Workgroup: HTTP
Internet-Draft:
draft-ietf-httpbis-message-signatures-01
Published: 17 November 2020
Intended Status: Standards Track
Expires: 21 May 2021
Authors: A. Backman, Ed. J. Richer M. Sporny
Amazon Bespoke Engineering Digital Bazaar

# Signing HTTP Messages

## Abstract

This document describes a mechanism for creating, encoding, and verifying digital signatures or message authentication codes over content within an HTTP message. This mechanism supports use cases where the full HTTP message may not be known to the signer, and where the message may be transformed (e.g., by intermediaries) before reaching the verifier.

## Note to Readers

*RFC EDITOR: please remove this section before publication*

This work was originally based on draft-cavage-http-signatures-12, but has since diverged from it, to reflect discussion since adoption by the HTTP Working Group. In particular, it addresses issues that have been identified, and adds features to support new use cases. It is a work-in-progress and not yet suitable for deployment.

## Status of This Memo

## Copyright Notice



## Table of Contents

1. Introduction
  1.1. Requirements Discussion
  1.2. HTTP Message Transformations
  1.3. Safe Transformations
  1.4. Conventions and Terminology
2. Identifying and Canonicalizing Content
  2.1. HTTP Header Fields
    2.1.1. Canonicalization Examples
  2.2. Dictionary Structured Field Members
    2.2.1. Canonicalization Examples
  2.3. List Prefixes
    2.3.1. Canonicalization Examples
  2.4. Signature Creation Time
  2.5. Signature Expiration Time
  2.6. Target Endpoint
    2.6.1. Canonicalization Examples
3. HTTP Message Signatures
  3.1. Signature Metadata
  3.2. Creating a Signature
    3.2.1. Choose and Set Signature Metadata Properties
    3.2.2. Create the Signature Input
    3.2.3. Sign the Signature Input
  3.3. Verifying a Signature
    3.3.1. Enforcing Application Requirements
4. Including a Message Signature in a Message
  4.1. The 'Signature-Input' HTTP Header
    4.1.1. Metadata Parameters
  4.2. The 'Signature' HTTP Header
  4.3. Examples
5. IANA Considerations
  5.1. HTTP Signature Algorithms Registry
    5.1.1. Registration Template

5.1.2. Initial Contents
5.2. HTTP Signature Metadata Parameters Registry
5.2.1. Registration Template
5.2.2. Initial Contents
6. Security Considerations
7. References
7.1. Normative References
7.2. Informative References
Appendix A. Examples
A.1. Example Keys
A.1.1. Example Key RSA test
A.2. Example keyId Values
A.3. Test Cases
A.3.1. Signature Generation
A.3.2. Signature Verification
Appendix B. Topics for Working Group Discussion
B.1. Issues
B.1.1. Confusing guidance on algorithm and key identification
B.1.2. Lack of definition of keyId hurts interoperability
B.1.3. Algorithm Registry duplicates work of JWA
B.1.4. Algorithm Registry should not be initialized with deprecated entries
B.1.5. No percent-encoding normalization of path/query
B.1.6. Misleading name for headers parameter
B.1.7. Changes to whitespace in header field values break verification
B.1.8. Multiple Set-Cookie headers are not well supported
B.1.9. Covered Content list is not signed
B.1.10. Algorithm is not signed
B.1.11. Verification key identifier is not signed
B.1.12. Max values, precision for Integer String and Decimal String not defined
B.1.13. keyId parameter value could break list syntax
B.1.14. Creation Time and Expiration Time do not allow for clock skew
B.1.15. Should require lowercased header field names as identifiers
B.1.16. Reconcile Date header and Creation Time
B.1.17. Remove algorithm-specific rules for content identifiers
B.1.18. Add guidance for signing compressed headers
B.1.19. Transformations to Via header field value break verification
B.1.20. Case changes to case-insensitive header field values break verification
B.1.21. Need more examples for Signature header
B.1.22. Expiration not needed
B.2. Features
B.2.1. Define more content identifiers
B.2.2. Multiple signature support

B.2.3. Support for incremental signing of header field value list items
B.2.4. Support expected authority changes
B.2.5. Support for signing specific cookies
Acknowledgements
Document History
Authors' Addresses

## 1. Introduction

Message integrity and authenticity are important security properties that are critical to the secure operation of many HTTP applications. Application developers typically rely on the transport layer to provide these properties, by operating their application over [TLS]. However, TLS only guarantees these properties over a single TLS connection, and the path between client and application may be composed of multiple independent TLS connections (for example, if the application is hosted behind a TLS-terminating gateway or if the client is behind a TLS Inspection appliance). In such cases, TLS cannot guarantee end-to-end message integrity or authenticity between the client and application. Additionally, some operating environments present obstacles that make it impractical to use TLS, or to use features necessary to provide message authenticity. Furthermore, some applications require the binding of an application-level key to the HTTP message, separate from any TLS certificates in use. Consequently, while TLS can meet message integrity and authenticity needs for many HTTP-based applications, it is not a universal solution.

This document defines a mechanism for providing end-to-end integrity and authenticity for content within an HTTP message. The mechanism allows applications to create digital signatures or message authentication codes (MACs) over only that content within the message that is meaningful and appropriate for the application. Strict canonicalization rules ensure that the verifier can verify the signature even if the message has been transformed in any of the many ways permitted by HTTP.

The mechanism described in this document consists of three parts:

* A common nomenclature and canonicalization rule set for the different protocol elements and other content within HTTP messages.

* Algorithms for generating and verifying signatures over HTTP message content using this nomenclature and rule set.

* A mechanism for attaching a signature and related metadata to an HTTP message.

### 1.1. Requirements Discussion

HTTP permits and sometimes requires intermediaries to transform messages in a variety of ways. This may result in a recipient receiving a message that is not bitwise equivalent to the message that was oringally sent. In such a case, the recipient will be unable to verify a signature over the raw bytes of the sender's HTTP message, as verifying digital signatures or MACs requires both signer and verifier to have the exact same signed content. Since the raw bytes of the message cannot be relied upon as signed content, the signer and verifier must derive the signed content from their respective versions of the message, via a mechanism that is resilient to safe changes that do not alter the meaning of the message.

For a variety of reasons, it is impractical to strictly define what constitutes a safe change versus an unsafe one. Applications use HTTP in a wide variety of ways, and may disagree on whether a particular piece of information in a message (e.g., the body, or the Date header field) is relevant. Thus a general purpose solution must provide signers with some degree of control over which message content is signed.

HTTP applications may be running in environments that do not provide complete access to or control over HTTP messages (such as a web browser's JavaScript environment), or may be using libraries that abstract away the details of the protocol (such as the Java HTTPClient library). These applications need to be able to generate and verify signatures despite incomplete knowledge of the HTTP message.

### 1.2. HTTP Message Transformations

As mentioned earlier, HTTP explicitly permits and in some cases requires implementations to transform messages in a variety of ways. Implementations are required to tolerate many of these transformations. What follows is a non-normative and non-exhaustive list of transformations that may occur under HTTP, provided as context:

* Re-ordering of header fields with different header field names ([MESSAGING], Section 3.2.2).

* Combination of header fields with the same field name ([MESSAGING], Section 3.2.2).

* Removal of header fields listed in the Connection header field ([MESSAGING], Section 6.1).

* Addition of header fields that indicate control options ([MESSAGING], Section 6.1).

* Addition or removal of a transfer coding ([MESSAGING], Section 5.7.2).

* Addition of header fields such as Via ([MESSAGING], Section 5.7.1) and Forwarded ([RFC7239], Section 4).

### 1.3. Safe Transformations

Based on the definition of HTTP and the requirements described above, we can identify certain types of transformations that should not prevent signature verification, even when performed on content covered by the signature. The following list describes those transformations:

* Combination of header fields with the same field name.

* Reordering of header fields with different names.

* Conversion between different versions of the HTTP protocol (e.g., HTTP/1.x to HTTP/2, or vice-versa).

* Changes in casing (e.g., "Origin" to "origin") of any case-insensitive content such as header field names, request URI scheme, or host.

