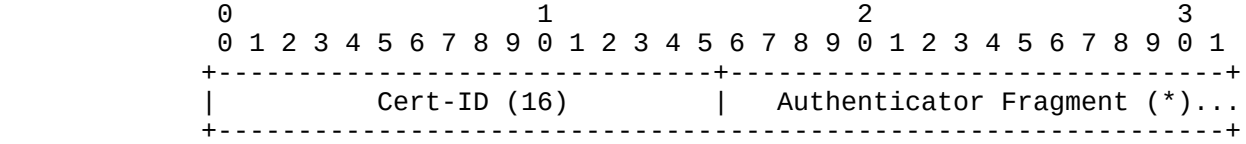


Figure 8: CERTIFICATE frame payload

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

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

If the "AUTOMATIC_USE" flag is set, the recipient MAY omit sending "CERTIFICATE_NEEDED" frames on future streams which would require a similar certificate and use the referenced certificate for authentication without further notice to the holder. This behavior is optional, and receipt of a "CERTIFICATE_NEEDED" frame does not imply that previously-presented certificates were unacceptable, even if "AUTOMATIC_USE" was set. Servers MUST set the "AUTOMATIC_USE" flag when sending a "CERTIFICATE" frame. A server MUST NOT send certificates for origins which it is not prepared to service on the current connection.

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

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

3.4.1. Exported Authenticator Characteristics

The Exported Authenticator API defined in [I-D.ietf-tls-exported-authenticator] takes as input a certificate, supporting information about the certificate (OCSP, SCT, etc.), and an optional "certificate_request_context". When generating exported authenticators for use with this extension, the "certificate_request_context" MUST be the two-octet Cert-ID.

Upon receipt of a completed authenticator, an endpoint MUST check that:

- the "validate" API confirms the validity of the authenticator itself
- the "certificate_request_context" matches the Cert-ID of the frame(s) in which it was received

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

4. Indicating failures during HTTP-Layer Certificate Authentication

Because this draft permits certificates to be exchanged at the HTTP framing layer instead of the TLS layer, several certificate-related errors which are defined at the TLS layer might now occur at the HTTP

framing layer. In this section, those errors are restated and added to the HTTP/2 error code registry.

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

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

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

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

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

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

5. Security Considerations

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

5.1. Impersonation

This mechanism could increase the impact of a key compromise. Rather than needing to subvert DNS or IP routing in order to use a compromised certificate, a malicious server now only needs a client to connect to _some_ HTTPS site under its control in order to present the compromised certificate. As recommended in [I-D.ietf-httpbis-origin-frame], clients opting not to consult DNS ought to employ some alternative means to increase confidence that the certificate is legitimate.

As noted in the Security Considerations of [I-D.ietf-tls-exported-authenticator], it difficult to formally prove that an endpoint is jointly authoritative over multiple certificates, rather than individually authoritative on each certificate. As a result, clients MUST NOT assume that because one origin was previously colocated with another, those origins will be reachable via the same endpoints in the future. Clients MUST NOT consider previous secondary certificates to be validated after TLS session resumption. However, clients MAY proactively query for previously-presented secondary certificates.

5.2. Fingerprinting

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

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

5.3. Denial of Service

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

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

5.4. Confusion About State

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

reevaluate the authorization state of a request as the set of certificates changes.

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

6. IANA Considerations

This draft adds entries in three registries.

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

6.1. HTTP/2 SETTINGS_HTTP_CERT_AUTH Setting

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

Name: SETTINGS_HTTP_CERT_AUTH

Code: 0xSETTING-TBD

Initial Value: 0

Specification: This document.

6.2. New HTTP/2 Frames

Four new frame types are registered in the "HTTP/2 Frame Types" registry established in [RFC7540]. The entries in the following table are registered by this document.

Frame Type	Code	Specification
CERTIFICATE_NEEDED	0xFRAME-TBD1	Section 3.1
CERTIFICATE_REQUEST	0xFRAME-TBD2	Section 3.3
CERTIFICATE	0xFRAME-TBD3	Section 3.4
USE_CERTIFICATE	0xFRAME-TBD4	Section 3.2

6.3. New HTTP/2 Error Codes

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

Name	Code	Specification
BAD_CERTIFICATE	0xERROR-TBD1	Section 4
UNSUPPORTED_CERTIFICATE	0xERROR-TBD2	Section 4
CERTIFICATE_REVOKED	0xERROR-TBD3	Section 4
CERTIFICATE_EXPIRED	0xERROR-TBD4	Section 4
CERTIFICATE_GENERAL	0xERROR-TBD5	Section 4

7. Acknowledgements

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

8. References

8.1. Normative References

[I-D.ietf-tls-exported-authenticator]
Sullivan, N., "Exported Authenticators in TLS", draft-ietf-tls-exported-authenticator-03 (work in progress), July 2017.

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-21 (work in progress), July 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>.

[RFC2459] Housley, R., Ford, W., Polk, W., and D. Solo, "Internet X.509 Public Key Infrastructure Certificate and CRL Profile", RFC 2459, DOI 10.17487/RFC2459, January 1999, <https://www.rfc-editor.org/info/rfc2459>.

[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>.

[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>.

[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>.

[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>.

[X690] ITU-T, "Information technology - ASN.1 encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", ISO ISO/IEC 8825-1:2002, 2002, <http://www.itu.int/ITU-T/studygroups/com17/languages/X.690-0207.pdf>.

8.2. Informative References

[I-D.ietf-httpbis-origin-frame]
Nottingham, M. and E. Nygren, "The ORIGIN HTTP/2 Frame", draft-ietf-httpbis-origin-frame-04 (work in progress), August 2017.

[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>.

Authors' Addresses

Mike Bishop

Email: mbishop@evequefou.be

Nick Sullivan
Cloudflare

Email: nick@cloudflare.com

Martin Thomson
Mozilla

Email: martin.thomson@gmail.com

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: May 4, 2018

A. Hope-Bailie
Ripple

October 31, 2017

HTTP-Payments
draft-hope-bailie-http-payments-00

Abstract

HTTP-Payments describes a mechanism for passing a standardized payment request in the headers of an HTTP 402 response and the expected behaviour of HTTP clients that receive such a response.

Feedback

This specification is an early experiment in bringing the work of the W3C Web Payments working group to the HTTP protocol. It is maintained at https://github.com/adrianhopebailie/http-payments [1].

The work is inspired by work in the Interledger community on [HTTP-ILP]

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 May 4, 2018.

Copyright Notice

Table of Contents

1. Introduction

The W3C Web Payments working group has defined a Web Platform API that is being widely deployed to browsers for requesting a payment. The PaymentRequest API [W3C.CR-payment-request-20170921], defines an interface for a website to pass a payment request to the user agent via this API.

The user agent will then, through interaction with the user, complete or reject the requested payment.

HTTP-Payments describes a manner in which an HTTP server can request payment from a client in the same manner as a website would from a user agent using the W3C APIs.

The critical portion of the payment request is the set of, one or more, supported payment methods and associated payment-method-specific data. HTTP-Payments defines a mechaism by which these are expressed in the response headers of an HTTP request for which the server requires a payment.

In the website and user-agent scenario, when handling the payment request, the user-agent will prompt the user to pick one of the supported payment methods and will then handle the payment in a manner that is appropriate for that payment method. In an HTTP-Payment, the HTTP client will perform this function, likely with no user interaction.

2. Terminology

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][].

3. Payment Methods

A payment method is a way that the payee can be paid. Examples include, via credit card, bank wire transfer, or Bitcoin.

A payment method is identified by a payment method identifier as specified in the Payment Method Identifiers specification [2]. This is either a standardized short-string, identified in a registry maintained by the W3C Web Payments WG, or a URL.

The most common case will be for the URL form to be used. In cases where there is no authority responsible for the payment method that can host the payment method URL, the WG will consider adding a new identifier for the payment method to the registry.

Payment methods define the data that the payer and payee need to exchange, to complete a payment, and the process by which this occurs.

4. HTTP Status Code 402

The HTTP Status Code, 402 (Payment Required) is currently defined in [RFC7231] as "reserved for future use". Using HTTP-Payment a service MAY respond to any request with the 402 response code and use the "Pay" header to specify the payment request details.

4.1. The "Pay" Header

The body of the "Pay" header is defined as follows:

```
Pay: <payment-method-identifier> <amount> <address> <payment-method-data>
```

Multiple "Pay" headers MAY be present in an HTTP 402 response.

The fields in the header are:

- payment-method: The payment method identifier for the accepted payment method. Either a standardized short-string or a URL.

- amount: The amount that must be paid, expressed as an integer. The currency, scale and precision of the destination account are expected to be expressed in the account address.

- address: A payment-method specific payee address. For example, if the payment method is Bitcoin this would be a Bitcoin address.

- payment-method-data: Payment method specific data. This is either a URI identifying the data or, if it is small enough, is the data itself, BASE64URL encoded as described in [RFC4648], Section 5.

4.2. The "Pay-Token" Header

An HTTP client that makes a paid-HTTP request, after paying for the request to be processed, MAY attach a "Pay-Token" header with a token referencing the payment.

This mechanism can be employed by services wishing to accept payments without binding these to an HTTP session.

4.3. The "Pay-Balance" Header

Ann HTTP Service that accepts payments may respond to any request with a "Pay-Balance" header. This contains an integer indicating the current balance of paid credit the client has with the HTTP service.

4.4. Flow

Upon receipt of a 402 response, an HTTP client MUST look for any "Pay" headers and parse these. The client can discard all headers for which it is not equipped to make a payment (i.e. filter on payment-method-identifier)

The client MUST then select the header that is preferred for processing based upon external interactions (such as with a human user) or pre-configured rules. The client MUST attempt to make a payment using the payment method identified in the header, for the amount specified, and to the destination address specified.

The payment-method specific data SHOULD be sufficient for the system processing the payment to reconcile the payment with the original HTTP request.

The client SHOULD receive a token in return for completeing the payment. If the payment method used does return a token to the payer, it MUST pass this token in subsequent HTTP requests.

The token MUST be passed in the "Pay-Token" header, BASE64URL encoded as described in [RFC4648], Section 5.

The HTTP service MUST process the "Pay-Token" header and use this to reconcile this HTTP request with the payment received prior.

4.5. Example

Client requests access to a paid resource:

```
POST /upload HTTP/1.1
Host: myservice.example
```

Server responds with payment request (and optionally indicates that the client has a zero balance):

```
HTTP/1.1 402 Payment Required
Pay: http://interledger.org 10 us.nexus.ankita.~recv.filepay SkTcFTZCBKgP6A6QOUV
cwWCCgYIP4rJPHlIzreavHdU
Pay-Balance: 0
```

Client makes the payment through an appropriate payment side-channel and then attempts the request again:

```
POST /upload HTTP/1.1
Host: myservice.example
Pay-Token: 7y0SfeN7lCuq0GFF5UsMYZofIjJ7LrvPvsePVWSv450
```

Server responds:

```
HTTP/1.1 200 Success
Pay-Balance: 0
```

5. References

5.1. Normative References

[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>.

[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>.

[W3C.CR-payment-method-id-20170914]
Bateman, A., Koch, Z., McElmurry, R., and M. Caceres, "Payment Method Identifiers", World Wide Web Consortium CR CR-payment-method-id-20170914, September 2017, <https://www.w3.org/TR/2017/CR-payment-method-id-20170914>.

[W3C.CR-payment-request-20170921]
Bateman, A., Koch, Z., McElmurry, R., Denicola, D., and M. Caceres, "Payment Request API", World Wide Web Consortium CR CR-payment-request-20170921, September 2017, <https://www.w3.org/TR/2017/CR-payment-request-20170921>.

5.2. Informative References

[HTTP-ILP]
Interledger Community Group, "HTTP-ILP", October 2017, <https://github.com/interledger/rfcs/blob/58d8dcb015b160a381313126fa3065c64406db05/0014-http-ilp/0014-http-ilp.md>.

5.3. URIs

[1] https://github.com/adrianhopebailie/http-payments

[2] W3C.CR-payment-method-id-20170914

Appendix A. Security Considerations

TBD

Appendix B. IANA Considerations

B.1. Payment Method Identifier Short-string Registry

The W3C maintains a registry of standardized short-string payment method identifiers as part of the [Payment Method Identifier] specification. If standardized short-string identifiers are to be used for HTTP-Payments this may be better served as an IANA registry.

Author’s Address

Adrian Hope-Bailie
Ripple
315 Montgomery Street
San Francisco, CA 94104
US

Phone: ----------
Email: adrian@ripple.com
URI: https://www.ripple.com

HTTP Working Group
Internet-Draft
Intended status: Experimental
Expires: January 3, 2019

K. Oku
Fastly
Y. Weiss
Akamai

July 2, 2018

Cache Digests for HTTP/2
draft-ietf-httpbis-cache-digest-05

Abstract

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

Note to Readers

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

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

Status of This Memo

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

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

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

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

Copyright Notice

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.

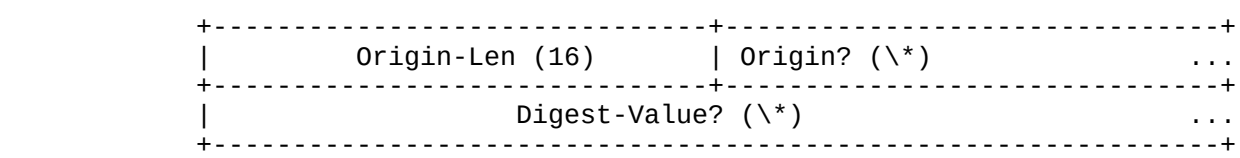
This specification defines a HTTP/2 frame type to allow clients to inform the server of their freshly cached contents using a Cuckoo-filter [Cuckoo] based digest. Servers can then use this to inform their choices of what to push to clients.

1.1. Notational Conventions

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

2. The CACHE_DIGEST Frame

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



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.

2.1. Client Behavior

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

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

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

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

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

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

2.1.1. Creating a digest

Given the following inputs:

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

- "N", an integer that represents the number of entries - a prime number smaller than 2**32

1. Let "f" be the number of bits per fingerprint, calculated as "P + 3"

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

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

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

5. Add 5 to "bytes"

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

7. Set the first byte to "P"

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

9. Return the "digest-value".

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

2.1.2. Adding a URL to the Digest-Value

Given the following inputs:

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

- "maxcount" - max number of cuckoo hops

- "digest-value"

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

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

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

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

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

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

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

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

9. While "maxcount" is larger than zero:

    1. Let "position_start" be 40 + "h" * "f" * "b".

    2. Let "position_end" be "position_start" + "f" * "b".

    3. While "position_start" < "position_end":

        1. Let "bits" be "f" bits from "digest_value" starting at "position_start".

        2. If "bits" is all zeros, set "bits" to "dest_fingerprint" and terminate these steps.

        3. Add "f" to "position_start".

    4. Let "e" be a random number from 0 to "b".

    5. Subtract "f" * ("b" - "e") from "position_start".

    6. Let "bits" be "f" bits from "digest_value" starting at "position_start".

    7. Let "fingerprint" be the value of bits, read as big endian.

    8. Set "bits" to "dest_fingerprint".

    9. Set "dest_fingerprint" to "fingerprint".

    10. Let "h" be Section 2.1.7 with "h", "dest_fingerprint" and "N" as inputs.

    11. Subtract 1 from "maxcount".

10. Subtract "f" from "position_start".

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

12. Let "h1" be "h"

13. Subtract 1 from "maxcount".

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

15. Go to step 7.

2.1.3. Removing a URL to the Digest-Value

Given the following inputs:

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

1. Let "f" be the value of the first byte of "digest-value".
2. Let "b" be the bucket size, defined as 4.
3. Let "N" be the value of the second to fifth bytes of "digest-value" in big endian form.
4. Let "key" be the return value of Section 2.1.5 with "URL" as input.
5. Let "h1" be the return value of Section 2.1.6 with "key" and "N" as inputs.
6. Let "fingerprint" be the return value of Section 2.1.4 with "key" and "f" as inputs.
7. Let "h2" be the return value of Section 2.1.7 with "h1", "fingerprint" and "N" as inputs.
8. Let "hashes" be an array containing "h1" and "h2".
9. For each "h" in "hashes":
    1. Let "position_start" be 40 + "h" * "f" * "b".
    2. Let "position_end" be "position_start" + "f" * "b".

