Internet Engineering Task Force (IETF)
Request for Comments: 9421
Category: Standards Track
ISSN: 2070-1721

A. Backman, Editor
Amazon
J. Richer, Editor
Bespoke Engineering
M. Sporny
Digital Bazaar
February 2024

HTTP Message Signatures

draft-ietf-httpbis-message-signatures-19

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.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841[1].

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc9421[2].

Copyright Notice

[1] https://www.rfc-editor.org/rfc/rfc7841.html#section-2
[2] https://www.rfc-editor.org/info/rfc9421
[3] https://trustee.ietf.org/license-info

Table of Contents

1 Introduction....5
1.1 Conventions and Terminology....5
1.2 Requirements....8
1.3 HTTP Message Transformations....8
1.4 Application of HTTP Message Signatures....9
2 HTTP Message Components....11
2.1 HTTP Fields....11
2.1.1 Strict Serialization of HTTP Structured Fields....13
2.1.2 Dictionary Structured Field Members....14
2.1.3 Binary-Wrapped HTTP Fields....14
2.1.4 Trailer Fields....15
2.2 Derived Components....16
2.2.1 Method....17
2.2.2 Target URI....17
2.2.3 Authority....18
2.2.4 Scheme....18
2.2.5 Request Target....18
2.2.6 Path....19
2.2.7 Query....20
2.2.8 Query Parameters....20
2.2.9 Status Code....22
2.3 Signature Parameters....22
2.4 Signing Request Components in a Response Message....23
2.5 Creating the Signature Base....27
3 HTTP Message Signatures....30
3.1 Creating a Signature....30
3.2 Verifying a Signature....30
3.2.1 Enforcing Application Requirements....32
3.3 Signature Algorithms....32
3.3.1 RSASSA-PSS Using SHA-512....33
3.3.2 RSASSA-PKCS1-v1_5 Using SHA-256....33
3.3.3 HMAC Using SHA-256....34
3.3.4 ECDSA Using Curve P-256 DSS and SHA-256....34
3.3.5 ECDSA Using Curve P-384 DSS and SHA-384....34
3.3.6 EdDSA Using Curve edwards25519....35
3.3.7 JSON Web Signature (JWS) Algorithms....35
4 Including a Message Signature in a Message....36
4.1 The Signature-Input HTTP Field....36
4.2 The Signature HTTP Field....36
4.3 Multiple Signatures....37
5 Requesting Signatures....40

5.1 The Accept-Signature Field....40
5.2 Processing an Accept-Signature....41

6 IANA Considerations....42

6.1 HTTP Field Name Registration....42
6.2 HTTP Signature Algorithms Registry....42
6.2.1 Registration Template....43
6.2.2 Initial Contents....43
6.3 HTTP Signature Metadata Parameters Registry....43
6.3.1 Registration Template....44
6.3.2 Initial Contents....44
6.4 HTTP Signature Derived Component Names Registry....44
6.4.1 Registration Template....45
6.4.2 Initial Contents....45
6.5 HTTP Signature Component Parameters Registry....46
6.5.1 Registration Template....46
6.5.2 Initial Contents....46

7 Security Considerations....48

7.1 General Considerations....48
7.1.1 Skipping Signature Verification....48
7.1.2 Use of TLS....48
7.2 Message Processing and Selection....48
7.2.1 Insufficient Coverage....48
7.2.2 Signature Replay....49
7.2.3 Choosing Message Components....49
7.2.4 Choosing Signature Parameters and Derived Components over HTTP Fields....49
7.2.5 Signature Labels....50
7.2.6 Multiple Signature Confusion....50
7.2.7 Collision of Application-Specific Signature Tag....50
7.2.8 Message Content....50
7.3 Cryptographic Considerations....51
7.3.1 Cryptography and Signature Collision....51
7.3.2 Key Theft....52
7.3.3 Symmetric Cryptography....52
7.3.4 Key Specification Mixup....52
7.3.5 Non-deterministic Signature Primitives....52
7.3.6 Key and Algorithm Specification Downgrades....52
7.3.7 Signing Signature Values....53
7.4 Matching Signature Parameters to the Target Message....53
7.4.1 Modification of Required Message Parameters....54
7.4.2 Matching Values of Covered Components to Values in the Target Message....54
7.4.3 Message Component Source and Context....54
7.4.4 Multiple Message Component Contexts....54
7.5 HTTP Processing....55
7.5.1 Processing Invalid HTTP Field Names as Derived Component Names....55
7.5.2 Semantically Equivalent Field Values....55
7.5.3 Parsing Structured Field Values....56
7.5.4 HTTP Versions and Component Ambiguity....56

7.5.5 Canonicalization Attacks 56
7.5.6 Non-List Field Values 57
7.5.7 Padding Attacks with Multiple Field Values 57
7.5.8 Ambiguous Handling of Query Elements 58

8 Privacy Considerations 59

8.1 Identification through Keys 59

8.2 Signatures do not provide confidentiality 59

8.3 Oracles 59

8.4 Required Content 59

9 References 60

9.1 Normative References 60

9.2 Informative References 61

Appendix A Detecting HTTP Message Signatures 63

Appendix B Examples 64

B.1 Example Keys 64
B.1.1 Example RSA Key 64
B.1.2 Example RSA-PSS Key 66
B.1.3 Example ECC P-256 Test Key 68
B.1.4 Example Ed25519 Test Key 69
B.1.5 Example Shared Secret 69

B.2 Test Cases 69
B.2.1 Minimal Signature Using rsa-pss-sha512 70
B.2.2 Selective Covered Components Using rsa-pss-sha512 71
B.2.3 Full Coverage Using rsa-pss-sha512 71
B.2.4 Signing a Response Using ecdsa-p256-sha256 72
B.2.5 Signing a Request Using hmac-sha256 73
B.2.6 Signing a Request Using ed25519 73

B.3 TLS-Terminating Proxies 74

B.4 HTTP Message Transformations 76

Authors' Addresses 80

1. Introduction

Message integrity and authenticity are 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 [TLS]. However, TLS only guarantees these properties over a single TLS connection, and the path between the 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 (such as the presentation of client certificates from a browser) or to use features necessary to provide message authenticity. Furthermore, some applications require the binding of a higher-level application-specific 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.

Additionally, many applications need to be able to generate and verify signatures despite incomplete knowledge of the HTTP message as seen on the wire, due to the use of libraries, proxies, or application frameworks that alter or hide portions of the message from the application at the time of signing or verification. These applications need a means to protect the parts of the message that are most relevant to the application without having to violate layering and abstraction.

Finally, object-based signature mechanisms such as JSON Web Signature [JWS] require the intact conveyance of the exact information that was signed. When applying such technologies to an HTTP message, elements of the HTTP message need to be duplicated in the object payload either directly or through the inclusion of a hash. This practice introduces complexity, since the repeated information needs to be carefully checked for consistency when the signature is verified.

This document defines a mechanism for providing end-to-end integrity and authenticity for components of an HTTP message by using a detached signature on HTTP messages. 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 many of the 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 (Section 2).
- Algorithms for generating and verifying signatures over HTTP message components using this signature base through the application of cryptographic primitives (Section 3).
- 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 (Section 4).

This document also provides a mechanism for negotiating the use of signatures in one or more subsequent messages via the "Accept-Signature" field (Section 5). This optional negotiation mechanism can be used along with opportunistic or application-driven message signatures by either party.

The mechanisms defined in this document are important tools that can be used to build an overall security mechanism for an application. This toolkit provides some powerful capabilities but is not sufficient in creating an overall security story. In particular, the requirements listed in Section 1.4 and the security considerations discussed in Section 7 are of high importance to all implementors of this specification. For example, this specification does not define a means to directly cover HTTP message content (defined in Section 6.4 of [HTTP]); rather, it relies on the Digest specification [DIGEST] to provide a hash of the message content, as discussed in Section 7.2.8.

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

The terms "HTTP message", "HTTP request", "HTTP response", "target URI", "gateway", "header field", "intermediary", "request target", "trailer field", "sender", "method", and "recipient" are used as defined in [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 signature base. The qualified term "digital signature" refers specifically to the output of an asymmetric cryptographic signing operation.

This document uses the following terminology from Section 3 of [STRUCTURED-FIELDS] to specify data types: List, Inner List, Dictionary, Item, String, Integer, Byte Sequence, and Boolean.

This document defines several string constructions using ABNF [ABNF] and uses the following ABNF rules: `VCHAR`, `SP`, `DQUOTE`, and `LF`. This document uses the following ABNF rules from [STRUCTURED-FIELDS]: `sf-string`, `inner-list`, and `parameters`. This document uses the following ABNF rules from [HTTP] and [HTTP/1.1]: `field-content`, `obs-fold`, and `obs-text`.

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.

Derived Component:
: An HTTP message component derived from the HTTP message through the use of a specified algorithm or process. See Section 2.2.

HTTP Message Component Name:
: A String that identifies an HTTP message component's source, such as a field name or derived component name.

HTTP Message Component Identifier:
: The combination of an HTTP message component name and any parameters. This combination uniquely identifies a specific

HTTP message component with 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 generated by the signer and verifier using the covered components set and the HTTP message. The signature base is processed by the cryptographic algorithm to produce or verify the HTTP message signature.

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.

Target Message: The HTTP message to which an HTTP message signature is applied.

Signature Context: The data source from which the HTTP message component values are drawn. The context includes the target message and any additional information the signer or verifier might have, such as the full target URI of a

request or the related request message for a response.

The term "UNIX timestamp" refers to what Section 4.16 of [POSIX.1] calls "seconds since the Epoch".

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

1.2. Requirements

HTTP permits, and sometimes requires, intermediaries to transform messages in a variety of ways. This can 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 integrity protections 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 have to 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 a particular header field) is relevant. Thus, a general-purpose solution needs to 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 HTTP Client (HttpClient) library[1]). These applications need to be able to generate and verify signatures despite incomplete knowledge of the HTTP message.

1.3. 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 could occur under HTTP, provided as context:

- Reordering of fields with different field names (Section 5.3 of [HTTP]).
- Combination of fields with the same field name (Section 5.2 of [HTTP]).
- Removal of fields listed in the Connection header field (Section 7.6.1 of [HTTP]).
- Addition of fields that indicate control options (Section 7.6.1 of [HTTP]).
- Addition or removal of a transfer coding (Section 7.7 of [HTTP]).
- Addition of fields such as Via (Section 7.6.3 of [HTTP]) and Forwarded (Section 4 of [RFC7239]).
- Conversion between different versions of HTTP (e.g., HTTP/1.x to HTTP/2, or vice versa).
- Changes in case (e.g., "Origin" to "origin") of any case-insensitive components such as field names, request URI scheme, or host.
- Changes to the request target and authority that, when applied together, do not result in a change to the message's target URI, as defined in Section 7.1 of [HTTP].

Additionally, there are some transformations that are either deprecated or otherwise not allowed but that could still occur in the wild. These transformations can still be handled without breaking the signature; they include such actions as:

- Use, addition, or removal of leading or trailing whitespace in a field value.
- Use, addition, or removal of `obs-fold` in field values (Section 5.2 of [HTTP/1.1]).

[1] https://openjdk.java.net/groups/net/httpclient/intro.html

We can identify these types of transformations as transformations that should not prevent signature verification, even when performed on message components covered by the signature. Additionally, all changes to components not covered by the signature should not prevent signature verification.

Some examples of these kinds of transformations, and the effect they have on the message signature, are found in Appendix B.4.

Other transformations, such as parsing and reserializing the field values of a covered component or changing the value of a derived component, can cause a signature to no longer validate against a target message. Applications of this specification need to take care to ensure that the transformations expected by the application are adequately handled by the choice of covered components.

1.4. Application of HTTP Message Signatures

HTTP message signatures are designed to be a general-purpose tool 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, at a minimum, all of the following items:

- The set of component identifiers (Section 2) and signature parameters (Section 2.3) that are expected and required to be included in the covered components list. For example, an authorization protocol could mandate that the Authorization field be covered to protect the authorization credentials and mandate that the signature parameters contain a `created` parameter (Section 2.3), while an API expecting semantically relevant HTTP message content could require the Content-Digest field defined in [DIGEST] to be present and covered as well as mandate a value for the `tag` parameter (Section 2.3) that is specific to the API being protected.
- The expected Structured Field types [STRUCTURED-FIELDS] of any required or expected covered component fields or parameters.
- A means of retrieving the key material used to verify the signature. An application will usually use the `keyid` parameter of the signature parameters (Section 2.3) and define rules for resolving a key from there, though the appropriate key could be known from other means such as preregistration of a signer's key.
- The set of allowable signature algorithms to be used by signers and accepted by verifiers.
- A means of determining that the signature algorithm used to verify the signature is appropriate for the key material and context of the message. For example, the process could use the `alg` parameter of the signature parameters (Section 2.3) to state the algorithm explicitly, derive the algorithm from the key material, or use some preconfigured algorithm agreed upon by the signer and verifier.
- A means of determining that a given key and algorithm used for a signature are appropriate for the context of the message. 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.
- A means of determining the context for derivation of message components from an HTTP message and its application context. While this is normally the target HTTP message itself, the context could include additional information known to the application through configuration, such as an external hostname.
- If binding between a request and response is needed using the mechanism provided in Section 2.4, all elements of the request message and the response message that would be required to provide properties of such a binding.
- The error messages and codes that are returned from the verifier to the signer when the signature is invalid, the key material is inappropriate, the validity time window is out of specification, a component value cannot be calculated, or any other errors occur during the signature verification process. For example, if a signature is being used as an authentication mechanism, an HTTP status code of 401 (Unauthorized) or 403 (Forbidden) could be appropriate. If the response is from an HTTP API, a response with an HTTP status code such as 400 (Bad Request) could include more details [RFC7807] [RFC9457], such as an indicator that the wrong key material was used.

