HTTP
Internet-Draft
Intended status: Standards Track
Expires: 27 November 2022

A. Backman, Ed.
Amazon
J. Richer
Bespoke Engineering
M. Sporny
Digital Bazaar

26 May 2022

HTTP Message Signatures
draft-ietf-httpbis-message-signatures-10

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.

About This Document

This note is to be removed before publishing as an RFC.

Status information for this document may be found at https://datatracker.ietf.org/doc/draft-ietf-httpbis-message-signatures/.

Discussion of this document takes place on the HTTP Working Group mailing list (mailto: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 for this draft and an issue tracker 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 27 November 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction 4
   1.1. Requirements Discussion 5
   1.2. HTTP Message Transformations 6
   1.3. Conventions and Terminology 7
   1.4. Application of HTTP Message Signatures 10
2. HTTP Message Components 11
   2.1. HTTP Fields 12
      2.1.1. Canonicalized Structured HTTP Fields 14
      2.1.2. Dictionary Structured Field Members 14
   2.2. Derived Components 15
      2.2.1. Signature Parameters 17
      2.2.2. Method 18
      2.2.3. Target URI 19
      2.2.4. Authority 19
      2.2.5. Scheme 20
      2.2.6. Request Target 20
      2.2.7. Path 22
      2.2.8. Query 22
      2.2.9. Query Parameters 23
      2.2.10. Status Code 25

2.3. Request-Response Signature Binding 25
2.4. Creating the Signature Base 28
3. HTTP Message Signatures 30
3.1. Creating a Signature 31
3.2. Verifying a Signature 33

3.2.1. Enforcing Application Requirements 35
3.3. Signature Algorithm Methods 36
3.3.1. RSASSA-PSS using SHA-512 37
3.3.2. RSASSA-PKCS1-v1_5 using SHA-256 37
3.3.3. HMAC using SHA-256 38
3.3.4. ECDSA using curve P-256 DSS and SHA-256 38
3.3.5. EdDSA using curve edwards25519 39
3.3.6. JSON Web Signature (JWS) algorithms 40
4. Including a Message Signature in a Message 40
4.1. The Signature-Input HTTP Field 41
4.2. The Signature HTTP Field 41
4.3. Multiple Signatures 42
5. Requesting Signatures 45
5.1. The Accept-Signature Field 46
5.2. Processing an Accept-Signature 47
6. IANA Considerations 47
6.1. HTTP Signature Algorithms Registry 48
6.1.1. Registration Template 48
6.1.2. Initial Contents 49
6.2. HTTP Signature Metadata Parameters Registry 49
6.2.1. Registration Template 49
6.2.2. Initial Contents 50
6.3. HTTP Signature Derived Component Names Registry 50
6.3.1. Registration Template 51
6.3.2. Initial Contents 51
6.4. HTTP Signature Component Parameters Registry 53
6.4.1. Registration Template 53
6.4.2. Initial Contents 53
7. Security Considerations 54
7.1. Signature Verification Skipping 55
7.2. Use of TLS 55
7.3. Signature Replay 55
7.4. Insufficient Coverage 56
7.5. Cryptography and Signature Collision 56
7.6. Key Theft 57
7.7. Modification of Required Message Parameters 57

7.8. Mismatch of Signature Parameters from Message 58
7.9. Multiple Signature Confusion 58
7.10. Signature Labels 58
7.11. Symmetric Cryptography 59
7.12. Canonicalization Attacks 59
7.13. Key Specification Mix-Up 60
7.14. HTTP Versions and Component Ambiguity 60
7.15. Key and Algorithm Specification Downgrades 60
7.16. Parsing Structured Field Values 61
7.17. Choosing Message Components 61
7.18. Confusing HTTP Field Names for Derived Component Names . 62
7.19. Non-deterministic Signature Primitives 62

7.20. Choosing Signature Parameters and Derived Components over HTTP Fields 63
7.21. Semantically Equivalent Field Values 63
7.22. Message Content 64
8. Privacy Considerations 65
8.1. Identification through Keys 65
8.2. Signatures do not provide confidentiality 65
8.3. Oracles 66
8.4. Required Content 66
9. References 66
9.1. Normative References 66
9.2. Informative References 68
Appendix A. Detecting HTTP Message Signatures 69
Appendix B. Examples 69
B.1. Example Keys 69
B.1.1. Example Key RSA test 70
B.1.2. Example RSA PSS Key 71
B.1.3. Example ECC P-256 Test Key 72
B.1.4. Example Ed25519 Test Key 73
B.1.5. Example Shared Secret 73
B.2. Test Cases 73
B.2.1. Minimal Signature Using rsa-pss-sha512 74
B.2.2. Selective Covered Components using rsa-pss-sha512 . . 75
B.2.3. Full Coverage using rsa-pss-sha512 76
B.2.4. Signing a Response using ecdsa-p256-sha256 77
B.2.5. Signing a Request using hmac-sha256 78
B.2.6. Signing a Request using ed25519 78
B.3. TLS-Terminating Proxies 79
Acknowledgements 81

Document History 82
Authors' Addresses 86

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, used to create the signature base.

* Algorithms for generating and verifying signatures over HTTP

message components using this signature base through application of cryptographic primitives.

- A mechanism for attaching a signature and related metadata to an HTTP message, and for parsing attached signatures and metadata from HTTP messages. To facilitate this, this document defines the "Signature-Input" and "Signature" fields.

This document also provides a mechanism for a potential verifier to signal to a potential signer 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. To facilitate this, this document defines the "Accept-Signature" field.

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 base. Since the exact raw bytes of the message cannot be relied upon as a reliable source for a signature base, the signer and verifier must independently

create the signature base 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 message content, the method, 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 (https://openjdk.java.net/groups/net/httpclient/intro.html)). 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 (Section 3.2.2 of [HTTP1]).

- Combination of header fields with the same field name (Section 3.2.2 of [HTTP1]).

- Removal of header fields listed in the Connection header field (Section 6.1 of [HTTP1]).

- Addition of header fields that indicate control options (Section 6.1 of [HTTP1]).

- Addition or removal of a transfer coding (Section 5.7.2 of [HTTP1]).

- Addition of header fields such as Via (Section 5.7.1 of [HTTP1]) and Forwarded (Section 4 of [RFC7239]).

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 [HTTP1].

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

1.3. 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 [HTTP1].

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

For brevity, the term "signature" on its own is used in this document to refer to both digital signatures (which use asymmetric cryptography) and keyed MACs (which use symmetric cryptography).

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 Name:
A name that identifies an HTTP Message Component.

HTTP Message Component Identifier:
The combination of an HTTP Message Component Name and any parameters 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 derived components (Section 2.2) that indicates the set of message components covered by the signature, never including the @signature-params identifier itself. The order of this set is preserved and communicated between the signer and verifier to facilitate reconstruction of the signature base.

Signature Base:
The sequence of bytes processed by the cryptographic algorithm to produce or verify the HTTP Message Signature. The signature base 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, defined in terms of the HTTP_SIGN and HTTP_VERIFY primitives described in Section 3.3.

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 after which the signature should no longer be accepted by the verifier, as asserted by the signer.

The term "Unix time" is defined by [POSIX.1], Section 4.16 (http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html#tag_04_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.

This document uses the following terminology from Section 3 of [STRUCTURED-FIELDS] to specify syntax and parsing: List, Inner List, Dictionary, String, Integer, Byte Sequence, and Boolean. This document uses the following ABNF rules from [STRUCTURED-FIELDS] where applicable: sf-string, key, parameters.

1.4. 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 field be covered to protect the authorization credentials and mandate the signature parameters contain a created parameter, while an API expecting semantically relevant HTTP message content could require the Content-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 2.2.1) 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 2.2.1) 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 base. 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, however some additional considerations are discussed in Section 7.

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 base. 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 [STRUCTURED-FIELDS], Section 4. The component identifier has a component name, which is a String value serialized using the sf-string ABNF rule. The component identifier MAY also include defined parameters which are serialized using the parameters rule.

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

Note that this means the serialization of the component name itself is encased in double quotes, with parameters following as a semicolon-separated list, such as "cache-control", "date", or "@signature-params", and "example-dictionary";key="foo".

One component identifier is distinct from another if either the component name or its parameters differ. Within a single list of covered components, each component identifier MUST be distinct from every other component identifier. Multiple component identifiers having the same component name MAY be included if they have parameters that make them distinct.

The component value associated with a component identifier is defined by the identifier itself. Component values MUST NOT contain newline (\n) characters.

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 base is defined in Section 2.4.

2.1. HTTP Fields

The component name 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 names.

Unless overridden by additional parameters and rules, the HTTP field value MUST be canonicalized as a single combined value as defined in Section 5.2 of [HTTP].

If the combined value is not available for a given header, the following algorithm will produce canonicalized results for an implementation:

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.

Note that since HTTP field values are not allowed to contain leading and trailing whitespace, this will be a no-op in a compliant implementation.

3. Remove any obsolete line-folding within the line and replace it with a single space (" "), as discussed in Section 5.2 of [HTTP1]. Note that this behavior is specific to [HTTP1] and does not apply to other versions of the HTTP specification.

4. Concatenate the list of values together with a single comma (",") and a single space (" ") between each item.

The resulting string is the canonicalized component value.

Note that some HTTP fields have values with multiple valid serializations that have equivalent semantics. Applications signing and processing such fields MUST consider how to handle the values of such fields to ensure that the signer and verifier can derive the same value, as discussed in Section 7.21.

Following are non-normative examples of canonicalized values for header fields, given the following example HTTP message fragment:

```
Host: www.example.com
Date: Tue, 20 Apr 2021 02:07:56 GMT
X-OWS-Header:   Leading and trailing whitespace.
X-Obs-Fold-Header: Obsolete
    line folding.
Cache-Control: max-age=60
Cache-Control:    must-revalidate
Example-Dict:  a=1,    b=2;x=1;y=2,   c=(a   b   c)
```

The following example shows canonicalized values for these example header fields, presented using the signature base format discussed in Section 2.4:

```
"host": www.example.com
"date": Tue, 20 Apr 2021 02:07:56 GMT
"x-ows-header": Leading and trailing whitespace.
"x-obs-fold-header": Obsolete line folding.
"cache-control": max-age=60, must-revalidate
```

```
"example-dict": a=1,    b=2;x=1;y=2,   c=(a   b   c)
```

Since empty HTTP header fields are allowed, they are also able to be signed when present in a message. The canonicalized value is the empty string. This means that the following empty header:

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

X-Empty-Header: \
```

Is serialized by the signature base generation algorithm (Section 2.4) with an empty string value following the colon and space added after the content identifier.

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

"x-empty-header": \
```

Note: these are shown here using the line wrapping algorithm in [RFC8792] due to limitations in the document format that strips trailing spaces from diagrams.