3. While "position_start" < "position_end":

    1. Let "bits" be "f" bits from "digest_value" starting at "position_start".

    2. If "bits" is "fingerprint", set "bits" to all zeros and terminate these steps.

    3. Add "f" to "position_start".

2.1.4. Computing a fingerprint value

Given the following inputs:

- "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

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

2. Return "key"

2.1.6. Computing a Hash Value

Given the following inputs:

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

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

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

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

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

2.2.1. Querying the Digest for a Value

Given the following inputs:

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

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

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

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

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

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

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

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

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

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

9. For each "h" in "hashes":

    1. Let "position_start" be 40 + "h" * "f" * "b".

    2. Let "position_end" be "position_start" + "f" * "b".

    3. While "position_start" < "position_end":

        1. Let "bits" be "f" bits from "digest_value" starting at "position_start".

        2. If "bits" is "fingerprint", return true

        3. Add "f" to "position_start".

10. Return false.

3. The SETTINGS_SENDING_CACHE_DIGEST SETTINGS Parameter

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

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

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

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

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

4. The SETTINGS_ACCEPT_CACHE_DIGEST SETTINGS Parameter

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

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

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

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

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

Some underlying transports allow the server’s first flight of application data to reach the client at around the same time when the client sends 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 SETTINGS_ACCEPT_CACHE_DIGEST settings value.

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

5. IANA Considerations

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

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

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

- 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: SETTINGS_ACCEPT_CACHE_DIGEST
- Initial Value: 0x0
- Reference: [this document]

6. Security Considerations

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

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

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

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

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

7. References

7.1. Normative References

[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 a digest of one resource and has only the "COMPLETE" flag set.

```
Cache-Digest: AfdA; complete
```

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

Also, due to the fact that any header that is supplied to Fetch is required to be end-to-end, there is an ambiguity in what a Cache-Digest header 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. Changes

B.1. Since draft-ietf-httpbis-cache-digest-04

- Remove ETag from the digest key calculations.
- Add SETTINGS_ prefix to parameter names.

B.2. Since draft-ietf-httpbis-cache-digest-03

- Yoav becomes an author; Mark steps down.

B.3. Since draft-ietf-httpbis-cache-digest-02

- Switch to Cuckoo Filter.

B.4. 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.

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

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

Appendix C. Acknowledgements

+{:numbered="false"}

Thanks to Stefan Eissing for his suggestions.

Authors' Addresses

Kazuho Oku
Fastly

Email: kazuhooku@gmail.com

Yoav Weiss
Akamai

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

HTTP Working Group
Internet-Draft
Intended status: Experimental
Expires: September 12, 2019

I. Grigorik
Google

March 11, 2019

HTTP Client Hints
draft-ietf-httpbis-client-hints-07

Abstract

HTTP defines proactive content negotiation to allow servers to select the appropriate response for a given request, based upon the user agent's characteristics, as expressed in request headers. In practice, clients are often unwilling to send those request headers, because it is not clear whether they will be used, and sending them impacts both performance and privacy.

This document defines two response headers, Accept-CH and Accept-CH-Lifetime, that servers can use to advertise their use of request headers for proactive content negotiation, along with a set of guidelines for the creation of such headers, colloquially known as "Client Hints."

It also defines an initial set of Client Hints.

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/client-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 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 12, 2019.

Table of Contents

1. Introduction

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

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

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

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

Proactive content negotiation (Section 3.4.1 of [RFC7231]) offers an alternative approach; user agents use specified, well-defined request headers to advertise their capabilities and characteristics, so that servers can select (or formulate) an appropriate response.

However, proactive content negotiation requires clients to send these request headers prolifically. This causes performance concerns (because it creates "bloat" in requests), as well as privacy issues; passively providing such information allows servers to silently fingerprint the user agent.

This document defines a new response header, Accept-CH, that allows an origin server to explicitly ask that clients send these headers in requests, for a period of time bounded by the Accept-CH-Lifetime response header. It also defines guidelines for content negotiation mechanisms that use it, colloquially referred to as Client Hints.

Client Hints mitigate the performance concerns by assuring that clients will only send the request headers when they're actually

going to be used, and the privacy concerns of passive fingerprinting by requiring explicit opt-in and disclosure of required headers by the server through the use of the Accept-CH response header.

This document defines the Client Hints infrastructure, a framework that enables servers to opt-in to specific proactive content negotiation features, which will enable them to adapt their content accordingly. However, it does not define any specific features that will use that infrastructure. Those features will be defined in their respective specifications.

This document does not supersede or replace the User-Agent header field. Existing device detection mechanisms can continue to use both mechanisms if necessary. By advertising user agent capabilities within a request header field, Client Hints allow for cache friendly and proactive content negotiation.

1.1. Notational Conventions

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

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

2. Client Hint Request Header Fields

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

2.1. Sending Client Hints

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

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

2.2. Server Processing of Client Hints

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

Further, depending on the hint used, the server can generate additional response header fields to convey related values to aid client processing.

2.2.1. Advertising Support via Accept-CH Header Field

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

```
Accept-CH = #field-name
```

For example:

```
Accept-CH: Sec-CH-Example, Sec-CH-Example-2
```

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

For example, based on Accept-CH example above, a user agent could append the Sec-CH-Example and Sec-CH-Example-2 header fields to all same-origin resource requests initiated by the page constructed from the response.

2.2.2. The Accept-CH-Lifetime Header Field

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

```
Accept-CH-Lifetime = #delta-seconds
```

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

```
Accept-CH: Sec-CH-Example, Sec-CH-Example-2
Accept-CH: Sec-CH-Example-3
Accept-CH-Lifetime: 86400
```

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

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

2.2.3. Interaction with Caches

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

```
Vary: Sec-CH-Example
```

Above example indicates that the cache key needs to include the Sec-CH-Example header field.

```
Vary: Sec-CH-Example, Sec-CH-Example-2
```

Above example indicates that the cache key needs to include the Sec-CH-Example and Sec-CH-Example-2 header fields.

3. Security Considerations

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

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

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

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

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

4. IANA Considerations

This document defines the "Accept-CH" and "Accept-CH-Lifetime" HTTP response fields, and registers them in the Permanent Message Header Fields registry.

4.1. Accept-CH

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

4.2. Accept-CH-Lifetime

- Header field name: Accept-CH-Lifetime
- Applicable protocol: HTTP
- Status: standard
- Author/Change controller: IETF
- Specification document(s): Section 2.2.2 of this document
- Related information: for Client Hints

5. References

5.1. Normative References

[HTML5] Hickson, I., Berjon, R., Faulkner, S., Leithead, T., Navara, E., O'Connor, T., and S. Pfeiffer, "HTML5", World Wide Web Consortium Recommendation REC-html5-20141028, October 2014, <http://www.w3.org/TR/2014/REC-html5-20141028>.

[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>.

[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>.

[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>.

[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>.

[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

[KEY] Fielding, R. and M. Nottingham, "The Key HTTP Response Header Field", draft-ietf-httpbis-key-01 (work in progress), March 2016.

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

Appendix A. Interaction with Key Response Header Field

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

```
Key: Sec-CH-Example;partition=1.5:2.5:4.0
```

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

```
Key: Width;Sec-CH-Example=320
```

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

Appendix B. Changes

B.1. Since -00

- Issue 168 (make Save-Data extensible) updated ABNF.
- Issue 163 (CH review feedback) editorial feedback from httpwg list.
- Issue 153 (NetInfo API citation) added normative reference.

B.2. Since -01

- Issue 200: Moved Key reference to informative.
- Issue 215: Extended passive fingerprinting and mitigation considerations.
- Changed document status to experimental.

B.3. Since -02

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

B.4. Since -03

- Issue 284: Extended guidance for Accept-CH
- Issue 308: Editorial cleanup
- Issue 306: Define Accept-CH-Lifetime

B.5. Since -04

- Issue 361: Removed Downlink
- Issue 361: Moved Key to appendix, plus other editorial feedback

B.6. Since -05

- Issue 372: Scoped CH opt-in and delivery to secure transports
- Issue 373: Bind CH opt-in to origin

B.7. Since -06

- Issue 524: Save-Data is now defined by NetInfo spec, dropping

B.8. Since -07

- Removed specific features to be defined in other specifications

Acknowledgements

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

Author's Address

Ilya Grigorik
Google

Email: ilya@igvita.com
URI: https://www.igvita.com/

HTTP
Internet-Draft
Intended status: Experimental
Expires: June 12, 2019

E. Stark
Google

December 9, 2018

Expect-CT Extension for HTTP
draft-ietf-httpbis-expect-ct-08

Abstract

This document defines a new HTTP header field named Expect-CT, which allows web host operators to instruct user agents to expect valid Signed Certificate Timestamps (SCTs) to be served on connections to these hosts. Expect-CT allows web host operators to discover misconfigurations in their Certificate Transparency deployments. Further, web host operaters can use Expect-CT to ensure that, if a UA which supports Expect-CT accepts a misissued certificate, that certificate will be 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

Table of Contents

1. Introduction

This document defines a new HTTP header field that enables UAs to identify web hosts that expect the presence of Signed Certificate Timestamps (SCTs) [I-D.ietf-trans-rfc6962-bis] in subsequent Transport Layer Security (TLS) [RFC8446] connections.

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

If misconfigured, Expect-CT can cause unwanted connection failures (for example, if a host deploys Expect-CT but then switches to a legitimate certificate that is not logged in Certificate Transparency logs, or if a web host operator believes their certificate to conform to all UAs' CT policies but is mistaken). Web host operators are advised to deploy Expect-CT with precautions, by using the reporting feature and gradually increasing the time interval during which the UA regards 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.

Expect-CT is similar to HSTS [RFC6797] and HPKP [RFC7469]. HSTS allows web sites to declare themselves accessible only via secure connections, and HPKP allows web sites to declare their cryptographic identifies. Similarly, Expect-CT allows web sites to declare themselves accessible only via connections that are compliant with CT policy.

This Expect-CT specification is compatible with [RFC6962] and [I-D.ietf-trans-rfc6962-bis], but not with future versions of Certificate Transparency. Expect-CT header fields will be ignore from web hosts which use future versions of Certificate Transparency, unless a future version of this document specifies how they should be processed.

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 BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

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 associated with TLS connections. The policy defines the properties of a connection that must be met in order for the UA to consider it CT-qualified.

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

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

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

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

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

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

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

- "Unknown Expect-CT Host" is an Expect-CT Host that the UA has not noted.