When choosing these parameters, an application of HTTP message signatures has to ensure that the verifier will have access to all required information needed to recreate the signature base. For example, a server behind a reverse proxy would need to know the original request URI to make use of the derived component `@target-uri`, even though the apparent target URI would be changed by the reverse proxy (see also Section 7.4.3). Additionally, an application using signatures in responses would need to ensure that clients receiving signed responses have access to all the signed portions of the message, including any portions of the request that were signed by the server using the `req` ("request-response") parameter (Section 2.4).

Details regarding this kind of profiling are within the purview of the application and outside the scope of this specification; however, some additional considerations are discussed in Section 7. In particular, when choosing the required set of component identifiers, care has to be taken to make sure that the coverage is sufficient for the application, as discussed in 7.2.1 and 7.2.8. This specification defines only part of a full security system for an application. When building a complete security system based on this tool, it is important to perform a security analysis of the entire system, of which HTTP message signatures is a part. Historical systems, such as AWS Signature Version 4 [AWS-SIGv4], can provide inspiration and examples of how to apply similar mechanisms to an application, though review of such historical systems does not negate the need for a security analysis of an application of HTTP message signatures.

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 signature context for deriving these values MUST be accessible to both the signer and the verifier of the message. The context MUST be the same across all components in a given signature. For example, it would be an error to use the raw query string for the `@query` derived component but combined query and form parameters for the `@query-param` derived component. For more considerations regarding the message component context, see Section 7.4.3.

A component identifier is composed of a component name and any parameters associated with that name. Each component name is either an HTTP field name (Section 2.1) or a registered derived component name (Section 2.2). The possible parameters for a component identifier are dependent on the component identifier. The "HTTP Signature Component Parameters" registry, which catalogs all possible parameters, is defined in Section 6.5.

Within a single list of covered components, each component identifier MUST occur only once. One component identifier is distinct from another if the component name differs or if any of the parameters differ for the same component name. Multiple component identifiers having the same component name MAY be included if they have parameters that make them distinct, such as `"foo";bar` and `"foo";baz`. The order of parameters MUST be preserved when processing a component identifier (such as when parsing during verification), but the order of parameters is not significant when comparing two component identifiers for equality checks. That is to say, `"foo";bar;baz` cannot be in the same message as `"foo";baz;bar`, since these two component identifiers are equivalent, but a system processing one form is not allowed to transform it into the other form.

The component value associated with a component identifier is defined by the identifier itself. Component values MUST NOT contain newline (`\n`) characters. Some HTTP message components can undergo transformations that change the bitwise value without altering the meaning of the component's value (for example, when combining field values). Message component values therefore need to be canonicalized before they are 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.

The following sections define component identifier names, their parameters, their associated values, and the canonicalization rules for their values. The method for combining message components into the signature base is defined in Section 2.5.

2.1. HTTP Fields

The component name for an HTTP field is the lowercased form of its field name as defined in Section 5.1 of [HTTP]. 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.

The component value for an HTTP field is the field value for the named field as defined in Section 5.5 of [HTTP]. The field value MUST be taken from the named header field of the target message unless this behavior is overridden by additional parameters and rules, such as the `req` and `tr` flags, below. For most fields, the field value is an ASCII string as recommended by [HTTP], and the component value is exactly that string. Other encodings could exist in some implementations, and all non-ASCII field values MUST be encoded to ASCII before being added to the signature base. The `bs` parameter, as described in Section 2.1.3, provides a method for wrapping such problematic field values.

Unless overridden by additional parameters and rules, HTTP field values MUST be combined into a single value as defined in Section 5.2 of [HTTP] to create the component value. Specifically, HTTP fields sent as multiple fields MUST be combined by concatenating the values using a single comma and a single space as a separator ("," + " "). Note that intermediaries are allowed to combine values of HTTP fields with any amount

of whitespace between the commas, and if this behavior is not accounted for by the verifier, the signature can fail, since the signer and verifier will see a different component value in their respective signature bases. For robustness, it is RECOMMENDED that signed messages include only a single instance of any field covered under the signature, particularly with the value for any list-based fields serialized using the algorithm below. This approach increases the chances of the field value remaining untouched through intermediaries. Where that approach is not possible and multiple instances of a field need to be sent separately, it is RECOMMENDED that signers and verifiers process any list-based fields taking all individual field values and combining them based on the strict algorithm below, to counter possible intermediary behavior. When the field in question is a Structured Field of type List or Dictionary, this effect can be accomplished more directly by requiring the strict Structured Field serialization of the field value, as described in Section 2.1.1.

Note that some HTTP fields, such as Set-Cookie [COOKIE], do not follow a syntax that allows for the combination of field values in this manner (such that the combined output is unambiguous from multiple inputs). Even though the component value is never parsed by the message signature process and is used only as part of the signature base (Section 2.5), caution needs to be taken when including such fields in signatures, since the combined value could be ambiguous. The `bs` parameter, as described in Section 2.1.3, provides a method for wrapping such problematic fields. See Section 7.5.6 for more discussion regarding this issue.

If the correctly combined value is not directly available for a given field by an implementation, the following algorithm will produce canonicalized results for list-based fields:

1. Create an ordered list of the field values of each instance of the field in the message, in the order 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 would 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 [HTTP/1.1]. Note that this behavior is specific to HTTP/1.1 and does not apply to other versions of the HTTP specification, which do not allow internal line folding.
4. Concatenate the list of values with a single comma (",") and a single space (" ") between each item.

The resulting string is the component value for the field.

Note that some HTTP fields have values with multiple valid serializations that have equivalent semantics, such as allowing case-insensitive values that intermediaries could change. 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 7.5.2.

The following are non-normative examples of component 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 the component values for these example header fields, presented using the signature base format defined in Section 2.5:

```
"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)
```

Empty HTTP fields can also be signed when present in a message. The canonicalized value is the empty string. This means that the following empty header field, with (SP) indicating a single trailing space character before the empty field value:

```
X-Empty-Header:(SP)
```

is serialized by the signature base generation algorithm (Section 2.5) with an empty string value following the colon and space added after the component identifier.

```
"x-empty-header":(SP)
```

Any HTTP field component identifiers MAY have the following parameters in specific circumstances, each described in detail in their own sections:

`sf` A Boolean flag indicating that the component value is serialized 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).

`bs` A Boolean flag indicating that individual field values are encoded using Byte Sequence data structures before being combined into the component value (Section 2.1.3).

`req` A Boolean flag for signed responses indicating that the component value is derived from the request that triggered this response message and not from the response message directly. Note that this parameter can also be applied to any derived component identifiers that target the request (Section 2.4).

`tr` A Boolean flag indicating that the field value is taken from the trailers of the message as defined in Section 6.5 of [HTTP]. If this flag is absent, the field value is taken from the header fields of the message as defined in Section 6.3 of [HTTP] (Section 2.1.4).

Multiple parameters MAY be specified together, though some combinations are redundant or incompatible. For example, the `sf` parameter's functionality is already covered when the `key` parameter is used on a Dictionary item, since `key` requires strict serialization of the value. The `bs` parameter, which requires the raw bytes of the field values from the message, is not compatible with the use of the `sf` or `key` parameters, which require the parsed data structures of the field values after combination.

Additional parameters can be defined in the "HTTP Signature Component Parameters" registry established in Section 6.5.

2.1.1. Strict Serialization of HTTP Structured Fields

If the value of an HTTP field is known by the application to be a Structured Field type (as defined in [STRUCTURED-FIELDS] or its extensions or updates) and the expected type of the Structured Field is known, the signer MAY include the `sf` parameter in the component identifier. If this parameter is included with a component identifier, the HTTP field value MUST be serialized using the formal serialization rules specified in Section 4 of [STRUCTURED-FIELDS] (or the applicable formal serialization section of its extensions or updates) 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.

If multiple field values occur within a message, these values MUST be combined into a single List or Dictionary structure before serialization.

If the application does not know the type of the field or does not know how to serialize the type of the field, the use of this flag will produce an error. As a consequence, the signer can only reliably sign fields using this flag when the verifier's system knows the type as well.

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 signature base without parameters, its value 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 reserialized 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; see 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.

If multiple field values occur within a message, these values MUST be combined into a single Dictionary structure before serialization.

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, not including the Dictionary key itself. Specifically, the value is serialized as an Item or Inner List (the two possible values of a Dictionary member), with all parameters and possible subfields serialized using the strict serialization rules defined in Section 4 of [STRUCTURED-FIELDS] (or the applicable section of its extensions or updates).

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.

The 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 2.5:

```
"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 reserialized according to the strict `member_value` algorithm, and the value for `key="d"` has been serialized as a Boolean value.

2.1.3. Binary-Wrapped HTTP Fields

If the value of the HTTP field in question is known by the application to cause problems with serialization, particularly with the combination of multiple values into a single line as discussed in Section 7.5.6, the signer SHOULD include the `bs` parameter in a component identifier to indicate that the values of the field need to be wrapped as binary structures before being combined.

If this parameter is included with a component identifier, the component value MUST be calculated using the following algorithm:

1. Let the input be the ordered set of values for a field, in the order they appear in the message.
2. Create an empty List for accumulating processed field values.
3. For each field value in the set:
4. The intermediate result is a List of Byte Sequence values.
5. Follow the strict serialization of a List as described in Section 4.1.1 of [STRUCTURED-FIELDS], and return this output.

For example, the following field with internal commas prevents the distinct field values from being safely combined:

```
Example-Header: value, with, lots
Example-Header: of, commas
```

In our example, the same field can be sent with a semantically different single value:

```
Example-Header: value, with, lots, of, commas
```

Both of these versions are treated differently by the application. However, if included in the signature base without parameters, the component value would be the same in both cases:

```
"example-header": value, with, lots, of, commas
```

However, if the `bs` parameter is added, the two separate instances are encoded and serialized as follows:

```
"example-header";bs: :dmFsdWUsIHdpdGgsIGxvdHM=:, :b2YsIGNvbW1hcw==:
```

For the single-instance field above, the encoding with the `bs` parameter is:

```
"example-header";bs: :dmFsdWUsIHdpdGgsIGxvdHMsIG9mLCBjb21tYXM=:
```

This component value is distinct from the multiple-instance field above, preventing a collision that could potentially be exploited.

2.1.4. Trailer Fields

If the signer wants to include a trailer field in the signature, the signer MUST include the `tr` Boolean parameter to indicate that the value MUST be taken from the trailer fields and not from the header fields.

For example, given the following message:

```
HTTP/1.1 200 OK
Content-Type: text/plain
Transfer-Encoding: chunked
Trailer: Expires

4
HTTP
7
Message
a
Signatures
0
Expires: Wed, 9 Nov 2022 07:28:00 GMT
```

The signer decides to add both the Trailer header field and the Expires trailer field to the signature base, along with the status code derived component:

```
"@status": 200
"trailer": Expires
"expires";tr: Wed, 9 Nov 2022 07:28:00 GMT
```

If a field is available as both a header and a trailer in a message, both values MAY be signed, but the values MUST be signed separately. The values of header fields and trailer fields of the same name MUST NOT be combined for purposes of the signature.

Since trailer fields could be merged into the header fields or dropped entirely by intermediaries as per Section 6.5.1 of [HTTP], it is NOT RECOMMENDED to include trailers in the signature unless the signer knows that the verifier will have access to the values of the trailers as sent.

2.2. Derived Components

In addition to HTTP fields, there are a number of different components that can be derived from the control data, signature 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, message targets, 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 7.5.1.

This specification defines the following derived components:

@method	The method used for a request (Section 2.2.1).
@target-uri	The full target URI for a request (Section 2.2.2).
@authority	The authority of the target URI for a request (Section 2.2.3).
@scheme	The scheme of the target URI for a request (Section 2.2.4).
@request-target	The request target (Section 2.2.5).
@path	The absolute path portion of the target URI for a request (Section 2.2.6).
@query	The query portion of the target URI for a request (Section 2.2.7).
@query-param	A parsed and encoded query parameter of the target URI for a request (Section 2.2.8).
@status	The status code for a response (Section 2.2.9).

Additional derived component names are defined in the "HTTP Signature Derived Component Names" registry (Section 6.4).

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 or verifier. In particular, when signing a response, the signer can include any derived components from the originating request by using the `req` parameter (Section 2.4).

request: Values derived from, and results applied to, an HTTP request message as described in Section 3.4 of [HTTP]. If the target message of the signature is a response, derived components that target request messages can be included by using the `req` parameter as defined in Section 2.4.

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

request, response: Values derived from, and results applied to, either a request message or a response message.

A derived component definition MUST define all target message types to which it can be applied.

Derived component values MUST be limited to printable characters and spaces and MUST NOT contain any newline characters. Derived component values MUST NOT start or end with whitespace characters.

2.2.1. 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. While conventionally standardized method names are uppercase [ASCII], no transformation to the input method value's case is performed.

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
```

2.2.2. Target URI

The `@target-uri` derived component refers to the target URI of a request message. The component value is the target URI of the request ([HTTP], Section 7.1), assembled from all available URI components, including the authority.