Any HTTP field component identifiers MAY have the following parameters in specific circumstances.

sf A boolean flag indicating that the field value is to be canonicalized using strict encoding of the structured field value. Section 2.1.1

key A string parameter used to select a single member value from a dictionary structured field. Section 2.1.2

2.1.1. Canonicalized Structured HTTP Fields

If the value of the the HTTP field in question is a structured field ([STRUCTURED-FIELDS]), the component identifier MAY include the sf parameter to indicate it is a known structured field. If this parameter is included with a component identifier, the HTTP field value MUST be serialized using the rules specified in Section 4 of [STRUCTURED-FIELDS] applicable to the type of the HTTP field. Note that this process will replace any optional internal whitespace with a single space character, among other potential transformations of the value.

For example, the following dictionary field is a valid serialization:

```
Example-Dict:  a=1,    b=2;x=1;y=2,   c=(a   b   c)
```

If included in the input string as-is, it would be:

```
"example-dict": a=1,    b=2;x=1;y=2,   c=(a   b   c)
```

However, if the sf parameter is added, the value is re-serialized as follows:

```
"example-dict";sf: a=1, b=2;x=1;y=2, c=(a b c)
```

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

2.1.2. Dictionary Structured Field Members

If a given field is known by the application to be a Dictionary structured field, an individual member in the value of that Dictionary is identified by using the parameter key and the Dictionary member key as a String value.

An individual member value of a Dictionary Structured Field is canonicalized by applying the serialization algorithm described in Section 4.1.2 of [STRUCTURED-FIELDS] on the member_value and its parameters, without the dictionary key. Specifically, the value is serialized as an Item or Inner List (the two possible values of a Dictionary member).

Each parameterized key for a given field MUST NOT appear more than once in the signature base. Parameterized keys MAY appear in any order in the signature base, regardless of the order they occur in the source Dictionary.

If a Dictionary key is named as a covered component but it does not occur in the Dictionary, this MUST cause an error in the signature base generation.

Following are non-normative examples of canonicalized values for Dictionary structured field members given the following example header field, whose value is known by the application to be a Dictionary:

```
Example-Dict:  a=1, b=2;x=1;y=2, c=(a   b    c), d
```

The following example shows canonicalized values for different component identifiers of this field, presented using the signature base format discussed in Section 2.4:

```
"example-dict";key="a": 1
"example-dict";key="d": ?1
"example-dict";key="b": 2;x=1;y=2
"example-dict";key="c": (a b c)
```

Note that the value for key="c" has been re-serialized according to the strict member_value algorithm.

2.2. Derived Components

In addition to HTTP fields, there are a number of different components that can be derived from the control data, processing context, or other aspects of the HTTP message being signed. Such derived components can be included in the signature base by defining a component name, possible parameters, and the derivation method for its component value.

Derived component names MUST start with the "at" @ character. This differentiates derived component names from HTTP field names, which cannot contain the @ character as per Section 5.1 of [HTTP]. Processors of HTTP Message Signatures MUST treat derived component names separately from field names, as discussed in Section 7.18.

This specification defines the following derived components:

@signature-params The signature metadata parameters for this signature. (Section 2.2.1)

@method The method used for a request. (Section 2.2.2)

@target-uri The full target URI for a request. (Section 2.2.3)

@authority The authority of the target URI for a request.

(Section 2.2.4)

@scheme The scheme of the target URI for a request. (Section 2.2.5)

@request-target The request target. (Section 2.2.6)

@path The absolute path portion of the target URI for a request. (Section 2.2.7)

@query The query portion of the target URI for a request. (Section 2.2.8)

@query-param A parsed query parameter of the target URI for a request. (Section 2.2.9)

@status The status code for a response. (Section 2.2.10).

Additional derived component names MAY be defined and registered in the HTTP Signatures Derived Component Name Registry. (Section 6.3)

Derived component values are taken from the context of the target message for the signature. This context includes information about the message itself, such as its control data, as well as any additional state and context held by the signer. In particular, when signing a response, the signer can include any derived components from the originating request by using the request-response signature binding parameter (Section 2.3).

request: Values derived from and results applied to an HTTP request message as described in Section 3.4 of [HTTP].

response: Values derived from and results applied to an HTTP response message as described in Section 3.4 of [HTTP].

A derived component definition MUST define all targets to which it can be applied.

The component value MUST be derived from the HTTP message being signed or the context in which the derivation occurs. The derived component value MUST be of the following form:

```
derived-component-value = *VCHAR
```

2.2.1. 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 name is @signature-params. This message component's value is REQUIRED as part of the signature base (Section 2.4) but the component identifier MUST NOT be enumerated within the set of covered components itself.

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 Integer UNIX timestamp value. Sub-second precision is not supported. Inclusion of this parameter is RECOMMENDED.

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

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

* alg: The HTTP message signature algorithm from the HTTP Message Signature Algorithm Registry, as a String value.

* keyid: The identifier for the key material as a 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 [STRUCTURED-FIELDS] as follows:

1. Let the output be an empty string.

2. Determine an order for the component identifiers of the covered components, not including the @signature-params component identifier itself. Once this order is chosen, it cannot be changed. This order MUST be the same order as used in creating the signature base (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 [STRUCTURED-FIELDS] 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 [STRUCTURED-FIELDS], 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 from Section 4.1.1.1 of [STRUCTURED-FIELDS] is used for the covered component value instead of the List serialization from Section 4.1.1 of [STRUCTURED-FIELDS] in order to facilitate parallelism with this value's inclusion in the Signature-Input field, 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")\
  ;keyid="test-key-rsa-pss";alg="rsa-pss-sha512";\
  created=1618884475;expires=1618884775
```

Note that an HTTP message could contain multiple signatures (Section 4.3), but only the signature parameters used for a single signature are included in an entry.

2.2.2. Method

The @method derived component refers to the HTTP method of a request message. The component value is canonicalized by taking the value of the method as a string. Note that the method name is case-sensitive as per [HTTP], 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 component value:

```
POST
```

And the following signature base line:

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

2.2.3. Target URI

The @target-uri derived component 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 [HTTP], 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 component value:

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

And the following signature base line:

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

2.2.4. Authority

The @authority derived component refers to the authority component of the target URI of the HTTP request message, as defined in [HTTP], 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 [HTTP], 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:

```
www.example.com
```

And the following signature base line:

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

The @authority derived component SHOULD be used instead of signing the Host header directly, see Section 7.20.

2.2.5. Scheme

The @scheme derived component 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 [HTTP], Section 4.2. While the scheme itself is case-insensitive, it MUST be normalized to lowercase for inclusion in the signature base. 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 component value:

```
http
```

And the following signature base line:

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

2.2.6. Request Target

The @request-target derived component refers to the full request target of the HTTP request message, as defined in [HTTP], 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 component 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:

```
/path?param=value
```

And the following signature base line:

```
"@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:

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

And the following signature base line:

```
"@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:

```
www.example.com:80
```

And the following signature base line:

```
"@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:

```
*
```

And the following signature base line:

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

2.2.7. Path

The @path derived component 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 [HTTP], 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 component value:

```
/path
```

And the following signature base line:

```
"@path": /path
```

2.2.8. Query

The @query derived component 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 [HTTP], 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 component value:

```
?param=value&foo=bar&baz=batman
```

And the following signature base line:

```
"@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 component value:

```
?queryString
```

And the following signature base line:

```
"@query": ?queryString
```

If the query string is absent from the request message, the value is the leading ? character alone:

```
?
```

Resulting in the following signature base line:

```
"@query": ?
```

2.2.9. Query Parameters

If a request target URI uses HTML form parameters in the query string as defined in HTMLURL, Section 5 [HTMLURL], the @query-param derived component allows addressing of individual query parameters. The query parameters MUST be parsed according to HTMLURL, Section 5.1 [HTMLURL], resulting in a list of (nameString, valueString) tuples. The REQUIRED name parameter of each component identifier contains the nameString of a single query parameter as a String value. 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 [HTMLURL], 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 query parameter is named as a covered component but it does not occur in the query parameters, this MUST cause an error in the signature base generation.

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 component values:

baz: batman

qux: an empty string

param: value

And the following signature base lines:

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

"@query-param";name="baz": batman
"@query-param";name="qux": \

"@query-param";name="param": value
```

If a parameter name occurs multiple times in a request, all parameter values of that name MUST be included in separate signature base lines in the order in which the parameters occur in the target URI. Note that in some implementations, the order of parsed query parameters is not stable, and this situation could lead to unexpected results. If multiple parameters are common within an application, it is RECOMMENDED to sign the entire query string using the @query component identifier defined in Section 2.2.8.

2.2.10. Status Code

The @status derived component refers to the three-digit numeric HTTP status code of a response message as defined in [HTTP], Section 15. The component value is the serialized three-digit integer of the HTTP status 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 component value:

```
200
```

And the following signature base line:

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

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

2.3. Request-Response Signature Binding

When a request message results in a signed response message, the signer can include portions of the request message in the signature base by adding the req parameter to the component identifier.

req Indicates that the component value is derived from the request
   that triggered this response message and not from the response
   message directly.

This parameter can be applied to both HTTP fields and derived components with the same semantics. The component value for a message component using this parameter is calculated in the same manner as it is normally, but data is pulled from the request message.

Note that the same component name MAY be included with and without the req parameter in a single signature base, indicating the same named component from both the request and response message.

The req parameter MAY be combined with other parameters as appropriate for the component identifier, such as the key parameter for a dictionary field.

For example, when serving a response for 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-Digest: sha-512=:WZDPaVn/7XgHaAy8pmojAkGWoRx2UFChF41A2svX+T\
  aPm+AbwAgBWnrIiYllu7BNNyealdVLvRwEmTHWXvJwew==:
Content-Length: 18
Signature-Input: sig1=("@method" "@authority" "@path" \
  "content-digest" "content-length" "content-type")\
  ;created=1618884475;keyid="test-key-rsa-pss"
Signature:  sig1=:LAH8BjcfcOcLojiuOBFWn0P5keD3xAOuJRGziCLuD8r5MW9S0\
  RoXXLzLSRfGY/3SF8kVIkHjE13SEFdTo4Af/fJ/Pu9wheqoLVdwXyY/UkBIS1M8Br\
  c8IODsn5DFIrG0IrburbLi0uCc+E2ZIIb6HbUJ+o+jP58JelMTe0QE3IpWINTEzpx\
  jqDf5/Df+InHCAkQCTuKsamjWXUpyOT1Wkxi7YPVNOjW4MfNuTZ9HdbD2Tr65+BXe\
  TG9ZS/9SWuXAc+BZ8WyPz0QRz//ec3uWXd7bYYODSjRAxHqX+S1ag3LZElYyUKaAI\
  jZ8MGOt4gXEwCSLDv/zqxZeWLj/PDkn6w==:

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

This would result in the following unsigned response message:

```
HTTP/1.1 503 Service Unavailable
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"}
```

To cryptographically link the response to the request, 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 base for this example is:

