Workgroup: HTTP
Internet-Draft:
draft-ietf-httpbis-message-signatures-03
Published: 7 April 2021
Intended Status: Standards Track
Expires: 9 October 2021
Authors: A. Backman, Ed.
Amazon

J. Richer
Bespoke Engineering

M. Sporny
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*

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

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

## 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 9 October 2021.

## Copyright Notice



## Table of Contents

## 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 originally 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:

**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.

The term "Unix time" is defined by [POSIX.1] section 4.16.

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.

Additionally, some examples use '\' line wrapping for long values that contain no whitespace, as per [RFC8792].

### 1.5. Application of HTTP Message Signatures

HTTP Message Signatures are designed to be a general-purpose security mechanism applicable in a wide variety of circumstances and applications. In order to properly and safely apply HTTP Message Signatures, an application or profile of this specification MUST specify all of the following items:

- The set of content identifiers (Section 2) that are expected and required. For example, an authorization protocol could mandate that the Authorization header be covered to protect the authorization credentials and mandate the signature parameters contain a created parameter, while an API expecting HTTP message bodies could require the Digest header to be present and covered.

- A means of retrieving the key material used to verify the signature. An application will usually use the keyid parameter of the signature parameters Section 2.4.2 and define rules for resolving a key from there, though the appropriate key could be known from other means.

- A means of determining the signature algorithm used to verify the signature content is appropriate for the key material. For example, the process could use the alg parameter of the signature parameters Section 2.4.2 to state the algorithm explicitly, derive the algorithm from the key material, or use some pre-configured algorithm agreed upon by the signer and verifier.

- A means of determining that a given key and algorithm presented in the request are appropriate for the request being made. For example, a server expecting only ECDSA signatures should know to reject any RSA signatures, or a server expecting asymmetric cryptography should know to reject any symmetric cryptography.

The details of this kind of profiling are the purview of the application and outside the scope of this specification.

## 2. HTTP Message Signature Covered Content

In order to allow signers and verifiers to establish which content is covered by a signature, this document defines content identifiers for data items covered by an HTTP Message Signature as well as the means for combining these canonicalized values into a signature input string.

Some content within HTTP messages can 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 intermediary transformations. This document defines rules for each content identifier that transform the identifier's associated content into such a canonical form.

Content identifiers are defined using production grammar defined by [RFC8941] section 4. The content identifier is an sf-string value. The content identifier type MAY define parameters which are included using the parameters rule.

```
content-identifier = sf-string parameters
```

Note that this means the value of the identifier itself is encased in double quotes, with parameters following as a semicolon-separated list, such as "cache-control", "date", or "@signature-params".

The following sections define content identifier types, their parameters, their associated content, and their canonicalization rules. The method for combining content identifiers into the signature input string is defined in Section 2.5.

### 2.1. HTTP Headers

The content identifier for an HTTP header is the lowercased form of 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.

Unless overridden by additional parameters and rules, the HTTP header field value MUST be canonicalized with the following steps:

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. Canonicalized Structured HTTP Headers

If value of the the HTTP header in question is a structured field [RFC8941], the content identifier MAY include the sf parameter. If this parameter is included, the HTTP header value MUST be

canonicalized using the rules specified in [RFC8941] section 4. Note that this process will replace any optional whitespace with a single space.

The resulting string is used as the field value input in Section 2.1.

### 2.1.2. 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 using the parameter key on the content identifier for the header. The value of this parameter is a the key being identified, without any parameters present on that key in the original dictionary.

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 [RFC8941] 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";key=a | 1 |
| "x-dictionary";key=b | 2;x=1;y=2 |
| "x-dictionary";key=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 parameter prefix with the value of N as an integer.

A list prefix value is canonicalized by applying the serialization algorithm described in Section 4.1.1 of [RFC8941] 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";prefix=0 | () |
| "x-list-a";prefix=1 | (a) |
| "x-list-a";prefix=3 | (a, b, c) |
| "x-list-a";prefix=6 | (a, b, c, d, e, f) |
| "x-list-b";prefix=0 | () |

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

### 2.4. Specialty Content Fields

Content not found in an HTTP header can be included in the signature base string by defining a content identifier and the canonicalization method for its content.

To differentiate specialty content identifiers from HTTP headers, specialty content identifiers MUST start with the "at" @ character. This specification defines the following specialty content identifiers:

**@request-target** The target request endpoint. Section 2.4.1

**@signature-params** The signature metadata parameters for this signature. Section 2.4.2

Additional specialty content identifiers MAY be defined and registered in the HTTP Signatures Specialty Content Identifier Registry. Section 5.3

#### 2.4.1. Request Target

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.