* Addition or removal of leading or trailing whitespace to a header field value.

* Addition or removal of obs-folds.

* Changes to the request-target and Host header field that when applied together do not result in a change to the message's effective request URI, as defined in Section 5.5 of [MESSAGING].

Additionally, all changes to content not covered by the signature are considered safe.

### 1.4. Conventions and Terminology

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

The terms "HTTP message", "HTTP request", "HTTP response", absolute-form, absolute-path, "effective request URI", "gateway", "header field", "intermediary", request-target, "sender", and "recipient" are used as defined in [MESSAGING].

The term "method" is to be interpreted as defined in Section 4 of [SEMANTICS].

For brevity, the term "signature" on its own is used in this document to refer to both digital signatures and keyed MACs. Similarly, the verb "sign" refers to the generation of either a digital signature or keyed MAC over a given input string. The qualified term "digital signature" refers specifically to the output of an asymmetric cryptographic signing operation.

In addition to those listed above, this document uses the following terms:

**Decimal String** An Integer String optionally concatenated with a period "." followed by a second Integer String, representing a positive real number expressed in base 10. The first Integer String represents the integral portion of the number, while the optional second Integer String represents the fractional portion of the number. (( Editor's note: There's got to be a definition for this that we can reference. ))

**Integer String** A US-ASCII string of one or more digits "0-9", representing a positive integer in base 10. (( Editor's note: There's got to be a definition for this that we can reference. ))

**Signer** The entity that is generating or has generated an HTTP Message Signature.

**Verifier** An entity that is verifying or has verified an HTTP Message Signature against an HTTP Message. Note that an HTTP Message Signature may be verified multiple times, potentially by different entities.

This document contains non-normative examples of partial and complete HTTP messages. To improve readability, header fields may be split into multiple lines, using the obs-fold syntax. This syntax is deprecated in [MESSAGING], and senders MUST NOT generate messages that include it.

## 2. Identifying and Canonicalizing Content

In order to allow signers and verifiers to establish which content is covered by a signature, this document defines content identifiers for signature metadata and discrete pieces of message content that may be covered by an HTTP Message Signature.

Some content within HTTP messages may undergo transformations that change the bitwise value without altering meaning of the content (for example, the merging together of header fields with the same name). Message content must therefore be canonicalized before it is signed, to ensure that a signature can be verified despite such innocuous transformations. This document defines rules for each content identifier that transform the identifier's associated content into such a canonical form.

The following sections define content identifiers, their associated content, and their canonicalization rules.

## 2.1. HTTP Header Fields

An HTTP header field is identified by its header field name. While HTTP header field names are case-insensitive, implementations MUST use lowercased field names (e.g., content-type, date, etag) when using them as content identifiers.

An HTTP header field value is canonicalized as follows:

1. Create an ordered list of the field values of each instance of the header field in the message, in the order that they occur (or will occur) in the message.

2. Strip leading and trailing whitespace from each item in the list.

3. Concatenate the list items together, with a comma "," and space " " between each item. The resulting string is the canonicalized value.

### 2.1.1. Canonicalization Examples

This section contains non-normative examples of canonicalized values for header fields, given the following example HTTP message:

```
HTTP/1.1 200 OK
Server: www.example.com
Date: Tue, 07 Jun 2014 20:51:35 GMT
X-OWS-Header:   Leading and trailing whitespace.
X-Obs-Fold-Header: Obsolete
    line folding.
X-Empty-Header:
Cache-Control: max-age=60
Cache-Control:    must-revalidate
```

The following table shows example canonicalized values for header fields, given that message:

| Header Field | Canonicalized Value |
|---|---|
| cache-control | max-age=60, must-revalidate |
| date | Tue, 07 Jun 2014 20:51:35 GMT |
| server | www.example.com |
| x-empty-header | |
| x-obs-fold-header | Obsolete line folding. |
| x-ows-header | Leading and trailing whitespace. |

Table 1: Non-normative examples of header field canonicalization.

### 2.2. Dictionary Structured Field Members

An individual member in the value of a Dictionary Structured Field is identified by the lowercased field name, followed by a semicolon ":", followed by the member name. An individual member in the value of a Dictionary Structured Field is canonicalized by applying the serialization algorithm described in Section 4.1.2 of [StructuredFields] on a Dictionary containing only that member.

#### 2.2.1. Canonicalization Examples

This section contains non-normative examples of canonicalized values for Dictionary Structured Field Members given the following example header field, whose value is assumed to be a Dictionary:

```
X-Dictionary:  a=1, b=2;x=1;y=2, c=(a, b, c)
```

The following table shows example canonicalized values for different content identifiers, given that field:

| Content Identifier | Canonicalized Value |
|---|---|
| x-dictionary:a | 1 |
| x-dictionary:b | 2;x=1;y=2 |
| x-dictionary:c | (a, b, c) |

Table 2: Non-normative examples of Dictionary member canonicalization.

### 2.3. List Prefixes

A prefix of a List Structured Field consisting of the first N members in the field's value (where N is an integer greater than 0 and less than or equal to the number of members in the List) is identified by the lowercased field name, followed by a semicolon ":", followed by N expressed as an Integer String. A list prefix is canonicalized by applying the serialization algorithm described in Section 4.1.1 of [StructuredFields] on a List containing only the first N members as specified in the list prefix, in the order they appear in the original List.

### 2.3.1. Canonicalization Examples

This section contains non-normative examples of canonicalized values for list prefixes given the following example header fields, whose values are assumed to be Dictionaries:

```
X-List-A: (a, b, c, d, e, f)
X-List-B: ()
```

The following table shows example canonicalized values for different content identifiers, given those fields:

| Content Identifier | Canonicalized Value |
|---|---|
| x-list-a:0 | () |
| x-list-a:1 | (a) |
| x-list-a:3 | (a, b, c) |
| x-list-a:6 | (a, b, c, d, e, f) |
| x-list-b:0 | () |

Table 3: Non-normative examples of list prefix canonicalization.

## 2.4. Signature Creation Time

The signature's Creation Time (Section 3.1) is identified by the *created identifier.

Its canonicalized value is an Integer String containing the signature's Creation Time expressed as the number of seconds since the Epoch, as defined in Section 4.16 of [POSIX.1].

> The use of seconds since the Epoch to canonicalize a timestamp simplifies processing and avoids timezone management required by specifications such as [RFC3339].

## 2.5. Signature Expiration Time

The signature's Expiration Time (Section 3.1) is identified by the *expires identifier.

Its canonicalized value is a Decimal String containing the signature's Expiration Time expressed as the number of seconds since the Epoch, as defined in Section 4.16 of [POSIX.1].

## 2.6. Target Endpoint

The request target endpoint, consisting of the request method and the path and query of the effective request URI, is identified by the *request-target identifier.

Its value is canonicalized as follows:

1. Take the lowercased HTTP method of the message.

2. Append a space " ".

3. Append the path and query of the request target of the message, formatted according to the rules defined for the :path pseudo-header in [HTTP2], Section 8.1.2.3. The resulting string is the canonicalized value.

### 2.6.1. Canonicalization Examples

The following table contains non-normative example HTTP messages and their canonicalized *request-target values.

| HTTP Message | *request-target |
|---|---|
| POST /?param=value HTTP/1.1<br>Host: www.example.com | post /?param=value |
| POST /a/b HTTP/1.1<br>Host: www.example.com | post /a/b |
| GET http://www.example.com/a/ HTTP/1.1 | get /a/ |
| GET http://www.example.com HTTP/1.1 | get / |
| CONNECT server.example.com:80 HTTP/1.1<br>Host: server.example.com | connect / |
| OPTIONS * HTTP/1.1<br>Host: server.example.com | options * |

| HTTP Message | *request-target |
| --- | --- |
| | |

Table 4: Non-normative examples of *request-target canonicalization.

## 3. HTTP Message Signatures

An HTTP Message Signature is a signature over a string generated from a subset of the content in an HTTP message and metadata about the signature itself. When successfully verified against an HTTP message, it provides cryptographic proof that with respect to the subset of content that was signed, the message is semantically equivalent to the message for which the signature was generated.