2. Server and Client Behavior

2.1. Response Header Field Syntax

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

Figure 1 describes the syntax (Augmented Backus-Naur Form) of the header field, using the grammar defined in [RFC5234] and the rules defined in Section 3.2 of [RFC7230]. The "#" ABNF extension is specified in Section 7 of [RFC7230].

```
Expect-CT           = 1#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

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), and Expect-CT is not disabled for local policy reasons (as discussed in Section 2.4.1), 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].

UAs MUST ignore "report-uri"s that do not use the HTTPS scheme. UAs MUST check Expect-CT compliance when the host in the "report-uri" is a Known Expect-CT Host; similarly, UAs MUST apply HSTS [RFC6797] if the host in the "report-uri" is a Known HSTS Host.

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.

Note that the report-uri need not necessarily be in the same Internet domain or web origin as the host being reported about. Hosts are in fact encouraged to use a separate host as the report-uri, so that CT failures on the Expect-CT host do not prevent reports from being sent.

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 in the same web browsing session.

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.

If a response contains an "Expect-CT" header field, then the response MUST contain an "Expect-CT" header field with a "max-age" directive. (A "max-age" directive need not appear in every "Expect-CT" header field in the response.) The "max-age" directive is REQUIRED to have a directive value, for which the syntax (after quoted-string unescaping, if necessary) is defined in Figure 3.

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

Figure 3: Syntax of the max-age directive value

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

2.1.4. Examples

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

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

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

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

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

2.2. Host Processing Model

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

2.2.1. HTTP-over-Secure-Transport Request Type

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

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

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

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

2.2.2. HTTP Request Type

Expect-CT Hosts SHOULD NOT include the Expect-CT header field in HTTP responses conveyed over non-secure transport.

2.3. User Agent Processing Model

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

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

2.3.1. Missing or Malformed Expect-CT Header Fields

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

2.3.2. Expect-CT Header Field Processing

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

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

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

- Note the host as a Known Expect-CT Host if it is not already so noted (see Section 2.3.2.1), or

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

If a UA receives an Expect-CT header field over a CT-compliant connection which uses a version of Certificate Transparency other than [RFC6962] or [I-D.ietf-trans-rfc6962-bis], the UA MUST ignore the Expect-CT header field and clear any Expect-CT metadata associated with the host.

2.3.2.1. Noting Expect-CT

Upon receipt of the Expect-CT response header field over an error-free TLS connection (with X.509 certificate chain validation as described in [RFC5280], as well as the validation described in Section 2.4), the UA MUST note the host as a Known Expect-CT Host,

storing the host's domain name and its associated Expect-CT directives in non-volatile storage.

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

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

2.3.2.2. Storage Model

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

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

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

(One way to achieve this behavior is for the UA to simply store a value of 60 days instead of the 90-day value provided by the Expect-CT host.)

2.3.3. Reporting

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

2.4. Evaluating Expect-CT Connections for CT Compliance

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

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

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

If a connection to a Known Expect-CT Host violates the UA's CT policy (i.e., the connection is not CT-qualified), and if the Known Expect-CT Host's Expect-CT metadata indicates an "enforce" configuration, the UA MUST treat the CT compliance failure as an error. The UA MAY allow the user to bypass the error, unless connection errors should have no user recourse due to other policies in effect (such as HSTS, as described in Section 12.1 of [RFC6797].

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

2.4.1. Skipping CT compliance checks

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

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

3. Reporting Expect-CT Failure

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

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

3.1. Generating a violation report

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

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

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

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

- "scheme": the value is the scheme with which the UA made the original request that failed the CT compliance check. The value is provided as a string. This key is optional and is assumed to be "https" if not present.

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

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

- "validated-certificate-chain": the value is the certificate chain as constructed by the UA during certificate chain verification. (This may differ from the value of the "served-certificate-chain" key.) The value is provided as an array of strings, which MUST appear in the order matching the chain that the UA validated; each string in the array is the Privacy-Enhanced Mail (PEM) representation of each X.509 certificate as described in [RFC7468]. The first certificate in the chain represents the end-entity certificate being verified. UAs that build certificate chains in more than one way during the validation process SHOULD send the last chain built.

- "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 4.5 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 a bad signature or an invalid timestamp).

* The "source" key, with a string value that indicates from where the UA obtained the SCT, as defined in Section 3 of [RFC6962] and Section 6 of [I-D.ietf-trans-rfc6962-bis]. The UA MUST set the value to one of "tls-extension", "ocsp", or "embedded". These correspond to the three methods of delivering SCTs in the TLS handshake that are described in Section 3.3 of [RFC6962].

* 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]. The base64 encoding is defined in Section 4 of [RFC4648]. If the value of the "version" key is "2", the UA MUST set this value to the base64 encoded [RFC4648] serialized "TransItem" structure representing the SCT, as defined in Section 4.5 of [I-D.ietf-trans-rfc6962-bis].

- "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 report 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 Expect-CT failures for Known Expect-CT Hosts: that is, 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 Expect-CT failures for hosts for which it does not have any stored Expect-CT metadata. That is, when the UA connects to a host and receives an Expect-CT header field which contains the "report-uri" directive, the UA SHOULD report an Expect-CT failure if the the connection 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].

Future versions of this specification may need to modify or extend the Expect-CT report format. They may do so by defining a new top-level key to contain the report, replacing the "expect-ct-report" key. Section 3.3 defines how report servers should handle report formats that they do not support.

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, the report conforms to the format described in Section 3.1, and it recognizes the scheme, hostname, and port in the "scheme", "hostname", and "port" fields of the report. If the report body cannot be parsed, or the report does not conform to the format described in Section 3.1, or the report server does not expect to receive reports for the scheme, hostname, or port in the report, then the report server MUST respond with a 400 Bad Request status code.

As described in Section 3.2, future versions of this specification may define new report formats that are sent with a different top-level key. If the report server does not recognize the report format, the report server MUST respond with a 501 Not Implemented 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. If the "test-report" key is absent or set to false, the server SHOULD store the report for processing and analysis by the owner of the Expect-CT Host.

4. Usability Considerations

When the UA detects a Known Expect-CT Host in violation of the UA's CT Policy, end users will experience denials of service. It is advisable for UAs to explain to users why they cannot access the Expect-CT Host, e.g., in a user interface that explains that the host's certificate cannot be validated.

5. Authoring Considerations

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

6. Privacy Considerations

Expect-CT can be used to infer what Certificate Transparency policy a UA is using, 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 store state about Known Expect-CT Hosts, and hence which domains the UA has contacted. Implementations may choose to not store this state subject to local policy (e.g., in the private browsing mode of a web browser).

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 end users to clear currently noted Expect-CT hosts, and allow users to query the current state of Known Expect-CT Hosts.

7. Security Considerations

7.1. Hostile header attacks

When UAs support the Expect-CT header field, it becomes a potential vector for hostile header attacks against site owners. If a site owner uses a certificate issued by a certificate authority which does not embed SCTs nor serve SCTs via OCSP or TLS extension, a malicious server operator or attacker could temporarily reconfigure the host to comply with the UA's CT policy, and add the Expect-CT header field in enforcing mode with a long "max-age". Implementing user agents would note this as an Expect-CT Host (see Section 2.3.2.1). After having done this, the configuration could then be reverted to not comply with the CT policy, prompting failures. Note this scenario would require the attacker to have substantial control over the infrastructure in question, being able to obtain different certificates, change server software, or act as a man-in-the-middle in connections.

Site operators can mitigate 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 4.

7.2. 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.

7.3. Amplification attacks

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

8. IANA Considerations

8.1. Header Field Registry

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

Header field name: Expect-CT

Applicable protocol: http

Status: experimental

Author/Change controller: IETF

Specification document(s): This document

Related information: (empty)

8.2. Media Types Registry

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

Type name: application

Subtype name: expect-ct-report+json

Required parameters: n/a

Optional parameters: n/a

Encoding considerations: binary

Security considerations: See Section 7

Interoperability considerations: n/a

Published specification: This document

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

Additional information:

Deprecated alias names for this type: n/a

Magic number(s): n/a

File extension(s): n/a

Macintosh file type code(s): n/a

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

Intended usage: COMMON

Restrictions on usage: none

Author: Emily Stark (estark@google.com)

Change controller: IETF

9. References

9.1. Normative References

[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-30 (work in progress), November 2018.

[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>.

[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>.

[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>.

[RFC6839] Hansen, T. and A. Melnikov, "Additional Media Type Structured Syntax Suffixes", RFC 6839, DOI 10.17487/RFC6839, January 2013, <https://www.rfc-editor.org/info/rfc6839>.

[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>.

[RFC7469] Evans, C., Palmer, C., and R. Sleevi, "Public Key Pinning Extension for HTTP", RFC 7469, DOI 10.17487/RFC7469, April 2015, <https://www.rfc-editor.org/info/rfc7469>.

[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>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

9.2. Informative References

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

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

9.3. URIs

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

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

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

[4] https://www.iana.org/assignments/message-headers

Appendix A. Changes

A.1. Since -07

- Editorial changes
- Specify that the end-entity certificate appears first in the "validated-certificate-chain" field of an Expect-CT report.
- Define how report format can be extended by future versions of this specification.

- Add optional "scheme" key to report format.
- Specify exact status codes for report server errors.
- Limit report-uris to HTTPS only.
- Note that this version of Expect-CT is only compatible with RFC 6962 and 6962-bis, not any future versions of CT.

A.2. Since -06

- Editorial changes

A.3. Since -05

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

A.4. Since -04

- Editorial changes

A.5. Since -03

- Editorial changes

A.6. Since -02

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

A.7. Since -01

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

A.8. Since -00

- Editorial changes

- Change Content-Type header of reports to 'application/expect-ct-report+json'
- Update header field syntax to match convention (issue #327)
- Reference RFC 6962-bis instead of RFC 6962

Author's Address

Emily Stark
Google

Email: estark@google.com

HTTP
Internet-Draft
Intended status: Standards Track
Expires: February 25, 2020

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

August 24, 2019

Structured Headers for HTTP
draft-ietf-httpbis-header-structure-13

Abstract

This document describes a set of data types and associated algorithms that are intended to make it easier and safer to define and handle HTTP header fields. It is intended for use by specifications of new HTTP header fields that wish to use a common syntax that is more restrictive than traditional HTTP field values.

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].

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

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

Copyright Notice

Table of Contents

1. Introduction

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

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

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

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

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

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

Section 2 describes how to specify a Structured Header.

Section 3 defines a number of abstract data types that can be used in Structured Headers. Those abstract types can be serialized into and parsed from textual HTTP headers using the algorithms described in Section 4.

1.1. Intentionally Strict Processing

This specification intentionally defines strict parsing and serialisation behaviours using step-by-step algorithms; the only error handling defined is to fail the operation altogether.

It is designed to encourage faithful implementation and therefore good interoperability. Therefore, an implementation that tried to be "helpful" by being more tolerant of input would make interoperability worse, since that would create pressure on other implementations to implement similar (but likely subtly different) workarounds.

In other words, strict processing is an intentional feature of this specification; it allows non-conformant input to be discovered and corrected by the producer early, and avoids both interoperability and security issues that might otherwise result.

Note that as a result of this strictness, if a header field is appended to by multiple parties (e.g., intermediaries, or different components in the sender), an error in one party's value is likely to cause the entire header field to fail parsing.

1.2. Notational Conventions

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

This document uses algorithms to specify parsing and serialisation behaviours, and the Augmented Backus-Naur Form (ABNF) notation of [RFC5234] to illustrate expected syntax in textual HTTP header fields. In doing so, uses the VCHAR, SP, DIGIT, ALPHA and DQUOTE rules from [RFC5234]. It also includes the OWS rule from [RFC7230].

When parsing from textual HTTP header fields, implementations MUST follow the algorithms, but MAY vary in implementation so as the behaviours are indistinguishable from specified behaviour. If there is disagreement between the parsing algorithms and ABNF, the specified algorithms take precedence. In some places, the algorithms are "greedy" with whitespace, but this should not affect conformance.

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

2. Defining New Structured Headers

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

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

- Specify the header field's allowed syntax for values, in terms of the types described in Section 3, along with their associated semantics. Syntax definitions are encouraged to use the ABNF rules beginning with "sh-" defined in this specification; other rules in this specification are not intended for use outside it.

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

Note that a header field definition cannot relax the requirements of this specification because doing so would preclude handling by generic software; they can only add additional constraints (for example, on the numeric range of integers and floats, the format of strings and tokens, the types allowed in a dictionary's values, or the number of items in a list). Likewise, header field definitions can only use Structured Headers for the entire header field value, not a portion thereof.

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

the size of individual header fields, the total number of fields, and/or the size of the entire header block.

For example, a fictitious Foo-Example header field might be specified as:

42. Foo-Example Header

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

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

```
Foo-Example = sh-dictionary
```

The dictionary MUST contain:

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

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

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

"barUrl" contains one or more URI-references (Section 4.1 of [RFC3986], Section 4.1). If barURL is not a valid URI-reference, it MUST be ignored. If barURL is a relative reference (Section 4.2 of [RFC3986]), it MUST be resolved (Section 5 of [RFC3986]) before being used.

For example:

```
Foo-Example: foo=2, barUrl=("https://bar.example.com/")
```

3. Structured Header Data Types

This section defines the abstract value types that can be composed into Structured Headers. The ABNF provided represents the on-wire format in textual HTTP headers.

3.1. Lists

Lists are arrays of zero or more members, each of which can be an item (Section 3.3) or an inner list (an array of zero or more items).

Each member of the top-level list can also have associated parameters - an ordered map of key-value pairs where the keys are short, textual strings and the values are items (Section 3.3). There can be zero or more parameters on a member, and their keys are required to be unique within that scope.

The ABNF for lists is:

```
sh-list       = list-member *( OWS "," OWS list-member )
list-member   = ( sh-item / inner-list ) *parameter
inner-list    = "(" OWS [ sh-item *( SP sh-item ) OWS ] ")"
parameter     = OWS ";" OWS param-name [ "=" param-value ]
param-name    = key
key           = lcalpha *( lcalpha / DIGIT / "_" / "-" / "*" )
lcalpha       = %x61-7A ; a-z
param-value   = sh-item
```

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

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

In textual HTTP headers, inner lists are denoted by surrounding parenthesis, and have their values delimited by a single space. A header field whose value is defined as a list of lists of strings could look like:

```
Example-StrListListHeader: ("foo" "bar"), ("baz"), ("bat" "one"), ()
```

Note that the last member in this example is an empty inner list.

In textual HTTP headers, members' parameters are separated from the member and each other by semicolons. For example:

```
Example-ParamListHeader: abc;a=1;b=2; cde_456, (ghi jkl);q="9";r=w
```

In textual HTTP headers, an empty list is denoted by not serialising the header at all.

Parsers MUST support lists containing at least 1024 members, support members with at least 256 parameters, support inner-lists containing

at least 256 members, and support parameter keys with at least 64 characters.

Header specifications can constrain the types of individual list values (including that of individual inner-list members and parameters) if necessary.

3.2. Dictionaries

Dictionaries are ordered maps of name-value pairs, where the names are short, textual strings and the values are items (Section 3.3) or arrays of items. There can be zero or more members, and their names are required to be unique within the scope of the dictionary they occur within.

Each member of the dictionary can also have associated parameters - an ordered map of key-value pairs where the keys are short, textual strings and the values are items (Section 3.3). There can be zero or more parameters on a member, and their keys are required to be unique within that scope.

Implementations MUST provide access to dictionaries both by index and by name. Specifications MAY use either means of accessing the members.

The ABNF for dictionaries in textual HTTP headers is:

```
sh-dictionary  = dict-member *( OWS "," OWS dict-member )
dict-member    = member-name "=" member-value *parameter
member-name    = key
member-value   = sh-item / inner-list
```

In textual HTTP headers, members are separated by a comma with optional whitespace, while names and values are separated by "=" (without whitespace). For example:

```
Example-DictHeader: en="Applepie", da=*w4ZibGV0w6ZydGU=*
```

A dictionary with a member whose value is an inner-list of tokens:

```
Example-DictListHeader: rating=1.5, feelings=(joy sadness)
```

A dictionary with a mix of singular and list values, some with parameters:

```
Example-MixDict: a=(1,2), b=3, c=4;aa=bb, d=(5,6);valid=?T
```

As with lists, an empty dictionary is represented in textual HTTP headers by omitting the entire header field.

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

Parsers MUST support dictionaries containing at least 1024 name/value pairs, and names with at least 64 characters.

3.3. Items

An item is can be a integer (Section 3.4), float (Section 3.5), string (Section 3.6), token (Section 3.7), byte sequence (Section 3.8), or Boolean (Section 3.9).

The ABNF for items in textual HTTP headers is:

```
sh-item = sh-integer / sh-float / sh-string / sh-token / sh-binary
          / sh-boolean
```

3.4. Integers

Integers have a range of -999,999,999,999,999 to 999,999,999,999,999 inclusive (i.e., up to fifteen digits, signed), for IEEE 754 compatibility ([IEEE754]).

The ABNF for integers in textual HTTP headers is:

```
sh-integer = ["-"] 1*15DIGIT
```

For example:

```
Example-IntegerHeader: 42
```

Note that commas in integers are used in this section's prose only for readability; they are not valid in the wire format.

3.5. Floats

Floats are decimal numbers with an integer and a fractional component. The fractional component has at most six digits of precision. Additionally, like integers, it can have no more than fifteen digits in total, which in some cases further constrains its precision.

The ABNF for floats in textual HTTP headers is:

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

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

```
Example-FloatHeader: 4.5
```

3.6. Strings

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

The ABNF for strings in textual HTTP headers is:

```
sh-string = DQUOTE *(chr) DQUOTE
chr       = unescaped / escaped
unescaped = %x20-21 / %x23-5B / %x5D-7E
escaped   = "\" ( DQUOTE / "\" )
```

In textual HTTP headers, strings are delimited with double quotes, using a backslash ("\") to escape double quotes and backslashes. For example:

```
Example-StringHeader: "hello world"
```

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

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

When it is necessary for a field value to convey non-ASCII string content, a byte sequence (Section 3.8) SHOULD be specified, along with a character encoding (preferably UTF-8).

Parsers MUST support strings with at least 1024 characters.

3.7. Tokens

Tokens are short textual words; their abstract model is identical to their expression in the textual HTTP serialisation.

The ABNF for tokens in textual HTTP headers is:

```
sh-token = ALPHA
           *( ALPHA / DIGIT / "_" / "-" / "." / ":" / "%"
              / "*" / "/" )
```

Parsers MUST support tokens with at least 512 characters.

Note that a Structured Header token is not the same as the "token" ABNF rule defined in [RFC7230].

3.8. Byte Sequences

Byte sequences can be conveyed in Structured Headers.

The ABNF for a byte sequence in textual HTTP headers is:

```
sh-binary = "*" *(base64) "*"
base64    = ALPHA / DIGIT / "+" / "/" / "="
```

In textual HTTP headers, a byte sequence is delimited with asterisks and encoded using base64 ([RFC4648], Section 4). For example:

```
Example-BinaryHdr: *cHJldGVuZCB0aGlzIGlzIGJpbmFyeSBjb250ZW50Lg==*
```

Parsers MUST support byte sequences with at least 16384 octets after decoding.

3.9. Booleans

Boolean values can be conveyed in Structured Headers.

The ABNF for a Boolean in textual HTTP headers is:

```
sh-boolean = "?" boolean
boolean    = "0" / "1"
```

In textual HTTP headers, a boolean is indicated with a leading "?" character. For example:

```
Example-BoolHdr: ?1
```

4. Working With Structured Headers in Textual HTTP Headers

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

4.1. Serializing Structured Headers

Given a structure defined in this specification, return an ASCII string suitable for use in a textual HTTP header value.

1. If the structure is a dictionary or list and its value is empty (i.e., it has no members), do not send the serialize field at all (i.e., omit both the field-name and field-value).

2. If the structure is a dictionary, let output_string be the result of Serializing a Dictionary (Section 4.1.2).

3. Else if the structure is a list, let output_string be the result of Serializing a List (Section 4.1.1).

4. Else if the structure is an item, let output_string be the result of Serializing an Item (Section 4.1.3).

5. Else, fail serialisation.

6. Return output_string converted into an array of bytes, using ASCII encoding [RFC0020].

4.1.1. Serializing a List

Given a list of (member-value, parameters) as input_list, return an ASCII string suitable for use in a textual HTTP header value.

1. Let output be an empty string.

2. For each (member-value, parameters) of input_list:

    1. If member-value is an array, append the result of applying Serialising an Inner List (Section 4.1.1.1) with member-value to output.

    2. Otherwise, append the result of applying Serializing an Item (Section 4.1.3) with member-value to output.

    3. Append the result of Serializing Parameters Section 4.1.1.2 with parameters to output.

4. If more member-values remain in input_plist:

    1. Append a COMMA to output.

    2. Append a single WS to output.

3. Return output.

4.1.1.1. Serialising an Inner List

Given an array as inner_list, return an ASCII string suitable for use in a textual HTTP header value.

1. Let output be the string "(".

2. For each member-value of inner_list:

    1. Append the result of applying Serializing an Item (Section 4.1.3) with member-value to output.

    2. If inner_list is not empty, append a single WS to output.

3. Append ")" to output.

4. Return output.

4.1.1.2. Serializing Parameters

Given an ordered dictionary as input_parameters (each member having a param-name and a param-value), return an ASCII string suitable for use in a textual HTTP header value.

1. Let output be an empty string.

2. For each parameter-name with a value of param-value in input_parameters:

    1. Append ";" to output.

    2. Append the result of applying Serializing a Key (Section 4.1.1.3) with param-name to output.

    3. If param-value is not null:

        1. Append "=" to output.

        2. Append the result of applying Serializing an Item (Section 4.1.3) with param-value to output.

3. Return output.

4.1.1.3. Serializing a Key

Given a key as input_key, return an ASCII string suitable for use in a textual HTTP header value.

1. If input_key is not a sequence of characters, or contains characters not in lcalpha, DIGIT, "*", "_", or "-", fail serialisation.

2. Let output be an empty string.

3. Append input_key to output.

4. Return output.

4.1.2. Serializing a Dictionary

Given an ordered dictionary as input_dictionary (each member having a member-name and a tuple value of (member-value, parameters)), return an ASCII string suitable for use in a textual HTTP header value.

1. Let output be an empty string.

2. For each member-name with a value of (member-value, parameters) in input_dictionary:

    1. Append the result of applying Serializing a Key (Section 4.1.1.3) with member's member-name to output.

    2. Append "=" to output.

    3. If member-value is an array, append the result of applying Serialising an Inner List (Section 4.1.1.1) with member-value to output.

    4. Otherwise, append the result of applying Serializing an Item (Section 4.1.3) with member-value to output.

    5. Append the result of Serializing Parameters Section 4.1.1.2 with parameters to output.

    6. If more members remain in input_dictionary:

        1. Append a COMMA to output.

        2. Append a single WS to output.

3. Return output.

4.1.3. Serializing an Item

Given an item as input_item, return an ASCII string suitable for use in a textual HTTP header value.

1. If input_item is an integer, return the result of applying Serializing an Integer (Section 4.1.4) to input_item.

2. If input_item is a float, return the result of applying Serializing a Float (Section 4.1.5) to input_item.

3. If input_item is a string, return the result of applying Serializing a String (Section 4.1.6) to input_item.

4. If input_item is a token, return the result of Serializing a Token (Section 4.1.7) to input_item.

5. If input_item is a Boolean, return the result of applying Serializing a Boolean (Section 4.1.9) to input_item.

6. If input_item is a byte sequence, return the result of applying Serializing a Byte Sequence (Section 4.1.8) to input_item.

7. Otherwise, fail serialisation.

4.1.4. Serializing an Integer

Given an integer as input_integer, return an ASCII string suitable for use in a textual HTTP header value.

1. If input_integer is not an integer in the range of -999,999,999,999,999 to 999,999,999,999,999 inclusive, fail serialisation.

2. Let output be an empty string.

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

4. Append input_integer’s numeric value represented in base 10 using only decimal digits to output.

5. Return output.

4.1.5. Serializing a Float

Given a float as input_float, return an ASCII string suitable for use in a textual HTTP header value.

1. Let output be an empty string.
2. If input_float is less than (but not equal to) 0, append "-" to output.
3. Append input_float's integer component represented in base 10 (using only decimal digits) to output; if it is zero, append "0".
4. Let integer_digits be the number of characters appended in the previous step.
5. If integer_digits is greater than 14, fail serialisation.
6. Let digits_avail be 15 minus integer_digits.
7. Let fractional_digits_avail be the minimum of digits_avail and 6.
8. Append "." to output.
9. Append at most fractional_digits_avail digits of input_float's fractional component represented in base 10 to output (using only decimal digits, and truncating any remaining digits); if it is zero, append "0".
10. Return output.

4.1.6. Serializing a String

Given a string as input_string, return an ASCII string suitable for use in a textual HTTP header value.

1. If input_string is not a sequence of characters, or contains characters outside the range %x00-1f or %x7f (i.e., is not in VCHAR or SP), fail serialisation.
2. Let output be an empty string.
3. Append DQUOTE to output.
4. For each character char in input_string:

1. If char is "\" or DQUOTE:

    1. Append "\" to output.

2. Append char to output.

5. Append DQUOTE to output.

6. Return output.

4.1.7. Serializing a Token

Given a token as input_token, return an ASCII string suitable for use in a textual HTTP header value.

1. If input_token is not a sequence of characters, or contains characters not in ALPHA, DIGIT, "_", "-", ".", ":", "%", "*" or "/", fail serialisation.

2. Let output be an empty string.

3. Append input_token to output.

4. Return output.

4.1.8. Serializing a Byte Sequence

Given a byte sequence as input_bytes, return an ASCII string suitable for use in a textual HTTP header value.

1. If input_bytes is not a sequence of bytes, fail serialisation.

2. Let output be an empty string.

3. Append "*" to output.

4. Append the result of base64-encoding input_bytes as per [RFC4648], Section 4, taking account of the requirements below.

5. Append "*" to output.

6. Return output.

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

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

4.1.9. Serializing a Boolean

Given a Boolean as input_boolean, return an ASCII string suitable for use in a textual HTTP header value.

1. If input_boolean is not a boolean, fail serialisation.
2. Let output be an empty string.
3. Append "?" to output.
4. If input_boolean is true, append "1" to output.
5. If input_boolean is false, append "0" to output.
6. Return output.

4.2. Parsing Header Fields into Structured Headers

When a receiving implementation parses textual HTTP header fields 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 array of bytes input_bytes that represents the chosen header's field-value (which is an empty string if that header is not present), and header_type (one of "dictionary", "list", or "item"), return the parsed header value.

1. Convert input_bytes into an ASCII string input_string; if conversion fails, fail parsing.
2. Discard any leading OWS from input_string.
3. If header_type is "list", let output be the result of Parsing a List from Text (Section 4.2.1).
4. If header_type is "dictionary", let output be the result of Parsing a Dictionary from Text (Section 4.2.2).
5. If header_type is "item", let output be the result of Parsing an Item from Text (Section 4.2.3).

6. Discard any leading OWS from input_string.

7. If input_string is not empty, fail parsing.

8. Otherwise, return output.

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

For Lists and Dictionaries, this has the effect of correctly concatenating all instances of the header field, as long as individual individual members of the top-level data structure are not split across multiple header instances.

Strings split across multiple header instances will have unpredictable results, because comma(s) and whitespace inserted upon combination will become part of the string output by the parser. Since concatenation might be done by an upstream intermediary, the results are not under the control of the serializer or the parser.

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

If parsing fails - including when calling another algorithm - the entire header field's value MUST be ignored (i.e., treated as if the header field were not present in the message). This is intentionally strict, to improve interoperability and safety, and specifications referencing this document are not allowed to loosen this requirement.

Note that this requirement does not apply to an implementation that is not parsing the header field; for example, an intermediary is not required to strip a failing header field from a message before forwarding it.

4.2.1. Parsing a List from Text

Given an ASCII string as input_string, return an array of (member, parameters). input_string is modified to remove the parsed value.

1. Let members be an empty array.

2. While input_string is not empty:

    1. Let member be the result of running Parsing a Parameterized Member from Text (Section 4.2.1.1) with input_string.

2. Append member to members.

3. Discard any leading OWS from input_string.

4. If input_string is empty, return members.

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

6. Discard any leading OWS from input_string.

7. If input_string is empty, there is a trailing comma; fail parsing.

3. No structured data has been found; return members (which is empty).

4.2.1.1. Parsing a Parameterized Member from Text

Given an ASCII string as input_string, return a member (either a list of items, or a single item) with an ordered map of parameters. input_string is modified to remove the parsed value.

1. If the first character of input_string is "(", let member be the result of running Parsing an Inner List (Section 4.2.1.2) with input_string.

2. Else, let member be the result of running Parsing an Item (Section 4.2.3) with input_string.

3. Let parameters be an empty, ordered map.

4. While input_string is not empty:

    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 key from Text (Section 4.2.1.3) from input_string.

    6. If param_name is already present in parameters, there is a duplicate; fail parsing.

7. Let param_value be a null value.

8. If the first character of input_string is "=":

    1. Consume the "=" character at the beginning of input_string.

    2. Let param_value be the result of Parsing an Item from Text (Section 4.2.3) from input_string.

9. Append key param_name with value param_value to parameters.

5. Return the tuple (member, parameters).

4.2.1.2. Parsing an Inner List

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

1. Consume the first character of input_string; if it is not "(", fail parsing.

2. Let inner_list be an empty array.

3. While input_string is not empty:

    1. Discard any leading OWS from input_string.

    2. If the first character of input_string is ")":

        1. Consume the first character of input_string.

        2. Return inner_list.

    3. Let item be the result of running Parsing an Item from Text (Section 4.2.3) with input_string.

    4. Append item to inner_list.

    5. If the first character of input_string is not SP or ")", fail parsing.

4. The end of the inner list was not found; fail parsing.

4.2.1.3. Parsing a Key from Text

Given an ASCII string as input_string, return a key. 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. If the first character of input_string is not one of lcalpha, DIGIT, "*", "_", or "-", return output_string.
    2. Let char be the result of removing the first character of input_string.
    3. Append char to output_string.
4. Return output_string.

4.2.2. Parsing a Dictionary from Text

Given an ASCII string as input_string, return an ordered map of (key, item). input_string is modified to remove the parsed value.

1. Let dictionary be an empty, ordered map.
2. While input_string is not empty:
    1. Let this_key be the result of running Parsing a Key from Text (Section 4.2.1.3) with input_string.
    2. If dictionary already contains the name this_key, there is a duplicate; fail parsing.
    3. Consume the first character of input_string; if it is not "=", fail parsing.
    4. Let member be the result of running Parsing a Parameterized Member from Text (Section 4.2.1.1) with input_string.
    5. Add name this_key with value member to dictionary.
    6. Discard any leading OWS from input_string.
    7. If input_string is empty, return dictionary.

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

9. Discard any leading OWS from input_string.

10. If input_string is empty, there is a trailing comma; fail parsing.

3. No structured data has been found; return dictionary (which is empty).

4.2.3. Parsing an Item from Text

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

1. If the first character of input_string is a "-" or a DIGIT, process input_string as a number (Section 4.2.4) and return the result.

2. If the first character of input_string is a DQUOTE, process input_string as a string (Section 4.2.5) and return the result.

3. If the first character of input_string is "*", process input_string as a byte sequence (Section 4.2.7) and return the result.

4. If the first character of input_string is "?", process input_string as a Boolean (Section 4.2.8) and return the result.

5. If the first character of input_string is an ALPHA, process input_string as a token (Section 4.2.6) and return the result.

6. Otherwise, the item type is unrecognized; fail parsing.

4.2.4. Parsing a Number from Text

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

NOTE: This algorithm parses both Integers (Section 3.4) and Floats (Section 3.5), and returns the corresponding structure.

1. Let type be "integer".

2. Let sign be 1.

3. Let input_number be an empty string.

4. If the first character of input_string is "-", consume it and set sign to -1.

5. If input_string is empty, there is an empty integer; fail parsing.

6. If the first character of input_string is not a DIGIT, fail parsing.

7. While input_string is not empty:

    1. Let char be the result of consuming the first character of input_string.

    2. If char is a DIGIT, append it to input_number.

    3. Else, if type is "integer" and char is ".", append char to input_number and set type to "float".

    4. Otherwise, prepend char to input_string, and exit the loop.

    5. If type is "integer" and input_number contains more than 15 characters, fail parsing.

    6. If type is "float" and input_number contains more than 16 characters, fail parsing.

8. If type is "integer":

    1. Parse input_number as an integer and let output_number be the product of the result and sign.

    2. If output_number is outside the range defined in Section 3.4, fail parsing.

9. Otherwise:

    1. If the final character of input_number is ".", fail parsing.

    2. If the number of characters after "." in input_number is greater than six, fail parsing.

    3. Parse input_number as a float and let output_number be the product of the result and sign.

10. Return output_number.

4.2.5. Parsing a String from Text

Given an ASCII string as 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 consuming the first character of input_string.
    2. If char is a backslash ("\"):
        1. If input_string is now empty, fail parsing.
        2. Let next_char be the result of consuming the first character of input_string.
        3. If next_char is not DQUOTE or "\", fail parsing.
        4. Append next_char to output_string.
    3. Else, if char is DQUOTE, return output_string.
    4. Else, if char is in the range %x00-1f or %x7f (i.e., is not in VCHAR or SP), fail parsing.
    5. Else, append char to output_string.
5. Reached the end of input_string without finding a closing DQUOTE; fail parsing.

4.2.6. Parsing a Token from Text

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

1. If the first character of input_string is not ALPHA, fail parsing.
2. Let output_string be an empty string.

3. While input_string is not empty:

    1. If the first character of input_string is not one of ALPHA, DIGIT, "_", "-", ".", ":", "%", "*" or "/", return output_string.

    2. Let char be the result of consuming the first character of input_string.

    3. Append char to output_string.

4. Return output_string.

4.2.7. Parsing a Byte Sequence from Text

Given an ASCII string as input_string, return a byte sequence. 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. If there is not a "*" character before the end of input_string, fail parsing.

4. Let b64_content be the result of consuming content of input_string up to but not including the first instance of the character "*".

5. Consume the "*" character at the beginning of input_string.

6. If b64_content contains a character not included in ALPHA, DIGIT, "+", "/" and "=", fail parsing.

7. Let binary_content be the result of Base 64 Decoding [RFC4648] b64_content, synthesizing padding if necessary (note the requirements about recipient behaviour below).

8. Return binary_content.

Because some implementations of base64 do not allow reject of encoded data that is not properly "=" padded (see [RFC4648], Section 3.2), parsers SHOULD NOT fail when it is not present, unless they cannot be configured to do so.

Because some implementations of base64 do not allow rejection of encoded data that has non-zero pad bits (see [RFC4648], Section 3.5),

parsers SHOULD NOT fail when it is present, unless they cannot be configured to do so.

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.

4.2.8. Parsing a Boolean from Text

Given an ASCII string as input_string, return a Boolean. 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. If the first character of input_string matches "1", discard the first character, and return true.
4. If the first character of input_string matches "0", discard the first character, and return false.
5. No value has matched; fail parsing.

5. IANA Considerations

This draft has no actions for IANA.

6. Security Considerations

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

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

7. References

7.1. Normative References

[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", IEEE 754-2008, DOI 10.1109/IEEESTD.2008.4610935, ISBN 978-0-7381-5752-8, August 2008, <http://ieeexplore.ieee.org/document/4610935/>.

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

[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>.

[RFC7493] Bray, T., Ed., "The I-JSON Message Format", RFC 7493, DOI 10.17487/RFC7493, March 2015, <https://www.rfc-editor.org/info/rfc7493>.

[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>.

[RFC7541] Peon, R. and H. Ruellan, "HPACK: Header Compression for HTTP/2", RFC 7541, DOI 10.17487/RFC7541, May 2015, <https://www.rfc-editor.org/info/rfc7541>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