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

"@status": 503
"content-length": 62
"content-type": application/json
"signature";req;key="sig1": :LAH8BjcfcOcLojiuOBFWn0P5keD3xAOuJRGziC\
  LuD8r5MW9S0RoXXLzLSRfGY/3SF8kVIkHjE13SEFdTo4Af/fJ/Pu9wheqoLVdwXyY\
  /UkBIS1M8Brc8IODsn5DFIrG0IrburbLi0uCc+E2ZIIb6HbUJ+o+jP58JelMTe0QE\
  3IpWINTEzpxjqDf5/Df+InHCAkQCTuKsamjWXUpyOT1Wkxi7YPVNOjW4MfNuTZ9Hd\
  bD2Tr65+BXeTG9ZS/9SWuXAc+BZ8WyPz0QRz//ec3uWXd7bYYODSjRAxHqX+S1ag3\
  LZElYyUKaAIjZ8MGOt4gXEwCSLDv/zqxZeWLj/PDkn6w==:
"@signature-params": ("@status" "content-length" "content-type" \
  "signature";req;key="sig1");created=1618884479\
  ;keyid="test-key-ecc-p256"
```

The signed response message is:

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

HTTP/1.1 503 Service Unavailable
Date: Tue, 20 Apr 2021 02:07:56 GMT
```

```
Content-Type: application/json
Content-Length: 62
Signature-Input: reqres=("@status" "content-length" "content-type" \
  "signature";req;key="sig1");created=1618884479\
  ;keyid="test-key-ecc-p256"
Signature: reqres=:vR1E+sDgh0J3dZyVdPc7mK0ZbEMW3N47eDpFjXLE9g95Gx1K\
  QLpdOmDQfedgdLzaFCqfD0WPn9e9/jubyUuZRw==:

{"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 that uses this component identifier.

Note that the ECDSA algorithm in use here is non-deterministic, meaning a different signature value will be created every time the algorithm is run. The signature value provided here can be validated against the given keys, but newly-generated signature values are not expected to match the example. See Section 7.19.

The req parameter MUST NOT be used with a request message.

2.4. Creating the Signature Base

The signature base is a US-ASCII string containing the canonicalized HTTP message components covered by the signature. The input to the signature base creation algorithm is the list of covered component identifiers and their associated values, along with any additional signature parameters. The output is the ordered set of bytes that form the signature base, which conforms to the following ABNF:

```
signature-base = *( signature-base-line LF ) signature-params-line
signature-base-line = component-identifier ":" SP
    ( derived-component-value / field-value )
signature-params-line = DQUOTE "@signature-params" DQUOTE ":" SP inner-list
```

To create the signature base, 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. Note that this serialization places the component name in double quotes and appends any parameters outside of the quotes.

    2. Append a single colon :

    3. Append a single space " "

    4. Determine the component value for the component identifier.

        * If the component name starts with an "at" character (@), derive the component's value from the message according to the specific rules defined for the derived component, as in Section 2.2. If the derived component name is unknown or the value cannot be derived, produce an error.

        * If the component name does not start with an "at" character (@), canonicalize the HTTP field value as described in Section 2.1. If the value cannot be calculated, produce an error.

    5. Append the covered component's canonicalized component value.

    6. Append a single newline \n

3. Append the signature parameters component (Section 2.2.1) 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 2.2.1

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 and not create an input string. Such situations are included but not limited to:

- The signer or verifier does not understand the derived component name.

- The component name identifies a field that is not present in the message or whose value is malformed.

- The component identifier indicates that a structured field serialization is used (via the sf parameter), but the field in question is known to not be a structured field or the type of structured field is not known to the implementation.

- 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 or a named query parameter identifier that references a member that is not present in the component value, or whose value is malformed. E.g., the identifier is "example-dict";key="c" and the value of the Example-Dict header field is a=1, b=2, which does not have the c value.

In the following non-normative example, the HTTP message being signed is the following 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-Digest: sha-512=:WZDPaVn/7XgHaAy8pmojAkGWoRx2UFChF41A2svX+T\
  aPm+AbwAgBWnrIiYllu7BNNyealdVLvRwEmTHWXvJwew==:
Content-Length: 18

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

The covered components consist of the @method, @authority, and @path derived components followed by the Content-Digest, Content-Length, and Content-Type HTTP header fields, in order. The signature parameters consist of a creation timestamp of 1618884473 and a key identifier of test-key-rsa-pss. Note that no explicit alg parameter is given here since the verifier is assumed by the application to correctly use the RSA PSS algorithm based on the identified key. The signature base for this message with these parameters is:

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

"@method": POST
"@authority": example.com
"@path": /foo
"content-digest": sha-512=:WZDPaVn/7XgHaAy8pmojAkGWoRx2UFChF41A2svX\
  +TaPm+AbwAgBWnrIiYllu7BNNyealdVLvRwEmTHWXvJwew==:
"content-length": 18
"content-type": application/json
"@signature-params": ("@method" "@authority" "@path" \
  "content-digest" "content-length" "content-type")\
  ;created=1618884473;keyid="test-key-rsa-pss"
```

Figure 1: Non-normative example Signature Base

Note that the example signature base here, or anywhere else within this specification, does not include the final newline that ends the displayed example.

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

Creation of an HTTP message signature is a process that takes as its input the message and the requirements for the application. The output is a signature value and set of signature parameters that can be applied to the message.

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. The expiration is a hint to the verifier, expressing the time at which the signer is no longer willing to vouch for the safety of the signature.

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 derived component listed in Section 2.2 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 derived 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 base. 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 base using these parameters and the signature base creation algorithm. (Section 2.4)

6. The signer uses the HTTP_SIGN primitive function to sign the signature base with the chosen signing algorithm using the key material chosen by the signer. The HTTP_SIGN primitive and several concrete applications of 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 base is signed with the test-key-rsa-pss key in Appendix B.1.2 and the RSA PSS algorithm described in Section 3.3.1, giving the following message signature output value, encoded in Base64:

NOTE: '\' line wrapping per RFC 8792

```
HIbjHC5rS0BYaa9v4QfD4193TORw7u9edguPh0AW3dMq9WImrlFrCGUDih47vAxi4L2\
YRZ3XMJc1uOKk/J0ZmZ+wcta4nKIgBkKq0rM9hs3CQyxXGxHLMCy8uqK488o+9jrptQ\
+xFPHK7a9sRL1IXNaagCNN3ZxJsYapFj+JXbmaI5rtAdSfSvzPuBCh+ARHBmWuNo1Uz\
VVdHXrl8ePL4cccqlazIJdC4QEjrF+Sn4IxBQzTZsL9y9TP5FsZYzHvDqbInkTNigBc\
E9cKOYNFCn4D/WM7F6TNuZO9EgtzepLWcjTymlHzK7aXq6Am6sfOrpIC49yXjj3ae6H\
RalVc/g==
```

Figure 2: Non-normative example signature value

Note that the RSA PSS algorithm in use here is non-deterministic, meaning a different signature value will be created every time the algorithm is run. The signature value provided here can be validated against the given keys, but newly-generated signature values are not expected to match the example. See Section 7.19.

3.2. Verifying a Signature

Verification of an HTTP message signature is a process that takes as its input the message (including Signature and Signature-Input fields) and the requirements for the application. The output of the verification is either a positive verification or an error.

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

1. Parse the Signature and Signature-Input fields as described in Section 4.1 and Section 4.2, 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 based on the policy and configuration of the verifier. 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 as a parameterized Inner List to get the ordered list of covered components and the signature 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 than 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 fail the verification.

7. Use the received HTTP message and the signature's metadata to recreate the signature base, using the algorithm defined in Section 2.4. The value of the @signature-params input is the

value of the Signature-Input field for this signature serialized according to the rules described in Section 2.2.1, not including the signature's label from the Signature-Input field.

8. If the key material is appropriate for the algorithm, apply the appropriate HTTP_VERIFY cryptographic verification algorithm to the signature, recalculated signature base, key material, signature value. The HTTP_VERIFY primitive and several concrete algorithms are defined in Section 3.3.

