Workgroup: HTTP
Internet-Draft:
draft-ietf-httpbis-message-signatures-06
Published: 13 August 2021
Intended Status: Standards Track
Expires: 14 February 2022
Authors: A. Backman, Ed.
Amazon

J. Richer
Bespoke Engineering

M. Sporny
Digital Bazaar

HTTP Message Signatures

Abstract

This document describes a mechanism for creating, encoding, and verifying digital signatures or message authentication codes over components of 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. This document also describes a means for requesting that a signature be applied to a subsequent HTTP message in an ongoing HTTP exchange.

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 14 February 2022.

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 components of an HTTP message. The mechanism allows applications to create digital signatures or message authentication codes (MACs) over only the components of the message that are 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 signing mechanism described in this document consists of three parts:

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

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

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

This document also provides a mechanism for one party to signal to another party that a signature is desired in one or more subsequent messages. This optional negotiation mechanism can be used along with opportunistic or application-driven message signatures by either party.

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 signature input. Since the exact raw bytes of the message cannot be relied upon as a reliable source of signature input, the signer and verifier must derive the signature input 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 components are 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 message components 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 components 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 components 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:

HTTP Message Signature:
: A digital signature or keyed MAC that covers one or more portions of an HTTP message. Note that a given HTTP Message can contain multiple HTTP Message Signatures.

Signer:
: The entity that is generating or has generated an HTTP Message Signature. Note that multiple entities can act as signers and apply separate HTTP Message Signatures to a given HTTP Message.

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.

HTTP Message Component:
: A portion of an HTTP message that is capable of being covered by an HTTP Message Signature.

HTTP Message Component Identifier:
: A value that uniquely identifies a specific HTTP Message Component in respect to a particular HTTP Message Signature and the HTTP Message it applies to.

HTTP Message Component Value:
: The value associated with a given component identifier within the context of a particular HTTP Message. Component values are derived from the HTTP Message and are usually subject to a canonicalization process.

Covered Components:
: An ordered set of HTTP message component identifiers for fields (Section 2.1) and specialty components (Section 2.3) that indicates the set of message components covered by the signature, not including the @signature-params specialty identifier itself. The order of this set is preserved and communicated between the signer and verifier to facilitate reconstruction of the signature input.

Signature Input:
: The sequence of bytes processed by the HTTP Message Signature algorithm to produce the HTTP Message Signature. The signature

input is generated by the signer and verifier using the covered components set and the HTTP Message.

HTTP Message Signature Algorithm:
A cryptographic algorithm that describes the signing and verification process for the signature. When expressed explicitly, the value maps to a string defined in the HTTP Signature Algorithms Registry defined in this document.

Key Material:
The key material required to create or verify the signature. The key material is often identified with an explicit key identifier, allowing the signer to indicate to the verifier which key was used.

Creation Time:
A timestamp representing the point in time that the signature was generated, as asserted by the signer.

Expiration Time:
A timestamp representing the point in time at which the signature expires, as asserted by the signer. A signature's expiration time could be undefined, indicating that the signature does not expire from the perspective of the signer.

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

This document contains non-normative examples of partial and complete HTTP messages. Some examples use a single trailing backslash '' to indicate line wrapping for long values, as per [RFC8792]. The \ character and leading spaces on wrapped lines are not part of the value.

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 component 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 2.3.1) 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 is appropriate for the key material. For example, the process could use the alg parameter of the signature parameters (Section 2.3.1) 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.

An application using signatures also has to ensure that the verifier will have access to all required information to re-create the signature input string. For example, a server behind a reverse proxy would need to know the original request URI to make use of identifiers like @target-uri. Additionally, an application using signatures in responses would need to ensure that clients receiving signed responses have access to all the signed portions, including any portions of the request that were signed by the server.

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

2. HTTP Message Components

In order to allow signers and verifiers to establish which components are covered by a signature, this document defines component identifiers for components covered by an HTTP Message Signature, a set of rules for deriving and canonicalizing the values associated with these component identifiers from the HTTP Message, and the means for combining these canonicalized values into a signature input string. The values for these items MUST be accessible to both the signer and the verifier of the message, which means these are usually derived from aspects of the HTTP message or signature itself.

Some HTTP message components can undergo transformations that change the bitwise value without altering meaning of the component's value (for example, the merging together of header fields with the same name). Message component values 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 component identifier that transform the identifier's associated component value into such a canonical form.

Component identifiers are serialized using the production grammar defined by RFC8941, Section 4 [RFC8941]. The component identifier itself is an sf-string value and MAY define parameters which are included using the parameters rule.

```
component-identifier = sf-string parameters
```

Note that this means the value of the component 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 component identifier types, their parameters, their associated values, and the canonicalization rules for their values. The method for combining component identifiers into the signature input is defined in Section 2.4.

2.1. HTTP Fields

The component identifier for an HTTP field is the lowercased form of its field name. While HTTP field names are case-insensitive, implementations MUST use lowercased field names (e.g., content-type, date, etag) when using them as component identifiers.

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

1. Create an ordered list of the field values of each instance of the 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 component value.

2.1.1. Canonicalized Structured HTTP Fields

If value of the the HTTP field in question is a structured field ([RFC8941]), the component identifier MAY include the sf parameter. If this parameter is included, the HTTP field value MUST be canonicalized using the rules specified in Section 4 of RFC8941

[RFC8941]. For example, this process will replace any optional internal whitespace with a single space character.

The resulting string is used as the component value 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:

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

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
"host"	www.example.com
"x-empty-header"	
"x-obs-fold-header"	Obsolete line folding.
"x-ows-header"	Leading and trailing whitespace.
"x-dictionary"	a=1, b=2;x=1;y=2, c=(a b c)
"x-dictionary";sf	a=1, b=2;x=1;y=2, c=(a b c)

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 component identifier for the field. 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 [RFC8941] on a Dictionary containing only that item.

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 known 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 component identifiers, given that field:

Component Identifier	Component 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. Specialty Components

Message components not found in an HTTP field can be included in the signature input by defining a component identifier and the canonicalization method for its component value.

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

@signature-params
: The signature metadata parameters for this signature. (Section 2.3.1)

@method
: The method used for a request. (Section 2.3.2)

@target-uri
: The full target URI for a request. (Section 2.3.3)

@authority
: The authority of the target URI for a request. (Section 2.3.4)

@scheme
: The scheme of the target URI for a request. (Section 2.3.5)

@request-target
: The request target. (Section 2.3.6)

@path
: The absolute path portion of the target URI for a request. (Section 2.3.7)

@query
: The query portion of the target URI for a request. (Section 2.3.8)

@query-params
: The parsed query parameters of the target URI for a request. (Section 2.3.9)

@status
: The status code for a response. (Section 2.3.10).

@request-response
: A signature from a request message that resulted in this response message. (Section 2.3.11)

Additional specialty component identifiers MAY be defined and registered in the HTTP Signatures Specialty Component Identifier Registry. (Section 6.3)

2.3.1. Signature Parameters

HTTP Message Signatures have metadata properties that provide information regarding the signature's generation and verification, such as the set of covered components, a timestamp, identifiers for verification key material, and other utilities.

The signature parameters component identifier is @signature-params.

The signature parameters component value is the serialization of the signature parameters for this signature, including the covered components set with all associated parameters. These parameters include any of the following:

- created: Creation time as an sf-integer UNIX timestamp value. Sub-second precision is not supported. Inclusion of this parameter is RECOMMENDED.

*expires: Expiration time as an sf-integer UNIX timestamp value. Sub-second precision is not supported.

*nonce: A random unique value generated for this signature.

*alg: The HTTP message signature algorithm from the HTTP Message Signature Algorithm Registry, as an sf-string value.

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

Additional parameters can be defined in the HTTP Signature Parameters Registry (Section 6.2.2).

The signature parameters component value is serialized as a parameterized inner list using the rules in Section 4 of RFC8941 [RFC8941] as follows:

1. Let the output be an empty string.

2. Determine an order for the component identifiers of the covered components. Once this order is chosen, it cannot be changed. This order MUST be the same order as used in creating the signature input (Section 2.4).

3. Serialize the component identifiers of the covered components, including all parameters, as an ordered inner-list according to Section 4.1.1.1 of RFC8941 [RFC8941] and append this to the output.

4. Determine an order for any signature parameters. Once this order is chosen, it cannot be changed.

5. Append the parameters to the inner-list in the chosen order according to Section 4.1.1.2 of RFC8941 [RFC8941], skipping parameters that are not available or not used for this message signature.