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

[4] https://github.com/httpwg/structured-header-tests

[5] https://github.com/httpwg/wiki/wiki/Structured-Headers

[6] https://github.com/httpwg/structured-header-tests

Appendix A. Acknowledgements

Many thanks to Matthew Kerwin for his detailed feedback and careful consideration during the development of this specification.

Appendix B. Frequently Asked Questions

B.1. Why not JSON?

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

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

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

Another example is JSON's ability to nest content to arbitrary depths. Since the resulting memory commitment might be unsuitable

(e.g., in embedded and other limited server deployments), it's necessary to limit it in some fashion; however, existing JSON implementations have no such limits, and even if a limit is specified, it's likely that some header field definition will find a need to violate it.

Because of JSON's broad adoption and implementation, it is difficult to impose such additional constraints across all implementations; some deployments would fail to enforce them, thereby harming interoperability. In short, if it looks like JSON, people will be tempted to use a JSON parser / serialiser on header fields.

Since a major goal for Structured Headers is to improve interoperability and simplify implementation, these concerns led to a format that requires a dedicated parser and serializer.

Additionally, there were widely shared feelings that JSON doesn't "look right" in HTTP headers.

B.2. Structured Headers don't "fit" my data.

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

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

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

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

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

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

Appendix C. Implementation Notes

A generic implementation of this specification should expose the top-level parse (Section 4.2) and serialize (Section 4.1) functions. They need not be functions; for example, it could be implemented as an object, with methods for each of the different top-level types.

For interoperability, it's important that generic implementations be complete and follow the algorithms closely; see Section 1.1. To aid this, a common test suite is being maintained by the community at https://github.com/httpwg/structured-header-tests [7].

Implementers should note that dictionaries and parameters are order-preserving maps. Some headers may not convey meaning in the ordering of these data types, but it should still be exposed so that applications which need to use it will have it available.

Likewise, implementations should note that it's important to preserve the distinction between tokens and strings. While most programming languages have native types that map to the other types well, it may be necessary to create a wrapper "token" object or use a parameter on functions to assure that these types remain separate.

Appendix D. Changes

RFC Editor: Please remove this section before publication.

D.1. Since draft-ietf-httpbis-header-structure-12

- Editorial improvements.
- Reworked float serialisation (#896).

D.2. Since draft-ietf-httpbis-header-structure-11

- Allow * in key (#844).
- Constrain floats to six digits of precision (#848).
- Allow dictionary members to have parameters (#842).

D.3. Since draft-ietf-httpbis-header-structure-10

- Update abstract (#799).
- Input and output are now arrays of bytes (#662).

- Implementations need to preserve difference between token and string (#790).
- Allow empty dictionaries and lists (#781).
- Change parameterized lists to have primary items (#797).
- Allow inner lists in both dictionaries and lists; removes lists of lists (#816).
- Subsume Parameterised Lists into Lists (#839).

D.4. Since draft-ietf-httpbis-header-structure-09

- Changed Boolean from T/F to 1/0 (#784).
- Parameters are now ordered maps (#765).
- Clamp integers to 15 digits (#737).

D.5. Since draft-ietf-httpbis-header-structure-08

- Disallow whitespace before items properly (#703).
- Created "key" for use in dictionaries and parameters, rather than relying on identifier (#702). Identifiers have a separate minimum supported size.
- Expanded the range of special characters allowed in identifier to include all of ALPHA, ".", ":", and "%" (#702).
- Use "?" instead of "!" to indicate a Boolean (#719).
- Added "Intentionally Strict Processing" (#684).
- Gave better names for referring specs to use in Parameterised Lists (#720).
- Added Lists of Lists (#721).
- Rename Identifier to Token (#725).
- Add implementation guidance (#727).

D.6. Since draft-ietf-httpbis-header-structure-07

- Make Dictionaries ordered mappings (#659).
- Changed "binary content" to "byte sequence" to align with Infra specification (#671).
- Changed "mapping" to "map" for #671.
- Don't fail if byte sequences aren't "=" padded (#658).
- Add Booleans (#683).
- Allow identifiers in items again (#629).
- Disallowed whitespace before items (#703).
- Explain the consequences of splitting a string across multiple headers (#686).

D.7. Since draft-ietf-httpbis-header-structure-06

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

D.8. Since draft-ietf-httpbis-header-structure-05

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

D.9. Since draft-ietf-httpbis-header-structure-04

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

D.10. Since draft-ietf-httpbis-header-structure-03

- Strengthen language around failure handling.

D.11. 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.

D.12. Since draft-ietf-httpbis-header-structure-01

- Replaced with draft-nottingham-structured-headers.

D.13. 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
Internet-Draft
Intended status: Experimental
Expires: September 7, 2019

C. Pratt

D. Thakore
CableLabs

B. Stark
AT&T

March 6, 2019

HTTP Random Access and Live Content
draft-ietf-httpbis-rand-access-live-04

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 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 September 7, 2019.

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, [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 accessible at the time the request was processed by the server. The client can infer from this response that bytes 0-1234567 of the representation can be requested and returned in a timely fashion (the bytes are immediately available).

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

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

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

Unlike a byte-range Range request, a byte-range Content-Range response header field cannot be "open ended", per [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 an 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, and much larger than the expected maximum size of the representation. See Section 6 for range value considerations.

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 [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. Recommendations for Very Large Values

While it would be ideal to define a single numerical Very Large Value, there's no single value that would work for all applications and platforms. e.g. JavaScript numbers cannot represent all integer values above 2^^53, so a JavaScript application may want to use 2^^53-1 for a Very Large Value. This value, however, would not be sufficient for all applications, such as continuously-streaming high-bitrate streams. So the value 2^^53-1 (9007199254740991) is recommended as a Very Large Value unless an application has a good justification to use a smaller or larger value. e.g. If it's always known that the resource won't exceed a value smaller than the recommended Very Large Value for an application, a smaller value can be used. And if it's likely that an application will utilize resources larger than the recommended Very Large Value - such as a continuously aggregating high-bitrate media stream - a larger value should be used.

Note that, in accordance with the semantics defined above, servers that support random-access live content will need to return the last-byte-pos provided in the Range request in some cases - even if the last-byte-pos cannot be represented as a numerical value internally by the server. As is the case with any live/continuously aggregating resource, the server should terminate the content transfer when the end of the resource is reached - whether the end is due to termination of the content source or the content length exceeds the server's maximum representation length.

5. IANA Considerations

This document has no actions for IANA.

6. Security Considerations

As described above, servers need to be prepared to receive last-byte-pos values in Range requests that are numerically larger than the server implementation supports - and return these values in Content-Range response header fields. Servers should check the last-byte-pos value before converting and storing them into numeric form to ensure the value doesn't cause an overflow or index incorrect data. The simplest way to satisfy the live-range semantics defined in this document without potential overflow issues is to store the last-byte-pos as a string value and return it in the byte-range Content-Range response header's last-byte-pos field.

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>.

[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>.

7.2. 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>.

Appendix A. 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
Internet-Draft
Intended status: Standards Track
Expires: December 29, 2018

M. Thomson
Mozilla
M. Nottingham
Fastly
W. Tarreau
HAProxy Technologies

June 27, 2018

Using Early Data in HTTP
draft-ietf-httpbis-replay-04

Abstract

Using TLS early data creates an exposure to the possibility of a replay attack. This document defines mechanisms that allow clients to communicate with servers about HTTP requests that are sent in early data. Techniques are described that use these mechanisms to mitigate the risk of replay.

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

Status of This Memo

Table of Contents

1. Introduction

TLS 1.3 [TLS13] introduces the concept of early data (also known as zero round trip data or 0-RTT data). Early data allows a client to send data to a server in the first round trip of a connection, without waiting for the TLS handshake to complete, if the client has spoken to the same server recently.

When used with HTTP [HTTP], early data allows clients to send requests immediately, avoiding the one or two round trip delay needed

for the TLS handshake. This is a significant performance enhancement; however, it has significant limitations.

The primary risk of using early data is that an attacker might capture and replay the request(s) it contains. TLS [TLS13] describes techniques that can be used to reduce the likelihood that an attacker can successfully replay a request, but these techniques can be difficult to deploy, and still leave some possibility of a successful attack.

Note that this is different from automated or user-initiated retries; replays are initiated by an attacker without the awareness of the client.

To help mitigate the risk of replays in HTTP, this document gives an overview of techniques for controlling these risks in servers, and defines requirements for clients when sending requests in early data.

The advice in this document also applies to use of 0-RTT in HTTP over QUIC [HQ].

1.1. Conventions and Definitions

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. Early Data in HTTP

Conceptually, early data is concatenated with other application data to form a single stream. This can mean that requests are entirely contained within early data, or only part of a request is early. In a multiplexed protocol, like HTTP/2 [RFC7540] or HTTP/QUIC [HQ], multiple requests might be partially delivered in early data.

The model that this document assumes is that once the TLS handshake completes, the early data received on that TLS connection is known to not be a replayed copy of that data. However, it is important to note that this does not mean that early data will not be or has not been replayed on another connection.

3. Supporting Early Data in HTTP Servers

A server decides whether or not to offer a client the ability to send early data on future connections when sending the TLS session ticket.

TLS [TLS13] mandates the use of replay detection strategies that reduce the ability of an attacker to successfully replay early data. These anti-replay techniques reduce but don't completely eliminate the chance of data being replayed and ensure a fixed upper limit to the number of replays.

When a server enables early data, there are a number of techniques it can use to mitigate the risks of replay:

1. The server can reject early data at the TLS layer. A server cannot selectively reject early data, so this results in all requests sent in early data being discarded.

2. The server can choose to delay processing of early data until after the TLS handshake completes. By deferring processing, it can ensure that only a successfully completed connection is used for the request(s) therein. This provides the server with some assurance that the early data was not replayed. If the server receives multiple requests in early data, it can determine whether to defer HTTP processing on a per-request basis.

3. The server can cause a client to retry individual requests and not use early data by responding with the 425 (Too Early) status code (Section 5.2), in cases where the risk of replay is judged too great.

Any of these techniques is equally effective and a server can use the method that best suits it.

For a given request, the level of tolerance to replay risk is specific to the resource it operates upon (and therefore only known to the origin server). The primary risk associated with using early data is in the actions a server takes when processing a request; processing a duplicated request might result in duplicated effects and side effects. Appendix E.5 of [TLS13] also describes other effects produced by processing duplicated requests.

The request method's safety ([RFC7231], Section 4.2.1) is one way to determine this. However, some resources do produce side effects with safe methods, so this cannot be universally relied upon.

It is RECOMMENDED that origin servers allow resources to explicitly configure whether early data is appropriate in requests. Absent such explicit information, origin servers MUST either reject early data or implement the techniques described in this document for ensuring that requests are not processed prior to TLS handshake completion.

A request might be sent partially in early data with the remainder of the request being sent after the handshake completes. This does not necessarily affect handling of that request; what matters is when the server starts acting upon the contents of a request. Any time any server instance might initiate processing prior to completion of the handshake, all server instances need to account for the possibility of replay of early data and how that could affect that processing (see also Section 6.2).

A server can partially process requests that are incomplete. Parsing header fields - without acting on the values - and determining request routing is likely to be safe from side-effects, but other actions might not be.

Intermediary servers do not have sufficient information to decide whether early data can be processed, so Section 5.2 describes a way for the origin to signal to them that a particular request isn't appropriate for early data. Intermediaries that accept early data MUST implement that mechanism.

Note that a server cannot choose to selectively reject early data at the TLS layer. TLS only permits a server to accept all early data, or none of it. Once a server has decided to accept early data, it MUST process all requests in early data, even if the server rejects the request by sending a 425 (Too Early) response.

A server can limit the amount of early data with the "max_early_data_size" field of the "early_data" TLS extension. This can be used to avoid committing an arbitrary amount of memory for requests that it might defer until the handshake completes.

4. Using Early Data in HTTP Clients

A client that wishes to use early data commences sending HTTP requests immediately after sending the TLS ClientHello.

By their nature, clients have control over whether a given request is sent in early data - thereby giving the client control over risk of replay. Absent other information, clients MAY send requests with safe HTTP methods (see [RFC7231], Section 4.2.1) in early data when it is available, and MUST NOT send unsafe methods (or methods whose safety is not known) in early data.

If the server rejects early data at the TLS layer, a client MUST start sending again as though the connection were new. This could entail using a different negotiated protocol [ALPN] than the one optimistically used for the early data. Any requests sent in early

data will need to be sent again, unless the client decides to abandon those requests.

Automatic retry creates the potential for a replay attack. An attacker intercepts a connection that uses early data and copies the early data to another server instance. The second server instance accepts and processes the early data, even though it will not complete the TLS handshake. The attacker then allows the original connection to complete. Even if the early data is detected as a duplicate and rejected, the first server instance might allow the connection to complete. If the client then retries requests that were sent in early data, the request will be processed twice.

Replays are also possible if there are multiple server instances that will accept early data, or if the same server accepts early data multiple times (though the latter would be in violation of requirements in Section 8 of [TLS13]).

Clients that use early data MUST retry requests upon receipt of a 425 (Too Early) status code; see Section 5.2.

An intermediary MUST NOT use early data when forwarding a request unless early data was used on a previous hop, or it knows that the request can be retried safely without consequences (typically, using out-of-band configuration). Absent better information, that means that an intermediary can only use early data if the request either arrived in early data or arrived with the "Early-Data" header field set to "1" (see Section 5.1).

5. Extensions for Early Data in HTTP

Because HTTP requests can span multiple "hops", it is necessary to explicitly communicate whether a request has been sent in early data on a previous hop. Likewise, some means of explicitly triggering a retry when early data is not desirable is necessary. Finally, it is necessary to know whether the client will actually perform such a retry.

To meet these needs, two signalling mechanisms are defined:

- The "Early-Data" header field is included in requests that might have been forwarded by an intermediary prior to the TLS handshake with its client completing.

- The 425 (Too Early) status code is defined for a server to indicate that a request could not be processed due to the consequences of a possible replay attack.

They are designed to enable better coordination of the use of early data between the user agent and origin server, and also when a gateway (also "reverse proxy", "Content Delivery Network", or "surrogate") is present.

Gateways typically don’t have specific information about whether a given request can be processed safely when it is sent in early data. In many cases, only the origin server has the necessary information to decide whether the risk of replay is acceptable. These extensions allow coordination between a gateway and its origin server.

5.1. The Early-Data Header Field

The "Early-Data" request header field indicates that the request has been conveyed in early data, and additionally indicates that a client understands the 425 (Too Early) status code.

It has just one valid value: "1". Its syntax is defined by the following ABNF [ABNF]:

```
Early-Data = "1"
```

For example:

```
GET /resource HTTP/1.0
Host: example.com
Early-Data: 1
```

An intermediary that forwards a request prior to the completion of the TLS handshake with its client MUST send it with the "Early-Data" header field set to "1" (i.e., it adds it if not present in the request). An intermediary MUST use the "Early-Data" header field if it - or another instance (see Section 6.2) - could have forwarded the request prior to handshake completion if circumstances were different.

An intermediary MUST NOT remove this header field if it is present in a request. "Early-Data" MUST NOT appear in a "Connection" header field.

The "Early-Data" header field is not intended for use by user agents (that is, the original initiator of a request). Sending a request in early data implies that the client understands this specification and is willing to retry a request in response to a 425 (Too Early) status code. A user agent that sends a request in early data does not need to include the "Early-Data" header field.

A server cannot make a request that contains the Early-Data header field safe for processing by waiting for the handshake to complete. A request that is marked with Early-Data was sent in early data on a previous hop. Requests that contain the Early-Data field and cannot be safely processed MUST be rejected using the 425 (Too Early) status code.

The "Early-Data" header field carries a single bit of information and clients MUST include at most one instance. Multiple or invalid instances of the header field MUST be treated as equivalent to a single instance with a value of 1 by a server.

A "Early-Data" header field MUST NOT be included in responses or request trailers.

5.2. The 425 (Too Early) Status Code

A 425 (Too Early) status code indicates that the server is unwilling to risk processing a request that might be replayed.

User agents that send a request in early data are expected to retry the request when receiving a 425 (Too Early) response status code. A user agent MAY do so automatically, but any retries MUST NOT be sent in early data.

In all cases, an intermediary can forward a 425 (Too Early) status code. Intermediaries MUST forward a 425 (Too Early) status code if the request that it received and forwarded contained an "Early-Data" header field. Otherwise, an intermediary that receives a request in early data MAY automatically retry that request in response to a 425 (Too Early) status code, but it MUST wait for the TLS handshake to complete on the connection where it received the request.

The server cannot assume that a client is able to retry a request unless the request is received in early data or the "Early-Data" header field is set to "1". A server SHOULD NOT emit the 425 status code unless one of these conditions is met.

The 425 (Too Early) status code is not cacheable by default. Its payload is not the representation of any identified resource.

6. Security Considerations

Using early data exposes a client to the risk that their request is replayed. A retried or replayed request can produce different side effects on the server. In addition to those side effects, replays and retries might be used for traffic analysis to recover information about requests or the resources those requests target. In

particular, a request that is replayed might result in a different response, which might be observable from the length of protected data even if the content remains confidential.

6.1. Gateways and Early Data

A gateway MUST NOT forward requests that were received in early data unless it knows that the origin server it will forward to understands the "Early-Data" header field and will correctly generate a 425 (Too Early) status code. A gateway that is uncertain about origin server support for a given request SHOULD either delay forwarding the request until the TLS handshake with its client completes, or send a 425 (Too Early) status code in response.

A gateway without at least one potential origin server that supports "Early-Data" header field expends significant effort for what can at best be a modest performance benefit from enabling early data. If no origin server supports early data, disabling early data entirely is more efficient.

6.2. Consistent Handling of Early Data

Consistent treatment of a request that arrives in - or partially in - early data is critical to avoiding inappropriate processing of replayed requests. If a request is not safe to process before the TLS handshake completes, then all instances of the server (including gateways) need to agree and either reject the request or delay processing.

Disabling early data, delaying requests, or rejecting requests with the 425 (Too Early) status code are all equally good measures for mitigating replay attacks on requests that might be vulnerable to replay. Server instances can implement any of these measures and be considered to be consistent, even if different instances use different methods. Critically, this means that it is possible to employ different mitigations in reaction to other conditions, such as server load.

A server MUST NOT act on early data before the handshake completes if it and any other server instance could make a different decision about how to handle the same data.

6.3. Denial of Service

Accepting early data exposes a server to potential denial of service through the replay of requests that are expensive to handle. A server that is under load SHOULD prefer rejecting TLS early data as a whole rather than accepting early data and selectively processing

requests. Generating a 503 (Service Unavailable) or 425 (Too Early) status code often leads to clients retrying requests, which could result in increased load.

6.4. Out of Order Delivery

In protocols that deliver data out of order (such as QUIC [HQ]) early data can arrive after the handshake completes. A server MAY process requests received in early data after handshake completion only if it can rely on other instances correctly handling replays of the same requests.

7. IANA Considerations

This document registers the "Early-Data" header field in the "Message Headers" registry located at https://www.iana.org/assignments/message-headers [4].

Header field name: Early-Data

Applicable protocol: http

Status: standard

Author/Change controller: IETF

Specification document(s): This document

Related information: (empty)

This document registers the 425 (Too Early) status code in the "Hypertext Transfer Protocol (HTTP) Status Code" registry located at https://www.iana.org/assignments/http-status-codes [5].

Value: 425

Description: Too Early

Reference: This document

8. References

8.1. Normative References

[ABNF] 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>.

[HTTP] 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>.

[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>.

[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>.

[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>.

[TLS13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-22 (work in progress), November 2017.

8.2. Informative References

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

[HQ] Bishop, M., "Hypertext Transfer Protocol (HTTP) over QUIC", draft-ietf-quic-http-08 (work in progress), December 2017.

[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>.

8.3. URIs

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

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

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

[4] https://www.iana.org/assignments/message-headers

[5] https://www.iana.org/assignments/http-status-codes

Acknowledgments

This document was not easy to produce. The following people made substantial contributions to the quality and completeness of the document: David Benjamin, Subodh Iyengar, Benjamin Kaduk, Ilari Liusavaara, Kazuho Oku, Eric Rescorla, Kyle Rose, and Victor Vasiliev.

Authors' Addresses

Martin Thomson
Mozilla

Email: martin.thomson@gmail.com

Mark Nottingham
Fastly

Email: mnot@mnot.net

Willy Tarreau
HAProxy Technologies

Email: willy@haproxy.org

HTTP Working Group
Internet-Draft
Obsoletes: 6265 (if approved)
Intended status: Standards Track
Expires: October 29, 2019

A. Barth
M. West
Google, Inc

April 27, 2019

Cookies: HTTP State Management Mechanism
draft-ietf-httpbis-rfc6265bis-03

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/ [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/6265bis [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 October 29, 2019.

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 ([RFC7230], Section 2).

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 refers to "request-target" as defined in Section 5.3 of [RFC7230].

Two sequences of octets are said to case-insensitively match each other if and only if they are equivalent under the i;ascii-casemap collation defined in [RFC4790].

The term string means a sequence of non-NUL octets.

The terms "active document", "ancestor browsing context", "browsing context", "dedicated worker", "Document", "WorkerGlobalScope", "sandboxed origin browsing context flag", "parent browsing context", "shared worker", "the worker's Documents", "nested browsing context", and "top-level browsing context" are defined in [HTML].

"Service Workers" are defined in the Service Workers specification [SERVICE-WORKERS].

The term "origin", the mechanism of deriving an origin from a URI, and the "the same" matching algorithm for origins are defined in [RFC6454].

"Safe" HTTP methods include "GET", "HEAD", "OPTIONS", and "TRACE", as defined in Section 4.2.1 of [RFC7231].

The term "public suffix" is defined in a note in Section 5.3 of [RFC6265] as "a domain that is controlled by a public registry", and are also known as "effective top-level domains" (eTLDs). For example, "example.com"'s public suffix is "com". User agents SHOULD use an up-to-date public suffix list, such as the one maintained by Mozilla at [PSL].

An origin's "registered domain" is the origin's host's public suffix plus the label to its left. That is, for "https://www.example.com", the public suffix is "com", and the registered domain is "example.com". This concept is defined more rigorously in [PSL], and is also known as "effective top-level domain plus one" (eTLD+1).

The term "request", as well as a request's "client", "current url", "method", and "target browsing context", are defined in [FETCH].

3. Overview

This section outlines a way for an origin server to send state information to a user agent and for the user agent to return the state information to the origin server.

To store state, the origin server includes a Set-Cookie header in an HTTP response. In subsequent requests, the user agent returns a Cookie request header to the origin server. The Cookie header contains cookies the user agent received in previous Set-Cookie headers. The origin server is free to ignore the Cookie header or use its contents for an application-defined purpose.

Origin servers MAY send a Set-Cookie response header with any response. User agents MAY ignore Set-Cookie headers contained in responses with 100-level status codes but MUST process Set-Cookie headers contained in other responses (including responses with 400- and 500-level status codes). An origin server can include multiple Set-Cookie header fields in a single response. The presence of a Cookie or a Set-Cookie header field does not preclude HTTP caches from storing and reusing a response.

Origin servers SHOULD NOT fold multiple Set-Cookie header fields into a single header field. The usual mechanism for folding HTTP headers fields (i.e., as defined in Section 3.2.2 of [RFC7230]) 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 [RFC7230], Section 3.2.6>

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  =
    <IMF-fixdate, defined in [RFC7231], Section 7.1.1.1>
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" / "None"
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 value is "None", the cookie will be sent with same-site and cross-site requests. If the "SameSite" attribute's value is something other than these three known keywords, the attribute's value will be treated as "None".

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]. A 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. Let "enforcement" be "None".

2. If cookie-av's attribute-value is a case-insensitive match for "Strict", set "enforcement" to "Strict".

3. If cookie-av's attribute-value is a case-insensitive match for "Lax", set "enforcement" to "Lax".

4. Append an attribute to the cookie-attribute-list with an attribute-name of "SameSite" and an attribute-value of "enforcement".

Note: This algorithm maps the "None" value, as well as any unknown value, to the "None" behavior, which is helpful for backwards compatibility when introducing new variants.

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/ [4].

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", "Lax", or "None"). 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, host-only-flag, and path as the newly-created cookie:

    1. Let old-cookie be the existing cookie with the same name, domain, host-only-flag, 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, <https://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, <https://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, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3490] Costello, A., "Internationalizing Domain Names in Applications (IDNA)", RFC 3490, DOI 10.17487/RFC3490, March 2003, <https://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, <https://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, <https://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, <https://www.rfc-editor.org/info/rfc5890>.

[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>.

[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>.

[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, <https://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, <https://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, <https://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, <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>.

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

[RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <https://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, <https://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/>.

10.3. URIs

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

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

[3] https://github.com/httpwg/http-extensions/labels/6265bis

[4] http://publicsuffix.org/

[5] https://github.com/httpwg/http-extensions/issues/243

[6] https://github.com/httpwg/http-extensions/issues/246

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

[8] https://github.com/httpwg/http-extensions/issues/247

[9] https://github.com/httpwg/http-extensions/issues/201

[10] https://github.com/httpwg/http-extensions/issues/204

[11] https://github.com/httpwg/http-extensions/issues/222

[12] https://github.com/httpwg/http-extensions/issues/248

[13] https://github.com/httpwg/http-extensions/issues/295

[14] https://github.com/httpwg/http-extensions/issues/302

[15] https://github.com/httpwg/http-extensions/issues/389

[16] https://github.com/httpwg/http-extensions/issues/199

[17] https://github.com/httpwg/http-extensions/issues/788

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 [5]
  - https://github.com/httpwg/http-extensions/issues/246 [6]
- 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 [7]
- Dropped "Cookie2" and "Set-Cookie2" from the IANA Considerations section: https://github.com/httpwg/http-extensions/issues/247 [8]
- 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 [9]
  * Added the word "Cookies" to the document's name: https://github.com/httpwg/http-extensions/issues/204 [10]
  * Clarified that the "__Host-" prefix requires an explicit "Path" attribute: https://github.com/httpwg/http-extensions/issues/222 [11]
  * 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 [12]
  * Noted that double-quotes in cookie values are part of the value, and are not stripped: https://github.com/httpwg/http-extensions/issues/295 [13]
  * Fixed the "site for cookies" algorithm to return something that makes sense: https://github.com/httpwg/http-extensions/issues/302 [14]

A.4. draft-ietf-httpbis-rfc6265bis-03

- Clarified handling of invalid SameSite values: https://github.com/httpwg/http-extensions/issues/389 [15]
- Reflect widespread implementation practice of including a cookie's "host-only-flag" when calculating its uniqueness: https://github.com/httpwg/http-extensions/issues/199 [16]
- Introduced an explicit "None" value for the SameSite attribute: https://github.com/httpwg/http-extensions/issues/788 [17]

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/

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: May 15, 2018

P. McManus
Mozilla

November 11, 2017

Bootstrapping WebSockets with HTTP/2
draft-mcmanus-httpbis-h2-websockets-02

Abstract

This document defines a mechanism for running the WebSocket Protocol [RFC6455] over a single stream of an HTTP/2 connection.

Status of This Memo

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

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

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

This Internet-Draft will expire on May 15, 2018.

Copyright Notice

Table of Contents

1. Introduction

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

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

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

This document extends the HTTP/2 CONNECT method. The extension allows the substitution of a new protocol name to connect to rather than the external host normally used by CONNECT. The result is a tunnel on a single HTTP/2 stream that can carry data for WebSockets (or any other protocol). The other streams on the connection may carry more extended CONNECT tunnels, traditional HTTP/2 data, or a mixture of both.

This tunneled stream will be multiplexed with other regular streams on the connection and enjoys the normal priority, cancellation, and flow control features of HTTP/2.

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

2. Terminology

In this document, the key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" are to be interpreted as described in BCP 14, [RFC2119].

3. The ENABLE_CONNECT_PROTOCOL SETTINGS Parameter

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

The new parameter is ENABLE_CONNECT_PROTOCOL (type = 0x8). The value of the parameter MUST be 0 or 1.

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

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

The use of a SETTINGS Parameter to opt-in to an otherwise incompatible protocol change is a use of "Extending HTTP/2" defined by section 5.5 of [RFC7540]. If a client were to use the provisions of the extended CONNECT method defined in this document without first receiving a ENABLE_CONNECT_PROTOCOL parameter with the value of 1 it would be a protocol violation.

4. The Extended CONNECT Method

The CONNECT Method of [RFC7540] Section 8.3 is modified in the following ways:

- A new pseudo-header :protocol MAY be included on request HEADERS indicating the desired protocol to be spoken on the tunnel created by CONNECT. The pseudo-header is single valued and contains a value from the HTTP Upgrade Token Registry defined by [RFC7230].

- On requests bearing the :protocol pseudo-header, the :scheme and :path pseudo-header fields SHOULD be included.

- o On requests bearing the :protocol pseudo-header, the :authority pseudo-header field is interpreted according to [RFC7540] Section 8.1.2.3 instead of [RFC7540] Section 8.3. In particular the server MUST not make a new TCP connection to the host and port indicated by the :authority.

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

5. Using Extended CONNECT To Bootstrap The WebSocket Protocol

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

The scheme of the Target URI [RFC7230] MUST be https for wss schemed WebSockets and http for ws schemed WebSockets. The websocket URI is still used for proxy autoconfiguration.

[RFC6455] requires the use of Connection and Upgrade headers that are not part of HTTP/2. They MUST not be included in the CONNECT request defined here.

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

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

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

After successfully processing the opening handshake the peers should proceed with The WebSocket Protocol [RFC6455] using the HTTP/2 stream from the CONNECT transaction as if it were the TCP connection

referred to in [RFC6455]. The state of the WebSocket connection at this point is OPEN as defined by [RFC6455] Section 4.1.

5.1. Example