### 3.1. Signature Metadata

HTTP Message Signatures have metadata properties that provide information regarding the signature's generation and/or verification. The following metadata properties are defined:

**Algorithm** An HTTP Signature Algorithm defined in the HTTP Signature Algorithms Registry defined in this document. It describes the signing and verification algorithms for the signature.

**Creation Time** A timestamp representing the point in time that the signature was generated. Sub-second precision is not supported. A signature's Creation Time MAY be undefined, indicating that it is unknown.

**Covered Content** An ordered list of content identifiers (Section 2) that indicates the metadata and message content that is covered by the signature. The order of identifiers in this list affects signature generation and verification, and therefore MUST be preserved.

**Expiration Time** A timestamp representing the point in time at which the signature expires. An expired signature always fails verification. A signature's Expiration Time MAY be undefined, indicating that the signature does not expire.

**Verification Key Material** The key material required to verify the signature.

### 3.2. Creating a Signature

In order to create a signature, a signer completes the following process:

1. Choose key material and algorithm, and set metadata properties Section 3.2.1

2. Create the Signature Input Section 3.2.2

3. Sign the Signature Input Section 3.2.3

The following sections describe each of these steps in detail.

### 3.2.1. Choose and Set Signature Metadata Properties

1. The signer chooses an HTTP Signature Algorithm from those registered in the HTTP Signature Algorithms Registry defined by this document, and sets the signature's Algorithm property to that value. The signer MUST NOT choose an algorithm marked "Deprecated". The mechanism by which the signer chooses an algorithm is out of scope for this document.

2. The signer chooses key material to use for signing and verification, and sets the signature's Verification Key Material property to the key material required for verification. The signer MUST choose key material that is appropriate for the signature's Algorithm, and that conforms to any requirements defined by the Algorithm, such as key size or format. The mechanism by which the signer chooses key material is out of scope for this document.

3. The signer sets the signature's Creation Time property to the current time.

4. The signer sets the signature's Expiration Time property to the time at which the signature is to expire, or to undefined if the signature will not expire.

5. The signer creates an ordered list of content identifiers representing the message content and signature metadata to be covered by the signature, and assigns this list as the signature's Covered Content.

   * Each identifier MUST be one of those defined in Section 2.

   * This list MUST NOT be empty, as this would result in creating a signature over the empty string.

   * If the signature's Algorithm name does not start with rsa, hmac, or ecdsa, signers SHOULD include *created and *request-target in the list.

   * If the signature's Algorithm starts with rsa, hmac, or ecdsa, signers SHOULD include date and *request-target in the list.

*Further guidance on what to include in this list and in what order is out of scope for this document. However, the list order is significant and once established for a given signature it MUST be preserved for that signature.

For example, given the following HTTP message:

```
GET /foo HTTP/1.1
Host: example.org
Date: Sat, 07 Jun 2014 20:51:35 GMT
X-Example: Example header
        with some whitespace.
X-EmptyHeader:
X-Dictionary: a=1, b=2
X-List: (a, b, c, d)
Cache-Control: max-age=60
Cache-Control: must-revalidate
```

The following table presents a non-normative example of metadata values that a signer may choose:

| **Property** | **Value** |
|---|---|
| Algorithm | hs2019 |
| Covered Content | *request-target, *created, host, date, cache-contol, x-emptyheader, x-example, x-dictionary:b, x-dictionary:a, x-list:3 |
| Creation Time | 1402174295 |
| Expiration Time | 1402174595 |
| Verification Key Material | The public key provided in Appendix A.1.1 and identified by the keyId value "test-key-a". |

Table 5: Non-normative example metadata values

#### 3.2.2. Create the Signature Input

The Signature Input is a US-ASCII string containing the content that will be signed. To create it, the signer concatenates together entries for each identifier in the signature's Covered Content in the order it occurs in the list, with each entry separated by a newline "\n". An identifier's entry is a US-ASCII string consisting of the lowercased identifier followed with a colon ":", a space " ", and the identifier's canonicalized value (described below).

If Covered Content contains *created and the signature's Creation Time is undefined or the signature's Algorithm name starts with rsa, hmac, or ecdsa an implementation MUST produce an error.

If Covered Content contains *expires and the signature does not have an Expiration Time or the signature's Algorithm name starts with rsa, hmac, or ecdsa an implementation MUST produce an error.

If Covered Content contains an identifier for a header field that is not present or malformed in the message, the implementation MUST produce an error.

If Covered Content contains an identifier for a Dictionary member that references a header field that is not present, is malformed in the message, or is not a Dictionary Structured Field, the implementation MUST produce an error. If the header field value does not contain the specified member, the implementation MUST produce an error.

If Covered Content contains an identifier for a List Prefix that references a header field that is not present, is malformed in the message, or is not a List Structured Field, the implementation MUST produce an error. If the header field value contains fewer than the specified number of members, the implementation MUST produce an error.

For the non-normative example Signature metadata in Table 5, the corresponding Signature Input is:

```
*request-target: get /foo
*created: 1402170695
host: example.org
date: Tue, 07 Jun 2014 20:51:35 GMT
cache-control: max-age=60, must-revalidate
x-emptyheader:
x-example: Example header with some whitespace.
x-dictionary: b=2
x-dictionary: a=1
x-list: (a, b, c)
```

Figure 1: Non-normative example Signature Input

### 3.2.3. Sign the Signature Input

The signer signs the Signature Input using the signing algorithm described by the signature's Algorithm property, and the key material chosen by the signer. The signer then encodes the result of that operation as a base 64-encoded string [RFC4648]. This string is the signature value.

For the non-normative example Signature metadata in Section 3.2.1 and Signature Input in Figure 1, the corresponding signature value is:

```
K2qGT5srn2OGbOIDzQ6kYT+ruaycnDAAUpKv+ePFfD0RAxn/1BUeZx/Kdrq32DrfakQ6b
PsvB9aqZqognNT6be4olHROIkeV879RrsrObury8L9SCEibeoHyqU/yCjphSmEdd7WD+z
rchK57quskKwRefy2iEC5S2uAH0EPyOZKWlvbKmKu5q4CaB8X/I5/+HLZLGvDiezqi6/7
p2Gngf5hwZ0lSdy39vyNMaaAT0tKo6nuVw0S1MVg1Q7MpWYZs0soHjttq0uLIA3DIbQfL
iIvK6/l0BdWTU7+2uQj7lBkQAsFZHoA96ZZgFquQrXRlmYOh+Hx5D9fJkXcXe5tmAg==
```


Figure 2: Non-normative example signature value

### 3.3. Verifying a Signature

In order to verify a signature, a verifier MUST:

1. Examine the signature's metadata to confirm that the signature meets the requirements described in this document, as well as any additional requirements defined by the application such as which header fields or other content are required to be covered by the signature.

2. Use the received HTTP message and the signature's metadata to recreate the Signature Input, using the process described in Section 3.2.2.

3. Use the signature's Algorithm and Verification Key Material with the recreated Signing Input to verify the signature value.

A signature with a Creation Time that is in the future or an Expiration Time that is in the past MUST NOT be processed.

The verifier MUST ensure that a signature's Algorithm is appropriate for the key material the verifier will use to verify the signature. If the Algorithm is not appropriate for the key material (for example, if it is the wrong size, or in the wrong format), the signature MUST NOT be processed.

#### 3.3.1. Enforcing Application Requirements

The verification requirements specified in this document are intended as a baseline set of restrictions that are generally applicable to all use cases. Applications using HTTP Message Signatures MAY impose requirements above and beyond those specified by this document, as appropriate for their use case.

Some non-normative examples of additional requirements an application might define are:

* Requiring a specific set of header fields to be signed (e.g., Authorization, Digest).

* Enforcing a maximum signature age.

* Prohibiting the use of certain algorithms, or mandating the use of an algorithm.

* Requiring keys to be of a certain size (e.g., 2048 bits vs. 1024 bits).

Application-specific requirements are expected and encouraged. When an application defines additional requirements, it MUST enforce them during the signature verification process, and signature verification MUST fail if the signature does not conform to the application's requirements.