6. The output contains the signature parameters component value.

Note that the inner-list serialization is used for the covered component value instead of the sf-list serialization in order to facilitate this value's inclusion in message fields such as the Signature-Input field'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

("@target-uri" "@authority" "date" "cache-control" "x-empty-header" \
  "x-example");keyid="test-key-rsa-pss";alg="rsa-pss-sha512";\
  created=1618884475;expires=1618884775
```

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

2.3.2. Method

The @method component identifier refers to the HTTP method of a request message. The component value of is canonicalized by taking the value of the method as a string. Note that the method name is case-sensitive as per [SEMANTICS] Section 9.1, and conventionally standardized method names are uppercase US-ASCII. If used, the @method component identifier MUST occur only once in the covered components.

For example, the following request message:

```
POST /path?param=value HTTP/1.1
Host: www.example.com
```

Would result in the following @method value:

```
"@method": POST
```

If used in a response message, the @method component identifier refers to the associated component value of the request that triggered the response message being signed.

2.3.3. Target URI

The @target-uri component identifier refers to the target URI of a request message. The component value is the full absolute target URI of the request, potentially assembled from all available parts including the authority and request target as described in [SEMANTICS] Section 7.1. If used, the @target-uri component identifier MUST occur only once in the covered components.

For example, the following message sent over HTTPS:

```
POST /path?param=value HTTP/1.1
Host: www.example.com
```

Would result in the following @target-uri value:

```
"@target-uri": https://www.example.com/path?param=value
```

If used in a response message, the @target-uri component identifier refers to the associated component value of the request that triggered the response message being signed.

2.3.4. Authority

The @authority component identifier refers to the authority component of the target URI of the HTTP request message, as defined in [SEMANTICS] Section 7.2. In HTTP 1.1, this is usually conveyed using the Host header, while in HTTP 2 and HTTP 3 it is conveyed using the :authority pseudo-header. The value is the fully-qualified authority component of the request, comprised of the host and, optionally, port of the request target, as a string. The component value MUST be normalized according to the rules in [SEMANTICS] Section 4.2.3. Namely, the host name is normalized to lowercase and the default port is omitted. If used, the @authority component identifier MUST occur only once in the covered components.

For example, the following request message:

```
POST /path?param=value HTTP/1.1
Host: www.example.com
```

Would result in the following @authority component value:

```
"@authority": www.example.com
```

If used in a response message, the @authority component identifier refers to the associated component value of the request that triggered the response message being signed.

2.3.5. Scheme

The @scheme component identifier refers to the scheme of the target URL of the HTTP request message. The component value is the scheme as a string as defined in [SEMANTICS] Section 4.2. While the scheme itself is case-insensitive, it MUST be normalized to lowercase for inclusion in the signature input string. If used, the @scheme component identifier MUST occur only once in the covered components.

For example, the following request message requested over plain HTTP:

```
POST /path?param=value HTTP/1.1
Host: www.example.com
```

Would result in the following @scheme value:

```
"@scheme": http
```

If used in a response message, the @scheme component identifier refers to the associated component value of the request that triggered the response message being signed.

2.3.6. Request Target

The @request-target component identifier refers to the full request target of the HTTP request message, as defined in [SEMANTICS] Section 7.1. The component value of the request target can take different forms, depending on the type of request, as described below. If used, the @request-target component identifier MUST occur only once in the covered components.

For HTTP 1.1, the component value is equivalent to the request target portion of the request line. However, this value is more difficult to reliably construct in other versions of HTTP. Therefore, it is NOT RECOMMENDED that this identifier be used when versions of HTTP other than 1.1 might be in use.

The origin form value is combination of the absolute path and query components of the request URL. For example, the following request message:

```
POST /path?param=value HTTP/1.1
Host: www.example.com
```

Would result in the following @request-target component value:

```
"@request-target": /path?param=value
```

The following request to an HTTP proxy with the absolute-form value, containing the fully qualified target URI:

```
GET https://www.example.com/path?param=value HTTP/1.1
```

Would result in the following @request-target component value:

```
"@request-target": https://www.example.com/path?param=value
```

The following CONNECT request with an authority-form value, containing the host and port of the target:

```
CONNECT www.example.com:80 HTTP/1.1
Host: www.example.com
```

Would result in the following @request-target component value:

```
"@request-target": www.example.com:80
```

The following OPTIONS request message with the asterisk-form value, containing a single asterisk * character:

```
OPTIONS * HTTP/1.1
Host: www.example.com
```

Would result in the following @request-target component value:

```
"@request-target": *
```

If used in a response message, the @request-target component identifier refers to the associated component value of the request that triggered the response message being signed.

2.3.7. Path

The @path component identifier refers to the target path of the HTTP request message. The component value is the absolute path of the request target defined by [RFC3986], with no query component and no trailing ? character. The value is normalized according to the rules in [SEMANTICS] Section 4.2.3. Namely, an empty path string is normalized as a single slash / character, and path components are represented by their values after decoding any percent-encoded octets. If used, the @path component identifier MUST occur only once in the covered components.

For example, the following request message:

```
POST /path?param=value HTTP/1.1
Host: www.example.com
```

Would result in the following @path value:

"@path": /path

If used in a response message, the @path identifier refers to the associated component value of the request that triggered the response message being signed.

2.3.8. Query

The @query component identifier refers to the query component of the HTTP request message. The component value is the entire normalized query string defined by [RFC3986], including the leading ? character. The value is normalized according to the rules in [SEMANTICS] Section 4.2.3. Namely, percent-encoded octets are decoded. If used, the @query component identifier MUST occur only once in the covered components.

For example, the following request message:

```
POST /path?param=value&foo=bar&baz=batman HTTP/1.1
Host: www.example.com
```

Would result in the following @query value:

"@query": ?param=value&foo=bar&baz=batman

The following request message:

```
POST /path?queryString HTTP/1.1
Host: www.example.com
```

Would result in the following @query value:

"@query": ?queryString

If used in a response message, the @query component identifier refers to the associated component value of the request that triggered the response message being signed.

2.3.9. Query Parameters

If a request target URI uses HTML form parameters in the query string as defined in [HTMLURL] Section 5, the @query-params component identifier allows addressing of individual query parameters. The query parameters MUST be parsed according to [HTMLURL] Section 5.1, resulting in a list of (nameString, valueString) tuples. The REQUIRED name parameter of each input

identifier contains the nameString of a single query parameter. Several different named query parameters MAY be included in the covered components. Single named parameters MAY occur in any order in the covered components.

The component value of a single named parameter is the the valueString of the named query parameter defined by [HTMLURL] Section 5.1, which is the value after percent-encoded octets are decoded. Note that this value does not include any leading ? characters, equals sign =, or separating & characters. Named query parameters with an empty valueString are included with an empty string as the component value.

If a parameter name occurs multiple times in a request, all parameter values of that name MUST be included in separate signature input lines in the order in which the parameters occur in the target URI.

For example for the following request:

```
POST /path?param=value&foo=bar&baz=batman&qux= HTTP/1.1
Host: www.example.com
```

Indicating the baz, qux and param named query parameters in would result in the following @query-param value:

```
"@query-params";name="baz": batman
"@query-params";name="qux":
"@query-params";name="param": value
```

If used in a response message, the @query-params component identifier refers to the associated component value of the request that triggered the response message being signed.

2.3.10. Status Code

The @status component identifier refers to the three-digit numeric HTTP status code of a response message as defined in [SEMANTICS] Section 15. The component value is the serialized three-digit integer of the HTTP response code, with no descriptive text. If used, the @status component identifier MUST occur only once in the covered components.

For example, the following response message:

```
HTTP/1.1 200 OK
Date: Fri, 26 Mar 2010 00:05:00 GMT
```

Would result in the following @status value:

```
"@status": 200
```

The @status component identifier MUST NOT be used in a request message.

2.3.11. Request-Response Signature Binding

When a signed request message results in a signed response message, the @request-response component identifier can be used to cryptographically link the request and the response to each other by including the identified request signature value in the response's signature input without copying the value of the request's signature to the response directly. This component identifier has a single REQUIRED parameter:

key Identifies which signature from the response to sign.

The component value is the sf-binary representation of the signature value of the referenced request identified by the key parameter.

For example, when serving this signed request:

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

POST /foo?param=value&pet=dog HTTP/1.1
Host: example.com
Date: Tue, 20 Apr 2021 02:07:55 GMT
Content-Type: application/json
Content-Length: 18
Signature-Input: sig1=("@authority" "content-type")\
  ;created=1618884475;keyid="test-key-rsa-pss"
Signature: sig1=:KuhJjsOKCiISnKHh2rln5ZNIrkRvue0DSu5rif3g7ckTbbX7C4\
  Jp3bcGmi8zZsFRURSQTcjbHdJtN8ZXlRptLOPGHkUa/3Qov79gBeqvHNUO4bhI27p\
  4WzD1bJDG9+6ml3gkrs7rOvMtROObPuc78A95fa4+skS/t2T7OjkfsHAm/enxf1fA\
  wkk15xj0n6kmriwZfgUlOqyff0XLwuH4XFvZ+ZTyxYNoo2+EfFg4NVfqtSJch2WDY\
  7n/qmhZOzMfyHlggWYFnDpyP27VrzQCQg8rM1Crp6MrwGLa94v6qP8pq0sQVq2DLt\
  4NJSoRRqXTvqlWIRnexmcKXjQFVz6YSA==:

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

This would result in the following unsigned response message:

```
HTTP/1.1 200 OK
Date: Tue, 20 Apr 2021 02:07:56 GMT
Content-Type: application/json
Content-Length: 62

{"busy": true, "message": "Your call is very important to us"}
```

The server signs the response with its own key and includes the signature of sig1 from the request in the covered components of the response. The signature input string for this example is:

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

"content-type": application/json
"content-length": 62
"@status": 200
"@request-response";key="sig1": :KuhJjsOKCiISnKHh2rln5ZNIrkRvue0DSu\
  5rif3g7ckTbbX7C4Jp3bcGmi8zZsFRURSQTcjbHdJtN8ZXlRptLOPGHkUa/3Qov79\
  gBeqvHNUO4bhI27p4WzD1bJDG9+6ml3gkrs7rOvMtROObPuc78A95fa4+skS/t2T7\
  OjkfsHAm/enxf1fAwkk15xj0n6kmriwZfgUlOqyff0XLwuH4XFvZ+ZTyxYNoo2+Ef\
  Fg4NVfqtSJch2WDY7n/qmhZOzMfyHlggWYFnDpyP27VrzQCQg8rM1Crp6MrwGLa94\
  v6qP8pq0sQVq2DLt4NJSoRRqXTvqlWIRnexmcKXjQFVz6YSA==:
"@signature-params": ("content-type" "content-length" "@status" \
  "@request-response";key="sig1");created=1618884475\
  ;keyid="test-key-ecc-p256"
```

The signed response message is:

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

HTTP/1.1 200 OK
Date: Tue, 20 Apr 2021 02:07:56 GMT
Content-Type: application/json
Content-Length: 62
Signature-Input: sig1=("content-type" "content-length" "@status" \
  "@request-response";key="sig1");created=1618884475\
  ;keyid="test-key-ecc-p256"
Signature: sig1=:crVqK54rxvdx0j7qnt2RL1oQSf+o21S/6Uk2hyFpoIfOT0q+Hv\
  msYAXUXzo0Wn8NFWh/OjWQOXHAQdVnTk87Pw==:

{"busy": true, "message": "Your call is very important to us"}
```

Since the request's signature value itself is not repeated in the response, the requester MUST keep the original signature value around long enough to validate the signature of the response.

The @request-response component identifier MUST NOT be used in a request message.

2.4. Creating the Signature Input String

The signature input is a US-ASCII string containing the canonicalized HTTP message components 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 components (including their parameters) using the following algorithm:

1. Let the output be an empty string.
2. For each message component item in the covered components set (in order):
    1. Append the component identifier for the covered component serialized according to the component-identifier rule.
    2. Append a single colon ":"
    3. Append a single space " "
    4. Append the covered component's canonicalized component value, as defined by the HTTP message component type. (Section 2.1 and Section 2.3)
    5. Append a single newline "\\n"
3. Append the signature parameters component (Section 2.3.1) as follows:
    1. Append the component identifier for the signature parameters serialized according to the component-identifier rule, i.e. "@signature-params"
    2. Append a single colon ":"
    3. Append a single space " "
    4. Append the signature parameters' canonicalized component value as defined in Section 2.3.1
4. Return the output string.

If covered components reference a component identifier that cannot be resolved to a component 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 component identifier.
- The component identifier identifies a field that is not present in the message or whose value is malformed.
- The component identifier is a Dictionary member identifier that references a field that is not present in the message, is not a Dictionary Structured Field, or whose value is malformed.
- The component identifier is a Dictionary member identifier that references a member that is not present in the 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

In the following non-normative example, the HTTP message being signed is the following request:

```
GET /foo HTTP/1.1
Host: example.org
Date: Tue, 20 Apr 2021 02:07:55 GMT
X-Example: Example header
        with some whitespace.
X-Empty-Header:
Cache-Control: max-age=60
Cache-Control: must-revalidate
```

The covered components consist of the @method, @path, and @authority specialty component identifiers followed by the Cache-Control, X-Empty-Header, X-Example HTTP headers, in order. The signature parameters consist of a creation timestamp is 1618884475 and the key identifier is test-key-rsa-pss. The signature input string for this message with these parameters is:

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

"@method": GET
"@path": /foo
"@authority": example.org
"cache-control": max-age=60, must-revalidate
"x-empty-header":
"x-example": Example header with some whitespace.
"@signature-params": ("@method" "@path" "@authority" \
  "cache-control" "x-empty-header" "x-example");created=1618884475\
  ;keyid="test-key-rsa-pss"
```

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 components of an HTTP message in addition to metadata about the signature itself. When successfully verified against an HTTP message, an HTTP Message Signature provides cryptographic proof that the message is semantically equivalent to the message for which the signature was generated, with respect to the subset of message components that was signed.

3.1. Creating a Signature

In order to create a signature, a signer MUST follow the following algorithm:

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 set of component identifiers representing the message components to be covered by the signature, and attaches signature metadata parameters to this set. The serialized value of this is later used as the value of the Signature-Input field as described in Section 4.1.

   *Once an order of covered components is chosen, the order MUST NOT change for the life of the signature.

* Each covered component identifier MUST be either an HTTP field in the message Section 2.1 or a specialty component identifier listed in Section 2.3 or its associated registry.

* Signers of a request SHOULD include some or all of the message control data in the covered components, such as the @method, @authority, @target-uri, or some combination thereof.

* Signers SHOULD include the created signature metadata parameter to indicate when the signature was created.

* The @signature-params specialty component identifier is not explicitly listed in the list of covered component 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.

* Further guidance on what to include in this set and in what order is out of scope for this document.

5. The signer creates the signature input string based on these signature parameters. (Section 2.4)

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 byte array output of the signature function is the HTTP message signature output value to be included in the Signature field as defined in Section 4.2.

For example, given the HTTP message and signature parameters in the example in Section 2.4, the example signature input string when signed with the test-key-rsa-pss key in Appendix B.1.2 gives the following message signature output value, encoded in Base64:

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

P0wLUszWQjoi54udOtydf9IWTfNhy+r53jGFj9XZuP4uKwxyJo1RSHi+oEF1FuX6O29\
d+lbxwwBao1BAgadijW+7O/PyezlTnqAOVPWx9GlyntiCiHzC87qmSQjvu1CFyFuWSj\
dGa3qLYYlNm7pVaJFalQiKWnUaqfT4LyttaXyoyZW84jS8gyarxAiWI97mPXU+OVM64\
+HVBHmnEsS+lTeIsEQo36T3NFf2CujWARPQg53r58RmpZ+J9eKR2CD6IJQvacn5A4Ix\
5BUAVGqlyp8JYm+S/CWJi31PNUjRRCusCVRj05NrxABNFv3r5S9IXf2fYJK+eyW4AiG\
VMvMcOg==
```

Figure 2: Non-normative example signature value

3.2. Verifying a Signature

A verifier processes a signature and its associated signature input parameters in concert with each other.

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

1. Parse the Signature and Signature-Input fields and extract the signatures to be verified.
    1. If there is more than one signature value present, determine which signature should be processed for this message. If an applicable signature is not found, produce an error.
    2. If the chosen Signature value does not have a corresponding Signature-Input value, produce an error.
2. Parse the values of the chosen Signature-Input field to get the parameters for the signature to be verified.
3. Parse the value of the corresponding Signature field to get the byte array value of the signature to be verified.
4. Examine the signature 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 message components are required to be covered by the signature. (Section 3.2.1)
5. 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. If a key is identified that the verifier does not know, does not trust for this request, or does not match something preconfigured, the verification MUST fail.
6. 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. If the algorithm is specified in more that one location, such as through static configuration and the algorithm signature parameter, or the algorithm signature parameter and from the key material itself, the resolved algorithms MUST be the same. If the algorithms are not the same, the verifier MUST vail the verification.

7. Use the received HTTP message and the signature's metadata to recreate the signature input, using the process described in Section 2.4. The value of the @signature-params input is the value of the SignatureInput field for this signature serialized according to the rules described in Section 2.3.1, not including the signature's label from the Signature-Input field.

8. If the key material is appropriate for the algorithm, apply the verification algorithm to the signature, recalculated signature input, signature parameters, key material, and algorithm. Several algorithms are defined in Section 3.3.

9. The results of the verification algorithm function are the final results of the signature verification.

If any of the above steps fail or produce an error, 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.

- *Prohibition of signature metadata parameters, such as runtime algorithm signaling with the alg parameter.

- *Prohibiting the use of certain algorithms, or mandating the use of a specific algorithm.

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

- *Enforcing uniqueness of a nonce value.

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 is appropriate for the key material, environment, and needs of the signer and verifier. All signatures are generated from and verified against the byte values of the signature input string defined in Section 2.4.

Each signature algorithm method takes as its input the signature input string as a set of byte values (I), the signing key material (Ks), and outputs the signature output as a set of byte values (S):