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

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

For example, verifying the signature with the key sig1 of the following message with the test-key-rsa-pss key in Appendix B.1.2 and the RSA PSS algorithm described in Section 3.3.1:

```
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-Digest: sha-512=:WZDPaVn/7XgHaAy8pmojAkGWoRx2UFChF41A2svX+T\
  aPm+AbwAgBWnrIiYllu7BNNyealdVLvRwEmTHWXvJwew==:
Content-Length: 18
Signature-Input: sig1=("@method" "@authority" "@path" \
  "content-digest" "content-length" "content-type")\
  ;created=1618884473;keyid="test-key-rsa-pss"
Signature: sig1=:HIbjHC5rS0BYaa9v4QfD4193TORw7u9edguPh0AW3dMq9WImrl\
  FrCGUDih47vAxi4L2YRZ3XMJc1uOKk/J0ZmZ+wcta4nKIgBkKq0rM9hs3CQyxXGxH\
  LMCy8uqK488o+9jrptQ+xFPHK7a9sRL1IXNaagCNN3ZxJsYapFj+JXbmaI5rtAdSf\
  SvzPuBCh+ARHBmWuNo1UzVVdHXrl8ePL4cccqlazIJdC4QEjrF+Sn4IxBQzTZsL9y\
  9TP5FsZYzHvDqbInkTNigBcE9cKOYNFCn4D/WM7F6TNuZO9EgtzepLWcjTymlHzK7\
  aXq6Am6sfOrpIC49yXjj3ae6HRalVc/g==:

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

With the additional requirements that at least the method, authority, path, content-digest, content-length, and content-type be signed, and that the signature creation timestamp is recent enough at the time of verification, the verification passes.

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 from the time of the created time stamp.

- Rejection of signatures past the expiration time in the expires time stamp. Note that the expiration time is a hint from the signer and that a verifier can always reject a signature ahead of its expiration time.

- Prohibition of certain signature metadata parameters, such as runtime algorithm signaling with the alg parameter when the algorithm is determined from the key information.

- Ensuring successful dereferencing of the keyid parameter to valid and appropriate key material.

- 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.

Each signature algorithm method takes as its input the signature base defined in Section 2.4 as a byte array (M), the signing key material (Ks), and outputs the signature output as a byte array (S):

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

Each verification algorithm method takes as its input the recalculated signature base defined in Section 2.4 as a byte array (M), the verification key material (Kv), and the presented signature to be verified as a byte array (S) and outputs the verification result (V) as a boolean:

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

This section contains several common algorithm methods. The method to use can be communicated through the explicit algorithm signature parameter alg defined in Section 2.2.1, 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 base (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 base 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 base (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 base 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 indicate if the signature presented is valid.

Note that the output of RSA PSS algorithms are non-deterministic, and therefore it is not correct to re-calculate a new signature on the signature base and compare the results to an existing signature. Instead, the verification algorithm defined here needs to be used. See Section 7.19.

Use of this algorithm can be indicated at runtime using the rsa-pss-sha512 value for the alg signature parameter.

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 base (M) (Section 2.4). The hash SHA-256 [RFC6234] is applied to the signature base 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 base (M) re-created as described in Section 3.2. The hash function SHA-256 [RFC6234] is applied to the signature base 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.

Use of this algorithm can be indicated at runtime using the rsa-v1_5-sha256 value for the alg signature parameter.

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 base (text) (Section 2.4). The hash function SHA-256 [RFC6234] is applied to the signature base 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.

Use of this algorithm can be indicated at runtime using the hmac-sha256 value for the alg signature parameter.

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 base (Section 2.4). The hash SHA-256 [RFC6234] is applied to the signature base to create the digest content to which the digital signature is applied, (M). The signature algorithm returns two integer values, r and s. These are both encoded in big-endian unsigned integers, zero-padded to 32-octets each. These encoded values are concatenated into a single 64-octet array consisting of the encoded value of r followed by the encoded value of s. The resulting concatenation of (r, s) is byte array of 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 base re-created as described in Section 3.2. The hash function SHA-256 [RFC6234] is applied to the signature base to create the digest content to which the signature verification function is applied, (M). The verifier extracts the HTTP message signature to be verified (S) as described in Section 3.2. This value is a 64-octet array consisting of the encoded values of r and s concatenated in order. These are both encoded in big-endian unsigned integers, zero-padded to 32-octets each. The resulting signature value (r, s) is used as input to the signature verification function. The results of the verification function indicate if the signature presented is valid.

Note that the output of ECDSA algorithms are non-deterministic, and therefore it is not correct to re-calculate a new signature on the signature base and compare the results to an existing signature. Instead, the verification algorithm defined here needs to be used. See Section 7.19.

Use of this algorithm can be indicated at runtime using the ecdsa-p256-sha256 value for the alg signature parameter.

3.3.5. EdDSA using curve edwards25519

To sign using this algorithm, the signer applies the Ed25519 algorithm Section 5.1.6 of [RFC8032] with the signer's private signing key and the signature base (Section 2.4). The signature base is taken as the input message (M) with no pre-hash function. The signature is a 64-octet concatenation of R and S as specified in Section 5.1.6 of [RFC8032], and this is taken as a byte array for the HTTP message signature output used in Section 3.1.

To verify using this algorithm, the signer applies the Ed25519 algorithm Section 5.1.7 of [RFC8032] using the public key portion of the verification key material (A) and the signature base re-created as described in Section 3.2. The signature base is taken as the input message (M) with no pre-hash function. The signature to be verified is processed as the 64-octet concatenation of R and S as specified in Section 5.1.7 of [RFC8032]. The results of the verification function indicate if the signature presented is valid.

Use of this algorithm can be indicated at runtime using the ed25519 value for the alg signature parameter.

3.3.6. JSON Web Signature (JWS) algorithms

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

For both signing and verification, the HTTP messages signature base (Section 2.4) is used as the entire "JWS Signing Input". The JOSE Header defined in [RFC7517] is not used, and the signature base 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.

JWA algorithm values from the JSON Web Signature and Encryption Algorithms Registry are not included as signature parameters. In fact, the explicit alg signature parameter is not used at all when using JOSE signing algorithms, as the JWS algorithm can be signaled using JSON Web Keys or other mechanisms common to JOSE implementations.

4. Including a Message Signature in a Message

Message signatures can be included within an HTTP message via the Signature-Input and Signature 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 fields. The label is chosen by the signer, except where a specific label is dictated by protocol negotiations such as described in Section 5

An HTTP message signature MUST use both fields and each field MUST contain the same labels. The Signature-Input and Signature Dictionaries are parallel data structures of each other, and the presence of any key in one field but not in the other is an error. The Signature-Input field identifies the covered components and parameters that describe how the signature was generated, while the Signature field contains the signature value. Each field MAY contain multiple labeled values.

4.1. The Signature-Input HTTP Field

The Signature-Input field is a Dictionary structured field defined in Section 3.2 of [STRUCTURED-FIELDS] 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 key 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 Inner List serialization process defined in Section 2.2.1.

NOTE: '\' line wrapping per RFC 8792

```
Signature-Input: sig1=("@method" "@target-uri" "@authority" \
  "content-digest" "cache-control");\
  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 base's @signature-params value. Note that in a structured field value, parameter order has to be preserved.

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 [HTTP], it is RECOMMENDED that the Signature-Input 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 field is a Dictionary structured field defined in Section 3.2 of [STRUCTURED-FIELDS] containing one or more message signatures generated from components and context of the HTTP message. The member's key is an identifier that uniquely identifies the message signature within the context of the HTTP message. The member's value is a Byte Sequence containing the signature value for the message signature identified by the member name.

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 [HTTP], it is RECOMMENDED that the Signature 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. Each distinct signature MUST have a unique label. These multiple signatures could be added all by the same signer or could come from several different signers. 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 non-normative example starts with a signed request from the client. The proxy takes this request validates the client's signature.

```
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-Digest: sha-512=:WZDPaVn/7XgHaAy8pmojAkGWoRx2UFChF41A2svX+T\
  aPm+AbwAgBWnrIiYllu7BNNyealdVLvRwEmTHWXvJwew==:
Content-Length: 18
Signature-Input: sig1=("@method" "@authority" "@path" \
  "content-digest" "content-length" "content-type")\
  ;created=1618884475;keyid="test-key-rsa-pss"
Signature:  sig1=:LAH8BjcfcOcLojiuOBFWn0P5keD3xAOuJRGziCLuD8r5MW9S0\
  RoXXLzLSRfGY/3SF8kVIkHjE13SEFdTo4Af/fJ/Pu9wheqoLVdwXyY/UkBIS1M8Br\
  c8IODsn5DFIrG0IrburbLi0uCc+E2ZIIb6HbUJ+o+jP58JelMTe0QE3IpWINTEzpx\
  jqDf5/Df+InHCAkQCTuKsamjWXUpyOT1Wkxi7YPVNOjW4MfNuTZ9HdbD2Tr65+BXe\
  TG9ZS/9SWuXAc+BZ8WyPz0QRz//ec3uWXd7bYYODSjRAxHqX+S1ag3LZElYyUKaAI\
  jZ8MGOt4gXEwCSLDv/zqxZeWLj/PDkn6w==:

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

The proxy then alters the message before forwarding it on to the origin server, changing the target host and adding the Forwarded header field defined in [RFC7239].

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

POST /foo?param=Value&Pet=dog HTTP/1.1
Host: origin.host.internal.example
Date: Tue, 20 Apr 2021 02:07:56 GMT
Content-Type: application/json
Content-Digest: sha-512=:WZDPaVn/7XgHaAy8pmojAkGWoRx2UFChF41A2svX+T\
  aPm+AbwAgBWnrIiYllu7BNNyealdVLvRwEmTHWXvJwew==:
Content-Length: 18
Forwarded: for=192.0.2.123
Signature-Input: sig1=("@method" "@authority" "@path" \
  "content-digest" "content-length" "content-type")\
  ;created=1618884475;keyid="test-key-rsa-pss"
Signature:  sig1=:LAH8BjcfcOcLojiuOBFWn0P5keD3xAOuJRGziCLuD8r5MW9S0\
  RoXXLzLSRfGY/3SF8kVIkHjE13SEFdTo4Af/fJ/Pu9wheqoLVdwXyY/UkBIS1M8Br\
  c8IODsn5DFIrG0IrburbLi0uCc+E2ZIIb6HbUJ+o+jP58JelMTe0QE3IpWINTEzpx\
  jqDf5/Df+InHCAkQCTuKsamjWXUpyOT1Wkxi7YPVNOjW4MfNuTZ9HdbD2Tr65+BXe\
  TG9ZS/9SWuXAc+BZ8WyPz0QRz//ec3uWXd7bYYODSjRAxHqX+S1ag3LZElYyUKaAI\
  jZ8MGOt4gXEwCSLDv/zqxZeWLj/PDkn6w==:

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

The proxy includes the client's signature value under the label sig1, which the proxy signs in addition to the Forwarded field. 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. The proxy identifies its own key and algorithm and, in this example, includes an expiration for the signature to indicate to downstream systems that the proxy will not vouch for this signed message past this short time window. This results in a signature base of:

NOTE: '\' line wrapping per RFC 8792

```
"signature";key="sig1": :LAH8BjcfcOcLojiuOBFWn0P5keD3xAOuJRGziCLuD8\
  r5MW9S0RoXXLzLSRfGY/3SF8kVIkHjE13SEFdTo4Af/fJ/Pu9wheqoLVdwXyY/UkB\
  IS1M8Brc8IODsn5DFIrG0IrburbLi0uCc+E2ZIIb6HbUJ+o+jP58JelMTe0QE3IpW\
  INTEzpxjqDf5/Df+InHCAkQCTuKsamjWXUpyOT1Wkxi7YPVNOjW4MfNuTZ9HdbD2T\
  r65+BXeTG9ZS/9SWuXAc+BZ8WyPz0QRz//ec3uWXd7bYYODSjRAxHqX+S1ag3LZEl\
  YyUKaAIjZ8MGOt4gXEwCSLDv/zqxZeWLj/PDkn6w==:
"forwarded": for=192.0.2.123
"@signature-params": ("signature";key="sig1" "forwarded")\
  ;created=1618884480;expires=1618884540;keyid="test-key-rsa"\
  ;alg="rsa-v1_5-sha256"
```

And a signature output value of:

NOTE: '\' line wrapping per RFC 8792