Applications MUST enforce the requirements defined in this document. Regardless of use case, applications MUST NOT accept signatures that do not conform to these requirements.

## 4. Including a Message Signature in a Message

Message signatures can be included within an HTTP message via the Signature-Input and Signature HTTP header fields, both defined within this specification. The Signature HTTP header field contains signature values, while the Signature-Input HTTP header field identifies the Covered Content and metadata that describe how each signature was generated.

### 4.1. The 'Signature-Input' HTTP Header

The Signature-Input HTTP header field is a Dictionary Structured Header [StructuredFields] containing the metadata for zero or more message signatures generated from content within the HTTP message. Each member describes a single message signature. The member's name is an identifier that uniquely identifies the message signature within the context of the HTTP message. The member's value is the message signature's Covered Content, expressed as a List of Tokens. Further signature metadata is expressed in parameters on the member value, as described below.

#### 4.1.1. Metadata Parameters

The parameters on each Signature-Input member value contain metadata about the signature. Each parameter name MUST be a parameter name registered in the IANA HTTP Signatures Metadata Parameters Registry defined in Section 5.2 of this document. This document defines the following parameters, and registers them as the initial contents of the registry:

**alg** RECOMMENDED. The alg parameter is a Token containing the name of the signature's Algorithm, as registered in the HTTP Signature Algorithms Registry defined by this document. Verifiers MUST determine the signature's Algorithm from the keyId parameter rather than from alg. If alg is provided and differs from or is incompatible with the algorithm or key material identified by keyId (for example, alg has a value of rsa-sha256 but keyId identifies an EdDSA key), then implementations MUST produce an error.

**created** RECOMMENDED. The created parameter is a Decimal containing the signature's Creation Time, expressed as the canonicalized value of the *created content identifier, as defined in Section 2. If not specified, the signature's Creation Time is undefined. This parameter is useful when signers are not capable of controlling the Date HTTP Header such as when operating in certain web browser environments.

**expires** OPTIONAL. The expires parameter is a Decimal containing the signature's Expiration Time, expressed as the canonicalized value of the *expires content identifier, as defined in Section 2. If the signature does not have an Expiration Time, this parameter MUST be omitted. If not specified, the signature's Expiration Time is undefined.

**keyId** REQUIRED. The keyId parameter is a String whose value can be used by a verifier to identify and/or obtain the signature's Verification Key Material. Further format and semantics of this value are out of scope for this document.

### 4.2. The 'Signature' HTTP Header

The Signature HTTP header field is a Dictionary Structured Header [StructuredFields] containing zero or more message signatures generated from content within the HTTP message. Each member's name is a signature identifier that is present as a member name in the Signature-Input Structured Header within the HTTP message. Each member's value is a Byte Sequence containing the signature value for the message signature identified by the member name. Any member in the Signature HTTP header field that does not have a corresponding member in the HTTP message's Signature-Input HTTP header field MUST be ignored.

### 4.3. Examples

The following is a non-normative example of Signature-Input and Signature HTTP header fields representing the signature in Figure 2:

```
Signature-Input: sig1=(*request-target, *created, host, date,
    cache-control, x-empty-header, x-example); keyId="test-key-a";
    alg=hs2019; created=1402170695; expires=1402170995
Signature: sig1=:K2qGT5srn2OGbOIDzQ6kYT+ruaycnDAAUpKv+ePFfD0RAxn/1BUe
    Zx/Kdrq32DrfakQ6bPsvB9aqZqognNT6be4olHROIkeV879RrsrObury8L9SCEibe
    oHyqU/yCjphSmEdd7WD+zrchK57quskKwRefy2iEC5S2uAH0EPyOZKWlvbKmKu5q4
    CaB8X/I5/+HLZLGvDiezqi6/7p2Gngf5hwZ0lSdy39vyNMaaAT0tKo6nuVw0S1MVg
    1Q7MpWYZs0soHjttq0uLIA3DIbQfLiIvK6/l0BdWTU7+2uQj7lBkQAsFZHoA96ZZg
    FquQrXRlmYOh+Hx5D9fJkXcXe5tmAg==:
```

Since Signature-Input and Signature are both defined as Dictionary Structured Headers, they can be used to easily include multiple signatures within the same HTTP message. For example, a signer may include multiple signatures signing the same content with different keys and/or algorithms to support verifiers with different capabilities, or a reverse proxy may include information about the client in header fields when forwarding the request to a service host, and may also include a signature over those fields and the client's signature. The following is a non-normative example of header fields a reverse proxy might add to a forwarded request that contains the signature in the above example:

```
X-Forwarded-For: 192.0.2.123
Signature-Input: reverse_proxy_sig=(*created, host, date,
    signature:sig1, x-forwarded-for); keyId="test-key-a";
    alg=hs2019; created=1402170695; expires=1402170695.25
Signature: reverse_proxy_sig=:ON3HsnvuoTlX41xfcGWaOEVo1M3bJDRBOp0Pc/O
    jAOWKQn0VMY0SvMMWXS7xG+xYVa152rRVAo6nMV7FS3rv0rR5MzXL8FCQ2A35DCEN
    LOhEgj/S1IstEAEFsKmE9Bs7McBsCtJwQ3hMqdtFenkDffSoHOZOInkTYGafkoy78
    l1VZvmb3Y4yf7McJwAvk2R3gwKRWiiRCw448Nt7JTWzhvEwbh7bN2swc/v3NJbg/w
    JYyYVbelZx4IywuZnYFxgPl/qvqbAjeEVvaLKLgSMr11y+uzxCHoMnDUnTYhMrmOT
    4O8lBLfRFOcoJPKBdoKg9U0a96U2mUug1bFOozEVYFg==:
```

## 5. IANA Considerations

### 5.1. HTTP Signature Algorithms Registry

This document defines HTTP Signature Algorithms, for which IANA is asked to create and maintain a new registry titled "HTTP Signature Algorithms". Initial values for this registry are given in Section 5.1.2. Future assignments and modifications to existing assignment are to be made through the Expert Review registration policy [RFC8126] and shall follow the template presented in Section 5.1.1.

#### 5.1.1. Registration Template

**Algorithm Name** An identifier for the HTTP Signature Algorithm. The name MUST be an ASCII string consisting only of lower-case characters ("a" - "z"), digits ("0" - "9"), and hyphens ("-"), and SHOULD NOT exceed 20 characters in length. The identifier MUST be unique within the context of the registry.

**Status** A brief text description of the status of the algorithm. The description MUST begin with one of "Active" or "Deprecated", and MAY provide further context or explanation as to the reason for the status.

**Description** A description of the algorithm used to sign the signing string when generating an HTTP Message Signature, or instructions on how to determine that algorithm. When the description specifies an algorithm, it MUST include a reference to the document or documents that define the algorithm.

#### 5.1.2. Initial Contents

(( MS: The references in this section are problematic as many of the specifications that they refer to are too implementation specific, rather than just pointing to the proper signature and hashing specifications. A better approach might be just specifying the signature and hashing function specifications, leaving implementers to connect the dots (which are not that hard to connect). ))

##### 5.1.2.1. hs2019

**Algorithm Name** hs2019

**Status** active

**Description** Derived from metadata associated with keyId. Recommend support for:

- RSASSA-PSS [RFC8017] using SHA-512 [RFC6234]
- HMAC [RFC2104] using SHA-512 [RFC6234]
- ECDSA using curve P-256 DSS [FIPS186-4] and SHA-512 [RFC6234]
- Ed25519ph, Ed25519ctx, and Ed25519 [RFC8032]

##### 5.1.2.2. rsa-sha1

**Algorithm Name** rsa-sha1

**Status** Deprecated; SHA-1 not secure.

**Description** RSASSA-PKCS1-v1_5 [RFC8017] using SHA-1 [RFC6234]

##### 5.1.2.3. rsa-sha256

**Algorithm Name**
rsa-sha256

**Status** Deprecated; specifying signature algorithm enables attack vector.

**Description** RSASSA-PKCS1-v1_5 [RFC8017] using SHA-256 [RFC6234]

#### 5.1.2.4. hmac-sha256

**Algorithm Name** hmac-sha256

**Status** Deprecated; specifying signature algorithm enables attack vector.

**Description** HMAC [RFC2104] using SHA-256 [RFC6234]

#### 5.1.2.5. ecdsa-sha256

**Algorithm Name** ecdsa-sha256

**Status** Deprecated; specifying signature algorithm enables attack vector.

**Description** ECDSA using curve P-256 DSS [FIPS186-4] and SHA-256 [RFC6234]

## 5.2. HTTP Signature Metadata Parameters Registry

This document defines the Signature-Input Structured Header, whose member values may have parameters containing metadata about a message signature. IANA is asked to create and maintain a new registry titled "HTTP Signature Metadata Parameters" to record and maintain the set of parameters defined for use with member values in the Signature-Input Structured Header. Initial values for this registry are given in Section 5.2.2. Future assignments and modifications to existing assignments are to be made through the Expert Review registration policy [RFC8126] and shall follow the template presented in Section 5.2.1.