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
```

2.2.3. 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 field, 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 provided in [HTTP], Section 4.2.3. Namely, the hostname is normalized to lowercase, and the default port is omitted.

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 field directly. See Section 7.2.4.

2.2.4. 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 lowercase 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.

For example, the following request message sent 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
```

2.2.5. 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.

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 a 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": *
```

2.2.6. 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 [URI], with no query component and no trailing question mark (`?`) character. The value is normalized according to the rules provided in [HTTP], Section 4.2.3. Namely, an empty path string is normalized as a single slash (`/`) character. Path components are represented by their

values before decoding any percent-encoded octets, as described in the simple string comparison rules provided in Section 6.2.1 of [URI].

For example, the following request message:

```
GET /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
```

2.2.7. 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 [URI], including the leading `?` character. The value is read using the simple string comparison rules provided in Section 6.2.1 of [URI]. Namely, percent-encoded octets are not decoded.

For example, the following request message:

```
GET /path?param=value&foo=bar&baz=bat%2Dman HTTP/1.1
Host: www.example.com
```

would result in the following `@query` component value:

```
?param=value&foo=bar&baz=bat%2Dman
```

and the following signature base line:

```
"@query": ?param=value&foo=bar&baz=bat%2Dman
```

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
```

Just like including an empty path component, the signer can include an empty query component to indicate that this component is not used in the message. If the query string is absent from the request message, the component value is the leading `?` character alone:

```
?
```

resulting in the following signature base line:

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

2.2.8. Query Parameters

If the query portion of a request target URI uses HTML form parameters in the format defined in 5 of [HTMLURL], the `@query-param` derived component allows addressing of these individual query parameters. The query parameters MUST be parsed according to 5.1 of [HTMLURL], resulting in a list of (`nameString`, `valueString`) tuples. The REQUIRED `name` parameter of each component identifier contains the encoded `nameString` of a single query parameter as a String value. The component value of a single named parameter is the encoded `valueString` of that single query parameter. Several different named query parameters MAY be included in the covered components. Single named parameters MAY occur in any order in the covered components, regardless of the order they occur in the query string.

The value of the `name` parameter and the component value of a single named parameter are calculated via the following process:

1. Parse the `nameString` or `valueString` of the named query parameter defined by 5.1 of [HTMLURL]; this is the value after percent-encoded octets are decoded.
2. Encode the `nameString` or `valueString` using the "percent-encode after encoding" process defined by 5.2 of [HTMLURL]; this results in an ASCII string [ASCII].
3. Output the ASCII string.

Note that the component value does not include any leading question mark (`?`) characters, equals sign (`=`) characters, or separating ampersand (`&`) characters. Named query parameters with an empty `valueString` have an empty string as the component value. Note that due to inconsistencies in implementations, some query parameter parsing libraries drop such empty values.

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:

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

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

baz: `batman`

qux: an empty string

param: `value`

and the following signature base lines, with (SP) indicating a single trailing space character before the empty component value:

```
"@query-param";name="baz": batman
"@query-param";name="qux":(SP)
"@query-param";name="param": value
```

This derived component has some limitations. Specifically, the algorithms provided in 5 of [HTMLURL] only support query parameters using percent-escaped UTF-8 encoding. Other encodings are not supported. Additionally, multiple instances of a named parameter are not reliably supported in the wild. If a parameter name occurs multiple times in a request, the named query parameter MUST NOT be included. 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 2.2.7.

The encoding process allows query parameters that include newlines or other problematic characters in their values, or with alternative encodings such as using the plus (+) character to represent spaces. For the query parameters in this message:

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

GET /parameters?var=this%20is%20a%20big%0Amultiline%20value&\
  bar=with+plus+whitespace&fa%C3%A7ade%22%3A%20=something HTTP/1.1
Host: www.example.com
Date: Tue, 20 Apr 2021 02:07:56 GMT
```

The resulting values are encoded as follows:

```
"@query-param";name="var": this%20is%20a%20big%0Amultiline%20value
"@query-param";name="bar": with%20plus%20whitespace
"@query-param";name="fa%C3%A7ade%22%3A%20": something
```

If the encoding were not applied, the resultant values would be:

```
"@query-param";name="var": this is a big
multiline value
"@query-param";name="bar": with plus whitespace
"@query-param";name="façade\": ": something
```

This base string contains characters that violate the constraints on component names and values and is therefore invalid.

2.2.9. 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.

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. Signature Parameters

HTTP message signatures have metadata properties that provide information regarding the signature's generation and verification, consisting of the ordered set of covered components and the ordered set of parameters, where the parameters include a timestamp of signature creation, identifiers for verification key material, and other utilities. This metadata is represented by a special message component in the signature base for signature parameters; this special message component is treated slightly differently from other message components. Specifically, the signature parameters message component is REQUIRED as the last line of the signature base (Section 2.5), and the component identifier MUST NOT be enumerated within the set of covered components for any signature, including itself.

The signature parameters component name is `@signature-params`.

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

`created`: Creation time as a UNIX timestamp value of type Integer. Sub-second precision is not supported. The inclusion of this parameter is RECOMMENDED.

`expires`: Expiration time as a UNIX timestamp value of type Integer. 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 Signature Algorithms" registry, as a String value.

`keyid`: The identifier for the key material as a String value.

`tag`: An application-specific tag for the signature as a String value. This value is used by applications to help identify signatures relevant for specific applications or protocols.

Additional parameters can be defined in the "HTTP Signature Metadata Parameters" registry (Section 6.3). Note that the parameters are not in any general order, but once an ordering is chosen for a given set of parameters, it cannot be changed without altering the signature parameters value.

The signature parameters component value is serialized as a parameterized Inner List using the rules provided 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 that used in creating the signature base (Section 2.5).
3. Serialize the component identifiers of the covered components, including all parameters, as an ordered Inner List of String values according to Section 4.1.1.1 of [STRUCTURED-FIELDS]; then, append this to the output. Note that the component identifiers can include their own parameters, and these parameters are ordered sets. Once an order is chosen for a component's parameters, the order cannot be changed.
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 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 the serialized component value for the parameters of an example message 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 a given signature parameters entry.

2.4. Signing Request Components in a Response Message

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` A Boolean flag indicating 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 that target the request, 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 instead of the target response message to which the signature is applied.

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 message and the 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 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-Digest: sha-512=:WZDPaVn/7XgHaAy8pmojAkGWoRx2UFChF41A2svX+T\
  aPm+AbwAgBWnrIiYllu7BNNyealdVLvRwEmTHWXvJwew==:
Content-Type: application/json
Content-Length: 18

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

This would result in the following unsigned response message:

```
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
Content-Digest: sha-512=:0Y6iCBzGg5rZtoXS95Ijz03mslf6KAMCloESHObfwn\
  HJDbkkWWQz6PhhU9kxsTbARtY2PTBOzq24uJFpHsMuAg==:

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

The server signs the response with its own key, including the `@status` code and several header fields in the covered components. While this covers a reasonable amount of the response for this application, the server additionally includes several components derived from the original request message that triggered this response. In this example, the server includes the method, authority, path, and content digest from the request in the covered components of the response. The Content-Digest for both the request and the response is included under the response signature. For the application in this example, the query is deemed not to be relevant to the response and is therefore not covered. Other applications would make different decisions based on application needs, as discussed in Section 1.4.

The signature base for this example is:

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

"@status": 503
"content-digest": sha-512=:0Y6iCBzGg5rZtoXS95Ijz03mslf6KAMCloESHObf\
  wnHJDbkkWWQz6PhhU9kxsTbARtY2PTBOzq24uJFpHsMuAg==:
"content-type": application/json
"@authority";req: example.com
"@method";req: POST
"@path";req: /foo
"content-digest";req: sha-512=:WZDPaVn/7XgHaAy8pmojAkGWoRx2UFChF41A\
  2svX+TaPm+AbwAgBWnrIiYllu7BNNyealdVLvRwEmTHWXvJwew==:
"@signature-params": ("@status" "content-digest" "content-type" \
  "@authority";req "@method";req "@path";req "content-digest";req)\
  ;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
Content-Digest: sha-512=:0Y6iCBzGg5rZtoXS95Ijz03mslf6KAMCloESHObfwn\
  HJDbkkWWQz6PhhU9kxsTbARtY2PTBOzq24uJFpHsMuAg==:
Signature-Input: reqres=("@status" "content-digest" "content-type" \
  "@authority";req "@method";req "@path";req "content-digest";req)\
  ;created=1618884479;keyid="test-key-ecc-p256"
Signature: reqres=:dMT/A/76ehrdBTD/2Xx8QuKV6FoyzEP/I9hdzKN8LQJLNgzU\
  4W767HK05rx1i8meNQQgQPgQp8wq2ive3tV5Ag==:

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

Note that the ECDSA signature algorithm in use here is non-deterministic, meaning that 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 7.3.5.

Since the component values from the request are not repeated in the response message, the requester MUST keep the original message component values around long enough to validate the signature of the response that uses this component identifier parameter. In most cases, this means the requester needs to keep the original request message around, since the signer could choose to include any portions of the request in its response, according to the needs of the application. Since it is possible for an intermediary to alter a request message before it is processed by the server, applications need to take care not to sign such altered values, as the client would not be able to validate the resulting signature.

It is also possible for a server to create a signed response in response to a signed request. For this example of a 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-Digest: sha-512=:WZDPaVn/7XgHaAy8pmojAkGWoRx2UFChF41A2svX+T\
  aPm+AbwAgBWnrIiYllu7BNNyealdVLvRwEmTHWXvJwew==:
Content-Type: application/json
Content-Length: 18
Signature-Input: sig1=("@method" "@authority" "@path" "@query" \
  "content-digest" "content-type" "content-length")\
  ;created=1618884475;keyid="test-key-rsa-pss"
Signature: sig1=:e8UJ5wMiRaonlth5ERtE8GIiEH7Akcr493nQ07VPNo6y3qvjdK\
  t0fo8VHO8xXDjmtYoatGYBGJVlMfIp06eVMEyNW2I4vN7XDAz7m5v1108vGzaDljr\
  d0H8+SJ28g7bzn6h2xeL/8q+qUwahWA/JmC8aOC9iVnwbOKCc0WSrLgWQwTY6VLp4\
  2Qt7jjhYT5W7/wCvfK9A1VmHH1lJXsV873Z6hpxesd50PSmO+xaNeYvDLvVdZlhtw\
  5PCtUYzKjHqwmaQ6DEuM8udRjYsoNqp2xZKcuCO1nKc0V3RjpqMZLuuyVbHDAbCzr\
  0pg2d2VM/OC33JAU7meEjjaNz+d7LWPg==:

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

The server could choose to sign portions of this response, including several portions of the request, resulting in this signature base:

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

"@status": 503
"content-digest": sha-512=:0Y6iCBzGg5rZtoXS95Ijz03mslf6KAMCloESHObf\
  wnHJDbkkWWQz6PhhU9kxsTbARtY2PTBOzq24uJFpHsMuAg==:
"content-type": application/json
"@authority";req: example.com
"@method";req: POST
"@path";req: /foo
"@query";req: ?param=Value&Pet=dog
"content-digest";req: sha-512=:WZDPaVn/7XgHaAy8pmojAkGWoRx2UFChF41A\
  2svX+TaPm+AbwAgBWnrIiYllu7BNNyealdVLvRwEmTHWXvJwew==:
"content-type";req: application/json
"content-length";req: 18
"@signature-params": ("@status" "content-digest" "content-type" \
  "@authority";req "@method";req "@path";req "@query";req \
  "content-digest";req "content-type";req "content-length";req)\
  ;created=1618884479;keyid="test-key-ecc-p256"
```

and the following signed response:

```
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
Content-Digest: sha-512=:0Y6iCBzGg5rZtoXS95Ijz03mslf6KAMCloESHObfwn\
  HJDbkkWWQz6PhhU9kxsTbARtY2PTBOzq24uJFpHsMuAg==:
Signature-Input: reqres=("@status" "content-digest" "content-type" \
  "@authority";req "@method";req "@path";req "@query";req \
  "content-digest";req "content-type";req "content-length";req)\
  ;created=1618884479;keyid="test-key-ecc-p256"
Signature: reqres=:C73J41GVKc+TYXbSobvZf0CmNcptRiWN+NY1Or0A36ISg6ym\
  dRN6ZgR2QfrtopFNzqAyv+CeWrMsNbcV2Ojsgg==:

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

Note that the ECDSA signature algorithm in use here is non-deterministic, meaning that 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 7.3.5.