```
G1WLTL4/9PGSKEQbSAMypZNk+I2dpLJ6qvl2JISahlP31OO/QEUd8/HdO2O7vYLi5k3\
JIiAK3UPK4U+kvJZyIUidsiXlzRI+Y2se3SGo0D8dLfhG95bKr6ukYXl60QHpsGRTfS\
iwdtvYKXGpKNrMlISJYd+oGrGRyI9gbCy0aFhc6I/okIMLeK4g9PgzpC3YTwhUQ98KI\
BNLWHgREfBgJxjPbxFlsgJ9ykPviLj8GKJ81HwsK3XM9P7WaS7fMGOt8h1kSqgkZQB9\
YqiIo+WhHvJa7iPy8QrYFKzx9BBEY6AwfStZAsXXz3LobZseyxsYcLJLs8rY0wVA9NP\
sxKrHGA==
```

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

POST /foo?param=Value&Pet=dog HTTP/1.1
Host: origin.host.internal.example
Date: Tue, 20 Apr 2021 02:07:56 GMT
Content-Type: application/json
Content-Digest: sha-512=:WZDPaVn/7XgHaAy8pmojAkGWoRx2UFChF41A2svX+T\
  aPm+AbwAgBWnrIiYllu7BNNyealdVLvRwEmTHWXvJwew==:
Content-Length: 18
Forwarded: for=192.0.2.123
Signature-Input: sig1=("@method" "@authority" "@path" \
    "content-digest" "content-length" "content-type")\
    ;created=1618884475;keyid="test-key-rsa-pss", \
  proxy_sig=("signature";key="sig1" "forwarded")\
    ;created=1618884480;expires=1618884540;keyid="test-key-rsa"\
    ;alg="rsa-v1_5-sha256"
Signature:  sig1=:LAH8BjcfcOcLojiuOBFWn0P5keD3xAOuJRGziCLuD8r5MW9S0\
    RoXXLzLSRfGY/3SF8kVIkHjE13SEFdTo4Af/fJ/Pu9wheqoLVdwXyY/UkBIS1M8\
    Brc8IODsn5DFIrG0IrburbLi0uCc+E2ZIIb6HbUJ+o+jP58JelMTe0QE3IpWINT\
    EzpxjqDf5/Df+InHCAkQCTuKsamjWXUpyOT1Wkxi7YPVNOjW4MfNuTZ9HdbD2Tr\
    65+BXeTG9ZS/9SWuXAc+BZ8WyPz0QRz//ec3uWXd7bYYODSjRAxHqX+S1ag3LZE\
    lYyUKaAIjZ8MGOt4gXEwCSLDv/zqxZeWLj/PDkn6w==:, \
  proxy_sig=:G1WLTL4/9PGSKEQbSAMypZNk+I2dpLJ6qvl2JISahlP31OO/QEUd8/\
    HdO2O7vYLi5k3JIiAK3UPK4U+kvJZyIUidsiXlzRI+Y2se3SGo0D8dLfhG95bKr\
    6ukYXl60QHpsGRTfSiwdtvYKXGpKNrMlISJYd+oGrGRyI9gbCy0aFhc6I/okIML\
    eK4g9PgzpC3YTwhUQ98KIBNLWHgREfBgJxjPbxFlsgJ9ykPviLj8GKJ81HwsK3X\
    M9P7WaS7fMGOt8h1kSqgkZQB9YqiIo+WhHvJa7iPy8QrYFKzx9BBEY6AwfStZAs\
    XXz3LobZseyxsYcLJLs8rY0wVA9NPsxKrHGA==:

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

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 request, the component identifiers can not include the @status component identifier.

5.1. The Accept-Signature Field

The Accept-Signature field is a Dictionary structured field defined in Section 3.2 of [STRUCTURED-FIELDS] 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 2.2.1.

NOTE: '\' line wrapping per RFC 8792

```
Accept-Signature: sig1=("@method" "@target-uri" "@authority" \
  "content-digest" "cache-control");\
  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 fields 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

IANA is requested to create three registries and to populate those registries with initial values as described in this section.

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.

Algorithms added to this registry MUST NOT be aliases for other entries in the registry.

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.

Description:

A brief description of the algorithm used to sign the signature base.

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.

Specification document(s):

Reference to the document(s) that specify the algorithm, 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

Algorithm Name	Description	Status	Specification document(s)
rsa-pss-sha512	RSASSA-PSS using SHA-512	Active	Section 3.3.1 of RFC nnnn
rsa-v1_5-sha256	RSASSA-PKCS1-v1_5 using SHA-256	Active	Section 3.3.2 of RFC nnnn
hmac-sha256	HMAC using SHA-256	Active	Section 3.3.3 of RFC nnnn
ecdsa-p256-sha256	ECDSA using curve P-256 DSS and SHA-256	Active	Section 3.3.4 of RFC nnnn

ed25519	Edwards Curve DSA using curve edwards25519	Active	Section 3.3.5 of RFC nnnn

Table 1: Initial contents of the HTTP Signature Algorithms Registry.

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

Name:

An identifier for the HTTP signature metadata parameter. 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.

Description:

A brief description of the metadata parameter and what it represents.

Specification document(s):

Reference to the document(s) that specify the parameter, 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.

6.2.2. 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	Description	Specification document(s)
alg	Explicitly declared signature algorithm	Section 2.2.1 of RFC nnnn
created	Timestamp of signature creation	Section 2.2.1 of RFC nnnn
expires	Timestamp of proposed signature expiration	Section 2.2.1 of RFC nnnn
keyid	Key identifier for the signing and verification keys used to create this signature	Section 2.2.1 of RFC nnnn
nonce	A single-use nonce value	Section 2.2.1 of RFC nnnn

Table 2: Initial contents of the HTTP Signature Metadata Parameters Registry.

6.3. HTTP Signature Derived Component Names Registry

This document defines a method for canonicalizing HTTP message components, including components that can be derived from the context of the HTTP message outside of the HTTP fields. These derived components are identified by a unique string, known as the component name. Component names for derived components always start with the "@" (at) symbol to distinguish them from HTTP header fields. IANA is asked to create and maintain a new registry typed "HTTP Signature Derived Component Names" to record and maintain the set of non-field component names 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

Name:
A name for the HTTP derived component. The name MUST begin with the "@" character followed by 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 name MUST be unique within the context of the registry.

Description:
A description of the derived component.

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.

Target:
The valid message targets for the derived parameter. MUST be one of the values "Request", "Response", or "Request, Response". The semantics of these are defined in Section 2.2.

Specification document(s):
Reference to the document(s) that specify the derived component, 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.

6.3.2. Initial Contents

The table below contains the initial contents of the HTTP Signature Derived Component Names Registry.

Name	Description	Status	Target	Specification document(s)	
@signature-params	Signature parameters,	Active	Request, Response	Section 2.2.1 of RFC nnnn	

	including covered content list				
@method	The HTTP request method	Active	Request	Section 2.2.2 of RFC nnnn	
@authority	The HTTP authority, or target host	Active	Request	Section 2.2.4 of RFC nnnn	
@scheme	The URI scheme of the request URI	Active	Request	Section 2.2.5 of RFC nnnn	
@target-uri	The full target URI of the request	Active	Request	Section 2.2.3 of RFC nnnn	
@request-target	The request target of the request	Active	Request	Section 2.2.6 of RFC nnnn	
@path	The full path of the request URI	Active	Request	Section 2.2.7 of RFC nnnn	
@query	The full query of the request URI	Active	Request	Section 2.2.8 of RFC nnnn	
@query-param		A single named query parameter	Active	Request	Section 2.2.9 of RFC nnnn
@status	The status code of the response	Active	Response	Section 2.2.10 of RFC nnnn	

Table 3: Initial contents of the HTTP Signature Derived Component Names Registry.

6.4. HTTP Signature Component Parameters Registry

This document defines several kinds of component identifiers, some of which can be parameterized in specific circumstances to provide unique modified behavior. IANA is asked to create and maintain a new registry typed "HTTP Signature Component Parameters" to record and maintain the set of parameters names, the component identifiers they are associated with, and the modifications these parameters make to the component value. Initial values for this registry are given in Section 6.4.2. 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 6.4.1.

6.4.1. Registration Template

Name:
A name for the parameter. The name MUST be an ASCII string that conforms to the key ABNF rule defined in [STRUCTURED-FIELDS] and SHOULD NOT exceed 20 characters in length. The name MUST be unique within the context of the registry.

Description:
A description of the parameter's function.

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

Target:
The applicable component identifiers for the parmeter. Can be a combination of one or more derived component identifiers as described in Section 2.2, one or more specific HTTP field names, the special value "Field Value" meaning all HTTP fields, or a separate human-readable description of the target.

Specification document(s):
Reference to the document(s) that specify the derived component, 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.

6.4.2. Initial Contents

The table below contains the initial contents of the HTTP Signature Derived Component Names Registry.

Name	Description	Status	Target	Specification document(s)
key	Dictionary structured fields	Active	Section 2.1.2 of RFC nnnn	
name	Named query parameters	Active	@query-param	Section 2.2.9 of RFC nnnn
sf	Strict structured field serialization	Active	Structured fields	Section 2.1.1 of RFC nnnn
req	Related request indicator	Active	Derived content identifiers with a target of Request or any field	Section 2.2.5 of RFC nnnn

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

7. Security Considerations

In order for an HTTP message to be considered covered by a signature, all of the following conditions have to be true:

* a signature is expected or allowed on the message by the verifier

* the signature exists on the message

- the signature is verified against the identified key material and algorithm

- the key material and algorithm are appropriate for the context of the message

- the signature is within expected time boundaries

- the signature covers the expected content, including any critical components

7.1. Signature Verification Skipping

HTTP Message Signatures only provide security if the signature is verified by the verifier. Since the message to which the signature is attached remains a valid HTTP message without the signature fields, it is possible for a verifier to ignore the output of the verification function and still process the message. Common reasons for this could be relaxed requirements in a development environment or a temporary suspension of enforcing verification during debugging an overall system. Such temporary suspensions are difficult to detect under positive-example testing since a good signature will always trigger a valid response whether or not it has been checked.

To detect this, verifiers should be tested using both valid and invalid signatures, ensuring that the invalid signature fails as expected.

7.2. Use of TLS

The use of HTTP Message Signatures does not negate the need for TLS or its equivalent to protect information in transit. Message signatures provide message integrity over the covered message components but do not provide any confidentiality for the communication between parties.

TLS provides such confidentiality between the TLS endpoints. As part of this, TLS also protects the signature data itself from being captured by an attacker, which is an important step in preventing signature replay (Section 7.3).

When TLS is used, it needs to be deployed according to the recommendations in [BCP195].

7.3. Signature Replay

Since HTTP Message Signatures allows sub-portions of the HTTP message to be signed, it is possible for two different HTTP messages to validate against the same signature. The most extreme form of this would be a signature over no message components. If such a signature were intercepted, it could be replayed at will by an attacker, attached to any HTTP message. Even with sufficient component coverage, a given signature could be applied to two similar HTTP messages, allowing a message to be replayed by an attacker with the signature intact.

To counteract these kinds of attacks, it's first important for the signer to cover sufficient portions of the message to differentiate it from other messages. In addition, the signature can use the nonce

signature parameter to provide a per-message unique value to allow the verifier to detect replay of the signature itself if a nonce value is repeated. Furthermore, the signer can provide a timestamp for when the signature was created and a time at which the signer considers the signature to be invalid, limiting the utility of a captured signature value.

If a verifier wants to trigger a new signature from a signer, it can send the Accept-Signature header field with a new nonce parameter. An attacker that is simply replaying a signature would not be able to generate a new signature with the chosen nonce value.

7.4. Insufficient Coverage

Any portions of the message not covered by the signature are susceptible to modification by an attacker without affecting the signature. An attacker can take advantage of this by introducing a header field or other message component that will change the processing of the message but will not be covered by the signature. Such an altered message would still pass signature verification, but when the verifier processes the message as a whole, the unsigned content injected by the attacker would subvert the trust conveyed by the valid signature and change the outcome of processing the message.

To combat this, an application of this specification should require as much of the message as possible to be signed, within the limits of the application and deployment. The verifier should only trust message components that have been signed. Verifiers could also strip out any sensitive unsigned portions of the message before processing of the message continues.

7.5. Cryptography and Signature Collision

The HTTP Message Signatures specification does not define any of its own cryptographic primitives, and instead relies on other specifications to define such elements. If the signature algorithm or key used to process the signature base is vulnerable to any attacks, the resulting signature will also be susceptible to these same attacks.