### 5.2.1. Registration Template

### 5.2.2. Initial Contents

The table below contains the initial contents of the HTTP Signature Metadata Parameters Registry. Each row in the table represents a distinct entry in the registry.

| Name | Status | Reference(s) |
|---|---|---|
| alg | Active | Section 4.1.1 of this document |
| created | Active | Section 4.1.1 of this document |
| expires | Active | Section 4.1.1 of this document |
| keyId | Active | Section 4.1.1 of this document |

Table 6: Initial contents of the HTTP Signature Metadata Parameters Registry.

## 6. Security Considerations

(( TODO: need to dive deeper on this section; not sure how much of what's referenced below is actually applicable, or if it covers everything we need to worry about. ))

(( TODO: Should provide some recommendations on how to determine what content needs to be signed for a given use case. ))

There are a number of security considerations to take into account when implementing or utilizing this specification. A thorough security analysis of this protocol, including its strengths and weaknesses, can be found in [WP-HTTP-Sig-Audit].

## 7. References

### 7.1. Normative References


**[FIPS186-4]** "Digital Signature Standard (DSS)", 2013, <https://csrc.nist.gov/publications/detail/fips/186/4/final>.

**[HTTP2]** 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>.

**[MESSAGING]** 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>.

**[POSIX.1]** "The Open Group Base Specifications Issue 7, 2018 edition", 2018, <https://pubs.opengroup.org/onlinepubs/9699919799/>.

**[RFC2104]** Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <https://www.rfc-editor.org/info/rfc2104>.

**[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>.

**[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>.

**[SEMANTICS]** 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>.

**[StructuredFields]** "Structured Field Vaues for HTTP", 2020, <https://datatracker.ietf.org/doc/draft-ietf-httpbis-header-structure>.

### 7.2. Informative References

**[RFC3230]** Mogul, J. and A. Van Hoff, "Instance Digests in HTTP", RFC 3230, DOI 10.17487/RFC3230, January 2002, <https://www.rfc-editor.org/info/rfc3230>.

**[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>.

**[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>.

**[RFC7239]** Petersson, A. and M. Nilsson, "Forwarded HTTP Extension", RFC 7239, DOI 10.17487/RFC7239, June 2014, <https://www.rfc-editor.org/info/rfc7239>.

**[RFC7518]** Jones, M., "JSON Web Algorithms (JWA)", RFC 7518, DOI 10.17487/RFC7518, May 2015, <https://www.rfc-editor.org/info/rfc7518>.

**[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>.

**[RFC8017]** Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <https://www.rfc-editor.org/info/rfc8017>.

**[RFC8032]** Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/info/rfc8032>.

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

**[TLS]** 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>.

**[WP-HTTP-Sig-Audit]** "Security Considerations for HTTP Signatures", 2013, <https://web-payments.org/specs/source/http-signatures-audit/>.

## Appendix A. Examples

### A.1. Example Keys

This section provides cryptographic keys that are referenced in example signatures throughout this document. These keys MUST NOT be used for any purpose other than testing.

#### A.1.1. Example Key RSA test

The following key is a 2048-bit RSA public and private key pair:

```
-----BEGIN RSA PUBLIC KEY-----
MIIBCgKCAQEAhAKYdtoeoy8zcAcR874L8cnZxKzAGwd7v36APp7Pv6Q2jdsPBRrw
WEBnez6d0UDKDwGbc6nxfEXAy5mbhgajzrw3MOEt8uA5txSKobBpKDeBLOsdJKFq
MGmXCQvEG7YemcxDTRPxAleIAgYYRjTSd/QBwVW9OwNFhekro3RtlinV0a75jfZg
kne/YiktSvLG34lw2zqXBDTC5NHROUqGTlML4PlNZS5Ri2U4aCNx2rUPRcKIlE0P
uKxI4T+HIaFpv8+rdV6eUgOrB2xeI1dSFFn/nnv5OoZJEIB+VmuKn3DCUcCZSFlQ
PSXSfBDiUGhwOw76WuSSsf1D4b/vLoJ10wIDAQAB
-----END RSA PUBLIC KEY-----

-----BEGIN RSA PRIVATE KEY-----
MIIEqAIBAAKCAQEAhAKYdtoeoy8zcAcR874L8cnZxKzAGwd7v36APp7Pv6Q2jdsP
BRrwWEBnez6d0UDKDwGbc6nxfEXAy5mbhgajzrw3MOEt8uA5txSKobBpKDeBLOsd
JKFqMGmXCQvEG7YemcxDTRPxAleIAgYYRjTSd/QBwVW9OwNFhekro3RtlinV0a75
jfZgkne/YiktSvLG34lw2zqXBDTC5NHROUqGTlML4PlNZS5Ri2U4aCNx2rUPRcKI
lE0PuKxI4T+HIaFpv8+rdV6eUgOrB2xeI1dSFFn/nnv5OoZJEIB+VmuKn3DCUcCZ
SFlQPSXSfBDiUGhwOw76WuSSsf1D4b/vLoJ10wIDAQABAoIBAG/JZuSWdoVHbi56
vjgCgkjg3lkO1KrO3nrdm6nrgA9P9qaPjxuKoWaKO1cBQlE1pSWp/cKncYgD5WxE
CpAnRUXG2pG4zdkzCYzAh1i+c34L6oZoHsirK6oNcEnHveydfzJL5934egm6p8DW
+m1RQ70yUt4uRc0YSor+q1LGJvGQHReF0WmJBZHrhz5e63Pq7lE0gIwuBqL8SMaA
yRXtK+JGxZpImTq+NHvEWWCu09SCq0r838ceQI55SvzmTkwqtC+8AT2zFviMZkKR
Qo6SPsrqItxZWRty2izawTF0Bf5S2VAx7O+6t3wBsQ1sLptoSgX3QblELY5asI0J
YFz7LJECgYkAsqeUJmqXE3LP8tYoIjMIAKiTm9o6psPlc8CrLI9CH0UbuaA2JCOM
cCNq8SyYbTqgnWlB9ZfcAm/cFpA8tYci9m5vYK8HNxQr+8FS3Qo8N9RJ8d0U5Csw
DzMYfRghAfUGwmlWj5hp1pQzAuhwbOXFtxKHVsMPhz1IBtF9Y8jvgqgYHLbmyiu1
mwJ5AL0pYF0G7x81prlARURwHo0Yf52kEw1dxpx+JXER7hQRWQki5/NsUEtv+8RT
qn2m6qte5DXLyn83b1qRscSdnCCwKtKWUug5q2ZbwVOCJCtmRwmnP131lWRYfj67
B/xJ1ZA6X3GEf4sNReNAtaucPEelgR2nsN0gKQKBiGoqHWbK1qYvBxX2X3kbPDkv
9C+celgZd2PW7aGYLCHq7nPbmfDV0yHcWjOhXZ8jRMjmANVR/eLQ2EfsRLdW69bn
f3ZD7JS1fwGnO3exGmHO3HZG+6AvberKYVYNHahNFEw5TsAcQWDLRpkGybBcxqZo
81YCqlqidwfeO5YtlO7etx1xLyqa2NsCeG9A86UjG+aeNnXEIDk1PDK+EuiThIUa
/2IxKzJKWl1BKr2d4xAfR0ZnEYuRrbeDQYgTImOlfW6/GuYIxKYgEKCFHFqJATAG
IxHrq1PDOiSwXd2GmVVYyEmhZnbcp8CxaEMQoevxAta0ssMK3w6UsDtvUvYvF22m
qQKBiD5GwESzsFPy3Ga0MvZpn3D6EJQLgsnrtUPZx+z2Ep2x0xc5orneB5fGyF1P
WtP+fG5Q6Dpdz3LRfm+KwBCWFKQjg7uTxcjerhBWEYPmEMKYwTJF5PBG9/ddvHLQ
EQeNC8fHGg4UXU8mhHnSBt3EA10qQJfRDs15M38eG2cYwB1PZpDHScDnDA0=
-----END RSA PRIVATE KEY-----
```