Applications signing a response to a signed request SHOULD sign all of the components of the request signature value to provide sufficient coverage and protection against a class of collision attacks, as discussed in Section 7.3.7. The server in this example has included all components listed in the Signature-Input field of the client's signature on the request in the response signature, in addition to components of the response.

While it is syntactically possible to include the Signature and Signature-Input fields of the request message in the signature components of a response to a message using this mechanism, this practice is NOT RECOMMENDED. This is because signatures of signatures do not provide transitive coverage of covered components as one might expect, and the practice is susceptible to several attacks as discussed in Section 7.3.7. An application that needs to signal successful processing or receipt of a signature would need to carefully specify alternative mechanisms for sending such a signal securely.

The response signature can only ever cover what is included in the request message when using this flag. Consequently, if an application needs to include the message content of the request under the signature of its response, the client needs to include a means for covering that content, such as a Content-Digest field. See the discussion in Section 7.2.8 for more information.

The `req` parameter MUST NOT be used for any component in a signature that targets a request message.

2.5. Creating the Signature Base

The signature base is an ASCII string [ASCII] containing the canonicalized HTTP message components covered by the signature. The input to the signature base creation algorithm is the ordered set of covered component identifiers and their associated values, along with any additional signature parameters discussed in Section 2.3.

Component identifiers are serialized using the strict serialization rules defined by [STRUCTURED-FIELDS], Section 4. The component identifier has a component name, which is a String Item value serialized using the `sf-string` ABNF rule. The component identifier MAY also include defined parameters that are serialized using the `parameters` ABNF rule. The signature parameters line defined in Section 2.3 follows this same pattern, but the component identifier is a String Item with a fixed value and no parameters, and the component value is always an Inner List with optional parameters.

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"`, `"@authority"`, `"@signature-params"`, or `"example-dictionary";key="foo"`.

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-content )
    ; no obs-fold nor obs-text
component-identifier = component-name parameters
component-name = sf-string
derived-component-value = *( VCHAR / SP )
signature-params-line = DQUOTE "@signature-params" DQUOTE
     ":" SP inner-list
```

To create the signature base, the signer or verifier concatenates entries for each component identifier in the signature's covered components (including their parameters) using the following algorithm. All errors produced as described MUST fail the algorithm immediately, without outputting a signature base.

1. Let the output be an empty string.
2. For each message component item in the covered components set (in order):
3. Append the signature parameters component (Section 2.3) according to the `signature-params-line` rule as follows:
4. Produce an error if the output string contains any non-ASCII characters [ASCII].
5. 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 a signature base. Such situations include, but are not limited to, the following:

- 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 includes a parameter that is unknown or does not apply to the component identifier to which it is attached.
- 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, that 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. For example, 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 known by the application to 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 above does not include the final newline that ends the displayed example, nor do other example signature bases displayed elsewhere in this specification.

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 signature context (including the target message) and the requirements for the application. The output is a signature value and set of signature parameters that can be communicated to the verifier by adding them to the message.

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

1. The signer chooses an HTTP signature algorithm and key material for signing from the set of potential signing algorithms. The set of potential algorithms is determined by the application and is out of scope for this document. 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 signature. An appropriate expiration length, and the processing requirements of this parameter, are application specific.
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 set is later used as the value of the Signature-Input field as described in Section 4.1.
5. The signer creates the signature base using these parameters and the signature base creation algorithm (Section 2.5).
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 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 2.5, the example signature base is signed with the `test-key-rsa-pss` key (see Appendix B.1.2) and the RSASSA-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\
VVdHXrl8ePL4ccqlazIJdC4QEjrF+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 that 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 7.3.5.

3.2. Verifying a Signature

Verification of an HTTP message signature is a process that takes as its input the signature context (including the target message, particularly its 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 apply the following algorithm:

1. Parse the Signature and Signature-Input fields as described in 4.1 and 4.2, and extract the signatures to be verified and their labels.
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 the applicable technique to obtain the key material from this external knowledge. If the key is identified in the signature parameters, the verifier will dereference the key identifier 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 or trust for this request or that does not match something preconfigured, the verification MUST fail.
6. Determine the algorithm to apply for verification:
7. Use the received HTTP message and the parsed signature parameters to recreate the signature base, using the algorithm defined in Section 2.5. 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 2.3. Note that this does not include 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, and 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 label `sig1` of the following message with the `test-key-rsa-pss` key (see Appendix B.1.2) and the RSASSA-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+ARHBmWuNo1UzVVdHXrl8ePL4ccccqlazIJdC4QEjrF+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 entries be signed, and that the signature creation timestamp be 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, Content-Digest).
- Enforcing a maximum signature age from the time of the `created` timestamp.
- Rejecting signatures past the expiration time in the `expires` timestamp. 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.
- Prohibiting 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 the `nonce` parameter.
- Requiring an application-specific value for the `tag` parameter.

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 Algorithms

An HTTP message signature MUST use a cryptographic digital signature or MAC method that is appropriate for the key material, environment, and needs of the signer and verifier. This specification does not strictly limit the available signature algorithms, and any signature algorithm that meets these basic requirements MAY be used by an application of HTTP message signatures.

For each signing method, `HTTP_SIGN` takes as its input the signature base defined in Section 2.5 as a byte array (`M`) and the signing key material (`Ks`), and outputs the resultant signature as a byte array (`S`):

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

For each verification method, `HTTP_VERIFY` takes as its input the regenerated signature base defined in Section 2.5 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
```

The following sections contain several common signature algorithms and demonstrate how these cryptographic primitives map to the `HTTP_SIGN` and `HTTP_VERIFY` definitions above. Which method to use can be communicated through the explicit algorithm (`alg`) signature parameter (Section 2.3), by reference to the key material, or through mutual agreement between the signer and verifier. Signature algorithms selected using the `alg` parameter MUST use values from the "HTTP Signature Algorithms" registry (Section 6.2).

3.3.1. RSASSA-PSS Using SHA-512

To sign using this algorithm, the signer applies the `RSASSA-PSS-SIGN (K, M)` function defined in [RFC8017] with the signer's private signing key (`K`) and the signature base (`M`) (Section 2.5). 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`) recreated 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 whether the signature presented is valid.

Note that the output of the RSASSA-PSS algorithm is non-deterministic; therefore, it is not correct to recalculate 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 7.3.5.

The 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 defined in [RFC8017] with the signer's private signing key (`K`) and the signature base (`M`) (Section 2.5). 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`) recreated 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 indicate whether the signature presented is valid.

The 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 2.5). 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 bytewise to the value of the HTTP message signature, and the results of the comparison determine the validity of the signature presented.

The 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` signature algorithm defined in [FIPS186-5] using curve P-256 with the signer's private signing key and the signature base (Section 2.5). 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 as 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 a byte array of the HTTP message signature output used in Section 3.1.

To verify using this algorithm, the verifier applies the `ECDSA` signature algorithm defined in [FIPS186-5] using the public key portion of the verification key material and the signature base recreated 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 as 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 whether the signature presented is valid.

Note that the output of ECDSA signature algorithms is non-deterministic; therefore, it is not correct to recalculate 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 7.3.5.

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

3.3.5. ECDSA Using Curve P-384 DSS and SHA-384

To sign using this algorithm, the signer applies the `ECDSA` signature algorithm defined in [FIPS186-5] using curve P-384 with the signer's private signing key and the signature base (Section 2.5). The hash SHA-384 [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 as big-endian unsigned integers, zero-padded to 48 octets each. These encoded values are concatenated into a single 96-octet array consisting of the encoded value of `r` followed by the encoded value of `s`. The resulting concatenation of `(r, s)` is a byte array of the HTTP message signature output used in Section 3.1.

To verify using this algorithm, the verifier applies the `ECDSA` signature algorithm defined in [FIPS186-5] using the public key portion of the verification key material and the signature base recreated as described in Section

3.2. The hash function SHA-384 [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 96-octet array consisting of the encoded values of `r` and `s` concatenated in order. These are both encoded as big-endian unsigned integers, zero-padded to 48 octets each. The resulting signature value `(r, s)` is used as input to the signature verification function. The results of the verification function indicate whether the signature presented is valid.

Note that the output of ECDSA signature algorithms is non-deterministic; therefore, it is not correct to recalculate 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 7.3.5.

The use of this algorithm can be indicated at runtime using the `ecdsa-p384-sha384` value for the `alg` signature parameter.

3.3.6. EdDSA Using Curve edwards25519

To sign using this algorithm, the signer applies the `Ed25519` algorithm defined in Section 5.1.6 of [RFC8032] with the signer's private signing key and the signature base (Section 2.5). The signature base is taken as the input message (`M`) with no prehash 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 defined in Section 5.1.7 of [RFC8032] using the public key portion of the verification key material (`A`) and the signature base recreated as described in Section 3.2. The signature base is taken as the input message (`M`) with no prehash 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 whether the signature presented is valid.

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

3.3.7. JSON Web Signature (JWS) Algorithms

If the signing algorithm is a JSON Object Signing and Encryption (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 message's signature base (Section 2.5) is used as the entire "JWS Signing Input". The JOSE Header [JWS] [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`".

JSON Web Algorithm (JWA) values from the "JSON Web Signature and Encryption Algorithms" registry are not included as signature parameters. Typically, the JWS algorithm can be signaled using JSON Web Keys (JWKs) or other mechanisms common to JOSE implementations. In fact, JWA values are not registered in the "HTTP Signature Algorithms" registry (Section 6.2), and so the explicit `alg` signature parameter is not used at all when using JOSE signing algorithms.

4. Including a Message Signature in a Message

HTTP message signatures can be included within an HTTP message via the Signature-Input and Signature fields, both defined within this specification.

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.

An HTTP message signature is identified by a label within an HTTP message. This label MUST be unique within a given HTTP message and MUST be used in both the Signature-Input field and the Signature field. The label is chosen by the signer, except where a specific label is dictated by protocol negotiations such as those described in Section 5.

An HTTP message signature MUST use both the Signature-Input field and the Signature field, and each field MUST contain the same labels. The presence of a label in one field but not the other is an error.

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 the label that uniquely identifies the message signature within the HTTP message. The member's value is the covered components ordered set serialized as an Inner List, including all signature metadata parameters identified by the label:

```
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 defined in Section 2.3. Note that in a Structured Field value, list order and parameter order have 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 field 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 the signature context of the target message. The member's key is the label that uniquely identifies the message signature within the HTTP message. The member's value is a Byte Sequence containing the signature value for the message signature identified by the label:

```
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 field 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 all be added by the same signer, or they 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. A reverse proxy takes this request and 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-Length: 18
Content-Digest: sha-512=:WZDPaVn/7XgHaAy8pmojAkGWoRx2UFChF41A2svX+T\
  aPm+AbwAgBWnrIiYllu7BNNyealdVLvRwEmTHWXvJwew==:
Signature-Input: sig1=("@method" "@authority" "@path" \
  "content-digest" "content-type" "content-length")\
  ;created=1618884475;keyid="test-key-ecc-p256"
Signature: sig1=:X5spyd6CFnAG5QnDyHfqoSNICd+BUP4LYMz2Q0JXlb//4Ijpzp\
  +kve2w4NIyqeAuM7jTDX+sNalzA8ESSaHD3A==:

{"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-Length: 18
Forwarded: for=192.0.2.123;host=example.com;proto=https
Content-Digest: sha-512=:WZDPaVn/7XgHaAy8pmojAkGWoRx2UFChF41A2svX+T\
  aPm+AbwAgBWnrIiYllu7BNNyealdVLvRwEmTHWXvJwew==:
Signature-Input: sig1=("@method" "@authority" "@path" \
  "content-digest" "content-type" "content-length")\
  ;created=1618884475;keyid="test-key-ecc-p256"
Signature: sig1=:X5spyd6CFnAG5QnDyHfqoSNICd+BUP4LYMz2Q0JXlb//4Ijpzp\
  +kve2w4NIyqeAuM7jTDX+sNalzA8ESSaHD3A==:

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

The proxy is in a position to validate the incoming client's signature and make its own statement to the origin server about the nature of the request that it is forwarding by adding its own signature over the new message before passing it along to the origin server. The proxy also includes all the elements from the original message that are relevant to the origin server's processing. In many cases, the proxy will want to cover all the same components that were covered by the client's signature, which is the case in the following example. Note that in this example, the proxy is signing over the new authority value, which it has changed. The proxy also adds the Forwarded header field to its own signature value. 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

"@method": POST
"@authority": origin.host.internal.example
"@path": /foo
"content-digest": sha-512=:WZDPaVn/7XgHaAy8pmojAkGWoRx2UFChF41A2svX\
  +TaPm+AbwAgBWnrIiYllu7BNNyealdVLvRwEmTHWXvJwew==:
"content-type": application/json
"content-length": 18
"forwarded": for=192.0.2.123;host=example.com;proto=https
"@signature-params": ("@method" "@authority" "@path" \
  "content-digest" "content-type" "content-length" "forwarded")\
  ;created=1618884480;keyid="test-key-rsa";alg="rsa-v1_5-sha256"\
  ;expires=1618884540
```

and a signature output value of:

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

S6ZzPXSdAMOPjN/6KXfXWNO/f7V6cHm7BXYUh3YD/fRad4BCaRZxP+JH+8XY1I6+8Cy\
+CM5g92iHgxtRPz+MjniOaYmdkDcnL9cCpXJleXsOckpURl49GwiyUpZ10KHgOEe11s\
x3G2gxI8S0jnxQB+Pu68U9vVcasqOWAEObtNKKZd8tSFu7LB5YAv0RAGhB8tmpv7sFn\
Im9y+7X5kXQfi8NMaZaA8i2ZHwpBdg7a6CMfwnnrtflzvZdXAsD3LH2TwevU+/PBPv0\
B6NMNk93wUs/vfJvye+YuI87HU38lZHowtznbLVdp770I6VHR6WfgS9ddzirrswsE1w\
5o0LV/g==
```