#### 2.4.1.1. 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 * |

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

### 2.4.2. Signature Parameters

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

The signature parameters special content is identified by the @signature-params identifier.

Its canonicalized value is the serialization of the signature parameters for this signature, including the covered content list with all associated parameters. The following metadata properties are defined:

**Covered Content:**
: An ordered list of content identifiers for headers Section 2.1 and specialty content Section 2.4 that indicates the metadata and message content that is covered by the signature. This list MUST NOT include the @signature-params specialty content identifier itself.

**Algorithm:**
: An HTTP Signature Algorithm defined in the HTTP Signature Algorithms Registry defined in this document, represented as a string. It describes the signing and verification algorithms for the signature.

**Key Material:**
: The key material required to create or verify the signature.

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

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

The signature parameters are serialized using the rules in [RFC8941] section 4 as follows:

1. Let the output be an empty string.

2. Serialize the content identifiers of the covered content as an ordered inner-list according to [RFC8941] section 4.1.1.1 and append this to the output.

3. Append the signature metadata as parameters according to [RFC8941] section 4.1.1.2 in the any order, skipping fields that are not available:

    *alg: Algorithm as an sf-string value.

*keyid: Identifier for the key material as an sf-string value.

*created: Creation time as an sf-integer timestamp value.

*expires: Expiration time as an sf-integer timestamp value.

Note that the inner-list serialization is used for the covered content instead of the sf-list serialization in order to facilitate this value's additional inclusion in the Signature-Input header's dictionary, as discussed in Section 4.1.

This example shows a canonicalized value for the parameters of a given signature:

```
# NOTE: '\' line wrapping per RFC 8792
("@request-target" "host" "date" "cache-control" "x-empty-header"
  "x-example"); keyid="test-key-a"; alg="rsa-pss-sha512"; \
  created=1402170695; expires=1402170995
```

Note that an HTTP message could contain multiple signatures, but only the signature parameters used for the current signature are included.

#### 2.4.2.1. Canonicalization Examples

Given the following signature parameters:

| Property | Value |
|---|---|
| Algorithm | rsa-pss-sha512 |
| Covered Content | @request-target, host, date, cache-control, x-emptyheader, x-example, x-dictionary;key=b, x-dictionary;key=a, x-list;prefix=3 |
| Creation Time | 1402174295 |
| Expiration Time | 1402174595 |
| Verification Key Material | The public key provided in Appendix B.1.1 and identified by the keyid value "test-key-a". |

Table 5

The signature parameter value is defined as:

```
# NOTE: '\' line wrapping per RFC 8792
"@signature-params": ("@request-target" "host" "date" "cache-control" \
  "x-empty-header" "x-example" "x-dictionary";key=b \
  "x-dictionary";key=a "x-list";prefix=3); keyid="test-key-a"; \
  alg="rsa-pss-sha512"; created=1402170695; expires=1402170995
```

### 2.5. Creating the Signature Input String

The signature input is a US-ASCII string containing the content that is covered by the signature. To create the signature input string, the signer or verifier concatenates together entries for each identifier in the signature's covered content and parameters using the following algorithm:

1. Let the output be an empty string.

2. For each covered content item in the covered content list (in order):

    1. Append the identifier for the covered content serialized according to the content-identifier rule.

    2. Append a single colon ":"

    3. Append a single space " "

    4. Append the covered content's canonicalized value, as defined by the covered content type. Section 2.1 and Section 2.4

    5. Append a single newline "\\n"

3. Append the signature parameters Section 2.4.2 as follows:

    1. Append the identifier for the signature parameters serialized according to the content-identifier rule, "@signature-params"

    2. Append a single colon ":"

    3. Append a single space " "

    4. Append the signature parameters' canonicalized value as defined in Section 2.4.2

4. Return the output string.

If covered content references an identifier that cannot be resolved to a value in the message, the implementation MUST produce an error. Such situations are included but not limited to: * The signer or verifier does not understand the content identifier. * The identifier identifies a header field that is not present in the message or whose value is malformed. * The identifier is a Dictionary member identifier that references a header field that is not present in the message, is not a Dictionary Structured Field, or whose value is malformed. * The identifier is a List Prefix member

identifier that references a header field that is not present in the message, is not a List Structured Field, or whose value is malformed. * The identifier is a Dictionary member identifier that references a member that is not present in the header field value, or whose value is malformed. E.g., the identifier is "x-dictionary";key=c and the value of the x-dictionary header field is a=1, b=2 * The identifier is a List Prefix member identifier that specifies more List members than are present the header field. E.g., the identifier is "x-list";prefix=3 and the value of the x-list header field is (1, 2).