```
HTTP_SIGN (I, Ks)  ->  S
```

Each verification algorithm method takes as its input the recalculated signature input string as a set of byte values (I), the verification key material (Kv), and the presented signature to be verified as a set of byte values (S) and outputs the verification result (V) as a boolean:

```
HTTP_VERIFY (I, Kv, S) -> V
```

This section contains several common algorithm methods. The method to use can be communicated through the algorithm signature parameter defined in Section 2.3.1, by reference to the key material, or through mutual agreement between the signer and verifier.

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 2.4). The mask generation function is MGF1 as specified in [RFC8017] with a hash function of SHA-512 [RFC6234]. The salt length (sLen) is 64 bytes. The hash function (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 byte array (S) is the HTTP message signature output used in Section 3.1.

To verify using this algorithm, the verifier applies the RSASSA-PSS-VERIFY ((n, e), M, S) function [RFC8017] using the public key portion of the verification key material ((n, e)) and the signature input string (M) re-created as described in Section 3.2. The mask generation function is MGF1 as specified in [RFC8017] with a hash function of SHA-512 [RFC6234]. The salt length (sLen) is 64 bytes. The hash function (Hash) SHA-512 [RFC6234] is applied to the signature input string to create the digest content to which the verification function is applied. The verifier extracts the HTTP message signature to be verified (S) as described in Section 3.2. 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] with the signer's private signing key (K) and the signature input string (M) (Section 2.4). 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 byte array (S) is the HTTP message signature output used in Section 3.1.

To verify using this algorithm, the verifier applies the RSASSA-PKCS1-V1_5-VERIFY ((n, e), M, S) function [RFC8017] using the public key portion of the verification key material ((n, e)) and the signature input string (M) re-created as described in Section 3.2. 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 extracts the HTTP message signature to be verified (S) as described in Section 3.2. 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 2.4). 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 value is the HTTP message signature output used in Section 3.1.

For verification, the verifier extracts the HTTP message signature to be verified (S) as described in Section 3.2. The output of the HMAC function is compared to the value of the HTTP message signature, and 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 the signer's private signing key and the signature input string (Section 2.4). 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 byte array is the HTTP message signature output used in Section 3.1.

To verify using this algorithm, the verifier applies the ECDSA algorithm [FIPS186-4] using the public key portion of the verification key material and the signature input string re-created as described in Section 3.2. 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 extracts the HTTP message signature to be verified (S) as described in Section 3.2. 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. There is no use of the explicit alg signature parameter when using JOSE signing algorithms.

For both signing and verification, the HTTP messages signature input string (Section 2.4) is used as the entire "JWS Signing Input". The

JOSE Header defined in [RFC7517] is not used, and the signature input string is not first encoded in Base64 before applying the algorithm. The output of the JWS signature is taken as a byte array prior to the Base64url encoding used in JOSE.

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 fields, both defined within this specification. When attached to a message, an HTTP message signature is identified by a label. This label MUST be unique within a given HTTP message and MUST be used in both the Signature-Input and Signature. The label is chosen by the signer, except where a specific label is dictated by protocol negotiations.

An HTTP message signature MUST use both fields containing the same labels: the Signature HTTP field contains the signature value, while the Signature-Input HTTP field identifies the covered components and parameters that describe how the signature was generated. Each field contains labeled values and MAY contain multiple labeled values, where the labels determine the correlation between the Signature and Signature-Input fields.

4.1. The 'Signature-Input' HTTP Field

The Signature-Input HTTP field is a Dictionary Structured Field [RFC8941] containing the metadata for one or more message signatures generated from components 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 components including all signature metadata parameters, using the serialization process defined in Section 2.3.1.

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

Signature-Input: sig1=("@method" "@target-uri" "host" "date" \
  "cache-control" "x-empty-header" "x-example");created=1618884475\
  ;keyid="test-key-rsa-pss"
```

To facilitate signature validation, the Signature-Input field value MUST contain the same serialized value used in generating the signature input string's @signature-params value.

The signer MAY include the Signature-Input field as a trailer to facilitate signing a message after its content has been processed by the signer. However, since intermediaries are allowed to drop trailers as per [SEMANTICS], it is RECOMMENDED that the Signature-Input HTTP field be included only as a header to avoid signatures being inadvertently stripped from a message.

Multiple Signature-Input fields MAY be included in a single HTTP message. The signature labels MUST be unique across all field values.

4.2. The 'Signature' HTTP Field

The Signature HTTP field is a Dictionary Structured field [RFC8941] containing one or more message signatures generated from components within the HTTP message. Each member's name is a signature identifier that is present as a member name in the Signature-Input Structured field 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 field that does not have a corresponding member in the HTTP message's Signature-Input HTTP field MUST be ignored.

NOTE: '\' line wrapping per RFC 8792

```
Signature: sig1=:P0wLUszWQjoi54udOtydf9IWTfNhy+r53jGFj9XZuP4uKwxyJo\
  1RSHi+oEF1FuX6O29d+lbxwwBao1BAgadijW+7O/PyezlTnqAOVPWx9GlyntiCiHz\
  C87qmSQjvu1CFyFuWSjdGa3qLYYlNm7pVaJFalQiKWnUaqfT4LyttaXyoyZW84jS8\
  gyarxAiWI97mPXU+OVM64+HVBHmnEsS+lTeIsEQo36T3NFf2CujWARPQg53r58Rmp\
  Z+J9eKR2CD6IJQvacn5A4Ix5BUAVGqlyp8JYm+S/CWJi31PNUjRRCusCVRj05NrxA\
  BNFv3r5S9IXf2fYJK+eyW4AiGVMvMcOg==:
```

The signer MAY include the Signature field as a trailer to facilitate signing a message after its content has been processed by the signer. However, since intermediaries are allowed to drop trailers as per [SEMANTICS], it is RECOMMENDED that the Signature-Input HTTP field be included only as a header to avoid signatures being inadvertently stripped from a message.

Multiple Signature fields MAY be included in a single HTTP message. The signature labels MUST be unique across all field values.

4.3. Multiple Signatures

Multiple distinct signatures MAY be included in a single message. Since Signature-Input and Signature are both defined as Dictionary Structured fields, they can be used to include multiple signatures within the same HTTP message by using distinct signature labels. For

example, a signer may include multiple signatures signing the same message components with different keys or algorithms to support verifiers with different capabilities, or a reverse proxy may include information about the client in fields when forwarding the request to a service host, including a signature over the client's original signature values.

The following is a non-normative example of header fields a reverse proxy sets in addition to the examples in the previous sections.