```
[[ From Client ]]                       [[ From Server ]]

                                        SETTINGS
                                        ENABLE_CONNECT_PROTOCOL = 1

HEADERS + END_HEADERS
:method = CONNECT
:protocol = websocket
:scheme = https
:path = /chat
:authority = server.example.com:443
sec-websocket-protocol = chat, superchat
sec-websocket-extensions = permessage-deflate
sec-websocket-version = 13
origin = http://www.example.com

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

DATA
WebSocket Data

                                        DATA + END_STREAM
                                        WebSocket Data

DATA + END_STREAM
WebSocket Data
```

6. Design Considerations

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

7. About Intermediaries

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

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

8. Security Considerations

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

- The CONNECT method is prohibited from being used by XMLHttpRequest
- The use of a pseudo-header is something that is connection specific and HTTP/2 does not ever allow to be created outside of the protocol stack.

9. IANA Considerations

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

Name: ENABLE_CONNECT_PROTOCOL

Code: 0x8

Initial Value: 0

Specification: This document

10. Acknowledgments

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

11. 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>.

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

[RFC6455] Fette, I. and A. Melnikov, "The WebSocket Protocol", RFC 6455, DOI 10.17487/RFC6455, December 2011, <https://www.rfc-editor.org/info/rfc6455>.

[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>.

[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>.

Author’s Address

Patrick McManus
Mozilla

Email: mcmanus@ducksong.com

Network Working Group
Internet-Draft
Obsoletes: 3205 (if approved)
Intended status: Best Current Practice
Expires: November 12, 2017

M. Nottingham
May 11, 2017

On the use of HTTP as a Substrate
draft-nottingham-bcp56bis-00

Abstract

HTTP is often used as a substrate for other application protocols. This document specifies best practices for these protocols' use of HTTP.

Note to Readers

The issues list for this draft can be found at https://github.com/mnot/I-D/labels/bcp56bis .

The most recent (often, unpublished) draft is at https://mnot.github.io/I-D/bcp56bis/ .

Recent changes are listed at https://github.com/mnot/I-D/commits/gh-pages/bcp56bis .

Status of This Memo

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

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

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

This Internet-Draft will expire on November 12, 2017.

Table of Contents

1. Introduction

HTTP [RFC7230] is often used as a substrate for other application protocols. This is done for a variety of reasons, including:

- familiarity by implementers, specifiers, administrators, developers and users,
- availability of a variety of client, server and proxy implementations,
- ease of use,
- ubiquity of Web browsers,
- reuse of existing mechanisms like authentication and encryption,
- presence of HTTP servers and clients in target deployments, and
- its ability to traverse firewalls.

The Internet community has a long tradition of protocol reuse, dating back to the use of Telnet [RFC0854] as a substrate for FTP [RFC0959] and SMTP [RFC2821]. However, layering new protocols over HTTP brings its own set of issues:

- Should an application using HTTP define a new URL scheme? Use new ports?
- Should it use standard HTTP methods and status codes, or define new ones?
- How can the maximum value be extracted from the use of HTTP?
- How does it coexist with other uses of HTTP - especially Web browsing?
- How can interoperability problems and "protocol dead ends" be avoided?

This document contains best current practices regarding the use of HTTP by applications other than Web browsing. Section 2 defines what applications it applies to; Section 3 surveys the properties of HTTP that are important to preserve, and Section 4 conveys best practices for those applications that do use HTTP.

It is written primarily to guide IETF efforts, but might be applicable in other situations. Note that the requirements herein do not necessarily apply to the development of generic HTTP extensions.

1.1. Notational Conventions

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

2. Is HTTP Being Used?

Different applications have different goals when using HTTP. In this document, we say an application is _using HTTP_ when any of the following conditions are true:

- The transport port in use is 80 or 443,
- The URL scheme "http" or "https" is used,
- The ALPN protocol ID [RFC7301] "http/1.1", "h2" or "h2c" is used, or
- The message formats described in [RFC7320] and/or [RFC7540] are used in conjunction with the IANA registries defined for HTTP.

When an application is using HTTP, all of the requirements of the HTTP protocol suite (including but not limited to [RFC7320], [RFC7321], [RFC7322], [RFC7233], [RFC7234], [RFC7325] and [RFC7540]) are in force.

An application might not be _using HTTP_ according to this definition, but still relying upon the HTTP specifications in some manner. For example, an application might wish to avoid re-specifying parts of the message format, but change others; or, it might want to use a different set of methods.

Such applications are referred to as _protocols based upon HTTP_ in this document. These have more freedom to modify protocol operation, but are also likely to lose at least a portion of the benefits outlined above, as most HTTP implementations won’t be easily adaptable to these changes, and as the protocol diverges from HTTP, the benefit of mindshare will be lost.

Protocols that are based upon HTTP MUST NOT reuse HTTP’s URL schemes, transport ports, ALPN protocol IDs or IANA registries; rather, they are encouraged to establish their own.

3. What's Important About HTTP

There are many ways that HTTP applications are defined and deployed, and sometimes they are brought to the IETF for standardisation. In that process, what might be workable for deployment in a limited fashion isn't appropriate for standardisation and the corresponding broader deployment.

This section examines the facets of the protocol that are important to preserve in these situations.

3.1. Generic Semantics

When writing an application's specification, it's often tempting to specify exactly how HTTP is to be implemented, supported and used.

However, this can easily lead to an unintended profile of HTTP's behaviour. For example, it's common to see specifications with language like this:

A '200 OK' response means that the widget has successfully been updated.

This sort of specification is bad practice, because it is adding new semantics to HTTP's status codes and methods, respectively; a recipient - whether it's an origin server, client library, intermediary or cache - now has to know these extra semantics to understand the message.

Some applications even require specific behaviours, such as:

A 'POST' request MUST result in a '201 Created' response.

This forms an expectation in the client that the response will always be "201 Created", when in fact there are a number of reasons why the status code might differ in a real deployment. If the client does not anticipate this, the application's deployment is brittle.

Much of the value of HTTP is in its _generic semantics_ - that is, the protocol elements defined by HTTP are potentially applicable to every resource, not specific to a particular context. Application-specific semantics are expressed in the payload; mostly, in the body, but also in header fields.

This allows a HTTP message to be examined by generic HTTP software (e.g., HTTP servers, intermediaries, client implementatiions), and its handling to be correctly determined. It also allows people to leverage their knowledge of HTTP semantics without special-casing them for a particular application.

Therefore, applications that use HTTP MUST NOT re-define, refine or overlay the semantics of defined protocol elements. Instead, they SHOULD focus their specifications on protocol elements that are specific to them; namely their HTTP resources.

See Section 4.2 for details.

3.2. Links

Another common practice is assuming that the HTTP server's name space (or a portion thereof) is exclusively for the use of a single application. This effectively overlays special, application-specific semantics onto that space, precludes other applications from using it.

As explained in [RFC7320], such "squatting" on a part of the URL space by a standard usurps the server's authority over its own resources, can cause deployment issues, and is therefore bad practice in standards.

Instead of statically defining URL paths, it is RECOMMENDED that applications using HTTP define links in payloads, to allow flexibility in deployment.

Using runtime links in this fashion has a number of other benefits. For example, navigating with a link allows a request to be routed to a different server without the overhead of a redirection, thereby supporting deployment across machines well. It becomes possible to "mix" different applications on the same server, and offers a natural path for extensibility, versioning and capability management.

3.3. Getting Value from HTTP

The simplest possible use of HTTP is to POST data to a single URL, thereby effectively tunnelling through the protocol.

This "RPC" style of communication does get some benefit from using HTTP - namely, message framing and the availability of implementations - but fails to realise many others:

- Caching for server scalability, latency and bandwidth reduction, and reliability;
- Authentication and access control;
- Automatic redirection;
- Partial content to selectively request part of a response;

- Natural support for extensions and versioning through protocol extension; and

- The ability to interact with the application easily using a Web browser.

Using such a high-level protocol to tunnel simple semantics has downsides too; because of its more advanced capabilities, breadth of deployment and age, HTTP's complexity can cause interoperability problems that could be avoided by using a simpler substrate (e.g., WebSockets [RFC6455], if browser support is necessary, or TCP [RFC0793] if not), or making the application be _based upon HTTP_, instead of using it (as defined in Section 2).

Applications that use HTTP are encouraged to accommodate the various features that the protocol offers, so that their users receive the maximum benefit from it. This document does not require specific features to be used, since the appropriate design tradeoffs are highly specific to a given situation. However, following the practices in Section 4 will help make them available.

4. Best Practices for Using HTTP

This section contains best practices regarding the use of HTTP by applications, including practices for specific HTTP protocol elements.

4.1. Specifying the Use of HTTP

When specifying the use of HTTP, an application SHOULD use [RFC7230] as the primary reference; it is not necessary to reference all of the specifications in the HTTP suite unless there are specific reasons to do so (e.g., a particular feature is called out).

Applications using HTTP MAY specify a minimum version to be supported (HTTP/1.1 is suggested), and MUST NOT specify a maximum version.

Likewise, applications need not specify what HTTP mechanisms - such as redirection, caching, authentication, proxy authentication, and so on - are to be supported. Full featured support for HTTP SHOULD be taken for granted in servers and clients, and the application's function SHOULD degrade gracefully if they are not (although this might be achieved by informing the user that their task cannot be completed).

For example, an application can specify that it uses HTTP like this:

Foo Application uses HTTP {{RFC7230}}. Implementations MUST support HTTP/1.1, and MAY support later versions. Support for common HTTP mechanisms such as redirection and caching are assumed.

4.2. Defining HTTP Resources

HTTP Applications SHOULD focus on defining the following application-specific protocol elements:

- Media types [RFC6838], often based upon a format convention such as JSON [RFC7159],
- HTTP header fields, as per Section 4.7, and
- The behaviour of resources, as identified by link relations [RFC5988].

By composing these protocol elements, an application can define a set of resources, identified by link relations, that implement specified behaviours, including:

- Retrieval of their state using GET, in one or more formats identified by media type;
- Resource creation or update using POST or PUT, with an appropriately identified request body format;
- Data processing using POST and identified request and response body format(s); and
- Resource deletion using DELETE.

For example, an application might specify:

Resources linked to with the "example-widget" link relation type are Widgets. The state of a Widget can be fetched in the "application/example-widget+json" format, and can be updated by PUT to the same link. Widget resources can be deleted.

The "Example-Count" response header field on Widget representations indicates how many Widgets are held by the sender.

The "application/example-widget+json" format is a JSON {{RFC7159}} format representing the state of a Widget. It contains links to related information in the link indicated by the Link header field value with the "example-other-info" link relation type.

4.3. HTTP URLs

In HTTP, URLs are opaque identifiers under the control of the server. As outlined in [RFC7320], standards cannot usurp this space, since it might conflict with existing resources, and constrain implementation and deployment.

In other words, applications that use HTTP MUST NOT associate application semantics with specific URL paths. For example, specifying that a "GET to the URL /foo retrieves a bar document" is bad practice. Likewise, specifying "The widget API is at the path /bar" violates [RFC7320].

Instead, applications that use HTTP are encouraged to use typed links [RFC5988] to convey the URIs that are in use, as well as the semantics of the resources that they identify. See Section 4.2 for details.

4.3.1. Initial URL Discovery

Generally, a client with begin interacting with a given application server by requesting an initial document that contains information about that particular deployment, potentially including links to other relevant resources.

Applications that use HTTP SHOULD allow an arbitrary URL to be used as that entry point. For example, rather than specifying "the initial document is at "/foo/v1", they should allow a deployment to use any URL as the entry point for the application.

In cases where doing so is impractical (e.g., it is not possible to convey a whole URL, but only a hostname) applications that use HTTP MAY define a well-known URL [RFC5785] as an entry point.

4.3.2. URL Schemes

Applications that use HTTP MUST allow use of the "https" URL scheme, and SHOULD NOT allow use of the "http" URL scheme, unless interoperability considerations with existing deployments require it. They MUST NOT use other URL schemes.

"https" is preferred to mitigate pervasive monitoring attacks [RFC7258].

Using other schemes to denote an application using HTTP makes it more difficult to use with existing implementations (e.g., Web browsers), and is likely to fail to meet the requirements of [RFC7595].

If it is necessary to advertise the application in use, this SHOULD be done in message payloads, not the URL scheme.

4.3.3. Transport Ports

Applications that use HTTP SHOULD use the default port for the URL scheme in use. If it is felt that networks might need to distinguish the application's traffic for operational reasons, it MAY register a separate port, but be aware that this has privacy implications for that protocol's users. The impact of doing so MUST be documented in Security Considerations.

4.4. Authentication and Application State

Applications that use HTTP MAY use stateful cookies [RFC6265] to identify a client and/or store client-specific data to contextualise requests.

If it is only necessary to identify clients, applications that use HTTP MAY use HTTP authentication [RFC7235]; if the Basic authentication scheme [RFC7617] is used, it MUST NOT be used with the 'http' URL scheme.

In either case, it is important to carefully specify the scoping and use of these mechanisms; if they expose sensitive data or capabilities (e.g., by acting as an ambiant authority), exploits are possible. Mitigations include using a request-specific token to assure the intent of the client.

4.5. HTTP Methods

Applications that use HTTP MUST confine themselves to using registered HTTP methods such as GET, POST, PUT, DELETE, and PATCH.

New HTTP methods are rare; they are required to be registered with IETF Review (see [RFC7232]), and are also required to be _generic_. That means that they need to be potentially applicable to all resources, not just those of one application.

While historically some applications (e.g., [RFC6352] and [RFC4791]) have defined non-generic methods, [RFC7231] now forbids this.

When it is believed that a new method is required, authors are encouraged to engage with the HTTP community early, and document their proposal as a separate HTTP extension, rather than as part of an application's specification.

4.6. HTTP Status Codes

Applications that use HTTP MUST only use registered HTTP status codes.

As with methods, new HTTP status codes are rare, and required (by [RFC7231]) to be registered with IETF review. Similarly, HTTP status codes are generic; they are required (by [RFC7231]) to be potentially applicable to all resources, not just to those of one application.

When it is believed that a new status code is required, authors are encouraged to engage with the HTTP community early, and document their proposal as a separate HTTP extension, rather than as part of an application's specification.

Status codes' primary function is to convey HTTP semantics for the benefit of generic HTTP software, not application-specific semantics. Therefore, applications MUST NOT specify additional semantics or refine existing semantics for status codes.

In particular, specifying that a particular status code has a specific meaning in the context of an application is harmful, as these are not generic semantics, since the consumer needs to be in the context of the application to understand them.

Furthermore, applications using HTTP MUST NOT re-specify the semantics of HTTP status codes, even if it is only by copying their definition. They MUST NOT require specific status phrases to be used; the status phrase has no function in HTTP, and is not guaranteed to be preserved by implementations.

Typically, applications using HTTP will convey application-specific information in the message body and/or HTTP header fields, not the status code.

Specifications sometimes also create a "laundry list" of potential status codes, in an effort to be helpful. The problem with doing so is that such a list is never complete; for example, if a network proxy is interposed, the client might encounter a "407 Proxy Authentication Required" response; or, if the server is rate limiting the client, it might receive a "429 Too Many Requests" response.

Since the list of HTTP status codes can be added to, it's safer to refer to it directly, and point out that clients SHOULD be able to handle all applicable protocol elements gracefully (i.e., falling back to the generic "n00" semantics of a given status code; e.g., "499" can be safely handled as "400" by clients that don't recognise it).

4.7. HTTP Header Fields

Applications that use HTTP MAY define new HTTP header fields, following the advice in [RFC7321], Section 8.3.1.

Typically, using HTTP header fields is appropriate in a few different situations:

- Their content is useful to intermediaries (who often wish to avoid parsing the body), and/or
- Their content is useful to generic HTTP software (e.g., clients, servers), and/or
- It is not possible to include their content in the message body (usually because a format does not allow it).

If none of these motivations apply, using a header field is NOT RECOMMENDED.

New header fields MUST be registered, as per [RFC7231] and [RFC3864].

It is RECOMMENDED that header field names be short (even when HTTP/2 header compression is in effect, there is an overhead) but appropriately specific. In particular, if a header field is specific to an application, an identifier for that application SHOULD form a prefix to the header field name, separated by a "-".

The semantics of existing HTTP header fields MUST NOT be re-defined without updating their registration or defining an extension to them (if allowed). For example, an application using HTTP cannot specify that the "Location" header has a special meaning in a certain context.

See Section 4.4 for requirements regarding header fields that carry application state (e.g,. Cookie).

5. IANA Considerations

This document has no requirements for IANA.

6. Security Considerations

Section 4.4 discusses the impact of using stateful mechanisms in the protocol as ambiant authority, and suggests a mitigation.

Section 4.3.2 requires support for ’https’ URLs, and discourages the use of ’http’ URLs, to mitigate pervasive monitoring attacks.

7. References

7.1. Normative References

[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>.

[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>.

[RFC5988] Nottingham, M., "Web Linking", RFC 5988, DOI 10.17487/RFC5988, October 2010, <http://www.rfc-editor.org/info/rfc5988>.

[RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, DOI 10.17487/RFC6838, January 2013, <http://www.rfc-editor.org/info/rfc6838>.

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

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

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

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

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

[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, <http://www.rfc-editor.org/info/rfc7301>.

[RFC7320] Nottingham, M., "URI Design and Ownership", BCP 190, RFC 7320, DOI 10.17487/RFC7320, July 2014, <http://www.rfc-editor.org/info/rfc7320>.

[RFC7321] McGrew, D. and P. Hoffman, "Cryptographic Algorithm Implementation Requirements and Usage Guidance for Encapsulating Security Payload (ESP) and Authentication Header (AH)", RFC 7321, DOI 10.17487/RFC7321, August 2014, <http://www.rfc-editor.org/info/rfc7321>.

[RFC7322] Flanagan, H. and S. Ginoza, "RFC Style Guide", RFC 7322, DOI 10.17487/RFC7322, September 2014, <http://www.rfc-editor.org/info/rfc7322>.

[RFC7325] Villamizar, C., Ed., Kompella, K., Amante, S., Malis, A., and C. Pignataro, "MPLS Forwarding Compliance and Performance Requirements", RFC 7325, DOI 10.17487/RFC7325, August 2014, <http://www.rfc-editor.org/info/rfc7325>.

[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>.

[RFC7595] Thaler, D., Ed., Hansen, T., and T. Hardie, "Guidelines and Registration Procedures for URI Schemes", BCP 35, RFC 7595, DOI 10.17487/RFC7595, June 2015, <http://www.rfc-editor.org/info/rfc7595>.

7.2. Informative References

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

[RFC0854] Postel, J. and J. Reynolds, "Telnet Protocol Specification", STD 8, RFC 854, DOI 10.17487/RFC0854, May 1983, <http://www.rfc-editor.org/info/rfc854>.

[RFC0959] Postel, J. and J. Reynolds, "File Transfer Protocol", STD 9, RFC 959, DOI 10.17487/RFC0959, October 1985, <http://www.rfc-editor.org/info/rfc959>.

[RFC2821] Klensin, J., Ed., "Simple Mail Transfer Protocol", RFC 2821, DOI 10.17487/RFC2821, April 2001, <http://www.rfc-editor.org/info/rfc2821>.

[RFC4791] Daboo, C., Desruisseaux, B., and L. Dusseault, "Calendaring Extensions to WebDAV (CalDAV)", RFC 4791, DOI 10.17487/RFC4791, March 2007, <http://www.rfc-editor.org/info/rfc4791>.

[RFC5785] Nottingham, M. and E. Hammer-Lahav, "Defining Well-Known Uniform Resource Identifiers (URIs)", RFC 5785, DOI 10.17487/RFC5785, April 2010, <http://www.rfc-editor.org/info/rfc5785>.

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

[RFC6352] Daboo, C., "CardDAV: vCard Extensions to Web Distributed Authoring and Versioning (WebDAV)", RFC 6352, DOI 10.17487/RFC6352, August 2011, <http://www.rfc-editor.org/info/rfc6352>.

[RFC6455] Fette, I. and A. Melnikov, "The WebSocket Protocol", RFC 6455, DOI 10.17487/RFC6455, December 2011, <http://www.rfc-editor.org/info/rfc6455>.

[RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, <http://www.rfc-editor.org/info/rfc7159>.

[RFC7235] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Authentication", RFC 7235, DOI 10.17487/RFC7235, June 2014, <http://www.rfc-editor.org/info/rfc7235>.

[RFC7258] Farrell, S. and H. Tschofenig, "Pervasive Monitoring Is an Attack", BCP 188, RFC 7258, DOI 10.17487/RFC7258, May 2014, <http://www.rfc-editor.org/info/rfc7258>.

[RFC7617] Reschke, J., "The ’Basic’ HTTP Authentication Scheme", RFC 7617, DOI 10.17487/RFC7617, September 2015, <http://www.rfc-editor.org/info/rfc7617>.

Author’s Address

Mark Nottingham

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

Network Working Group
Internet-Draft
Intended status: Informational
Expires: May 3, 2018

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

October 30, 2017

Structured Headers for HTTP
draft-nottingham-structured-headers-00

Abstract

This document describes Structured Headers, a way of simplifying HTTP header field definition and parsing. It is intended for use by new HTTP header fields.

Note to Readers

RFC EDITOR: please remove this section before publication

The issues list for this draft can be found at https://github.com/mnot/I-D/labels/structured-headers [1].

The most recent (often, unpublished) draft is at https://mnot.github.io/I-D/structured-headers/ [2].

Recent changes are listed at https://github.com/mnot/I-D/commits/gh-pages/structured-headers [3].

See also the draft's current status in the IETF datatracker, at https://datatracker.ietf.org/doc/draft-nottingham-structured-headers/ [4].

Status of This Memo

This Internet-Draft will expire on May 3, 2018.

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.

Likewise, bespoke parsers often need to be written for specific HTTP headers, because each 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 data, 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, of which only three are allowed at the "top" level: lists, dictionaries, or items.

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

HTTP headers that use 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.

Field definitions MUST NOT relax or otherwise modify the requirements of this specification; doing so would preclude handling by generic software.

However, field definitions are encouraged to clearly state additional constraints upon the syntax, as well as the consequences when those constraints are violated.

For example:

FooExample Header

The FooExample HTTP header field conveys a list of numbers 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:

* A member whose key is "foo", and whose value is an integer ([RFCxxxx], Section Y.Y), indicating the number of foos in the message.
* A member whose key is "bar", and whose value is a string ([RFCxxxx], Section Y.Y), conveying the characteristic bar-ness of the message.

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

Note that empty header field values are not allowed by the syntax, and therefore will be considered errors.

3. Parsing Requirements for Textual 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. Note that input_string may incorporate multiple header lines combined into one comma-separated field-value, as per [RFC7230], Section 3.2.2.

1. Discard any OWS from the beginning of input_string.

2. If the field-value is defined to be a dictionary, return the result of Parsing a Dictionary from Textual headers (Section 4.7).

3. If the field-value is defined to be a list, return the result of Parsing a List from Textual Headers (Section 4.8).

4. If the field-value is defined to be a parameterised label, return the result of Parsing a Parameterised Label from Textual headers (Section 4.4).

5. Otherwise, return the result of Parsing an Item from Textual Headers (Section 4.6).

Note that in the case of lists and dictionaries, this has the effect of combining multiple instances of the header field into one. However, for singular items and parameterised labels, it has the effect of selecting the first value and ignoring any subsequent instances of the field, as well as extraneous text afterwards.

Additionally, note that the effect of the parsing algorithms as specified is generally intolerant of syntax errors; if one is encountered, the typical response is to throw an error, thereby discarding the entire header field value. This includes any 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. Numbers

Abstractly, numbers are integers with an optional fractional part. They have a maximum of fifteen digits available to be used in one or 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 numbers allows a maximum of fifteen digits between the integer and fractional part, along with an optional "-" indicating negative numbers.