For the non-normative example Signature metadata in Table 6, the corresponding Signature Input is:

```
# NOTE: '\' line wrapping per RFC 8792
"@request-target": get /foo
"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";key=b: 2
"x-dictionary";key=a: 1
"x-list";prefix=3: (a, b, c)
"@signature-params": ("@request-target" "host" "date" "cache-control" \
  "x-empty-header" "x-example" "x-dictionary";key=b \
  "x-dictionary";key=a "x-list";prefix=3); keyid="test-key-a"; \
  created=1402170695; expires=1402170995
```

Figure 1: Non-normative example Signature Input

## 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. Creating a Signature

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

1. The signer chooses an HTTP signature algorithm and key material for signing. 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 the algorithm and key material is out of scope for this document.

2. The signer sets the signature's creation time to the current time.

3. If applicable, the signer sets the signature's expiration time property to the time at which the signature is to expire.

4. 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 covered content identifier MUST reference either an HTTP header or a specialty content field listed in Section 2.4 or its associated registry.

    *Signers SHOULD include @request-target in the covered content list list.

    *Signers SHOULD include a date stamp in some form, such as using the date header. Alternatively, the created signature metadata parameter can fulfil this role.

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

    *Note that the @signature-params specialty identifier is not explicitly listed in the list of covered content identifiers, because it is required to always be present as the last line in the signature input. This ensures that a signature always covers its own metadata.

5. The signer creates the signature input string. Section 2.5

6. The signer signs the signature input with the chosen signing algorithm using the key material chosen by the signer. Several signing algorithms are defined in in Section 3.3.

7. The signer then encodes the result of that operation as a Base64-encoded string [RFC4648]. This string is the signature output value.

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 |
|---|---|
| Covered Content | @request-target, host, date, cache-control, x-emptyheader, x-example, x-dictionary;key=b, x-dictionary;key=a, x-list;prefix=3 |
| Creation Time | 1402174295 |
| Expiration Time | 1402174595 |
| Verification Key Material | The public key provided in Appendix B.1.1 and identified by the keyid value "test-key-a". |

Table 6: Non-normative example metadata values

For the non-normative example signature metadata and signature input in Figure 1, the corresponding signature value is:

```
# NOTE: '\' line wrapping per RFC 8792
K2qGT5srn2OGbOIDzQ6kYT+ruaycnDAAUpKv+ePFfD0RAxn/1BUeZx/Kdrq32DrfakQ6b\
PsvB9aqZqognNT6be4olHROIkeV879RrsrObury8L9SCEibeoHyqU/yCjphSmEdd7WD+z\
rchK57quskKwRefy2iEC5S2uAH0EPyOZKWlvbKmKu5q4CaB8X/I5/+HLZLGvDiezqi6/7\
p2Gngf5hwZ0lSdy39vyNMaaAT0tKo6nuVw0S1MVg1Q7MpWYZs0soHjttq0uLIA3DIbQfL\
iIvK6/l0BdWTU7+2uQj7lBkQAsFZHoA96ZZgFquQrXRlmYOh+Hx5D9fJkXcXe5tmAg==
```

Figure 2: Non-normative example signature value

### 3.2. Verifying a Signature

In order to verify a signature, a verifier MUST follow the following algorithm:

1. Examine the signature's parameters 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 contents are required to be covered by the signature. Section 3.2.1

2. Determine the verification key material for this signature. If the key material is known through external means such as static configuration or external protocol negotiation, the verifier will use that. If the key is identified in the signature parameters, the verifier will dereference this to appropriate key material to use with the signature. The verifier has to determine the trustworthiness of the key material for the context in which the signature is presented.

3. Determine the algorithm to apply for verification:

    1. If the algorithm is known through external means such as static configuration or external protocol negotiation, the verifier will use this algorithm.

    2. If the algorithm is explicitly stated in the signature parameters using a value from the HTTP Message Signatures registry, the verifier will use the referenced algorithm.

    3. If the algorithm can be determined from the keying material, such as through an algorithm field on the key value itself, the verifier will use this algorithm.