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

Forwarded: for=192.0.2.123
Signature-Input: sig1=("@method" "@path" "@authority" \
    "cache-control" "x-empty-header" "x-example")\
    ;created=1618884475;keyid="test-key-rsa-pss"
Signature: sig1=:P0wLUszWQjoi54udOtydf9IWTfNhy+r53jGFj9XZuP4uKwxyJo\
    1RSHi+oEF1FuX6O29d+lbxwwBao1BAgadijW+7O/PyezlTnqAOVPWx9GlyntiCi\
    HzC87qmSQjvu1CFyFuWSjdGa3qLYYlNm7pVaJFalQiKWnUaqfT4LyttaXyoyZW8\
    4jS8gyarxAiWI97mPXU+OVM64+HVBHmnEsS+lTeIsEQo36T3NFf2CujWARPQg53\
    r58RmpZ+J9eKR2CD6IJQvacn5A4Ix5BUAVGqlyp8JYm+S/CWJi31PNUjRRCusCV\
    Rj05NrxABNFv3r5S9IXf2fYJK+eyW4AiGVMvMcOg==:
```

The client's request includes a signature value under the label sig1, which the proxy signs in addition to the Forwarded header defined in [RFC7239]. Note that since the client's signature already covers the client's Signature-Input value for sig1, this value is transitively covered by the proxy's signature and need not be added explicitly. This results in a signature input string of:

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

"signature";key="sig1": :P0wLUszWQjoi54udOtydf9IWTfNhy+r53jGFj9XZuP\
  4uKwxyJo1RSHi+oEF1FuX6O29d+lbxwwBao1BAgadijW+7O/PyezlTnqAOVPWx9Gl\
  yntiCiHzC87qmSQjvu1CFyFuWSjdGa3qLYYlNm7pVaJFalQiKWnUaqfT4LyttaXyo\
  yZW84jS8gyarxAiWI97mPXU+OVM64+HVBHmnEsS+lTeIsEQo36T3NFf2CujWARPQg\
  53r58RmpZ+J9eKR2CD6IJQvacn5A4Ix5BUAVGqlyp8JYm+S/CWJi31PNUjRRCusCV\
  Rj05NrxABNFv3r5S9IXf2fYJK+eyW4AiGVMvMcOg==:
"forwarded": for=192.0.2.123
"@signature-params": ("signature";key="sig1" "forwarded")\
  ;created=1618884480;keyid="test-key-rsa";alg="rsa-v1_5-sha256"
```

And a signature output value of:

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

cjGvZwbsq9JwexP9TIvdLiivxqLINwp/ybAc19KOSQuLvtmMt3EnZxNiE+797dXK2cj\
PPUFqoZxO8WWx1SnKhAU9SiXBr99NTXRmA1qGBjqus/1Yxwr8keB8xzFt4inv3J3zP0\
k6TlLkRJstkVnNjuhRIUA/ZQCo8jDYAl4zWJJjppy6Gd1XSg03iUa0sju1yj6rcKbMA\
BBuzhUz4G0u1hZkIGbQprCnk/FOsqZHpwaWvY8P3hmcDHkNaavcokmq+3EBDCQTzgwL\
qfDmV0vLCXtDda6CNO2Zyum/pMGboCnQn/VkQ+j8kSydKoFg6EbVuGbrQijth6I0dDX\
2/HYcJg==
```

These values are added to the HTTP request message by the proxy. The original signature is included under the identifier sig1, and the reverse proxy's signature is included under the label proxy_sig. The proxy uses the key test-key-rsa to create its signature using the rsa-v1_5-sha256 signature algorithm, while the client's original signature was made using the key id of test-key-rsa-pss and an RSA PSS signature algorithm.

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

Forwarded: for=192.0.2.123
Signature-Input: sig1=("@method" "@path" "@authority" \
    "cache-control" "x-empty-header" "x-example")\
    ;created=1618884475;keyid="test-key-rsa-pss", \
  proxy_sig=("signature";key="sig1" "forwarded")\
    ;created=1618884480;keyid="test-key-rsa";alg="rsa-v1_5-sha256"
Signature: sig1=:P0wLUszWQjoi54udOtydf9IWTfNhy+r53jGFj9XZuP4uKwxyJo\
    1RSHi+oEF1FuX6O29d+lbxwwBao1BAgadijW+7O/PyezlTnqAOVPWx9GlyntiCi\
    HzC87qmSQjvu1CFyFuWSjdGa3qLYYlNm7pVaJFalQiKWnUaqfT4LyttaXyoyZW8\
    4jS8gyarxAiWI97mPXU+OVM64+HVBHmnEsS+lTeIsEQo36T3NFf2CujWARPQg53\
    r58RmpZ+J9eKR2CD6IJQvacn5A4Ix5BUAVGqlyp8JYm+S/CWJi31PNUjRRCusCV\
    Rj05NrxABNFv3r5S9IXf2fYJK+eyW4AiGVMvMcOg==:, \
  proxy_sig=:cjGvZwbsq9JwexP9TIvdLiivxqLINwp/ybAc19KOSQuLvtmMt3EnZx\
    NiE+797dXK2cjPPUFqoZxO8WWx1SnKhAU9SiXBr99NTXRmA1qGBjqus/1Yxwr8k\
    eB8xzFt4inv3J3zP0k6TlLkRJstkVnNjuhRIUA/ZQCo8jDYAl4zWJJjppy6Gd1X\
    Sg03iUa0sju1yj6rcKbMABBuzhUz4G0u1hZkIGbQprCnk/FOsqZHpwaWvY8P3hm\
    cDHkNaavcokmq+3EBDCQTzgwLqfDmV0vLCXtDda6CNO2Zyum/pMGboCnQn/VkQ+\
    j8kSydKoFg6EbVuGbrQijth6I0dDX2/HYcJg==:
```

The proxy's signature and the client's original signature can be verified independently for the same message, based on the needs of the application. Since the proxy's signature covers the client signature, the backend service fronted by the proxy can trust that the proxy has validated the incoming signature.

5. Requesting Signatures

While a signer is free to attach a signature to a request or response without prompting, it is often desirable for a potential

verifier to signal that it expects a signature from a potential signer using the Accept-Signature field.

The message to which the requested signature is applied is known as the "target message". When the Accept-Signature field is sent in an HTTP Request message, the field indicates that the client desires the server to sign the response using the identified parameters and the target message is the response to this request. All responses from resources that support such signature negotiation SHOULD either be uncacheable or contain a Vary header field that lists Accept-Signature, in order to prevent a cache from returning a response with a signature intended for a different request.

When the Accept-Signature field is used in an HTTP Response message, the field indicates that the server desires the client to sign its next request to the server with the identified parameters, and the target message is the client's next request. The client can choose to also continue signing future requests to the same server in the same way.

The target message of an Accept-Signature field MUST include all labeled signatures indicated in the Accept-Header signature, each covering the same identified components of the Accept-Signature field.

The sender of an Accept-Signature field MUST include identifiers that are appropriate for the type of the target message. For example, if the target message is a response, the identifiers can not include the @status identifier.

5.1. The Accept-Signature Field

The Accept-Signature HTTP header field is a Dictionary Structured field [RFC8941] containing the metadata for one or more requested message signatures to be generated from message components of the target HTTP message. Each member describes a single message signature. The member's name is an identifier that uniquely identifies the requested message signature within the context of the target HTTP message. The member's value is the serialization of the desired covered components of the target message, including any allowed signature metadata parameters, using the serialization process defined in Section 2.3.1.

NOTE: '\' line wrapping per RFC 8792

```
Accept-Signature: sig1=("@method" "@target-uri" "host" "date" \
  "cache-control" "x-empty-header" "x-example")\
  ;keyid="test-key-rsa-pss"
```

The requested signature MAY include parameters, such as a desired algorithm or key identifier. These parameters MUST NOT include parameters that the signer is expected to generate, including the created and nonce parameters.

5.2. Processing an Accept-Signature

The receiver of an Accept-Signature field fulfills that header as follows:

1. Parse the field value as a Dictionary
2. For each member of the dictionary:
    1. The name of the member is the label of the output signature as specified in Section 4.1
    2. Parse the value of the member to obtain the set of covered component identifiers
    3. Process the requested parameters, such as the signing algorithm and key material. If any requested parameters cannot be fulfilled, or if the requested parameters conflict with those deemed appropriate to the target message, the process fails and returns an error.
    4. Select any additional parameters necessary for completing the signature
    5. Create the Signature-Input and Signature header values and associate them with the label
3. Optionally create any additional Signature-Input and Signature values, with unique labels not found in the Accept-Signature field
4. Combine all labeled Signature-Input and Signature values and attach both headers to the target message

Note that by this process, a signature applied to a target message MUST have the same label, MUST have the same set of covered component, and MAY have additional parameters. Also note that the target message MAY include additional signatures not specified by the Accept-Signature field.

6. IANA Considerations

6.1. 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 6.1.2. 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 6.1.1.

Algorithms referenced by algorithm identifiers have to be fully defined with all parameters fixed. Algorithm identifiers in this registry are to be interpreted as whole string values and not as a combination of parts. That is to say, it is expected that implementors understand rsa-pss-sha512 as referring to one specific algorithm with its hash, mask, and salt values set as defined here. Implementors do not parse out the rsa, pss, and sha512 portions of the identifier to determine parameters of the signing algorithm from the string.

6.1.1. 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.