### A.2. Example keyId Values

The table below maps example keyId values to associated algorithms and/or keys. These are example mappings that are valid only within the context of examples in examples within this and future documents that reference this section. Unless otherwise specified, within the context of examples it should be assumed that the signer and verifier understand these keyId mappings. These keyId values are not reserved, and deployments are free to use them, with these associations or others.

| keyId | Algorithm | Verification Key |
|---|---|---|
| test-key-a | hs2019, using RSASSA-PSS [RFC8017] and SHA-512 [RFC6234] | The public key specified in Appendix A.1.1 |
| test-key-b | rsa-sha256 | The public key specified in Appendix A.1.1 |

Table 7

### A.3. Test Cases

This section provides non-normative examples that may be used as test cases to validate implementation correctness. These examples are based on the following HTTP message:

```
POST /foo?param=value&pet=dog HTTP/1.1
Host: example.com
Date: Tue, 07 Jun 2014 20:51:35 GMT
Content-Type: application/json
Digest: SHA-256=X48E9qOokqqrvdts8nOJRJN3OWDUoyWxBf7kbu9DBPE=
Content-Length: 18

{"hello": "world"}
```

#### A.3.1. Signature Generation

##### A.3.1.1. hs2019 signature over minimal recommended content

This presents metadata for a Signature using hs2019, over minimum recommended data to sign:

| Property | Value |
|---|---|
| Algorithm | hs2019, using RSASSA-PSS [RFC8017] using SHA-512 [RFC6234] |
| Covered Content | *created, *request-target |
| Creation Time | 8:51:35 PM GMT, June 7th, 2014 |
| Expiration Time | Undefined |
| Verification Key Material | The public key specified in Appendix A.1.1. |

Table 8

The Signature Input is:

```
*created: 1402170695
*request-target: post /foo?param=value&pet=dog
```

The signature value is:

QaVaWYfF2da6tG66Xtd0GrVFChJ0fOWUe/C6kaYESPiYYwnMH9egOgyKqgLLY9NQJFk7b
QY834sHEUwjS5ByEBaO3QNwIvqEY1qAAU/2MX14tc9Yn7ELBnaaNHaHkV3xVO9KIuLT7V
6e4OUuGb1axfbXpMgPEql6CEFrn6K95CLuuKP5/gOEcBtmJp5L58gN4VvZrk2OVA6U971
YiEDNuDa4CwMcQMvcGssbc/L3OULTUffD/1VcPtdGImP2uvVQntpT8b2lBeBpfh8MuaV2
vtzidyBYFtAUoYhRWO8+ntqA1q2OK4LMjM2XgDScSVWvGdVd459A0wI9lRlnPap3zg==

A possible Signature-Input and Signature header containing this signature is:

```
Signature-Input: sig1=(*created, *request-target);
    keyId="test-key-a"; created=1402170695
Signature: sig1=:QaVaWYfF2da6tG66Xtd0GrVFChJ0fOWUe/C6kaYESPiYYwnMH9eg
    OgyKqgLLY9NQJFk7bQY834sHEUwjS5ByEBaO3QNwIvqEY1qAAU/2MX14tc9Yn7ELB
    naaNHaHkV3xVO9KIuLT7V6e4OUuGb1axfbXpMgPEql6CEFrn6K95CLuuKP5/gOEcB
    tmJp5L58gN4VvZrk2OVA6U971YiEDNuDa4CwMcQMvcGssbc/L3OULTUffD/1VcPtd
    GImP2uvVQntpT8b2lBeBpfh8MuaV2vtzidyBYFtAUoYhRWO8+ntqA1q2OK4LMjM2X
    gDScSVWvGdVd459A0wI9lRlnPap3zg==:
```

#### A.3.1.2. hs2019 signature covering all header fields

This presents metadata for a Signature using hs2019 that covers all header fields in the request:

| Property | Value |
| --- | --- |
| Algorithm | hs2019, using RSASSA-PSS [RFC8017] using SHA-512 [RFC6234] |
| Covered Content | *created, *request-target, host, date, content-type, digest, content-length |
| Creation Time | 8:51:35 PM GMT, June 7th, 2014 |
| Expiration Time | Undefined |
| Verification Key Material | The public key specified in Appendix A.1.1. |

Table 9

The Signature Input is:

```
*created: 1402170695
*request-target: post /foo?param=value&pet=dog
host: example.com
date: Tue, 07 Jun 2014 20:51:35 GMT
content-type: application/json
digest: SHA-256=X48E9qOokqqrvdts8nOJRJN3OWDUoyWxBf7kbu9DBPE=
content-length: 18
```

The signature value is:

B24UG4FaiE2kSXBNKV4DA91J+mElAhS3mncrgyteAye1GKMpmzt8jkHNjoudtqw3GngGY
3n0mmwjdfn1eA6nAjgeHwl0WXced5tONcCPNzLswqPOiobGeA5y4WE8iBveel30OKYVel
0lZ1OnXOmN5TIEIIPo9LrE+LzZis6A0HA1FRMtKgKGhT3N965pkqfhKbq/V48kpJKT8+c
Zs0TOn4HFMG+OIy6c9ofSBrXD68yxP6QYTz6xH0GMWawLyPLYR52j3I05fK1ylAb6K0ox
PxzQ5nwrLD+mUVPZ9rDs1En6fmOX9xfkZTblG/5D+s1fHHs9dDXCOVkT5dLS8DjdIA==

A possible Signature-Input and Signature header containing this signature is:

```
Signature-Input: sig1=(*request-target, *created, host, date,
        content-type, digest, content-length); keyId="test-key-a";
    alg=hs2019; created=1402170695
Signature: sig1=:B24UG4FaiE2kSXBNKV4DA91J+mElAhS3mncrgyteAye1GKMpmzt8
    jkHNjoudtqw3GngGY3n0mmwjdfn1eA6nAjgeHwl0WXced5tONcCPNzLswqPOiobGe
    A5y4WE8iBveel30OKYVel0lZ1OnXOmN5TIEIIPo9LrE+LzZis6A0HA1FRMtKgKGhT
    3N965pkqfhKbq/V48kpJKT8+cZs0TOn4HFMG+OIy6c9ofSBrXD68yxP6QYTz6xH0G
    MWawLyPLYR52j3I05fK1ylAb6K0oxPxzQ5nwrLD+mUVPZ9rDs1En6fmOX9xfkZTbl
    G/5D+s1fHHs9dDXCOVkT5dLS8DjdIA==:
```

### A.3.2. Signature Verification

#### A.3.2.1. Minimal Required Signature Header

This presents a Signature-Input and Signature header containing only the minimal required parameters:

```
Signature-Input: sig1=(); keyId="test-key-a"; created=1402170695
Signature: sig1=:cxieW5ZKV9R9A70+Ua1A/1FCvVayuE6Z77wDGNVFSiluSzR9TYFV
    vwUjeU6CTYUdbOByGMCee5q1eWWUOM8BIH04Si6VndEHjQVdHqshAtNJk2Quzs6WC
    2DkV0vysOhBSvFZuLZvtCmXRQfYGTGhZqGwq/AAmFbt5WNLQtDrEe0ErveEKBfaz+
    IJ35zhaj+dun71YZ82b/CRfO6fSSt8VXeJuvdqUuVPWqjgJD4n9mgZpZFGBaDdPiw
    pfbVZHzcHrumFJeFHWXH64a+c5GN+TWlP8NPg2zFdEc/joMymBiRelq236WGm5VvV
    9a22RW2/yLmaU/uwf9v40yGR/I1NRA==:
```