4. Use the received HTTP message and the signature's metadata to recreate the signature input, using the process described in Section 2.5. The value of the @signature-params input is the value of the SignatureInput header field for this signature serialized according to the rules described in Section 2.4.2, not including the signature's label from the SignatureInput header.

5. If the key material is appropriate for the algorithm, apply the verification algorithm to the signature, signature input, signature parameters, key material, and algorithm. The results of the verification algorithm function are the final results of the signature verification. Several algorithms are defined in Section 3.3.

If any of the above steps fail, the signature validation fails.

### 3.2.1. 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.

### 3.3. Signature Algorithm Methods

HTTP Message signatures MAY use any cryptographic digital signature or MAC method that allows for the signing of the signature input string. This section contains several common algorithm parameters that can be communicated through the algorithm signature parameter defined in Section 2.4.2, by reference to the key material, or through agreement between the signer and verifier.

Signatures are generated from and verified against the byte values of the signature input string defined in Section 2.5.

#### 3.3.1. RSASSA-PSS using SHA-512

To sign using this algorithm, the signer applies the RSASSA-PSS-SIGN (K, M) function [RFC8017] with the signer's private signing key (K) and the signature input string (M) Section 2.5. The hash SHA-512 [RFC6234] is applied to the signature input string to create the digest content to which the digital signature is applied. The resulting signed content (S) is Base64-encoded as described in Section 3.1. The resulting encoded value is the HTTP message signature output.

To verify using this algorithm, the verifier applies the RSASSA-PSS-VERIFY ((n, e), M, S) function [RFC8017] using the public key material ((n, e)). The verifier re-creates the signature input

string (M) from the received message, as defined in Section 2.5. The hash function SHA-512 [RFC6234] is applied to the signature input string to create the digest content to which the verification function is applied. The verifier decodes the HTTP message signature from Base64 as described in Section 3.2 to give the http message signature to be verified (S). The results of the verification function are compared to the http message signature to determine if the signature presented is valid.

#### 3.3.2. RSASSA-PKCS1-v1_5 using SHA-256

To sign using this algorithm, the signer applies the RSASSA-PKCS1-V1_5-SIGN (K, M) function [RFC8017] to signer's private signing key (K) and the signature input string (M) Section 2.5. The hash SHA-256 [RFC6234] is applied to the signature input string to create the digest content to which the digital signature is applied. The resulting signed content (S) is Base64-encoded as described in Section 3.1. The resulting encoded value is the HTTP message signature output.

To verify using this algorithm, the verifier applies the RSASSA-PKCS1-V1_5-VERIFY ((n, e), M, S) function [RFC8017] using the public key material ((n, e)). The verifier re-creates the signature input string (M) from the received message, as defined in Section 2.5. The hash function SHA-256 [RFC6234] is applied to the signature input string to create the digest content to which the verification function is applied. The verifier decodes the HTTP message signature from Base64 as described in Section 3.2 to give the http message signature to be verified (S). The results of the verification function are compared to the http message signature to determine if the signature presented is valid.

#### 3.3.3. HMAC using SHA-256

To sign and verify using this algorithm, the signer applies the HMAC function [RFC2104] with the shared signing key (K) and the signature input string (text) Section 2.5. The hash function SHA-256 [RFC6234] is applied to the signature input string to create the digest content to which the HMAC is applied, giving the signature result.

For signing, the resulting signed content is Base64-encoded as described in Section 3.1. The resulting encoded value is the HTTP message signature output.

For verification, the verifier decodes the HTTP message signature from Base64 as described in Section 3.2 to give the signature to be compared to the output of the HMAC function. The results of the comparison determine the validity of the signature presented.

#### 3.3.4. ECDSA using curve P-256 DSS and SHA-256

To sign using this algorithm, the signer applies the ECDSA algorithm [FIPS186-4] using curve P-256 with signer's private signing key and the signature input string Section 2.5. The hash function SHA-256 [RFC6234] is applied to the signature input string to create the digest content to which the digital signature is applied. The resulting signed content is Base64-encoded as described in Section 3.1. The resulting encoded value is the HTTP message signature output.

To verify using this algorithm, the verifier applies the ECDSA algorithm [FIPS186-4] using the public key material. The verifier re-creates the signature input string defined in Section 2.5. The hash function SHA-256 [RFC6234] is applied to the signature input string to create the digest content to which the verification function is applied. The verifier decodes the HTTP message signature from Base64 as described in Section 3.2 to give the signature to be verified. The results of the verification function are compared to the http message signature to determine if the signature presented is valid.