6.1.2. Initial Contents

6.1.2.1. 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

6.1.2.2. 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

6.1.2.3. hmac-sha256

Algorithm Name:
hmac-sha256

Status:
Active

Description:
HMAC using SHA-256

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

6.1.2.4. 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

6.2. HTTP Signature Metadata Parameters Registry

This document defines the signature parameters structure, the values of which 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 parameters structure. Initial values for this registry are given in Section 6.2.2. 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 6.2.1.

6.2.1. Registration Template

6.2.2. 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 2.3.1 of this document
created	Active	Section 2.3.1 of this document
expires	Active	Section 2.3.1 of this document
keyid	Active	Section 2.3.1 of this document
nonce	Active	Section 2.3.1 of this document

Table 3: Initial contents of the HTTP Signature Metadata Parameters Registry.

6.3. HTTP Signature Specialty Component Identifiers Registry

This document defines a method for canonicalizing HTTP message components, including components that can be generated from the context of the HTTP message outside of the HTTP fields. These components are identified by a unique string, known as the component identifier. IANA is asked to create and maintain a new registry typed "HTTP Signature Specialty Component Identifiers" to record and maintain the set of non-field component identifiers and the methods to produce their associated component values. Initial values for this registry are given in Section 6.3.2. 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 6.3.1.

6.3.1. Registration Template

6.3.2. Initial Contents

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

Name	Status	Target	Reference
@signature-params	Active	Request, Response	Section 2.3.1 of this document
@method	Active	Request, Related-Response	Section 2.3.2 of this document
@authority	Active	Request, Related-Response	Section 2.3.4 of this document
@scheme	Active	Request, Related-Response	Section 2.3.5 of this document
@target-uri	Active	Request, Related-Response	Section 2.3.3 of this document
@request-target	Active	Request, Related-Response	Section 2.3.6 of this document
@path	Active	Request, Related-Response	Section 2.3.7 of this document
@query	Active	Request, Related-Response	Section 2.3.8 of this document
@query-params	Active	Request, Related-Response	Section 2.3.9 of this document
@status	Active	Response	Section 2.3.10 of this document
@request-response	Active	Section 2.3.11 of this document	

Table 4: Initial contents of the HTTP Signature Specialty Component Identifiers Registry.

7. 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 components need 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].

8. References

8.1. Normative References

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

[HTMLURL] "URL (Living Standard)", 2021, <https://url.spec.whatwg.org/>.

[MESSAGING] Fielding, R. T., Nottingham, M., and J. Reschke, "HTTP/1.1", Work in Progress, Internet-Draft, draft-ietf-httpbis-messaging-17, 25 July 2021, <https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-messaging-17>.

[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>.

[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/rfc/rfc3986>.

[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. T., Nottingham, M., and J. Reschke, "HTTP Semantics", Work in Progress, Internet-Draft, draft-ietf-

httpbis-semantics-17, 25 July 2021, <https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-semantics-17>.

8.2. Informative References

[I-D.ietf-httpbis-client-cert-field] Campbell, B. and M. Bishop, "Client-Cert HTTP Header Field: Conveying Client Certificate Information from TLS Terminating Reverse Proxies to Origin Server Applications", Work in Progress, Internet-Draft, draft-ietf-httpbis-client-cert-field-00, 8 June 2021, <https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-client-cert-field-00>.

[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 field used within this specification. It is recommended that developers wishing to support both this specification and other historical drafts do so carefully and deliberately, as incompatibilities between this specification and various versions of other drafts could lead to unexpected problems.

It is recommended that implementers first detect and validate the Signature-Input field defined in this specification to detect that this standard is in use and not an alternative. If the Signature-Input field is present, all Signature fields 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.

The key identifiers for each key are used throughout the examples in this specification. It is assumed for these examples that the signer and verifier can unambiguously dereference all key identifiers used here, and that the keys and algorithms used are appropriate for the context in which the signature is presented.

B.1.1. Example Key RSA test

The following key is a 2048-bit RSA public and private key pair, referred to in this document as test-key-rsa:

```
-----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.1.2. Example RSA PSS Key

The following key is a 2048-bit RSA public and private key pair, referred to in this document as test-key-rsa-pss:

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAr4tmm3r20Wd/PbqvP1s2
+QEtvpuRaV8Yq40gjUR8y2Rjxa6dpG2GXHbPfvMs8ct+Lh1GH45x28Rw3Ry53mm+
oAXjyQ86OnDkZ5N8lYbggD4O3w6M6pAvLkhk95AndTrifbIFPNU8PPMO7OyrFAHq
gDsznjPFmTOtCEcN2Z1FpWgchwuYLPL+Wokqltd11nqqzi+bJ9cvSKADYdUAAN5W
Utzdpiy6LbTgSxP7ociU4Tn0g5I6aDZJ7A8Lzo0KSyZYoA485mqcO0GVAdVw9lq4
aOT9v6d+nb4bnNkQVklLQ3fVAvJm+xdDOp9LCNCN48V2pnDOkFV6+U9nV5oyc6XI
2wIDAQAB
-----END PUBLIC KEY-----

-----BEGIN PRIVATE KEY-----
MIIEvgIBADALBgkqhkiG9w0BAQoEggSqMIIEpgIBAAKCAQEAr4tmm3r20Wd/Pbqv
P1s2+QEtvpuRaV8Yq40gjUR8y2Rjxa6dpG2GXHbPfvMs8ct+Lh1GH45x28Rw3Ry5
3mm+oAXjyQ86OnDkZ5N8lYbggD4O3w6M6pAvLkhk95AndTrifbIFPNU8PPMO7Oyr
FAHqgDsznjPFmTOtCEcN2Z1FpWgchwuYLPL+Wokqltd11nqqzi+bJ9cvSKADYdUA
AN5WUtzdpiy6LbTgSxP7ociU4Tn0g5I6aDZJ7A8Lzo0KSyZYoA485mqcO0GVAdVw
9lq4aOT9v6d+nb4bnNkQVklLQ3fVAvJm+xdDOp9LCNCN48V2pnDOkFV6+U9nV5oy
c6XI2wIDAQABAoIBAQCUB8ip+kJiiZVKF8AqfB/aUP0jTAqOQewK1kKJ/iQCXBCq
pbo360gvdt05H5VZ/RDVkEgO2k73VSsbulqezKs8RFs2tEmU+JgTI9MeQJPWcP6X
aKy6LIYs0E2cWgp8GADgoBs8llBq0UhX0KffglIeek3n7Z6Gt4YFge2TAcW2WbN4
XfK7lupFyo6HHyWRiYHMMARQXLJeOSdTn5aMBP0PO4bQyk5ORxTUSeOciPJUFktQ
HkvGbym7KryEfwH8Tks0L7WhzyP60PL3xS9FNOJi9m+zztwYIXGDQuKM2GDsITeD
2mI2oHoPMyAD0wdI7BwSVW18p1h+jgfc4dlexKYRAoGBAOVfuiEiOchGghV5vn5N
RDNscAFnpHj1QgMr6/UG05RTgmcLfVsI1I4bSkbrIuVKviGGf7atlkROALOG/xRx
DLadgBEeNyHL5lz6ihQaFJLVQ0u3U4SB67J0YtVO3R6lXcIjBDHuY8SjYJ7Ci6Z6
vuDcoaEujnlrtUhaMxvSfcUJAoGBAMPsCHXte1uWNAqYad2WdLjPDlKtQJK1diCm
rqmB2g8QE99hDOHItjDBEdpyFBKOIP+NpVtM2KLhRajjcL9Ph8jrID6XUqikQuVi
4J9FV2m42jXMuioTT13idAILanYg8D3idvy/3isDVkON0X3UAVKrgMEne0hJpkPL
FYqgetvDAoGBAKLQ6JZMbSe0pPIJkSamQhsehgL5Rs51iX4m1z7+sYFAJfhvN3Q/
OGIHDRp6HjMUcxHpHw7U+S1TETxePwKLnLKj6hw8jnX2/nZRgWHzgVcY+sPsReRx
NJVf+Cfh6yOtznfX00p+JWOXdSY8glSSHJwRAMog+hFGW1AYdt7w80XBAoGBAImR
NUugqapgaEA8TrFxkJmngXYaAqpA0iYRA7kv3S4QavPBUGtFJHBNULzitydkNtVZ
3w6hgce0h9YThTo/nKc+OZDZbgfN9s7cQ75x0PQCAO4fx2P91Q+mDzDUVTeG30mE
t2m3S0dGe47JiJxifV9P3wNBNrZGSIF3mrORBVNDAoGBAI0QKn2Iv7Sgo4T/XjND
dl2kZTXqGAk8dOhpUiw/HdM3OGWbhHj2NdCzBliOmPyQtAr770GITWvbAI+IRYyF
S7Fnk6ZVVVHsxjtaHy1uJGFlaZzKR4AGNaUTOJMs6NadzCmGPAxNQQOCqoUjn4XR
rOjr9w349JooGXhOxbu8nOxX
-----END PRIVATE KEY-----
```