These values are added to the HTTP request message by the proxy. The original signature is included under the label `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 `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-Length: 18
Forwarded: for=192.0.2.123;host=example.com;proto=https
Content-Digest: sha-512=:WZDPaVn/7XgHaAy8pmojAkGWoRx2UFChF41A2svX+T\
  aPm+AbwAgBWnrIiYllu7BNNyealdVLvRwEmTHWXvJwew==:
Signature-Input: sig1=("@method" "@authority" "@path" \
    "content-digest" "content-type" "content-length")\
    ;created=1618884475;keyid="test-key-ecc-p256", \
  proxy_sig=("@method" "@authority" "@path" "content-digest" \
    "content-type" "content-length" "forwarded")\
    ;created=1618884480;keyid="test-key-rsa";alg="rsa-v1_5-sha256"\
    ;expires=1618884540
Signature: sig1=:X5spyd6CFnAG5QnDyHfqoSNICd+BUP4LYMz2Q0JXlb//4Ijpzp\
    +kve2w4NIyqeAuM7jTDX+sNalzA8ESSaHD3A==:, \
  proxy_sig=:S6ZzPXSdAMOPjN/6KXfXWNO/f7V6cHm7BXYUh3YD/fRad4BCaRZxP+\
    JH+8XY1I6+8Cy+CM5g92iHgxtRPz+MjniOaYmdkDcnL9cCpXJleXsOckpURl49G\
    wiyUpZ10KHgOEe11sx3G2gxI8S0jnxQB+Pu68U9vVcasqOWAEObtNKKZd8tSFu7\
    LB5YAv0RAGhB8tmpv7sFnIm9y+7X5kXQfi8NMaZaA8i2ZHwpBdg7a6CMfwnnrtf\
    lzvZdXAsD3LH2TwevU+/PBPv0B6NMNk93wUs/vfJvye+YuI87HU38lZHowtznbL\
    Vdp770I6VHR6WfgS9ddzirrswsE1w5o0LV/g==:

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

While the proxy could additionally include the client's Signature field value and Signature-Input fields from the original message in the new signature's covered components, this practice is NOT RECOMMENDED due to known weaknesses in signing signature values as discussed in Section 7.3.7. The proxy is in a position to validate the client's signature; the changes the proxy makes to the message will invalidate the existing signature when the message is seen by the origin server. In this example, it is possible for the origin server to have additional information in its signature context to account for the change in authority, though this practice requires additional configuration and extra care as discussed in Section 7.4.4. In other applications, the origin server will not be able to verify the original signature itself but will still want to verify that the proxy has done the appropriate validation of the client's signature. An application that needs to signal successful processing or receipt of a signature would need to carefully specify alternative mechanisms for sending such a signal securely.

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.

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-Signature field, each covering the same identified components of the Accept-Signature field.

The sender of an Accept-Signature field MUST include only identifiers that are appropriate for the type of the target message. For example, if the target message is a request, the covered components cannot 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 key is the label 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 component metadata parameters, using the serialization process defined in Section 2.3:

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

Accept-Signature: sig1=("@method" "@target-uri" "@authority" \
  "content-digest" "cache-control");\
  keyid="test-key-rsa-pss";created;tag="app-123"
```

The list of component identifiers indicates the exact set of component identifiers to be included in the requested signature, including all applicable component parameters.

The signature request MAY include signature metadata parameters that indicate desired behavior for the signer. The following behavior is defined by this specification:

`created`: The signer is requested to generate and include a creation time. This parameter has no associated value when sent as a signature request.

`expires`: The signer is requested to generate and include an expiration time. This parameter has no associated value when sent as a signature request.

`nonce`: The signer is requested to include the value of this parameter as the signature `nonce` in the target signature.

`alg`: The signer is requested to use the indicated signature algorithm from the "HTTP Signature Algorithms" registry to create the target signature.

`keyid`: The signer is requested to use the indicated key material to create the target signature.

`tag`: The signer is requested to include the value of this parameter as the signature `tag` in the target signature.

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:
3. Optionally create any additional Signature-Input and Signature field values, with unique labels not found in the Accept-Signature field.
4. Combine all labeled Signature-Input and Signature field values, and attach both fields to the target message.

By this process, a signature applied to a target message MUST have the same label, MUST include the same set of covered components, MUST process all requested parameters, and MAY have additional parameters.

The receiver of an Accept-Signature field MAY ignore any signature request that does not fit application parameters.

The target message MAY include additional signatures not specified by the Accept-Signature field. For example, to cover additional message components, the signer can create a second signature that includes the additional components as well as the signature output of the requested signature.

6. IANA Considerations

IANA has updated one registry and created four new registries, according to the following sections.

6.1. HTTP Field Name Registration

IANA has updated the entries in the "Hypertext Transfer Protocol (HTTP) Field Name Registry" as follows:

Field Name	Status	Reference
Signature-Input	permanent	Section 4.1 of RFC 9421
Signature	permanent	Section 4.2 of RFC 9421
Accept-Signature	permanent	Section 5.1 of RFC 9421

Table 1: Updates to the Hypertext Transfer Protocol (HTTP) Field Name Registry

6.2. HTTP Signature Algorithms Registry

This document defines HTTP signature algorithms, for which IANA has created and now maintains a new registry titled "HTTP Signature Algorithms". 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 Specification Required registration policy [RFC8126].

The algorithms listed in this registry identify some possible cryptographic algorithms for applications to use with this specification, but the entries neither represent an exhaustive list of possible algorithms nor indicate fitness for purpose with any particular application of this specification. An application is free to implement any algorithm that suits its needs, provided the signer and verifier can agree to the parameters of that algorithm in a secure and deterministic fashion. When an application needs to signal the use of a particular algorithm at runtime using the `alg` signature parameter, this registry provides a mapping between the value of that parameter and a particular algorithm. However, the use of the `alg` parameter needs to be treated with caution to avoid various forms of algorithm confusion and substitution attacks, as discussed in Section 7.

The Status value should reflect standardization status and the broad opinion of relevant interest groups such as the IETF or security-related Standards Development Organizations (SDOs). When an algorithm is first registered, the designated expert (DE) should set the Status field to "Active" if there is consensus for the algorithm to be generally recommended as secure or "Provisional" if the algorithm has not reached that consensus, e.g., for an experimental algorithm. A status of "Provisional" does not mean that the algorithm is known to be insecure but instead indicates that the algorithm has not reached consensus regarding its properties. If at a future time the algorithm as registered is found to have flaws, the registry entry can be updated and the algorithm can be marked as "Deprecated" to indicate that the algorithm has been found to have problems. This status does not preclude an application from using a particular algorithm; rather, it serves to provide a warning regarding possible known issues with an algorithm that need to be considered by the application. The DE can further ensure that the registration includes an explanation and reference for the Status value; this is particularly important for deprecated algorithms.

The DE is expected to do the following:

- Ensure that the algorithms referenced by a registered algorithm identifier are fully defined with all parameters (e.g., salt, hash, required key length) fixed by the defining text.
- Ensure that the algorithm definition fully specifies the `HTTP_SIGN` and `HTTP_VERIFY` primitive functions, including how all defined inputs and outputs map to the underlying cryptographic algorithm.
- Reject any registrations that are aliases of existing registrations.
- Ensure that all registrations follow the template presented in Section 6.2.1; this includes ensuring that the length of the name is not excessive while still being unique and recognizable.

This specification creates algorithm identifiers by including major parameters in the identifier String in order to make the algorithm name unique and recognizable by developers. However, 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 in the defining text that establishes the identifier in question. Implementors do not parse out the `rsa`, `pss`, and `sha512` portions of the identifier to determine parameters of the signing algorithm from the String, and the registration of one combination of parameters does not imply the registration of other combinations.

6.2.1. Registration Template

Algorithm Name: An identifier for the HTTP signature algorithm. The name MUST be an ASCII string that conforms to the `sf-string` ABNF rule in Section 3.3.3 of [STRUCTURED-FIELDS] 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: The status of the algorithm. MUST start with one of the following values and MAY contain additional explanatory text. The options are:

- "Active": For algorithms without known problems. The signature algorithm is fully specified, and its security properties are understood.
- "Provisional": For unproven algorithms. The signature algorithm is fully specified, but its security properties are not known or proven.
- "Deprecated": For algorithms with known security issues. The signature algorithm is no longer recommended for general use and might be insecure or unsafe in some known circumstances.

Reference: Reference to the document or documents 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.

6.2.2. Initial Contents

The table below contains the initial contents of the "HTTP Signature Algorithms" registry.

Algorithm Name	Description	Status	Reference
`rsa-pss-sha512`	RSASSA-PSS using SHA-512	Active	Section 3.3.1 of RFC 9421
`rsa-v1_5-sha256`	RSASSA-PKCS1-v1_5 using SHA-256	Active	Section 3.3.2 of RFC 9421
`hmac-sha256`	HMAC using SHA-256	Active	Section 3.3.3 of RFC 9421
`ecdsa-p256-sha256`	ECDSA using curve P-256 DSS and SHA-256	Active	Section 3.3.4 of RFC 9421
`ecdsa-p384-sha384`	ECDSA using curve P-384 DSS and SHA-384	Active	Section 3.3.5 of RFC 9421
`ed25519`	EdDSA using curve edwards25519	Active	Section 3.3.6 of RFC 9421

Table 2: Initial Contents of the HTTP Signature Algorithms Registry

6.3. HTTP Signature Metadata Parameters Registry

This document defines the signature parameters structure (Section 2.3), which may have parameters containing metadata about a message signature. IANA has created and now maintains 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 6.3.2. Future assignments and modifications to existing assignments are to be made through the Expert Review registration policy [RFC8126].

The DE is expected to do the following:

- Ensure that the name follows the template presented in Section 6.3.1; this includes ensuring that the length of the name is not excessive while still being unique and recognizable for its defined function.
- Ensure that the defined functionality is clear and does not conflict with other registered parameters.
- Ensure that the definition of the metadata parameter includes its behavior when used as part of the normal signature process as well as when used in an Accept-Signature field.

6.3.1. Registration Template

Name: An identifier for the HTTP signature metadata parameter. The name MUST be an ASCII string that conforms to the `key` ABNF rule defined in Section 3.1.2 of [STRUCTURED-FIELDS] 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.

Reference: Reference to the document or documents 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.

6.3.2. 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	Reference
`alg`	Explicitly declared signature algorithm	Section 2.3 of RFC 9421
`created`	Timestamp of signature creation	Section 2.3 of RFC 9421
`expires`	Timestamp of proposed signature expiration	Section 2.3 of RFC 9421
`keyid`	Key identifier for the signing and verification keys used to create this signature	Section 2.3 of RFC 9421
`nonce`	A single-use nonce value	Section 2.3 of RFC 9421
`tag`	An application-specific tag for a signature	Section 2.3 of RFC 9421

Table 3: Initial Contents of the HTTP Signature Metadata Parameters Registry

6.4. 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 target 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 field names. IANA has created and now maintains a new registry titled "HTTP Signature Derived Component Names" to record and maintain the set of

non-field component names and the methods used to produce their associated component values. 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].

The DE is expected to do the following:

- Ensure that the name follows the template presented in Section 6.4.1; this includes ensuring that the length of the name is not excessive while still being unique and recognizable for its defined function.
- Ensure that the component value represented by the registration request can be deterministically derived from the target HTTP message.
- Ensure that any parameters defined for the registration request are clearly documented, along with their effects on the component value.

The DE should ensure that a registration is sufficiently distinct from existing derived component definitions to warrant its registration.

When setting a registered item's status to "Deprecated", the DE should ensure that a reason for the deprecation is documented, along with instructions for moving away from the deprecated functionality.

6.4.1. Registration Template

Name: A name for the HTTP derived component. The name MUST begin with the "at" (@) character followed by an ASCII string consisting only of lowercase 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. A value of "Deprecated" indicates that the derived component name is no longer recommended for use.

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

Reference: Reference to the document or documents 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	Reference
`@signature-params`	Reserved for signature parameters line in signature base	Active	Request, Response	Section 2.3 of RFC 9421
`@method`	The HTTP request method	Active	Request	Section 2.2.1 of RFC 9421
`@authority`	The HTTP authority, or target host	Active	Request	Section 2.2.3 of RFC 9421
`@scheme`	The URI scheme of the request URI	Active	Request	Section 2.2.4 of RFC 9421

Name	Description	Status	Target	Reference
`@target-uri`	The full target URI of the request	Active	Request	Section 2.2.2 of RFC 9421
`@request-target`	The request target of the request	Active	Request	Section 2.2.5 of RFC 9421
`@path`	The full path of the request URI	Active	Request	Section 2.2.6 of RFC 9421
`@query`	The full query of the request URI	Active	Request	Section 2.2.7 of RFC 9421
`@query-param`	A single named query parameter	Active	Request	Section 2.2.8 of RFC 9421
`@status`	The status code of the response	Active	Response	Section 2.2.9 of RFC 9421

Table 4: Initial Contents of the HTTP Signature Derived Component Names Registry

6.5. 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 has created and now maintains a new registry titled "HTTP Signature Component Parameters" to record and maintain the set of parameter names, the component identifiers they are associated with, and the modifications these parameters make to the component value. Definitions of parameters MUST define the targets to which they apply (such as specific field types, derived components, or contexts). Initial values for this registry are given in Section 6.5.2. Future assignments and modifications to existing assignments are to be made through the Expert Review registration policy [RFC8126].

The DE is expected to do the following:

- Ensure that the name follows the template presented in Section 6.5.1; this includes ensuring that the length of the name is not excessive while still being unique and recognizable for its defined function.
- Ensure that the definition of the field sufficiently defines any interactions or incompatibilities with other existing parameters known at the time of the registration request.
- Ensure that the component value defined by the component identifier with the parameter applied can be deterministically derived from the target HTTP message in cases where the parameter changes the component value.

6.5.1. Registration Template

Name: A name for the parameter. The name MUST be an ASCII string that conforms to the `key` ABNF rule defined in Section 3.1.2 of [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.

Reference: Reference to the document or documents 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.

6.5.2. Initial Contents

The table below contains the initial contents of the "HTTP Signature Component Parameters" registry.

Name	Description	Reference
`sf`	Strict Structured Field serialization	Section 2.1.1 of RFC 9421
`key`	Single key value of Dictionary Structured Fields	Section 2.1.2 of RFC 9421
`bs`	Byte Sequence wrapping indicator	Section 2.1.3 of RFC 9421
`tr`	Trailer	Section 2.1.4 of RFC 9421
`req`	Related request indicator	Section 2.4 of RFC 9421
`name`	Single named query parameter	Section 2.2.8 of RFC 9421

Table 5: 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.
- The list of covered components is applicable to the context of the message.

In addition to the application requirement definitions listed in Section 1.4, the following security considerations provide discussion and context regarding the requirements of creating and verifying signatures on HTTP messages.

7.1. General Considerations

7.1.1. Skipping Signature Verification

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 or Signature-Input 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 while 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 an invalid signature fails as expected.

7.1.2. 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 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. This is an important step in preventing signature replay (Section 7.2.2).

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

7.2. Message Processing and Selection

7.2.1. 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 or modifying 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.

7.2.2. Signature Replay

Since HTTP message signatures allow 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 expired, 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.

7.2.3. 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, this document 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 using 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 circumstances. The Via and Forwarded header fields, for example, are expected to be manipulated by proxies and other middleboxes, 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.

7.2.4. 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 header 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 7.2.2 and 7.2.1.

"host" The Host header field is specific to HTTP/1.1, and its functionality is subsumed by the `@authority` derived component, defined in Section 2.2.3. In order to preserve the value across

different HTTP versions, applications should always use the `@authority` derived component. See also Section 7.5.4.

7.2.5. 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 during the signing process. An intermediary is allowed to relabel 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 parameters can be used to convey whatever additional meaning is required to be attached to, and covered by, the signature. In particular, the `tag` parameter can be used to define an application-specific value as described in Section 7.2.7.

7.2.6. 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 needs 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 signatures.

7.2.7. Collision of Application-Specific Signature Tag

Multiple applications and protocols could apply HTTP signatures on the same message simultaneously. In fact, this is a desired feature in many circumstances; see Section 4.3. A naive verifier could become confused while processing multiple signatures, either accepting or rejecting a message based on an unrelated or irrelevant signature. In order to help an application select which signatures apply to its own processing, the application can declare a specific value for the `tag` signature parameter as defined in Section 2.3. For example, a signature targeting an application gateway could require `tag="app-gateway"` as part of the signature parameters for that application.

The use of the `tag` parameter does not prevent an attacker from also using the same value as a target application, since the parameter's value is public and otherwise unrestricted. As a consequence, a verifier should only use a value of the `tag` parameter to limit which signatures to check. Each signature still needs to be examined by the verifier to ensure that sufficient coverage is provided, as discussed in Section 7.2.1.

7.2.8. Message Content

On its own, this specification does not provide coverage for the content of an HTTP message under the signature, in either a request or a 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 to pass the signature. Since only the field value is covered by the signature directly, checking only the signature is not sufficient protection against such a substitution attack.

As discussed in [DIGEST], the value of the Content-Digest field is dependent on the content encoding of the message. If an intermediary changes the content encoding, the resulting Content-Digest value would change. This would in turn invalidate the signature. Any intermediary performing such an action would need to apply a new signature with the updated Content-Digest field value, similar to the reverse proxy use case discussed in Section 4.3.

Applications that make use of the `req` parameter (Section 2.4) also need to be aware of the limitations of this functionality. Specifically, if a client does not include something like a Content-Digest header field in the request, the server is unable to include a signature that covers the request's content.

7.3. Cryptographic Considerations

7.3.1. Cryptography and Signature Collision

This document 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 signature base from 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 Signature Algorithms" registry is one source of trusted signature algorithms for applications to apply to their messages.

While it is possible for an attacker to substitute the signature parameters value or the signature value separately, the signature base generation algorithm (Section 2.5) always covers the signature parameters as the final value in the signature base 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 base.

7.3.2. 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 that 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 7.3.3).

7.3.3. Symmetric Cryptography

This document allows both asymmetric and symmetric cryptography to be applied to HTTP messages. By their 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 7.3.6).

7.3.4. Key Specification Mixup

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 not only that the signature can be validated for a message but that the key and algorithm used are appropriate.

7.3.5. 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.

7.3.6. 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 any key that uses the weaker RSA 1.5 specification.

Another example of a downgrade attack would be when an asymmetric algorithm is expected, such as RSA-PSS, but an attacker substitutes a signature using a 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 a higher-protocol-level algorithm specification instead, preventing an attacker from substituting the algorithm specified.

7.3.7. Signing Signature Values

When applying the `req` parameter (Section 2.4) or multiple signatures (Section 4.3) to a message, it is possible to sign the value of an existing Signature field, thereby covering the bytes of the existing signature output in the new signature's value. While it would seem that this practice would transitively cover the components under the original signature in a verifiable fashion, the attacks described in [JACKSON2019] can be used to impersonate a signature output value on an unrelated message.

In this example, Alice intends to send a signed request to Bob, and Bob wants to provide a signed response to Alice that includes a cryptographic proof that Bob is responding to Alice's incoming message. Mallory wants to intercept this traffic and replace Alice's message with her own, without Alice being aware that the interception has taken place.

1. Alice creates a message `Req_A` and applies a signature `Sig_A` using her private key `Key_A_Sign`.
2. Alice believes she is sending `Req_A` to Bob.
3. Mallory intercepts `Req_A` and reads the value `Sig_A` from this message.
4. Mallory generates a different message `Req_M` to send to Bob instead.
5. Mallory crafts a signing key `Key_M_Sign` such that she can create a valid signature `Sig_M` over her request `Req_M` using this key, but the byte value of `Sig_M` exactly equals that of `Sig_A`.
6. Mallory sends `Req_M` with `Sig_M` to Bob.
7. Bob validates `Sig_M` against Mallory's verification key `Key_M_Verify`. At no time does Bob think that he's responding to Alice.
8. Bob responds with response message `Res_B` to `Req_M` and creates signature `Sig_B` over this message using his key `Key_B_Sign`. Bob includes the value of `Sig_M` under `Sig_B`'s covered components but does not include anything else from the request message.
9. Mallory receives the response `Res_B` from Bob, including the signature `Sig_B` value. Mallory replays this response to Alice.
10. Alice reads `Res_B` from Mallory and verifies `Sig_B` using Bob's verification key `Key_B_Verify`. Alice includes the bytes of her original signature `Sig_A` in the signature base, and the signature verifies.
11. Alice is led to believe that Bob has responded to her message and believes she has cryptographic proof of this happening, but in fact Bob responded to Mallory's malicious request and Alice is none the wiser.

To mitigate this, Bob can sign more portions of the request message than just the Signature field, in order to more fully differentiate Alice's message from Mallory's. Applications using this feature, particularly for non-repudiation purposes, can stipulate that any components required in the original signature also be covered separately in the second signature. For signed messages, requiring coverage of the corresponding Signature-Input field of the first signature ensures that unique items such as nonces and timestamps are also covered sufficiently by the second signature.

7.4. Matching Signature Parameters to the Target Message

7.4.1. 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 verifier turning off signature checking in order to make systems work again (see Section 7.1.1) or application minimizing the requirements related to the signed component.

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. If an expected modification is found, the signer and verifier can agree on an alternative set of requirements that will pass. However, the signer and verifier should not remove the requirement to sign the modified component when it is suspected that an attacker is modifying the component.

7.4.2. Matching Values of Covered Components to Values in the Target Message

The verifier needs to make sure that the signed message components match those in the message itself. For example, the `@method` derived component requires that the value within the signature base be the same as the HTTP method used when presenting this message. This specification encourages this by requiring the verifier to derive the signature base from the message, but lazy caching or conveyance of a raw signature base to a processing subsystem could lead to downstream verifiers accepting a message that does not match the presented signature.

To counter this, the component that generates the signature base needs to be trusted by both the signer and verifier within a system.

7.4.3. Message Component Source and Context

The signature context for deriving message component values includes the target HTTP message itself, any associated messages (such as the request that triggered a response), and additional information that the signer or verifier has access to. Both signers and verifiers need to carefully consider the source of all information when creating component values for the signature base and take care not to take information from untrusted sources. Otherwise, an attacker could leverage such a loosely defined message context to inject their own values into the signature base string, overriding or corrupting the intended values.

For example, in most situations, the target URI of the message is as defined in [HTTP], Section 7.1. However, let's say that there is an application that requires signing of the `@authority` of the incoming request, but the application doing the processing is behind a reverse proxy. Such an application would expect a change in the `@authority` value, and it could be configured to know the external target URI as seen by the client on the other side of the proxy. This application would use this configured value as its target URI for the purposes of deriving message component values such as `@authority` instead of using the target URI of the incoming message.

This approach is not without problems, as a misconfigured system could accept signed requests intended for different components in the system. For this scenario, an intermediary could instead add its own signature to be verified by the application directly, as demonstrated in Section 4.3. This alternative approach requires a more active intermediary but relies less on the target application knowing external configuration values.

As another example, Section 2.4 defines a method for signing response messages and also including portions of the request message that triggered the response. In this case, the context for component value calculation is the combination of the response and request messages, not just the single message to which the signature is applied. For this feature, the `req` flag allows both signers to explicitly signal which part of the context is being sourced for a component identifier's value. Implementations need to ensure that only the intended message is being referred to for each component; otherwise, an attacker could attempt to subvert a signature by manipulating one side or the other.

7.4.4. Multiple Message Component Contexts

It is possible that the context for deriving message component values could be distinct for each signature present within a single message. This is particularly the case when proxies mutate messages and include signatures over the mutated values, in addition to any existing signatures. For example, a reverse proxy can replace a public hostname in a request to a service with the hostname for the individual service host to which it is forwarding the request. If both the client and the reverse proxy add signatures covering `@authority`, the service host will see two signatures on the request, each signing different values for the `@authority` message component, reflecting the change to that component as the message made its way from the client to the service host.

In such a case, it's common for the internal service to verify only one of the signatures or to use externally configured information, as discussed in Section 7.4.3. However, a verifier processing both signatures has to use a different message component context for each signature, since the component value for the `@authority` component will be different for each signature. Verifiers like this need to be aware of both the reverse proxy's context for incoming messages and the target service's context for the message coming from the reverse proxy. The verifier needs to take particular care to apply the correct context to the correct signature; otherwise, an attacker could use knowledge of this complex setup to confuse the inputs to the verifier.

Such verifiers also need to ensure that any differences in message component contexts between signatures are expected and permitted. For example, in the above scenario, the reverse proxy could include the original hostname in a Forwarded header field and could sign `@authority`, forwarded, and the client's entry in the Signature field. The verifier can use the hostname from the Forwarded header field to confirm that the hostname was transformed as expected.

7.5. HTTP Processing

7.5.1. Processing Invalid HTTP Field Names as 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 field names. 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 7.3.1) attack or other functional substitution attack (such as using the signature from a GET request on a crafted POST request).

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 processed as an HTTP field. 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 2.5 provides a safe order of operations.

7.5.2. Semantically Equivalent Field Values

The signature base generation algorithm (Section 2.5) 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 values 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 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 on 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 7.2.1) 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.

7.5.3. Parsing Structured Field Values

Several parts of this specification rely on the parsing of Structured Field values [STRUCTURED-FIELDS] -- in particular, strict serialization 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, on both the signer side and the verifier side.

7.5.4. 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 header 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 define a set of derived components that define a single way to get the value in question, such as the `@authority` derived component (Section 2.2.3) in lieu of the Host header field. Applications should therefore prefer derived components for such options where possible.

7.5.5. 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 7.3.1) attack. Alternatively, if header fields that appear multiple times are not joined into a single string value, as required by this specification, similar attacks can be mounted, as a signed component value would show up in the signature base more than once and could be substituted or otherwise attacked in this way.

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

7.5.6. Non-List Field Values

When an HTTP field occurs multiple times in a single message, these values need to be combined into a single one-line string value to be included in the HTTP signature base, as described in Section 2.5. Not all HTTP fields can be combined into a single value in this way and still be a valid value for the field. For the purposes of generating the signature base, the message component value is never meant to be read back out of the signature base string or used in the application. Therefore, it is considered best practice to treat the signature base generation algorithm separately from processing the field values by the application, particularly for fields that are known to have this property. If the field values that are being signed do not validate, the signed message should also be rejected.

If an HTTP field allows for unquoted commas within its values, combining multiple field values can lead to a situation where two semantically different messages produce the same line in a signature base. For example, take the following hypothetical header field with an internal comma in its syntax, here used to define two separate lists of values:

```
Example-Header: value, with, lots
Example-Header: of, commas
```

For this header field, sending all of these values as a single field value results in a single list of values:

```
Example-Header: value, with, lots, of, commas
```

Both of these messages would create the following line in the signature base:

```
"example-header": value, with, lots, of, commas
```

Since two semantically distinct inputs can create the same output in the signature base, special care has to be taken when handling such values.

Specifically, the Set-Cookie field [COOKIE] defines an internal syntax that does not conform to the List syntax provided in [STRUCTURED-FIELDS]. In particular, some portions allow unquoted commas, and the field is typically sent as multiple separate field lines with distinct values when sending multiple cookies. When multiple Set-Cookie fields are sent in the same message, it is not generally possible to combine these into a single line and be able to parse and use the results, as discussed in [HTTP], Section 5.3. Therefore, all the cookies need to be processed from their separate field values, without being combined, while the signature base needs to be processed from the special combined value generated solely for this purpose. If the cookie value is invalid, the signed message ought to be rejected, as this is a possible padding attack as described in Section 7.5.7.

To deal with this, an application can choose to limit signing of problematic fields like Set-Cookie, such as including the field in a signature only when a single field value is present and the results would be unambiguous. Similar caution needs to be taken with all fields that could have non-deterministic mappings into the signature base. Signers can also make use of the `bs` parameter to armor such fields, as described in Section 2.1.3.

7.5.7. Padding Attacks with Multiple Field Values

Since HTTP field values need to be combined into a single string value to be included in the HTTP signature base (see Section 2.5), it is possible for an attacker to inject an additional value for a given field and add this to the signature base of the verifier.

In most circumstances, this causes the signature validation to fail as expected, since the new signature base value will not match the one used by the signer to create the signature. However, it is theoretically possible for the attacker to inject both a garbage value into a field and a desired value into another field in order to force a particular input. This is a variation of the collision attack described in Section 7.3.1, where the attacker accomplishes their change in the message by adding to existing field values.

To counter this, an application needs to validate the content of the fields covered in the signature in addition to ensuring that the signature itself validates. With such protections, the attacker's padding attack would be rejected by the field value processor, even in the case where the attacker could force a signature collision.

7.5.8. Ambiguous Handling of Query Elements

The HTML form parameters format defined in 5 of [HTMLURL] is widely deployed and supported by many application frameworks. For convenience, some of these frameworks in particular combine query parameters that are found in the HTTP query and those found in the message content, particularly for POST messages with a Content-Type value of "application/x-www-form-urlencoded". The `@query-param` derived component identifier defined in Section 2.2.8 draws its values only from the query section of the target URI of the request. As such, it would be possible for an attacker to shadow or replace query parameters in a request by overriding a signed query parameter with an unsigned form parameter, or vice versa.

To counter this, an application needs to make sure that values used for the signature base and the application are drawn from a consistent context, in this case the query component of the target URI. Additionally, when the HTTP request has content, an application should sign the message content as well, as discussed in Section 7.2.8.

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 also result in 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 for 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 7.1.2.

8.3. Oracles

It is important to balance the need for providing useful feedback to developers regarding 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 surprising behavior where, for fear of breaking the signature, an intermediary cannot remove privacy-sensitive information from a message before forwarding it on for processing. One way to mitigate this specific situation would be for the intermediary to verify the signature itself and then modify 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 shown in the example in Section 4.3.

9. References

9.1. Normative References

[FIPS186-5] NIST, "Digital Signature Standard (DSS)", DOI 10.6028/NIST.FIPS.186-5, February 2023, <https://doi.org/10.6028/NIST.FIPS.186-5>.

[HTMLURL] WHATWG, "URL (Living Standard)", January 2024, <https://url.spec.whatwg.org/>.

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

[ASCII] Cerf, V., "ASCII format for network interchange", STD 80, RFC 20, DOI 10.17487/RFC0020, October 1969, <https://www.rfc-editor.org/info/rfc20>.

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

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[URI] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[ABNF] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <https://www.rfc-editor.org/info/rfc6234>.

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

[RFC7518] Jones, M., "JSON Web Algorithms (JWA)", RFC 7518, DOI 10.17487/RFC7518, May 2015, <https://www.rfc-editor.org/info/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/info/rfc8017>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/

RFC8032, January 2017, <https://www.rfc-editor.org/info/rfc8032>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[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/info/rfc8941>.

[HTTP] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022, <https://www.rfc-editor.org/info/rfc9110>.

[HTTP/1.1] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP/1.1", STD 99, RFC 9112, DOI 10.17487/RFC9112, June 2022, <https://www.rfc-editor.org/info/rfc9112>.

9.2. Informative References

[AWS-SIGv4] Amazon Simple Storage Service, "Authenticating Requests (AWS Signature Version 4)", March 2006, <https://docs.aws.amazon.com/AmazonS3/latest/API/sig-v4-authenticating-requests.html>.

[BCP195] Moriarty, K. and S. Farrell, "Deprecating TLS 1.0 and TLS 1.1", BCP 195, RFC 8996, DOI 10.17487/RFC8996, March 2021. Sheffer, Y., Saint-Andre, P., and T. Fossati, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 9325, DOI 10.17487/RFC9325, November 2022. https://www.rfc-editor.org/info/bcp195>

[DIGEST] Polli, R. and L. Pardue, "Digest Fields", RFC 9530, DOI 10.17487/RFC9530, February 2024, <https://www.rfc-editor.org/info/rfc9530>.

[SIGNING-HTTP-MESSAGES] Cavage, M. and M. Sporny, "Signing HTTP Messages", Work in Progress, draft-cavage-http-signatures-12, October 2019, <https://datatracker.ietf.org/doc/html/draft-cavage-http-signatures-12>.

[JACKSON2019] Jackson, D., Cremers, C., Cohn-Gordon, K., and R. Sasse, "Seems Legit: Automated Analysis of Subtle Attacks on Protocols that Use Signatures", DOI 10.1145/3319535.3339813, CCS '19: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pp. 2165-2180, November 2019, <https://dl.acm.org/doi/10.1145/3319535.3339813>.

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

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

[RFC7468] Josefsson, S. and S. Leonard, "Textual Encodings of PKIX, PKCS, and CMS Structures", RFC 7468, DOI 10.17487/RFC7468, April 2015, <https://www.rfc-editor.org/info/rfc7468>.

[JWS] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <https://www.rfc-editor.org/info/rfc7515>.

[RFC7807] Nottingham, M. and E. Wilde, "Problem Details for HTTP APIs", RFC 7807, DOI 10.17487/RFC7807, March 2016, <https://www.rfc-editor.org/info/rfc7807>.

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

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

[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/info/rfc8792>.

[CLIENT-CERT] Campbell, B. and M. Bishop, "Client-Cert HTTP Header Field", RFC 9440, DOI 10.17487/RFC9440, July 2023, <https://www.rfc-editor.org/info/rfc9440>.

[RFC9457] Nottingham, M., Wilde, E., and S. Dalal, "Problem Details for HTTP APIs", RFC 9457, DOI 10.17487/RFC9457, July 2023, <https://www.rfc-editor.org/info/rfc9457>.

[SIGNING-HTTP-REQS-OAUTH] Richer, J., Ed., Bradley, J., and H. Tschofenig, "A Method for Signing HTTP Requests for OAuth", Work in Progress, draft-ietf-oauth-signed-http-request-03, August 2016.

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

It is recommended that implementors first detect and validate the Signature-Input field defined in this specification to detect that the mechanism described in this document 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 specification.

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 non-deterministic, 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 [RFC7468], 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 all of these keys directly and may require additional processing. All keys are also made available in JWK format.

B.1.1. Example RSA Key

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-----
```

The same public and private key pair in JWK format:

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

{
  "kty": "RSA",
  "kid": "test-key-rsa",
  "p": "sqeUJmqXE3LP8tYoIjMIAKiTm9o6psPlc8CrLI9CH0UbuaA2JCOMcCNq8Sy\
  YbTqgnWlB9ZfcAm_cFpA8tYci9m5vYK8HNxQr-8FS3Qo8N9RJ8d0U5CswDzMYfRgh\
  AfUGwmlWj5hp1pQzAuhwbOXFtxKHVsMPhz1IBtF9Y8jvgqgYHLbmyiu1mw",
  "q": "vSlgXQbvHzWmuUBFRHAejRh_naQTDV3GnH4lcRHuFBFZCSLn82xQS2_7xFO\
  qfabqq17kNcvKfzdvWpGxxJ2cILAq0pZS6DmrZlvBU4IkK2ZHCac_XfWVZFh-PrsH\
  _EnVkDpfcYR_iw1F40C1q5w8R6WBHaew3SAp",
  "d": "b8lm5JZ2hUduLnq-OAKCSODeWQ7Uqs7eet2bqeuAD0_2po-PG4qhZoo7VwF\
  CUTWlJan9wqdxiAPlbEQKkCdFRcbakbjN2TMJjMCHWL5zfgvqhmgeyKsrqg1wSce9\
  7J1_Mkvn3fh6CbqnwNb6bVFDvTJS3i5FzRhKiv6rUsYm8ZAdF4XRaYkFkeuHPl7rc\
  -ruUTSAjC4GovxIxoDJFe0r4kbFmkiZOr40e8RZYK7T1IKrSvzfxx5AjnlK_OZOTC\
  q0L7wBPbMW-IxmQpFCjpI-yuoi3FlZG3LaLNrBMXQF_lLZUDHs77q3fAGxDWwum2h\
  KBfdBuUQtjlqwjQlgXPsskQ",
  "e": "AQAB",
  "qi": "PkbARLOwU_LcZrQy9mmfcPoQlAuCyeu1Q9nH7PYSnbHTFzmiud4Hl8bIXU\
  9a0_58blDoOl3PctF-b4rAEJYUpCODu5PFyN6uEFYRg-YQwpjBMkXk8Eb39128ctA\
  RB40Lx8caDhRdTyaEedIG3cQDXSpAl9EOzXkzfx4bZxjAHU9mkMdJwOcMDQ",
  "dp": "aiodZsrWpi8HFfZfeRs8OS_0L5x6WBl3Y9btoZgsIeruc9uZ8NXTIdxaM6\
  FdnyNEyOYA1VH94tDYR-xEt1br1ud_dkPslLV_Aac7d7EaYc7cdkb7oC9t6sphVg0\
  dqE0UTDlOwBxBYMtGmQbJsFzGpmjzVgKqWqJ3B947li2U7t63HXEvKprY2w",
  "dq": "b0DzpSMb5p42dcQgOTU8Mr4S6JOEhRr_YjErMkpaXUEqvZ3jEB9HRmcRi5\
  Gtt4NBiBMiY6V9br8a5gjEpiAQoIUcWokBMAYjEeurU8M6JLBd3YaZVVjISaFmdty\
  nwLFoQxCh6_EC1rSywwrfDpSwO29S9i8Xbaap",
  "n": "hAKYdtoeoy8zcAcR874L8cnZxKzAGwd7v36APp7Pv6Q2jdsPBRrwWEBnez6\
  d0UDKDwGbc6nxfEXAy5mbhgajzrw3MOEt8uA5txSKobBpKDeBLOsdJKFqMGmXCQvE\
  G7YemcxDTRPxAleIAgYYRjTSd_QBwVW9OwNFhekro3RtlinV0a75jfZgkne_YiktS\
  vLG34lw2zqXBDTC5NHROUqGTlML4PlNZS5Ri2U4aCNx2rUPRcKIlE0PuKxI4T-HIa\
  Fpv8-rdV6eUgOrB2xeI1dSFFn_nnv5OoZJEIB-VmuKn3DCUcCZSFlQPSXSfBDiUGh\
  wOw76WuSSsf1D4b_vLoJ10w"
}
```

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 PKCS #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
aKy6LIYs0E2cWgp8GADgoBs8llBq0UhX0KffglIeek3n7Z6Gt4YFge2TAcW2WbN4
XfK7lupFyo6HHyWRiYHMMARQXLJeOSdTn5aMBP0PO4bQyk5ORxTUSeOciPJUFktQ
HkvGbym7KryEfwH8Tks0L7WhzyP60PL3xS9FNOJi9m+zztwYIXGDQuKM2GDsITeD
2mI2oHoPMyAD0wdI7BwSVW18p1h+jgfc4dlexKYRAoGBAOVfuiEiOchGghV5vn5N
RDNscAFnpHj1QgMr6/UG05RTgmcLfVsI1I4bSkbrIuVKviGGf7atlkROALOG/xRx
DLadgBEeNyHL5lz6ihQaFJLVQ0u3U4SB67J0YtVO3R6lXcIjBDHuY8SjYJ7Ci6Z6
vuDcoaEujnlrtUhaMxvSfcUJAoGBAMPsCHXte1uWNAqYad2WdLjPDlKtQJK1diCm
rqmB2g8QE99hDOHItjDBEdpyFBKOIP+NpVtM2KLhRajjcL9Ph8jrID6XUqikQuVi
4J9FV2m42jXMuioTT13idAILanYg8D3idvy/3isDVkON0X3UAVKrgMEne0hJpkPL
FYqgetvDAoGBAKLQ6JZMbSe0pPIJkSamQhsehgL5Rs51iX4m1z7+sYFAJfhvN3Q/
OGIHDRp6HjMUcxHpHw7U+S1TETxePwKLnLKj6hw8jnX2/nZRgWHzgVcY+sPsReRx
NJVf+Cfh6yOtznfX00p+JWOXdSY8glSSHJwRAMog+hFGW1AYdt7w80XBAoGBAImR
NUugqapgaEA8TrFxkJmngXYaAqpA0iYRA7kv3S4QavPBUGtFJHBNULzitydkNtVZ
3w6hgce0h9YThTo/nKc+OZDZbgfN9s7cQ75x0PQCAO4fx2P91Q+mDzDUVTeG30mE
t2m3S0dGe47JiJxifV9P3wNBNrZGSIF3mrORBVNDAoGBAI0QKn2Iv7Sgo4T/XjND
dl2kZTXqGAk8dOhpUiw/HdM3OGWbhHj2NdCzBliOmPyQtAr770GITWvbAI+IRYyF
S7Fnk6ZVVVHsxjtaHy1uJGFlaZzKR4AGNaUTOJMs6NadzCmGPAxNQQOCqoUjn4XR
rOjr9w349JooGXhOxbu8nOxX
-----END PRIVATE KEY-----
```