A common attack against signature systems is to force a signature collision, where the same signature value successfully verifies against multiple different inputs. Since this specification relies on reconstruction of the input string based on an HTTP message, and the list of components signed is fixed in the signature, it is difficult but not impossible for an attacker to effect such a collision. An attacker would need to manipulate the HTTP message and its covered message components in order to make the collision effective.

To counter this, only vetted keys and signature algorithms should be used to sign HTTP messages. The HTTP Message Signatures Algorithm Registry is one source of potential trusted algorithms.

While it is possible for an attacker to substitute the signature parameters value or the signature value separately, the signature base generation algorithm (Section 2.4) always covers the signature parameters as the final value in the input string using a deterministic serialization method. This step strongly binds the signature base with the signature value in a way that makes it much more difficult for an attacker to perform a partial substitution on the signature bases.

7.6. Key Theft

A foundational assumption of signature-based cryptographic systems is that the signing key is not compromised by an attacker. If the keys used to sign the message are exfiltrated or stolen, the attacker will be able to generate their own signatures using those keys. As a consequence, signers have to protect any signing key material from exfiltration, capture, and use by an attacker.

To combat this, signers can rotate keys over time to limit the amount of time stolen keys are useful. Signers can also use key escrow and storage systems to limit the attack surface against keys. Furthermore, the use of asymmetric signing algorithms exposes key material less than the use of symmetric signing algorithms (Section 7.11).

7.7. Modification of Required Message Parameters

An attacker could effectively deny a service by modifying an otherwise benign signature parameter or signed message component. While rejecting a modified message is the desired behavior, consistently failing signatures could lead to the verifier turning off signature checking in order to make systems work again (see Section 7.1).

If such failures are common within an application, the signer and verifier should compare their generated signature bases with each other to determine which part of the message is being modified. However, the signer and verifier should not remove the requirement to sign the modified component when it is suspected an attacker is modifying the component.

7.8. Mismatch of Signature Parameters from Message

The verifier needs to make sure that the signed message components match those in the message itself. This specification encourages this by requiring the verifier to derive these values from the message, but lazy cacheing or conveyance of the signature base to a processing system could lead to downstream verifiers accepting a message that does not match the presented signature.

7.9. Multiple Signature Confusion

Since multiple signatures can be applied to one message (Section 4.3), it is possible for an attacker to attach their own signature to a captured message without modifying existing signatures. This new signature could be completely valid based on the attacker's key, or it could be an invalid signature for any number of reasons. Each of these situations need to be accounted for.

A verifier processing a set of valid signatures needs to account for all of the signers, identified by the signing keys. Only signatures from expected signers should be accepted, regardless of the cryptographic validity of the signature itself.

A verifier processing a set of signatures on a message also needs to determine what to do when one or more of the signatures are not valid. If a message is accepted when at least one signature is valid, then a verifier could drop all invalid signatures from the request before processing the message further. Alternatively, if the verifier rejects a message for a single invalid signature, an attacker could use this to deny service to otherwise valid messages by injecting invalid signatures alongside the valid ones.

7.10. Signature Labels

HTTP Message Signature values are identified in the Signature and Signature-Input field values by unique labels. These labels are chosen only when attaching the signature values to the message and are not accounted for in the signing process. An intermediary adding its own signature is allowed to re-label an existing signature when processing the message.

Therefore, applications should not rely on specific labels being present, and applications should not put semantic meaning on the labels themselves. Instead, additional signature parmeters can be used to convey whatever additional meaning is required to be attached to and covered by the signature.

7.11. Symmetric Cryptography

The HTTP Message Signatures specification allows for both asymmetric and symmetric cryptography to be applied to HTTP messages. By its nature, symmetric cryptographic methods require the same key material to be known by both the signer and verifier. This effectively means that a verifier is capable of generating a valid signature, since they have access to the same key material. An attacker that is able to compromise a verifier would be able to then impersonate a signer.

Where possible, asymmetric methods or secure key agreement mechanisms should be used in order to avoid this type of attack. When symmetric methods are used, distribution of the key material needs to be protected by the overall system. One technique for this is the use of separate cryptographic modules that separate the verification process (and therefore the key material) from other code, minimizing the vulnerable attack surface. Another technique is the use of key derivation functions that allow the signer and verifier to agree on unique keys for each message without having to share the key values directly.

Additionally, if symmetric algorithms are allowed within a system, special care must be taken to avoid key downgrade attacks (Section 7.15).

7.12. Canonicalization Attacks

Any ambiguity in the generation of the signature base could provide an attacker with leverage to substitute or break a signature on a message. Some message component values, particularly HTTP field values, are potentially susceptible to broken implementations that could lead to unexpected and insecure behavior. Naive implementations of this specification might implement HTTP field processing by taking the single value of a field and using it as the direct component value without processing it appropriately.

For example, if the handling of obs-fold field values does not remove the internal line folding and whitespace, additional newlines could be introduced into the signature base by the signer, providing a potential place for an attacker to mount a signature collision (Section 7.5) attack. Alternatively, if header fields that appear multiple times are not joined into a single string value, as is required by this specification, similar attacks can be mounted as a signed component value would show up in the input string more than once and could be substituted or otherwise attacked in this way.

To counter this, the entire field processing algorithm needs to be implemented by all implementations of signers and verifiers.

7.13. Key Specification Mix-Up

The existence of a valid signature on an HTTP message is not sufficient to prove that the message has been signed by the appropriate party. It is up to the verifier to ensure that a given key and algorithm are appropriate for the message in question. If the verifier does not perform such a step, an attacker could substitute their own signature using their own key on a message and force a verifier to accept and process it. To combat this, the verifier needs to ensure that not only does the signature validate for a message, but that the key and algorithm used are appropriate.

7.14. HTTP Versions and Component Ambiguity

Some message components are expressed in different ways across HTTP versions. For example, the authority of the request target is sent using the Host header field in HTTP 1.1 but with the :authority pseudo-header in HTTP 2. If a signer sends an HTTP 1.1 message and signs the Host field, but the message is translated to HTTP 2 before it reaches the verifier, the signature will not validate as the Host header field could be dropped.

It is for this reason that HTTP Message Signatures defines a set of derived components that define a single way to get value in question, such as the @authority derived component (Section 2.2.4) in lieu of the Host header field. Applications should therefore prefer derived components for such options where possible.

7.15. Key and Algorithm Specification Downgrades

Applications of this specification need to protect against key specification downgrade attacks. For example, the same RSA key can be used for both RSA-PSS and RSA v1.5 signatures. If an application expects a key to only be used with RSA-PSS, it needs to reject signatures for that key using the weaker RSA 1.5 specification.

Another example of a downgrade attack occurs when an asymmetric algorithm is expected, such as RSA-PSS, but an attacker substitutes a signature using symmetric algorithm, such as HMAC. A naive verifier implementation could use the value of the public RSA key as the input to the HMAC verification function. Since the public key is known to

the attacker, this would allow the attacker to create a valid HMAC signature against this known key. To prevent this, the verifier needs to ensure that both the key material and the algorithm are appropriate for the usage in question. Additionally, while this specification does allow runtime specification of the algorithm using the alg signature parameter, applications are encouraged to use other mechanisms such as static configuration or higher protocol-level algorithm specification instead.

7.16. Parsing Structured Field Values

Several parts of this specification rely on the parsing of structured field values [STRUCTURED-FIELDS]. In particular, normalization of HTTP structured field values (Section 2.1.1), referencing members of a dictionary structured field (Section 2.1.2), and processing the @signature-input value when verifying a signature (Section 3.2). While structured field values are designed to be relatively simple to parse, a naive or broken implementation of such a parser could lead to subtle attack surfaces being exposed in the implementation.

For example, if a buggy parser of the @signature-input value does not enforce proper closing of quotes around string values within the list of component identifiers, an attacker could take advantage of this and inject additional content into the signature base through manipulating the Signature-Input field value on a message.

To counteract this, implementations should use fully compliant and trusted parsers for all structured field processing, both on the signer and verifier side.

7.17. Choosing Message Components

Applications of HTTP Message Signatures need to decide which message components will be covered by the signature. Depending on the application, some components could be expected to be changed by intermediaries prior to the signature's verification. If these components are covered, such changes would, by design, break the signature.

However, the HTTP Message Signature standard allows for flexibility in determining which components are signed precisely so that a given application can choose the appropriate portions of the message that need to be signed, avoiding problematic components. For example, a web application framework that relies on rewriting query parameters might avoid use of the @query derived component in favor of sub-indexing the query value using @query-param derived components instead.

Some components are expected to be changed by intermediaries and ought not to be signed under most circumstance. The Via and Forwarded header fields, for example, are expected to be manipulated by proxies and other middle-boxes, including replacing or entirely dropping existing values. These fields should not be covered by the signature except in very limited and tightly-coupled scenarios.

Additional considerations for choosing signature aspects are discussed in Section 1.4.

7.18. Confusing HTTP Field Names for Derived Component Names

The definition of HTTP field names does not allow for the use of the @ character anywhere in the name. As such, since all derived component names start with the @ character, these namespaces should be completely separate. However, some HTTP implementations are not sufficiently strict about the characters accepted in HTTP headers. In such implementations, a sender (or attacker) could inject a header field starting with an @ character and have it passed through to the application code. These invalid header fields could be used to override a portion of the derived message content and substitute an arbitrary value, providing a potential place for an attacker to mount a signature collision (Section 7.5) attack.

To combat this, when selecting values for a message component, if the component name starts with the @ character, it needs to be processed as a derived component and never taken as a fields. Only if the component name does not start with the @ character can it be taken from the fields of the message. The algorithm discussed in

Section 2.4 provides a safe order of operations.

7.19. Non-deterministic Signature Primitives

Some cryptographic primitives such as RSA PSS and ECDSA have non-deterministic outputs, which include some amount of entropy within the algorithm. For such algorithms, multiple signatures generated in succession will not match. A lazy implementation of a verifier could ignore this distinction and simply check for the same value being created by re-signing the signature base. Such an implementation would work for deterministic algorithms such as HMAC and EdDSA but fail to verify valid signatures made using non-deterministic algorithms. It is therefore important that a verifier always use the correctly-defined verification function for the algorithm in question and not do a simple comparison.

7.20. Choosing Signature Parameters and Derived Components over HTTP Fields

Some HTTP fields have values and interpretations that are similar to HTTP signature parameters or derived components. In most cases, it is more desirable to sign the non-field alternative. In particular, the following fields should usually not be included in the signature unless the application specifically requires it:

"date" The "date" field value represents the timestamp of the HTTP message. However, the creation time of the signature itself is encoded in the created signature parameter. These two values can be different, depending on how the signature and the HTTP message are created and serialized. Applications processing signatures for valid time windows should use the created signature parameter for such calculations. An application could also put limits on how much skew there is between the "date" field and the created signature parameter, in order to limit the application of a generated signature to different HTTP messages. See also Section 7.3 and Section 7.4.