B.1.3. Example ECC P-256 Test Key

The following key is an elliptical curve key over the curve P-256, referred to in this document as test-key-ecc-p256.

```
-----BEGIN EC PRIVATE KEY-----
MHcCAQEEIFKbhfNZfpDsW43+0+JjUr9K+bTeuxopu653+hBaXGA7oAoGCCqGSM49
AwEHoUQDQgAEqIVYZVLCrPZHGHjP17CTW0/+D9Lfw0EkjqF7xB4FivAxzic30tMM
4GF+hR6Dxh71Z50VGGdldkkDXZCnTNnoXQ==
-----END EC PRIVATE KEY-----
```

```
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEqIVYZVLCrPZHGHjP17CTW0/+D9Lf
w0EkjqF7xB4FivAxzic30tMM4GF+hR6Dxh71Z50VGGdldkkDXZCnTNnoXQ==
-----END PUBLIC KEY-----
```

B.1.4. Example Shared Secret

The following shared secret is 64 randomly-generated bytes encoded in Base64, referred to in this document as test-shared-secret.

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

uzvJfB4u3N0Jy4T7NZ75MDVcr8zSTInedJtkgcu46YW4XByzNJjxBdtjUkdJPBt\
  bmHhIDi6pcl8jsasjlTMtDQ==
```

B.2. 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 messages:

For requests, this test-request message is used:

```
POST /foo?param=value&pet=dog HTTP/1.1
Host: example.com
Date: Tue, 20 Apr 2021 02:07:55 GMT
Content-Type: application/json
Digest: SHA-256=X48E9qOokqqrvdts8nOJRJN3OWDUoyWxBf7kbu9DBPE=
Content-Length: 18

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

For responses, this test-response message is used:

```
HTTP/1.1 200 OK
Date: Tue, 20 Apr 2021 02:07:56 GMT
Content-Type: application/json
Digest: SHA-256=X48E9qOokqqrvdts8nOJRJN3OWDUoyWxBf7kbu9DBPE=
Content-Length: 18

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

B.2.1. Minimal Signature Using rsa-pss-sha512

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

request, covering none of the components of the HTTP message request but providing a timestamped signature proof of possession of the key.

The corresponding signature input is:

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

"@signature-params": ();created=1618884475\
  ;keyid="test-key-rsa-pss";alg="rsa-pss-sha512"
```

This results in the following Signature-Input and Signature headers being added to the message:

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

Signature-Input: sig1=();created=1618884475\
  ;keyid="test-key-rsa-pss";alg="rsa-pss-sha512"
Signature: sig1=:HWP69ZNiom9Obu1KIdqPPcu/C1a5ZUMBbqS/xwJECV8bhIQVmE\
  AAAzz8LQPvtP1iFSxxluDO1KE9b8L+O64LEOvhwYdDctV5+E39Jy1eJiD7nYREBgx\
  TpdUfzTO+Trath0vZdTylFlxK4H3l3s/cuFhnOCxmFYgEa+cw+StBRgY1JtafSFwN\
  cZgLxVwialuH5VnqJS4JN8PHD91XLfkjMscTo4jmVMpFd3iLVe0hqVFl7MDt6TMkw\
  IyVFnEZ7B/VIQofdShO+C/7MuupCSLVjQz5xA+Zs6Hw+W9ESD/6BuGs6LF1TcKLxW\
  +5K+2zvDY/Cia34HNpRW5io7Iv9/b7iQ==:
```

Note that since the covered components list is empty, this signature could be applied by an attacker to an unrelated HTTP message. Therefore, use of an empty covered components set is discouraged.

B.2.2. Selective Covered Components using rsa-pss-sha512

This example covers additional components in test-request using the rsa-pss-sha512 algorithm.

The corresponding signature input is:

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

"@authority": example.com
"content-type": application/json
"@signature-params": ("@authority" "content-type")\
  ;created=1618884475;keyid="test-key-rsa-pss"
```

This results in the following Signature-Input and Signature headers being added to the message:

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

Signature-Input: sig1=("@authority" "content-type")\
  ;created=1618884475;keyid="test-key-rsa-pss"
Signature: sig1=:ik+OtGmM/kFqENDf9Plm8AmPtqtC7C9a+zYSaxr58b/E6h81gh\
  JS3PcH+m1asiMp8yvccnO/RfaexnqanVB3C72WRNZN7skPTJmUVmoIeqZncdP2mlf\
  xlLP6UbkrgYsk91NS6nwkKC6RRgLhBFqzP42oq8D2336OiQPDAo/04SxZt4Wx9nDG\
  uy2SfZJUhsJqZyEWRk4204x7YEB3VxDAAlVgGt8ewilWbIKKTOKp3ymUeQIwptqYw\
  v0l8mN404PPzRBTpB7+HpClyK4CNp+SVv46+6sHMfJU4taz10s/NoYRmYCGXyadzY\
  YDj0BYnFdERB6NblI/AOWFGl5Axhhmjg==:
```

B.2.3. Full Coverage using rsa-pss-sha512

This example covers all headers in test-request (including the message body Digest) plus various elements of the control data, using the rsa-pss-sha512 algorithm.

The corresponding signature input is:

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

"date": Tue, 20 Apr 2021 02:07:56 GMT
"@method": POST
"@path": /foo
"@query": ?param=value&pet=dog
"@authority": example.com
"content-type": application/json
"digest": SHA-256=X48E9qOokqqrvdts8nOJRJN3OWDUoyWxBf7kbu9DBPE=
"content-length": 18
"@signature-params": ("date" "@method" "@path" "@query" \
  "@authority" "content-type" "digest" "content-length")\
  ;created=1618884475;keyid="test-key-rsa-pss"
```

This results in the following Signature-Input and Signature headers being added to the message:

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

Signature-Input: sig1=("date" "@method" "@path" "@query" \
  "@authority" "content-type" "digest" "content-length")\
  ;created=1618884475;keyid="test-key-rsa-pss"
Signature: sig1=:JuJnJMFGD4HMysAGsfOY6N5ZTZUknsQUdClNG51VezDgPUOW03\
  QMe74vbIdndKwW1BBrHOHR3NzKGYZJ7X3ur23FMCdANe4VmKb3Rc1Q/5YxOO8p7Ko\
  yfVa4uUcMk5jB9KAn1M1MbgBnqwZkRWsbv8ocCqrnD85Kavr73lx51k1/gU8w673W\
  T/oBtxPtAn1eFjUyIKyA+XD7kYph82I+ahvm0pSgDPagu917SlqUjeaQaNnlZzO03\
  Iy1RZ5XpgbNeDLCqSLuZFVID80EohC2CQ1cL5svjslrlCNstd2JCLmhjL7xV3NYXe\
  rLim4bqUQGRgDwNJRnqobpS6C1NBns/Q==:
```

Note in this example that the value of the Date header and the value of the created signature parameter need not be the same. This is due to the fact that the Date header is added when creating the HTTP Message and the created parameter is populated when creating the signature over that message, and these two times could vary. If the Date header is covered by the signature, it is up to the verifier to determine whether its value has to match that of the created parameter or not.

B.2.4. Signing a Response using ecdsa-p256-sha256

This example covers portions of the test-response response message using the ecdsa-p256-sha256 algorithm and the key test-key-ecc-p256.

The corresponding signature input is:

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

"content-type": application/json
"digest": SHA-256=X48E9qOokqqrvdts8nOJRJN3OWDUoyWxBf7kbu9DBPE=
"content-length": 18
"@signature-params": ("content-type" "digest" "content-length")\
  ;created=1618884475;keyid="test-key-ecc-p256"
```

This results in the following Signature-Input and Signature headers being added to the message:

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

Signature-Input: sig1=("content-type" "digest" "content-length")\
  ;created=1618884475;keyid="test-key-ecc-p256"
Signature: sig1=:n8RKXkj0iseWDmC6PNSQ1GX2R9650v+lhbb6rTGoSrSSx18zmn\
  6fPOtBx48/WffYLO0n1RHHf9scvNGAgGq52Q==:
```

B.2.5. Signing a Request using hmac-sha256

This example covers portions of the test-request using the hmac-sha256 algorithm and the secret test-shared-secret.

The corresponding signature input is:

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

"@authority": example.com
"date": Tue, 20 Apr 2021 02:07:55 GMT
"content-type": application/json
"@signature-params": ("@authority" "date" "content-type")\
  ;created=1618884475;keyid="test-shared-secret"
```