The corresponding signature metadata derived from this header field is:

| Property | Value |
|---|---|
| Algorithm | hs2019, using RSASSA-PSS using SHA-256 |
| Covered Content | *created |
| Creation Time | 8:51:35 PM GMT, June 7th, 2014 |
| Expiration Time | Undefined |
| Verification Key Material | The public key specified in Appendix A.1.1. |

Table 10

The corresponding Signature Input is:

```
*created: 1402170695
```

#### A.3.2.2. Minimal Recommended Signature Header

This presents a Signature-Input and Signature header containing only the minimal required and recommended parameters:

```
Signature-Input: sig1=(); alg=hs2019; keyId="test-key-a";
    created=1402170695
Signature: sig1=:cxieW5ZKV9R9A70+Ua1A/1FCvVayuE6Z77wDGNVFSiluSzR9TYFV
    vwUjeU6CTYUdbOByGMCee5q1eWWUOM8BIH04Si6VndEHjQVdHqshAtNJk2Quzs6WC
    2DkV0vysOhBSvFZuLZvtCmXRQfYGTGhZqGwq/AAmFbt5WNLQtDrEe0ErveEKBfaz+
    IJ35zhaj+dun71YZ82b/CRfO6fSSt8VXeJuvdqUuVPWqjgJD4n9mgZpZFGBaDdPiw
    pfbVZHzcHrumFJeFHWXH64a+c5GN+TWlP8NPg2zFdEc/joMymBiRelq236WGm5VvV
    9a22RW2/yLmaU/uwf9v40yGR/I1NRA==:
```

The corresponding signature metadata derived from this header field is:

| Property | Value |
|---|---|
| Algorithm | hs2019, using RSASSA-PSS using SHA-512 |
| Covered Content | *created |
| Creation Time | 8:51:35 PM GMT, June 7th, 2014 |
| Expiration Time | Undefined |
| Verification Key Material | The public key specified in Appendix A.1.1. |

Table 11

The corresponding Signature Input is:

```
*created: 1402170695
```

#### A.3.2.3. Minimal Signature Header using rsa-sha256

This presents a minimal Signature-Input and Signature header for a signature using the rsa-sha256 algorithm:

```
Signature: sig1=(date); alg=rsa-sha256; keyId="test-key-b"
Signature: sig1=:HtXycCl97RBVkZi66ADKnC9c5eSSlb57GnQ4KFqNZplOpNfxqk62
    JzZ484jXgLvoOTRaKfR4hwyxlcyb+BWkVasApQovBSdit9Ml/YmN2IvJDPncrlhPD
    VDv36Z9/DiSO+RNHD7iLXugdXo1+MGRimW1RmYdenl/ITeb7rjfLZ4b9VNnLFtVWw
    rjhAiwIqeLjodVImzVc5srrk19HMZNuUejK6I3/MyN3+3U8tIRW4LWzx6ZgGZUaEE
    P0aBlBkt7Fj0Tt5/P5HNW/Sa/m8smxbOHnwzAJDa10PyjzdIbywlnWIIWtZKPPsoV
    oKVopUWEU3TNhpWmaVhFrUL/O6SN3w==:
```

The corresponding signature metadata derived from this header field is:

| Property | Value |
|---|---|
| Algorithm | rsa-sha256 |
| Covered Content | date |

| Property | Value |
|---|---|
| Creation Time | Undefined |
| Expiration Time | Undefined |
| Verification Key Material | The public key specified in Appendix A.1.1. |

Table 12

The corresponding Signature Input is:

```
date: Tue, 07 Jun 2014 20:51:35 GMT
```

## Appendix B. Topics for Working Group Discussion

*RFC EDITOR: please remove this section before publication*

The draft has known issues that will need to be addressed during development, and these issues have been enumerated but not addressed in this version. Topics are not listed in any particular order.

### B.1. Issues

#### B.1.1. Confusing guidance on algorithm and key identification

The current draft encourages determining the Algorithm metadata property from the keyId field, both in the guidance for the use of algorithm and keyId, and the definition for the hs2019 algorithm and deprecation of the other algorithms in the registry. The current state arose from concern that a malicious party could change the value of the algorithm parameter, potentially tricking the verifier into accepting a signature that would not have been verified under the actual parameter.

Punting algorithm identification into keyId hurts interoperability, since we aren't defining the syntax or semantics of keyId. It actually goes against that claim, as we are dictating that the signing algorithm must be specified by keyId or derivable from it. It also renders the algorithm registry essentially useless. Instead of this approach, we can protect against manipulation of the Signature header field by adding support for (and possibly mandating) including Signature metadata within the Signature Input.

#### B.1.2. Lack of definition of keyId hurts interoperability

The current text leaves the format and semantics of keyId completely up to the implementation. This is primarily due to the fact that most implementers of Cavage have extensive investment in key distribution and management, and just need to plug an identifier into the header. We should support those cases, but we also need to provide guidance for the developer that doesn't have that and just wants to know how to identify a key. It may be enough to punt this to profiling specs, but this needs to be explored more.

### B.1.3. Algorithm Registry duplicates work of JWA

[RFC7518] already defines an IANA registry for cryptographic algorithms. This wasn't used by Cavage out of concerns about complexity of JOSE, and issues with JWE and JWS being too flexible, leading to insecure combinations of options. Using JWA's definitions does not need to mean we're using JOSE, however. We should look at if/how we can leverage JWA's work without introducing too many sharp edges for implementers.

In any use of JWS algorithms, this spec would define a way to create the JWS Signing Input string to be applied to the algorithm. It should be noted that this is incompatible with JWS itself, which requires the inclusion of a structured header in the signature input.

A possible approach is to incorporate all elements of the JWA signature algorithm registry into this spec using a prefix or other marker, such as jws-RS256 for the RSA 256 JSON Web Signature algorithm.

### B.1.4. Algorithm Registry should not be initialized with deprecated entries

The initial entries in this document reflect those in Cavage. The ones that are marked deprecated were done so because of the issue explained in Appendix B.1.1, with the possible exception of rsa-sha1. We should probably just remove that one.

### B.1.5. No percent-encoding normalization of path/query

See: issue #26

The canonicalization rules for *request-target do not perform handle minor, semantically meaningless differences in percent-encoding, such that verification could fail if an intermediary normalizes the effective request URI prior to forwarding the message.

At a minimum, they should be case and percent-encoding normalized as described in sections 6.2.2.1 and 6.2.2.2 of [RFC3986].

### B.1.6. Misleading name for headers parameter

The Covered Content list contains identifiers for more than just headers, so the header parameter name is no longer appropriate. Some alternatives: "content", "signed-content", "covered-content".

#### B.1.7. Changes to whitespace in header field values break verification

Some header field values contain RWS, OWS, and/or BWS. Since the header field value canonicalization rules do not address whitespace, changes to it (e.g., removing OWS or BWS or replacing strings of RWS with a single space) can cause verification to fail.

#### B.1.8. Multiple Set-Cookie headers are not well supported

The Set-Cookie header can occur multiple times but does not adhere to the list syntax, and thus is not well supported by the header field value concatenation rules.

#### B.1.9. Covered Content list is not signed

The Covered Content list should be part of the Signature Input, to protect against malicious changes.

#### B.1.10. Algorithm is not signed

The Algorithm should be part of the Signature Input, to protect against malicious changes.

#### B.1.11. Verification key identifier is not signed

The Verification key identifier (e.g., the value used for the keyId parameter) should be part of the Signature Input, to protect against malicious changes.

#### B.1.12. Max values, precision for Integer String and Decimal String not defined

The definitions for Integer String and Decimal String do not specify a maximum value. The definition for Decimal String (used to provide sub-second precision for Expiration Time) does not define minimum or maximum precision requirements. It should set a sane requirement here (e.g., MUST support up to 3 decimal places and no more).