```
number   = ["-"] ( "." 1*15DIGIT /
             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 /
           15DIGIT )

integer  = ["-"] 1*15DIGIT
unsigned = 1*15DIGIT
```

integer and unsigned are defined as conveniences to specification authors; if their use is specified and their ABNF is not matched, a parser MUST consider it to be invalid.

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

```
ExampleNumberHeader: 4.5
```

4.1.1. Parsing Numbers from Textual Headers

TBD

4.2. Strings

Abstractly, strings are ASCII strings [RFC0020], excluding control characters (i.e., the range 0x20 to 0x7E). Note that this excludes tabs, newlines and carriage returns. They may be at most 1024 characters long.

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

```
string    = DQUOTE 1*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 generate an error.

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 4.5) SHOULD be specified, along with a character encoding (most likely, UTF-8).

4.2.1. Parsing a String from Textual Headers

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, throw an error.
3. Discard the first character of input_string.

4. If input_string contains more than 1025 characters, throw an error.

5. 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, throw an error.

        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 "\", throw an error.

            3. Append next_char to output_string.

    3. Else, if char is DQUOTE, remove the first character of input_string and return output_string.

    4. Else, append char to output_string.

6. Otherwise, throw an error.

4.3. Labels

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

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

Note that labels can only contain lowercase letters.

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

```
ExampleLabelHeader: foo/bar
```

4.3.1. Parsing a Label from Textual Headers

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

1. If input_string contains more than 256 characters, throw an error.
2. If the first character of input_string is not lcalpha, throw an error.
3. Let output_string be an empty 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 not one of lcalpha, DIGIT, "_", "-", "*" or "/":
        1. Prepend char to input_string.
        2. Return output_string.
    3. Append char to output_string.
5. Return output_string.

4.4. Parameterised Labels

Parameterised Labels are labels (Section 4.3) with up to 256 parameters; each parameter has a label and an optional value that is an item (Section 4.6). Ordering between parameters is not significant, and duplicate parameters MUST be considered an error.

The textual HTTP serialisation uses semicolons (";") to delimit the parameters from each other, and equals ("=") to delimit the parameter name from its value.

```
parameterised = label *256( OWS ";" OWS label [ "=" item ] )
```

For example,

```
ExampleParamHeader: abc; a=1; b=2; c
```

4.4.1. Parsing a Parameterised Label from Textual Headers

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

1. Let primary_label be the result of Parsing a Label from Textual Headers (Section 4.3) from input_string.
2. Let parameters be an empty mapping.
3. In a loop:
    1. Consume any OWS from the beginning of 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. Consume any OWS from the beginning of input_string.
    5. let param_name be the result of Parsing a Label from Textual Headers (Section 4.3) from input_string.
    6. If param_name is already present in parameters, throw an error.
    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 Textual Headers (Section 4.6) from input_string.
    9. If parameters has more than 255 members, throw an error.
    10. Add param_name to parameters with the value param_value.
4. Return the tuple (primary_label, parameters).

4.5. Binary Content

Arbitrary binary content up to 16K in size can be conveyed in Structured Headers.

The textual HTTP serialisation indicates their presence by a leading "*", with the data encoded using Base 64 Encoding [RFC4648], without padding (as "=" might be confused with the use of dictionaries).

```
binary = "*" 1*21846(base64)
base64 = ALPHA / DIGIT / "+" / "/"
```

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

```
ExampleBinaryHeader: *cHJldGVuZCB0aGlzIGlzIGJpbmFyeSBjb250ZW50Lg
```

4.5.1. Parsing Binary Content from Textual Headers

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 "*", throw an error.
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 character that is not in ALPHA, DIGIT, "+" or "/".
4. Let binary_content be the result of Base 64 Decoding [RFC4648] b64_content, synthesising padding if necessary. If an error is encountered, throw it.
5. Return binary_content.

4.6. Items

An item is can be a number (Section 4.1), string (Section 4.2), label (Section 4.3) or binary content (Section 4.5).

```
item = number / string / label / binary
```

4.6.1. Parsing an Item from Textual Headers

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

1. Discard any OWS from the beginning of input_string.

2. If the first character of input_string is a "-" or a DIGIT, process input_string as a number (Section 4.1) and return the result, throwing any errors encountered.

3. If the first character of input_string is a DQUOTE, process input_string as a string (Section 4.2) and return the result, throwing any errors encountered.

4. If the first character of input_string is "*", process input_string as binary content (Section 4.5) and return the result, throwing any errors encountered.

5. If the first character of input_string is an lcalpha, process input_string as a label (Section 4.3) and return the result, throwing any errors encountered.

6. Otherwise, throw an error.

4.7. Dictionaries

Dictionaries are unordered maps of key-value pairs, where the keys are labels (Section 4.3) and the values are items (Section 4.6). 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.

```
dictionary = label "=" item *1023( OWS "," OWS label "=" item )
```

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

```
ExampleDictHeader: foo=1.23, da="Applepie", en=*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.

4.7.1. Parsing a Dictionary from Textual Headers

Given an ASCII string input_string, return a mapping of (label, item). input_string is modified to remove the parsed value.

1. Let dictionary be an empty mapping.

2. While input_string is not empty:

    1. Let this_key be the result of running Parse Label from Textual Headers (Section 4.3) with input_string. If an error is encountered, throw it.

    2. If dictionary already contains this_key, raise an error.

    3. Consume a "=" from input_string; if none is present, raise an error.

    4. Let this_value be the result of running Parse Item from Textual Headers (Section 4.6) with input_string. If an error is encountered, throw it.

    5. Add key this_key with value this_value to dictionary.

    6. Discard any leading OWS from input_string.

    7. If input_string is empty, return dictionary.

    8. Consume a COMMA from input_string; if no comma is present, raise an error.

    9. Discard any leading OWS from input_string.

3. Return dictionary.

4.8. Lists

Lists are arrays of items (Section 4.6) or parameterised labels (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 1*1024( OWS "," OWS list_member )
list_member = item / paramterised_label
```

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

```
ExampleLabelListHeader: foo, bar, baz_45
```

and a header field whose value is defined as a list of parameterised labels could look like:

```
ExampleParamListHeader: abc/def; g="hi";j, klm/nop
```

4.8.1. Parsing a List from Textual Headers

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 Textual Headers (Section 4.6) with input_string. If an error is encountered, throw it.
    2. Append item to items.
    3. Discard any leading OWS from input_string.
    4. If input_string is empty, return items.
    5. Consume a COMMA from input_string; if no comma is present, raise an error.
    6. Discard any leading OWS from input_string.
3. Return items.

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/>.

[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://github.com/mnot/I-D/labels/structured-headers

[2] https://mnot.github.io/I-D/structured-headers/

[3] https://github.com/mnot/I-D/commits/gh-pages/structured-headers

[4] https://datatracker.ietf.org/doc/draft-nottingham-structured-headers/

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