#### 3.3.5. JSON Web Signature (JWS) algorithms

If the signing algorithm is a JOSE signing algorithm from the JSON Web Signature and Encryption Algorithms Registry established by [RFC7518], the JWS algorithm definition determines the signature and hashing algorithms to apply for both signing and verification.

For both signing and verification, the HTTP messages signature input string Section 2.5 is used as the entire "JWS Signing Input". The JWS Header defined in [RFC7517] is not used, nor is the input string first encoded in Base64 before applying the algorithm.

The JWS algorithm MUST NOT be none and MUST NOT be any algorithm with a JOSE Implementation Requirement of Prohibited.

## 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 [RFC8941] 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 serialization of the covered content including all signature metadata parameters, using the serialization process defined in Section 2.4.2.

```
# NOTE: '\' line wrapping per RFC 8792
Signature-Input: sig1=("@request-target" "host" "date"
    "cache-control" "x-empty-header" "x-example"); keyid="test-key-a";
    alg="rsa-pss-sha512"; created=1402170695; expires=1402170995
```

To facilitate signature validation, the Signature-Input header MUST contain the same serialization value used in generating the signature input.

### 4.2. The 'Signature' HTTP Header

The Signature HTTP header field is a Dictionary Structured Header [RFC8941] 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.

```
# NOTE: '\' line wrapping per RFC 8792
Signature: sig1=:K2qGT5srn2OGbOIDzQ6kYT+ruaycnDAAUpKv+ePFfD0RAxn/1BUe\
    Zx/Kdrq32DrfakQ6bPsvB9aqZqognNT6be4olHROIkeV879RrsrObury8L9SCEibe\
    oHyqU/yCjphSmEdd7WD+zrchK57quskKwRefy2iEC5S2uAH0EPyOZKWlvbKmKu5q4\
    CaB8X/I5/+HLZLGvDiezqi6/7p2Gngf5hwZ0lSdy39vyNMaaAT0tKo6nuVw0S1MVg\
    1Q7MpWYZs0soHjttq0uLIA3DIbQfLiIvK6/l0BdWTU7+2uQj7lBkQAsFZHoA96ZZg\
    FquQrXRlmYOh+Hx5D9fJkXcXe5tmAg==:
```

### 4.3. Examples

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

```
# NOTE: '\' line wrapping per RFC 8792

Signature-Input: sig1=("@request-target" "host" "date"
    "cache-control" "x-empty-header" "x-example"); keyid="test-key-a";
    alg="rsa-pss-sha512"; 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:

```
# NOTE: '\' line wrapping per RFC 8792

X-Forwarded-For: 192.0.2.123
Signature-Input: reverse_proxy_sig=("host" "date"
    "signature";key=sig1 "x-forwarded-for"); keyid="test-key-a";
    alg="rsa-pss-sha512"; created=1402170695; expires=1402170695
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 brief description of the algorithm used to sign the signature input string.

**Specification document(s):**
Reference to the document(s) that specify the token endpoint authorization method, preferably including a URI that can be used to retrieve a copy of the document(s). An indication of the relevant sections may also be included but is not required.

#### 5.1.2. Initial Contents

##### 5.1.2.1. rsa-pss-sha512

**Algorithm Name:**
rsa-pss-sha512

**Status:**
Active

**Definition:**
RSASSA-PSS using SHA-256

**Specification document(s):**
[[This document]] Section 3.3.1

#### 5.1.2.2. rsa-v1_5-sha256

**Algorithm Name:**
rsa-v1_5-sha256

**Status:**
Active

**Description:**
RSASSA-PKCS1-v1_5 using SHA-256

**Specification document(s):**
[[This document]] Section 3.3.2

#### 5.1.2.3. hmac-sha256

**Algorithm Name:**
hmac-sha256

**Status:**
Active

**Description:**
HMAC using SHA-256

**Specification document(s):**
[[This document]] Section 3.3.3

#### 5.1.2.4. ecdsa-p256-sha256

**Algorithm Name:**
ecdsa-p256-sha256

**Status:**
Active

**Description:**
ECDSA using curve P-256 DSS and SHA-256

**Specification document(s):**
[[This document]] Section 3.3.4

### 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 2.4.2 of this document |
| created | Active | Section 2.4.2 of this document |
| expires | Active | Section 2.4.2 of this document |
| keyid | Active | Section 2.4.2 of this document |

Table 7: Initial contents of the HTTP Signature Metadata Parameters Registry.