#### B.1.13. keyId parameter value could break list syntax

The keyId parameter value needs to be constrained so as to not break list syntax (e.g., by containing a comma).

#### B.1.14. Creation Time and Expiration Time do not allow for clock skew

The processing instructions for Creation Time and Expiration Time imply that verifiers are not permitted to account for clock skew during signature verification.

### B.1.15. Should require lowercased header field names as identifiers

The current text allows mixed-case header field names when they are being used as content identifiers. This is unnecessary, as header field names are case-insensitive, and creates opportunity for incompatibility. Instead, content identifiers should always be lowercase.

### B.1.16. Reconcile Date header and Creation Time

The draft is missing guidance on if/how the Date header relates to signature Creation Time. There are cases where they may be different, such as if a signature was pre-created. Should Creation Time default to the value in the Date header if the created parameter is not specified?

### B.1.17. Remove algorithm-specific rules for content identifiers

The rules that restrict when the signer can or must include certain identifiers appear to be related to the pseudo-revving of the Cavage draft that happened when the hs2019 algorithm was introduced. We should drop these rules, as it can be expected that anyone implementing this draft will support all content identifiers.

### B.1.18. Add guidance for signing compressed headers

The draft should provide guidance on how to sign headers when [RFC7541] is used. This guidance might be as simple as "sign the uncompressed header field value."

### B.1.19. Transformations to Via header field value break verification

Intermediaries are permitted to strip comments from the Via header field value, and consolidate related sequences of entries. The canonicalization rules do not account for these changes, and thus they cause signature verification to fail if the Via header is signed. At the very least, guidance on signing or not signing Via headers needs to be included.

### B.1.20. Case changes to case-insensitive header field values break verification

Some header field values are case-insensitive, in whole or in part. The canonicalization rules do not account for this, thus a case change to a covered header field value causes verification to fail.

### B.1.21. Need more examples for Signature header

Add more examples showing different cases e.g, where created or expires are not present.

### B.1.22. Expiration not needed

In many cases, putting the expiration of the signature into the hands of the signer opens up more options for failures than necessary. Instead of the expires, any verifier can use the created field and an internal lifetime or offset to calculate expiration. We should consider dropping the expires field.

## B.2. Features

### B.2.1. Define more content identifiers

It should be possible to independently include the following content and metadata properties in Covered Content:

* The signature's Algorithm
* The signature's Covered Content
* The value used for the keyId parameter
* Request method
* Individual components of the effective request URI: scheme, authority, path, query
* Status code
* Request body (currently supported via Digest header [RFC3230] )

### B.2.2. Multiple signature support

(( Editor's note: I believe this use case is theoretical. Please let me know if this is a use case you have. ))

There may be scenarios where attaching multiple signatures to a single message is useful:

* A gateway attaches a signature over headers it adds (e.g., Forwarded) to messages already signed by the user agent.
* A signer attaches two signatures signed by different keys, to be verified by different entities.

This could be addressed by changing the Signature header syntax to accept a list of parameter sets for a single signature, e.g., by separating parameters with ";" instead of ",". It may also be necessary to include a signature identifier parameter.

#### B.2.3. Support for incremental signing of header field value list items

(( Editor's note: I believe this use case is theoretical. Please let me know if this is a use case you have. ))

Currently, signing a header field value is all-or-nothing: either the entire value is signed, or none of it is. For header fields that use list syntax, it would be useful to be able to specify which items in the list are signed.

A simple approach that allowed the signer to indicate the list size at signing time would allow a signer to sign header fields that are may be appended to by intermediaries as the message makes its way to the recipient. Specifying list size in terms of number of items could introduce risks of list syntax is not strictly adhered to (e.g., a malicious party crafts a value that gets parsed by the application as 5 items, but by the verifier as 4). Specifying list size in number of octets might address this, but more exploration is required.

#### B.2.4. Support expected authority changes

In some cases, the authority of the effective request URI may be expected to change, for example from "public-service-name.example.com" to "service-host-1.public-service-name.example.com". This is commonly the case for services that are hosted behind a load-balancing gateway, where the client sends requests to a publicly known domain name for the service, and these requests are transformed by the gateway into requests to specific hosts in the service fleet.

One possible way to handle this would be to special-case the Host header field to allow verifier to substitute a known expected value, or a value provided in another header field (e.g., Via) when generating the Signature Input, provided that the verifier also recognizes the real value in the Host header. Alternatively, this logic could apply to an (audience) content identifier.

#### B.2.5. Support for signing specific cookies

A signer may only wish to sign one or a few cookies, for example if the website requires its authentication state cookie to be signed, but also sets other cookies (e.g., for analytics, ad tracking, etc.)

## Acknowledgements

This specification is based on the draft-cavage-http-signatures draft. The editor would like to thank the authors of that draft, Mark Cavage and Manu Sporny, for their work on that draft and their continuing contributions.

The editor would also like to thank the following individuals for feedback on and implementations of the draft-cavage-http-signatures draft (in alphabetical order): Mark Adamcin, Mark Allen, Paul Annesley, Karl Boehlmark, Stephane Bortzmeyer, Sarven Capadisli, Liam Dennehy, ductm54, Stephen Farrell, Phillip Hallam-Baker, Eric Holmes, Andrey Kislyuk, Adam Knight, Dave Lehn, Dave Longley, James H. Manger, Ilari Liusvaara, Mark Nottingham, Yoav Nir, Adrian Palmer, Lucas Pardue, Roberto Polli, Julian Reschke, Michael Richardson, Wojciech Rygielski, Adam Scarr, Cory J. Slep, Dirk Stein, Henry Story, Lukasz Szewc, Chris Webber, and Jeffrey Yasskin

## Document History

*RFC EDITOR: please remove this section before publication*

* draft-ietf-httpbis-message-signatures

  - Since -01

    - Replaced unstructured Signature header with Signature-Input and Signature Dictionary Structured Header Fields.

    - Defined content identifiers for individual Dictionary members, e.g., x-dictionary-field:member-name.

    - Defined content identifiers for first N members of a List, e.g., x-list-field:4.

    - Fixed up examples.

    - Updated introduction now that it's adopted.

  - -01

    - Strengthened requirement for content identifiers for header fields to be lower-case (changed from SHOULD to MUST).

    - Added real example values for Creation Time and Expiration Time.

    - Minor editorial corrections and readability improvements.

  - -00

    - Initialized from draft-richanna-http-message-signatures-00, following adoption by the working group.

*draft-richanna-http-message-signatures

--00

- oConverted to xml2rfc v3 and reformatted to comply with RFC style guides.
- oRemoved Signature auth-scheme definition and related content.
- oRemoved conflicting normative requirements for use of algorithm parameter. Now MUST NOT be relied upon.
- oRemoved Extensions appendix.
- oRewrote abstract and introduction to explain context and need, and challenges inherent in signing HTTP messages.
- oRewrote and heavily expanded algorithm definition, retaining normative requirements.
- oAdded definitions for key terms, referenced RFC 7230 for HTTP terms.
- oAdded examples for canonicalization and signature generation steps.
- oRewrote Signature header definition, retaining normative requirements.
- oAdded default values for algorithm and expires parameters.
- oRewrote HTTP Signature Algorithms registry definition. Added change control policy and registry template. Removed suggested URI.
- oAdded IANA HTTP Signature Parameter registry.
- oAdded additional normative and informative references.
- oAdded Topics for Working Group Discussion section, to be removed prior to publication as an RFC.

**Authors' Addresses**

Annabelle Backman (editor)
Amazon
P.O. Box 81226
Seattle, WA 98108-1226
United States of America

Email: richanna@amazon.com
URI: https://www.amazon.com/

Justin Richer
Bespoke Engineering

Email: ietf@justin.richer.org
URI: https://bspk.io/

Manu Sporny
Digital Bazaar
203 Roanoke Street W.
Blacksburg, VA 24060
United States of America

Email: msporny@digitalbazaar.com
URI: https://manu.sporny.org/