"host" The "host" header field is specific to HTTP 1.1, and its functionality is subsumed by the "@authority" derived component, defined in Section 2.2.4. In order to preserve the value across different HTTP versions, applications should always use the "@authority" derived component.

7.21. Semantically Equivalent Field Values

The signature base generation algorithm (Section 2.4) uses the value of an HTTP field as its component value. In the common case, this amounts to taking the actual bytes of the field value as the component value for both the signer and verifier. However, some field values allow for transformation of the values in semantically equivalent ways that alter the bytes used in the value itself. For example, a field definition can declare some or all of its value to be case-insensitive, or to have special handling of internal whitespace characters. Other fields have expected transformations from intermediaries, such as the removal of comments in the Via header field. In such cases, a verifier could be tripped up by using the equivalent transformed field value, which would differ from the byte value used by the signer. The verifier would have have a difficult time finding this class of errors since the value of the field is still acceptable for the application, but the actual bytes required by the signature base would not match.

When processing such fields, the signer and verifier have to agree how to handle such transformations, if at all. One option is to not sign problematic fields, but care must be taken to ensure that there is still sufficient signature coverage (Section 7.4) for the application. Another option is to define an application-specific canonicalization value for the field before it is added to the HTTP message, such as to always remove internal comments before signing, or to always transform values to lowercase. Since these transformations are applied prior to the field being used as input to the signature base generation algorithm, the signature base will still simply contain the byte value of the field as it appears within the message. If the transformations were to be applied after the value is extracted from the message but before it is added to the signature base, different attack surfaces such as value substitution attacks could be launched against the application. All application-specific additional rules are outside the scope of this specification, and by their very nature these transformations would harm interoperability of the implementation outside of this specific application. It is recommended that applications avoid the use of such additional rules wherever possible.

7.22. Message Content

On its own, this specification does not provide coverage for the content of an HTTP message under the signature, either in request or response. However, [DIGEST] defines a set of fields that allow a cryptographic digest of the content to be represented in a field. Once this field is created, it can be included just like any other field as defined in Section 2.1.

For example, in the following response message:

```
HTTP/1.1 200 OK
Content-Type: application/json

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

The digest of the content can be added to the Content-Digest field as follows:

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

HTTP/1.1 200 OK
Content-Type: application/json
Content-Digest: \
  sha-256=:X48E9qOokqqrvdts8nOJRJN3OWDUoyWxBf7kbu9DBPE=:

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

This field can be included in a signature base just like any other field along with the basic signature parameters:

```
"@status": 200
"content-digest": \
  sha-256=:X48E9qOokqqrvdts8nOJRJN3OWDUoyWxBf7kbu9DBPE=:
"@signature-input": ("@status" "content-digest")
```

From here, the signing process proceeds as usual.

Upon verification, it is important that the verifier validate not only the signature but also the value of the Content-Digest field itself against the actual received content. Unless the verifier performs this step, it would be possible for an attacker to substitute the message content but leave the Content-Digest field value untouched. Since only the field value is covered by the signature directly, checking only the signature is not sufficient protection against such a substitution attack.

8. Privacy Considerations

8.1. Identification through Keys

If a signer uses the same key with multiple verifiers, or uses the same key over time with a single verifier, the ongoing use of that key can be used to track the signer throughout the set of verifiers that messages are sent to. Since cryptographic keys are meant to be functionally unique, the use of the same key over time is a strong indicator that it is the same party signing multiple messages.

In many applications, this is a desirable trait, and it allows HTTP Message Signatures to be used as part of authenticating the signer to the verifier. However, it could be unintentional tracking that a signer might not be aware of. To counter this kind of tracking, a signer can use a different key for each verifier that it is in communication with. Sometimes, a signer could also rotate their key when sending messages to a given verifier. These approaches do not negate the need for other anti-tracking techniques to be applied as necessary.

8.2. Signatures do not provide confidentiality

HTTP Message Signatures do not provide confidentiality of any of the information protected by the signature. The content of the HTTP message, including the value of all fields and the value of the signature itself, is presented in plaintext to any party with access to the message.

To provide confidentiality at the transport level, TLS or its equivalent can be used as discussed in Section 7.2.

8.3. Oracles

It is important to balance the need for providing useful feedback to developers on error conditions without providing additional information to an attacker. For example, a naive but helpful server implementation might try to indicate the required key identifier needed for requesting a resource. If someone knows who controls that key, a correlation can be made between the resource's existence and the party identified by the key. Access to such information could be used by an attacker as a means to target the legitimate owner of the resource for further attacks.

8.4. Required Content

A core design tenet of this specification is that all message components covered by the signature need to be available to the verifier in order to recreate the signature base and verify the signature. As a consequence, if an application of this specification requires that a particular field be signed, the verifier will need access to the value of that field.

For example, in some complex systems with intermediary processors this could cause the surprising behavior of an intermediary not being able to remove privacy-sensitive information from a message before forwarding it on for processing, for fear of breaking the signature. A possible mitigation for this specific situation would be for the intermediary to verify the signature itself, then modifying the message to remove the privacy-sensitive information. The intermediary can add its own signature at this point to signal to the next destination that the incoming signature was validated, as is shown in the example in Section 4.3.

9. References

9.1. 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/>.

[HTTP] Fielding, R. T., Nottingham, M., and J. Reschke, "HTTP Semantics", Work in Progress, Internet-Draft, draft-ietf-httpbis-semantics-19, 12 September 2021, <https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-semantics-19>.

[HTTP1] Fielding, R. T., Nottingham, M., and J. Reschke, "HTTP/1.1", Work in Progress, Internet-Draft, draft-ietf-httpbis-messaging-19, 12 September 2021, <https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-messaging-19>.

[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>.

[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>.

[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>.

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

[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>.

[STRUCTURED-FIELDS] 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>.

9.2. Informative References

[BCP195] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 7525, May 2015.

Moriarty, K. and S. Farrell, "Deprecating TLS 1.0 and TLS 1.1", BCP 195, RFC 8996, March 2021.

<https://www.rfc-editor.org/info/bcp195>

[CLIENT-CERT] Campbell, B. and M. Bishop, "Client-Cert HTTP Header Field", Work in Progress, Internet-Draft, draft-ietf-httpbis-client-cert-field-02, 25 May 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-client-cert-field-02>.

[DIGEST] Polli, R. and L. Pardue, "Digest Fields", Work in

Progress, Internet-Draft, draft-ietf-httpbis-digest-headers-09, 25 May 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-digest-headers-09>.

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

[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>.

Appendix A. Detecting HTTP Message Signatures

There have been many attempts to create signed HTTP messages in the past, including other non-standardized definitions of the Signature field, which is 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

The following non-normative examples are provided as a means of testing implementations of HTTP Message Signatures. The signed messages given can be used to create the signature base with the stated parameters, creating signatures using the stated algorithms and keys.

The private keys given can be used to generate signatures, though since several of the demonstrated algorithms are nondeterministic, the results of a signature are expected to be different from the exact bytes of the examples. The public keys given can be used to validate all signed 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.

The components for each private key in PEM format can be displayed by executing the following OpenSSL command:

```
openssl pkey -text
```

This command was tested with all the example keys on OpenSSL version 1.1.1m. Note that some systems cannot produce or use these keys directly, and may require additional processing.

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. This key is encoded in PEM Format, with no encryption.

```
   -----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. This key is PCKS#8 encoded in PEM format, with no encryption.

```
-----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
aKy6LIYs0E2cWgp8GADgoBs8llBq0UhX0KfglIeek3n7Z6Gt4YFge2TAcW2WbN4
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 a public and private elliptical curve key pair over the curve P-256, referred to in this document as `test-key-ecc-p256. This key is encoded in PEM format, with no encryption.

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

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

B.1.4. Example Ed25519 Test Key

The following key is an elliptical curve key over the Edwards curve ed25519, referred to in this document as test-key-ed25519. This key is PCKS#8 encoded in PEM format, with no encryption.

```
-----BEGIN PUBLIC KEY-----
MCowBQYDK2VwAyEAJrQLj5P/89iXES9+vFgrIy29clF9CC/oPPsw3c5D0bs=
-----END PUBLIC KEY-----
```

```
-----BEGIN PRIVATE KEY-----
MC4CAQAwBQYDK2VwBCIEIJ+DYvh6SEqVTm50DFtMDoQikTmiCqirVv9mWG9qfSnF
-----END PRIVATE KEY-----
```

B.1.5. 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:

```
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-Digest: sha-512=:WZDPaVn/7XgHaAy8pmojAkGWoRx2UFChF41A2svX+T\
  aPm+AbwAgBWnrIiYllu7BNNyealdVLvRwEmTHWXvJwew==:
Content-Length: 18

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

For responses, this test-response message is used:

```
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-Digest: sha-512=:JlEy2bfUz7WrWIjc1qV6KVLpdr/7L5/L4h7Sxvh6sN\
  HpDQWDCL+GauFQWcZBvVDhiyOnAQsxzZFYwi0wDH+1pw==:
Content-Length: 23

{"message": "good dog"}
```

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

This example presents a minimal signature using the rsa-pss-sha512 algorithm over test-request, covering none of the components of the HTTP message, but providing a timestamped signature proof of possession of the key with a signer-provided nonce.

The corresponding signature base is:

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

"@signature-params": ();created=1618884473;keyid="test-key-rsa-pss"\
  ;nonce="b3k2pp5k7z-50gnwp.yemd"
```

This results in the following Signature-Input and Signature headers being added to the message under the signature label sig-b21:

NOTE: '\' line wrapping per RFC 8792

```
Signature-Input: sig-b21=();created=1618884473\
  ;keyid="test-key-rsa-pss";nonce="b3k2pp5k7z-50gnwp.yemd"
Signature: sig-b21=:d2pmTvmbncD3xQm8E9ZV2828BjQWGgiwAaw5bAkgibUopem\
  LJcWDy/lkbbHAve4cRAtx31Iq786U7it++wgGxbtRxf8Udx7zFZsckzXaJMkA7ChG\
  52eSkFxykJeNqsrWH5S+oxNFlD4dzVuwe8DhTSja8xxbR/Z2cOGdCbzR72rgFWhzx\
  2VjBqJzsPLMIQKhO4DGezXehhWwE56YCE+O6c0mKZsfxVrogUvA4HELjVKWmAvtl6\
  UnCh8jYzuVG5WSb/QEVPnP5TmcAnLH1g+s++v6d4s8m0gCw1fV5/SITLq9mhho8K3\
  +7EPYTU8IU1bLhdxO5Nyt8C8ssinQ98Xw9Q==:
```

Note that since the covered components list is empty, this signature could be applied by an attacker to an unrelated HTTP message. In this example, the nonce parameter is included to prevent the same signature from being replayed more than once, but if an attacker intercepts the signature and prevents its delivery to the verifier, the attacker could apply this signature to another message. Therefore, use of an empty covered components set is discouraged. See Section 7.4 for more discussion.

Note that the RSA PSS algorithm in use here is non-deterministic, meaning a different signature value will be created every time the algorithm is run. The signature value provided here can be validated against the given keys, but newly-generated signature values are not expected to match the example. See Section 7.19.

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 base is:

NOTE: '\' line wrapping per RFC 8792