This results in the following Signature-Input and Signature headers being added to the message:

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

Signature-Input: sig1=("@authority" "date" "content-type")\
  ;created=1618884475;keyid="test-shared-secret"
Signature: sig1=:fN3AMNGbx0V/cIEKkZOvLOoC3InI+lM2+gTv22x3ia8=:
```

B.3. TLS-Terminating Proxies

In this example, there is a TLS-terminating reverse proxy sitting in front of the resource. The client does not sign the request but instead uses mutual TLS to make its call. The terminating proxy validates the TLS stream and injects a Client-Cert header according to [I-D.ietf-httpbis-client-cert-field]. By signing this header field, a reverse proxy can not only attest to its own validation of the initial request but also authenticate itself to the backend system independently of the client's actions. The client makes the following request to the TLS terminating proxy using mutual TLS:

```
POST /foo?Param=value&pet=Dog HTTP/1.1
Host: example.com
Date: Tue, 20 Apr 2021 02:07:55 GMT
Content-Type: application/json
Content-Length: 18

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

The proxy processes the TLS connection and extracts the client's TLS certificate to a Client-Cert header field and passes it along to the internal service hosted at service.internal.example. This results in the following unsigned request:

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

POST /foo?Param=value&pet=Dog HTTP/1.1
Host: service.internal.example
Date: Tue, 20 Apr 2021 02:07:55 GMT
Content-Type: application/json
Content-Length: 18
Client-Cert: :MIIBqDCCAU6gAwIBAgIBBzAKBggqhkjOPQQDAjA6MRswGQYDVQQKD\
  BJMZXQncyBBdXRoZW50aWNhdGUxGzAZBgNVBAMMEkxBIEludGVybWVkaWF0ZSBDQT\
  AeFw0yMDAxMTQyMjU1MzNaFw0yMTAxMjMyMjU1MzNaMA0xCzAJBgNVBAMMAkJDMFk\
  wEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAE8YnXXfaUgmnMtOXU/IncWalRhebrXmck\
  C8vdgJ1p5Be5F/3YC8OthxM4+k1M6aEAEFcGzkJiNy6J84y7uzo9M6NyMHAwCQYDV\
  R0TBAIwADAfBgNVHSMEGDAWgBRm3WjLa38lbEYCuiCPct0ZaSED2DAOBgNVHQ8BAf\
  8EBAMCBsAwEwYDVR0lBAwwCgYIKwYBBQUHAwIwHQYDVR0RAQH/BBMwEYEPYmRjQGV\
  4YW1wbGUuY29tMAoGCCqGSM49BAMCA0gAMEUCIBHda/r1vaL6G3VliL4/Di6YK0Q6\
  bMjeSkC3dFCOOB8TAiEAx/kHSB4urmiZ0NX5r5XarmPk0wmuydBVoU4hBVZ1yhk=:

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

Without a signature, the internal service would need to trust that the incoming connection has the right information. By signing the Client-Cert header and other portions of the internal request, the internal service can be assured that the correct party, the trusted proxy, has processed the request and presented it to the correct service. The proxy's signature input consists of the following:

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

"@path": /foo
"@query": Param=value&pet=Dog
"@method": POST
"@authority": service.internal.example
"client-cert": :MIIBqDCCAU6gAwIBAgIBBzAKBggqhkjOPQQDAjA6MRswGQYDVQQ\
  KDBJMZXQncyBBdXRoZW50aWNhdGUxGzAZBgNVBAMMEkxBIEludGVybWVkaWF0ZSBD\
  QTAeFw0yMDAxMTQyMjU1MzNaFw0yMTAxMjMyMjU1MzNaMA0xCzAJBgNVBAMMAkJDM\
  FkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAE8YnXXfaUgmnMtOXU/IncWalRhebrXm\
  ckC8vdgJ1p5Be5F/3YC8OthxM4+k1M6aEAEFcGzkJiNy6J84y7uzo9M6NyMHAwCQY\
  DVR0TBAIwADAfBgNVHSMEGDAWgBRm3WjLa38lbEYCuiCPct0ZaSED2DAOBgNVHQ8B\
  Af8EBAMCBsAwEwYDVR0lBAwwCgYIKwYBBQUHAwIwHQYDVR0RAQH/BBMwEYEPYmRjQ\
  GV4YW1wbGUuY29tMAoGCCqGSM49BAMCA0gAMEUCIBHda/r1vaL6G3VliL4/Di6YK0\
  Q6bMjeSkC3dFCOOB8TAiEAx/kHSB4urmiZ0NX5r5XarmPk0wmuydBVoU4hBVZ1yhk=:
"@signature-params": ("@path" "@query" "@method" "@authority" \
  "client-cert");created=1618884475;keyid="test-key-ecc-p256"
```

This results in the following signature:

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

5gudRjXaHrAYbEaQUOoY9TuvqWOdPcspkp7YyKCB0XhyAG9cB715hucPPanEK0OVyiN\
LJqcoq2Yn1DPWQcnbog==
```

Which results in the following signed request sent from the proxy to the internal service:

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

POST /foo?Param=value&pet=Dog HTTP/1.1
Host: service.internal.example
Date: Tue, 20 Apr 2021 02:07:55 GMT
Content-Type: application/json
Content-Length: 18
Client-Cert: :MIIBqDCCAU6gAwIBAgIBBzAKBggqhkjOPQQDAjA6MRswGQYDVQQKD\
  BJMZXQncyBBdXRoZW50aWNhdGUxGzAZBgNVBAMMEkxBIEludGVybWVkaWF0ZSBDQT\
  AeFw0yMDAxMTQyMjU1MzNaFw0yMTAxMjMyMjU1MzNaMA0xCzAJBgNVBAMMAkJDMFk\
  wEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAE8YnXXfaUgmnMtOXU/IncWalRhebrXmck\
  C8vdgJ1p5Be5F/3YC8OthxM4+k1M6aEAEFcGzkJiNy6J84y7uzo9M6NyMHAwCQYDV\
  R0TBAIwADAfBgNVHSMEGDAWgBRm3WjLa38lbEYCuiCPct0ZaSED2DAOBgNVHQ8BAf\
  8EBAMCBsAwEwYDVR0lBAwwCgYIKwYBBQUHAwIwHQYDVR0RAQH/BBMwEYEPYmRjQGV\
  4YW1wbGUuY29tMAoGCCqGSM49BAMCA0gAMEUCIBHda/r1vaL6G3VliL4/Di6YK0Q6\
  bMjeSkC3dFCOOB8TAiEAx/kHSB4urmiZ0NX5r5XarmPk0wmuydBVoU4hBVZ1yhk=:
Signature-Input: ttrp=("@path" "@query" "@method" "@authority" \
  "client-cert");created=1618884475;keyid="test-key-ecc-p256"
Signature: ttrp=:5gudRjXaHrAYbEaQUOoY9TuvqWOdPcspkp7YyKCB0XhyAG9cB7\
  15hucPPanEK0OVyiNLJqcoq2Yn1DPWQcnbog==:

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

The internal service can validate the proxy's signature and therefore be able to trust that the client's certificate has been appropriately processed.

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 editors would also like to thank the following individuals for feedback, insight, and implementation of this draft and its predecessors (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, Ilari Liusvaara, James H. Manger, Kathleen Moriarty, 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

--06

- Updated language for message components, including identifiers and values.
- Clarified that Signature-Input and Signature are fields which can be used as headers or trailers.
- Add "Accept-Signature" field and semantics for signature negotiation.
- Define new specialty content identifiers, re-defined request-target identifier.
- Added request-response binding.

--05

- Remove list prefixes.
- Clarify signature algorithm parameters.
- Update and fix examples.
- Add examples for ECC and HMAC.

--04

- Moved signature component definitions up to intro.
- Created formal function definitions for algorithms to fulfill.
- Updated all examples.
- Added nonce parameter field.

--03

- Clarified signing and verification processes.

- Updated algorithm and key selection method.

- Clearly defined core algorithm set.

- Defined JOSE signature mapping process.

- Removed legacy signature methods.

- Define signature parameters separately from "signature" object model.

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

--02

- Removed editorial comments on document sources.

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

- 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";key=member-name.

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

- Fixed up examples.

- Updated introduction now that it's adopted.

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

- Required signature parameters to be included in signature.

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

--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

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/