### 5.3. HTTP Signature Specialty Content Identifiers Registry

This document defines a method for canonicalizing HTTP message content, including content that can be generated from the context of the HTTP message outside of the HTTP headers. This content is identified by a unique key. IANA is asked to create and maintain a new registry typed "HTTP Signature Specialty Content Identifiers" to record and maintain the set of non-header content identifiers and their canonicalization method. Initial values for this registry are given in Section 5.3.2. 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 5.3.1.

#### 5.3.1. Registration Template

#### 5.3.2. Initial Contents

The table below contains the initial contents of the HTTP Signature Specialty Content Identifiers Registry.

| Name | Status | Reference(s) |
|---|---|---|
| @request-target | Active | Section 2.4.1 of this document |
| @signature-params | Active | Section 2.4.2 of this document |

Table 8: Initial contents of the HTTP Signature Specialty Content Identifiers 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/rfc/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/rfc/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/rfc/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/rfc/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/rfc/rfc8174>.

**[RFC8792]** Watsen, K., Auerswald, E., Farrel, A., and Q. Wu, "Handling Long Lines in Content of Internet-Drafts and RFCs", RFC 8792, DOI 10.17487/RFC8792, June 2020, <https://www.rfc-editor.org/rfc/rfc8792>.

**[RFC8941]** Nottingham, M. and P-H. Kamp, "Structured Field Values for HTTP", RFC 8941, DOI 10.17487/RFC8941, February 2021, <https://www.rfc-editor.org/rfc/rfc8941>.

**[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/rfc/rfc7231>.

### 7.2. Informative References

**[RFC4648]** Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/rfc/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/rfc/rfc6234>.

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

**[RFC7517]** Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, <https://www.rfc-editor.org/rfc/rfc7517>.

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

**[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/rfc/rfc8017>.

**[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/rfc/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/rfc/rfc8446>.

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

## Appendix A. Detecting HTTP Message Signatures

There have been many attempts to create signed HTTP messages in the past, including other non-standard definitions of the Signature header used within this specification. It is recommended that developers wishing to support both this specification and other historial drafts do so carefully and deliberately, as incompatibilities between this specification and various versions of other drafts could lead to problems.

It is recommended that implementers first detect and validate the Signature-Input header defined in this specification to detect that this standard is in use and not an alternative. If the Signature-Input header is present, all Signature headers can be parsed and interpreted in the context of this draft.

## Appendix B. Examples

### B.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.

#### B.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-----
```

### B.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 | rsa-pss-sha512 | The public key specified in Appendix B.1.1 |
| test-key-b | rsa-v1_5-sha256 | The public key specified in Appendix B.1.1 |

Table 9

### B.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"}
```

#### B.3.1. Signature Verification

##### B.3.1.1. Minimal Signature Header using rsa-pss-sha512

This presents a minimal Signature-Input and Signature header for a signature using the rsa-pss-sha512 algorithm:

```
# NOTE: '\' line wrapping per RFC 8792

Signature: sig1=("date"); alg="rsa-pss-sha512"; 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-pss-sha512 |
| Covered Content | date |

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

Table 10

The corresponding Signature Input is:

```
"date": Tue, 07 Jun 2014 20:51:35 GMT
"@signature-params": ("date"); alg="rsa-pss-sha512"; keyid="test-key-b"
```


## Acknowledgements

This specification was initially based on the draft-cavage-http-signatures internet draft. The editors 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

--03

oClarified signing and verification processes.

oUpdated algorithm and key selection method.

oClearly defined core algorithm set.

oDefined JOSE signature mapping process.

oRemoved legacy signature methods.

oDefine signature parameters separately from "signature" object model.

- oDefine serialization values for signature-input header based on signature input.

--02

- oRemoved editorial comments on document sources.

- oRemoved in-document issues list in favor of tracked issues.

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

- oDefined content identifiers for individual Dictionary members, e.g., "x-dictionary-field";key=member-name.

- oDefined content identifiers for first N members of a List, e.g., "x-list-field":prefix=4.

- oFixed up examples.

- oUpdated introduction now that it's adopted.

- oDefined specialty content identifiers and a means to extend them.

- oRequired signature parameters to be included in signature.

- oAdded guidance on backwards compatibility, detection, and use of signature methods.

--01

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

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

- oMinor editorial corrections and readability improvements.

--00

- oInitialized 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/