```
"@authority": example.com
"content-digest": sha-512=:WZDPaVn/7XgHaAy8pmojAkGWoRx2UFChF41A2svX\
  +TaPm+AbwAgBWnrIiYllu7BNNyealdVLvRwEmTHWXvJwew==:
"@signature-params": ("@authority" "content-digest")\
  ;created=1618884473;keyid="test-key-rsa-pss"
```

This results in the following Signature-Input and Signature headers being added to the message under the label sig-b22:

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

Signature-Input: sig-b22=("@authority" "content-digest")\
  ;created=1618884473;keyid="test-key-rsa-pss"
Signature: sig-b22=:Fee1uy9YGZq5UUwwYU6vz4dZNvfw3GYrFl1L6YlVIyUMuWs\
  wWDNSvql4dVtSeidYjYZUm7SBCENIb5KYy2ByoC3bI+7gydd2i4OAT5lyDtmeapnA\
  a8uP/b9xUpg+VSPElbBs6JWBIQsd+nMdHDe+ls/IwVMwXktC37SqsnbNyhNp6kcvc\
  WpevjzFcD2VqdZleUz4jN7P+W5A3wHiMGfIjIWn36KXNB+RKyrlGnIS8yaBBrom5r\
  cZWLrLbtg6VlrH1+/07RV+kgTh/l10h8qgpl9zQHu7mWbDKTq0tJ8K4ywcPoC4s2I\
  4rU88jzDKDGdTTQFZoTVZxZmuTM1FvHfzIw==:
```

Note that the RSA PSS algorithm in use here is non-deterministic, meaning a different signature value will be created every time the algorithm is run. The signature value provided here can be validated against the given keys, but newly-generated signature values are not expected to match the example. See Section 7.19.

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

This example covers all applicable message components in test-request (including the content type and length) plus many derived components, again using the rsa-pss-sha512 algorithm. Note that the Host header field is not covered because the @authority derived component is included instead.

The corresponding signature base is:

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

"date": Tue, 20 Apr 2021 02:07:55 GMT
"@method": POST
"@path": /foo
"@query": ?param=Value&Pet=dog
```

```
"@authority": example.com
"content-type": application/json
"content-digest": sha-512=:WZDPaVn/7XgHaAy8pmojAkGWoRx2UFChF41A2svX\
  +TaPm+AbwAgBWnrIiYllu7BNNyealdVLvRwEmTHWXvJwew==:
"content-length": 18
"@signature-params": ("date" "@method" "@path" "@query" \
  "@authority" "content-type" "content-digest" "content-length")\
  ;created=1618884473;keyid="test-key-rsa-pss"
```

This results in the following Signature-Input and Signature headers being added to the message under the label sig-b23:

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

Signature-Input: sig-b23=("date" "@method" "@path" "@query" \
  "@authority" "content-type" "content-digest" "content-length")\
  ;created=1618884473;keyid="test-key-rsa-pss"
Signature: sig-b23=:bbN8oArOxYoyylQQUU6QYwrTuaxLwjAC9fbY2F6SVWvh0yB\
  iMIRGOnMYwZ/5MR6fb0Kh1rIRASVxFkeGt683+qRpRRU5p2voTp768ZrCUb38K0fU\
  xN0O0iC59DzYx8DFll5GmydPxSmme9v6ULbMFkl+V5B1TP/yPViV7KsLNmvKiLJH1\
  pFkh/aYA2HXXZzNBXmIkoQoLd7YfW91kE9o/CCoC1xMy7JA1ipwvKvfrs65ldmlu9\
  bpG6A9BmzhuzF8Eim5f8ui9eH8LZH896+QIF61ka39VBrohr9iyMUJpvRX2Zbhl5Z\
  JzSRxpJyoEZAFL2FUo5fTIztsDZKEgM4cUA==:
```

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. See Section 7.20 for more discussion.

Note that the RSA PSS algorithm in use here is non-deterministic, meaning a different signature value will be created every time the algorithm is run. The signature value provided here can be validated against the given keys, but newly-generated signature values are not expected to match the example. See Section 7.19.

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 base is:

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

"@status": 200
"content-type": application/json
"content-digest": sha-512=:mEWXIS7MaLRuGgxOBdODa3xqM1XdEvxoYhvlCFJ4\
  1QJgJc4GTsPp29l5oGX69wWdXymyU0rjJuahq4l5aGgfLQ==:
"content-length": 23
"@signature-params": ("@status" "content-type" "content-digest" \
  "content-length");created=1618884473;keyid="test-key-ecc-p256"
```

This results in the following Signature-Input and Signature headers being added to the message under the label sig-b24:

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

Signature-Input: sig-b24=("@status" "content-type" \
  "content-digest" "content-length");created=1618884473\
  ;keyid="test-key-ecc-p256"
Signature: sig-b24=:wNmSUAhwb5LxtOtOpNa6W5xj067m5hFrj0XQ4fvpaCLx0NK\
  ocgPquLgyahnzDnDAUy5eCdlYUEkLIj+32oiasw==:
```

Note that the ECDSA algorithm in use here is non-deterministic, meaning a different signature value will be created every time the algorithm is run. The signature value provided here can be validated against the given keys, but newly-generated signature values are not expected to match the example. See Section 7.19.

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 base is:

NOTE: '\' line wrapping per RFC 8792

```
"date": Tue, 20 Apr 2021 02:07:55 GMT
"@authority": example.com
"content-type": application/json
"@signature-params": ("date" "@authority" "content-type")\
  ;created=1618884473;keyid="test-shared-secret"
```

This results in the following Signature-Input and Signature headers being added to the message under the label sig-b25:

NOTE: '\' line wrapping per RFC 8792

```
Signature-Input: sig-b25=("date" "@authority" "content-type")\
  ;created=1618884473;keyid="test-shared-secret"
Signature: sig-b25=:pxcQw6G3AjtMBQjwo8XzkZf/bws5LelbaMk5rGIGtE8=:
```

Before using symmetric signatures in practice, see the discussion of the security tradeoffs in Section 7.11.

B.2.6. Signing a Request using ed25519

This example covers portions of the test-request using the ed25519 algorithm and the key test-key-ed25519.

The corresponding signature base is:

NOTE: '\' line wrapping per RFC 8792

```
"date": Tue, 20 Apr 2021 02:07:55 GMT
"@method": POST
"@path": /foo
"@authority": example.com
"content-type": application/json
"content-length": 18
"@signature-params": ("date" "@method" "@path" "@authority" \
  "content-type" "content-length");created=1618884473\
  ;keyid="test-key-ed25519"
```

This results in the following Signature-Input and Signature headers being added to the message under the label sig-b26:

NOTE: '\' line wrapping per RFC 8792

```
Signature-Input: sig-b26=("date" "@method" "@path" "@authority" \
  "content-type" "content-length");created=1618884473\
  ;keyid="test-key-ed25519"
Signature: sig-b26=:wqcAqbmYJ2ji2glfAMaRy4gruYYnx2nEFN2HN6jrnDnQCK1\
  u02Gb04v9EDgwUPiu4A0w6vuQv5lIp5WPpBKRCw==:
```

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 [CLIENT-CERT], and then applies a signature to this field. By signing this header field, a reverse proxy can not only attest to its own validation of the initial request's TLS parameters 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 base 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=1618884473;keyid="test-key-ecc-p256"
```

This results in the following signature:

NOTE: '\' line wrapping per RFC 8792

```
xVMHVpawaAC/0SbHrKRs9i8I3eOs5RtTMGCWXm/9nvZzoHsIg6Mce9315T6xoklyy0y\
zhD9ah4JHRwMLOgmizw==
```

Which results in the following signed request sent from the proxy to the internal service with the proxy's signature under the label ttrp:

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=1618884473;keyid="test-key-ecc-p256"
Signature: ttrp=:xVMHVpawaAC/0SbHrKRs9i8I3eOs5RtTMGCWXm/9nvZzoHsIg6\
  Mce9315T6xoklyy0yzhD9ah4JHRwMLOgmizw==:

{"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 specification also includes contributions from the draft-oauth-signed-http-request internet draft and other similar efforts.

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 Böhlmark, Stéphane Bortzmeyer, Sarven Capadisli, Liam Dennehy, Stephen Farrell, Phillip Hallam-Baker, Tyler Ham, 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, Rich Salz, 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
  - -10
    - Removed "related response" and "@request-response" in favor of generic "req" parameter.
    - Editorial fixes to comply with HTTP extension style guidelines.
    - Add security consideration on message content.
  - -09
    - Explained key formats better.
    - Removed "host" and "date" from most examples.
    - Fixed query component generation.
    - Renamed "signature input" and "signature input string" to "signature base".
    - Added consideration for semantically equivalent field values.

- -08

   - Editorial fixes.

   - Changed "specialty component" to "derived component".

   - Expanded signature input generation and ABNF rules.

   - Added Ed25519 algorithm.

   - Clarified encoding of ECDSA signature.

   - Clarified use of non-deterministic algorithms.

- -07

   - Added security and privacy considerations.

   - Added pointers to algorithm values from definition sections.

   - Expanded IANA registry sections.

   - Clarified that the signing and verification algorithms take application requirements as inputs.

   - Defined "signature targets" of request, response, and related-response for specialty components.

- -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.

      o Added request-response binding.

   - -05

      o Remove list prefixes.

      o Clarify signature algorithm parameters.

      o Update and fix examples.

      o Add examples for ECC and HMAC.

   - -04

      o Moved signature component definitions up to intro.

      o Created formal function definitions for algorithms to fulfill.

      o Updated all examples.

      o Added nonce parameter field.

   - -03

      o Clarified signing and verification processes.

      o Updated algorithm and key selection method.

      o Clearly defined core algorithm set.

      o Defined JOSE signature mapping process.

      o Removed legacy signature methods.

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

      o 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

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

* draft-richanna-http-message-signatures

  - -00

    o Converted to xml2rfc v3 and reformatted to comply with RFC style guides.

    o Removed Signature auth-scheme definition and related content.

    o Removed conflicting normative requirements for use of algorithm parameter. Now MUST NOT be relied upon.

    o Removed Extensions appendix.

    o Rewrote abstract and introduction to explain context and need, and challenges inherent in signing HTTP messages.

    o Rewrote and heavily expanded algorithm definition, retaining normative requirements.

    o Added definitions for key terms, referenced RFC 7230 for HTTP terms.

    o Added examples for canonicalization and signature generation steps.

    o Rewrote Signature header definition, retaining normative requirements.

    o Added default values for algorithm and expires parameters.

    o Rewrote HTTP Signature Algorithms registry definition. Added change control policy and registry template. Removed suggested URI.

    o Added IANA HTTP Signature Parameter registry.

- Added additional normative and informative references.

- Added 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/