The same public and private key pair in JWK format:

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

{
  "kty": "RSA",
  "kid": "test-key-rsa-pss",
  "p": "5V-6ISI5yEaCFXm-fk1EM2xwAWekePVCAyvr9QbTlFOCZwt9WwjUjhtKRus\
  i5Uq-IYZ_tq2WRE4As4b_FHEMtp2AER43IcvmXPqKFBoUktVDS7dThIHrsnRi1U7d\
  HqVdwiMEMe5jxKNgnsKLpnq-4NyhoS6OeWu1SFozG9J9xQk",
  "q": "w-wIde17W5Y0Cphp3ZZ0uM8OUq1AkrV2IKauqYHaDxAT32EM4ci2MMER2nI\
  UEo4g_42lW0zYouFFqONwv0-HyOsgPpdSqKRC5WLgn0VXabjaNcy6KhNPXeJ0Agtq\
  diDwPeJ2_L_eKwNWQ43RfdQBUquAwSd7SEmmQ8sViqB628M",
  "d": "lAfIqfpCYomVShfAKnwf2lD9I0wKjkHsCtZCif4kAlwQqqW6N-tIL3bdOR-\
  VWf0Q1ZBIDtpO91UrG7pansyrPERbNrRJlPiYEyPTHkCT1nD-l2isuiyGLNBNnFoK\
  fBgA4KAbPJZQatFIV9Cn34JSHnpN5-2ehreGBYHtkwHFtlmzeF3yu5bqRcqOhx8lk\
  YmBzDAEUFyyXjknU5-WjAT9DzuG0MpOTkcU1EnjnIjyVBZLUB5Lxm8puyq8hH8B_E\
  5LNC-1oc8j-tDy98UvRTTiYvZvs87cGCFxg0LijNhg7CE3g9piNqB6DzMgA9MHSOw\
  cElVtfKdYfo4H3OHZXsSmEQ",
  "e": "AQAB",
  "qi": "jRAqfYi_tKCjhP9eM0N2XaRlNeoYCTx06GlSLD8d0zc4ZZuEePY10LMGWI\
  6Y_JC0CvvvQYhNa9sAj4hFjIVLsWeTplVVUezGO1ofLW4kYWVpnMpHgAY1pRM4kyz\
  o1p3MKYY8DE1BA4KqhSOfhdGs6Ov3Dfj0migZeE7Fu7yc7Fc",
  "dp": "otDolkxtJ7Sk8gmRJqZCGx6GAvlGznWJfibXPv6xgUAl-G83dD84YgcNGn\
  oeMxRzEekfDtT5LVMRPF4_AoucsqPqHDyOdfb-dlGBYfOBVxj6w-xF5HE0lV_4J-H\
  rI63Od9fTSn4lY5d1JjyCVJIcnBEAyiD6EUZbUBh23vDzRcE",
  "dq": "iZE1S6CpqmBoQDxOsXGQmaeBdhoCqkDSJhEDuS_dLhBq88FQa0UkcE1QvO\
  K3J2Q21VnfDqGBx7SH1hOFOj-cpz45kNluB832ztxDvnHQ9AIA7h_HY_3VD6YPMNR\
  VN4bfSYS3abdLR0Z7jsmInGJ9X0_fA0E2tkZIgXeas5EFU0M",
  "n": "r4tmm3r20Wd_PbqvP1s2-QEtvpuRaV8Yq40gjUR8y2Rjxa6dpG2GXHbPfvM\
  s8ct-Lh1GH45x28Rw3Ry53mm-oAXjyQ86OnDkZ5N8lYbggD4O3w6M6pAvLkhk95An\
  dTrifbIFPNU8PPMO7OyrFAHqgDsznjPFmTOtCEcN2Z1FpWgchwuYLPL-Wokqltd11\
  nqqzi-bJ9cvSKADYdUAAN5WUtzdpiy6LbTgSxP7ociU4Tn0g5I6aDZJ7A8Lzo0KSy\
  ZYoA485mqcO0GVAdVw9lq4aOT9v6d-nb4bnNkQVklLQ3fVAvJm-xdDOp9LCNCN48V\
  2pnDOkFV6-U9nV5oyc6XI2w"
}
```

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-----
```

The same public and private key pair in JWK format:

```
{
  "kty": "EC",
  "crv": "P-256",
  "kid": "test-key-ecc-p256",
  "d": "UpuF81l-kOxbjf7T4mNSv0r5tN67Gim7rnf6EFpcYDs",
  "x": "qIVYZVLCrPZHGHjP17CTW0_-D9Lfw0EkjqF7xB4FivA",
  "y": "Mc4nN9LTDOBhfoUeg8Ye9WedFRhnZXZJA12Qp0zZ6F0"
}
```

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 PKCS #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-----
```

The same public and private key pair in JWK format:

```
{
  "kty": "OKP",
  "crv": "Ed25519",
  "kid": "test-key-ed25519",
  "d": "n4Ni-HpISpVObnQMW0wOhCKROaIKqKtW_2ZYb2p9KcU",
  "x": "JrQLj5P_89iXES9-vFgrIy29clF9CC_oPPsw3c5D0bs"
}
```

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=:mEWXIS7MaLRuGgxOBdODa3xqM1XdEvxoYhvlCFJ41Q\
  JgJc4GTsPp29l5oGX69wWdXymyU0rjJuahq4l5aGgfLQ==:
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 header fields 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, the use of an empty covered components set is discouraged. See Section 7.2.1 for more discussion.

Note that the RSA-PSS algorithm in use here is non-deterministic, meaning that 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 7.3.5.

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

This example covers additional components (the authority, the Content-Digest header field, and a single named query parameter) in `test-request` using the `rsa-pss-sha512` algorithm. This example also adds a `tag` parameter with the application-specific value of `header-example`.

The corresponding signature base is:

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

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

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

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

Signature-Input: sig-b22=("@authority" "content-digest" \
  "@query-param";name="Pet");created=1618884473\
  ;keyid="test-key-rsa-pss";tag="header-example"
Signature: sig-b22=:LjbtqUbfmvjj5C5kr1Ugj4PmLYvx9wVjZvD9GsTT4F7GrcQ\
  EdJzgI9qHxICagShLRiLMlAJjtq6N4CDfKtjvuJyE5qH7KT8UCMkSowOB4+ECxCmT\
  8rtAmj/0PIXxi0A0nxKyB09RNrCQibbUjsLS/2YyFYXEu4TRJQzRw1rLEuEfY17SA\
  RYhpTlaqwZVtR8NV7+4UKkjqpcAoFqWFQh62s7Cl+H2fjBSpqfZUJcsIk4N6wiKYd\
  4je2U/lankenQ99PZfB4jY3I5rSV2DSBVkSFsURIjYErOs0tFTQosMTAoxk//0RoK\
  UqiYY8Bh0aaUEb0rQl3/XaVe4bXTugEjHSw==:
```

Note that the RSA-PSS algorithm in use here is non-deterministic, meaning that 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 7.3.5.

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 header fields 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 field and the value of the `created` signature parameter need not be the same. This is due to the fact that the Date header field 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 field 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 7.2.4 for more discussion.

Note that the RSA-PSS algorithm in use here is non-deterministic, meaning that 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 7.3.5.

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

This example covers portions of the `test-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 header fields 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 signature algorithm in use here is non-deterministic, meaning that 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 7.3.5.

B.2.5. Signing a Request Using hmac-sha256

This example covers portions of the `test-request` message 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 header fields 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 regarding security trade-offs in Section 7.3.3.

B.2.6. Signing a Request Using ed25519

This example covers portions of the `test-request` message 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 header fields 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 field according to [CLIENT-CERT], and then applies a signature to this field. By signing this header field, a reverse proxy not only can attest to its own validation of the initial request's TLS parameters but can 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 field 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.

B.4. HTTP Message Transformations

HTTP allows intermediaries and applications to transform an HTTP message without affecting the semantics of the message itself. HTTP message signatures are designed to be robust against many of these transformations in different circumstances.

For example, the following HTTP request message has been signed using the `Ed25519` algorithm and the key `test-key-ed25519`:

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

GET /demo?name1=Value1&Name2=value2 HTTP/1.1
Host: example.org
Date: Fri, 15 Jul 2022 14:24:55 GMT
Accept: application/json
Accept: */*
Signature-Input: transform=("@method" "@path" "@authority" \
  "accept");created=1618884473;keyid="test-key-ed25519"
Signature: transform=:ZT1kooQsEHpZ0I1IjCqtQppOmIqlJPeo7DHR3SoMn0s5J\
  Z1eRGS0A+vyYP9t/LXlh5QMFFQ6cpLt2m0pmj3NDA==:
```

The signature base string for this message is:

```
"@method": GET
"@path": /demo
"@authority": example.org
"accept": application/json, */*
"@signature-params": ("@method" "@path" "@authority" "accept")\
  ;created=1618884473;keyid="test-key-ed25519"
```

The following message has been altered by adding the Accept-Language header field as well as adding a query parameter. However, since neither the Accept-Language header field nor the query is covered by the signature, the same signature is still valid:

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

GET /demo?name1=Value1&Name2=value2&param=added HTTP/1.1
Host: example.org
Date: Fri, 15 Jul 2022 14:24:55 GMT
Accept: application/json
Accept: */*
Accept-Language: en-US,en;q=0.5
Signature-Input: transform=("@method" "@path" "@authority" \
  "accept");created=1618884473;keyid="test-key-ed25519"
Signature: transform=:ZT1kooQsEHpZ0I1IjCqtQppOmIqlJPeo7DHR3SoMn0s5J\
  Z1eRGS0A+vyYP9t/LXlh5QMFFQ6cpLt2m0pmj3NDA==:
```

The following message has been altered by removing the Date header field, adding a Referer header field, and collapsing the Accept header field into a single line. The Date and Referer header fields are not covered by the signature, and the collapsing of the Accept header field is an allowed transformation that is already accounted for by the canonicalization algorithm for HTTP field values. The same signature is still valid:

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

GET /demo?name1=Value1&Name2=value2 HTTP/1.1
Host: example.org
Referer: https://developer.example.org/demo
Accept: application/json, */*
Signature-Input: transform=("@method" "@path" "@authority" \
  "accept");created=1618884473;keyid="test-key-ed25519"
Signature: transform=:ZT1kooQsEHpZ0I1IjCqtQppOmIqlJPeo7DHR3SoMn0s5J\
  Z1eRGS0A+vyYP9t/LXlh5QMFFQ6cpLt2m0pmj3NDA==:
```

The following message has been altered by reordering the field values of the original message but not reordering the individual Accept header fields. The same signature is still valid:

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

GET /demo?name1=Value1&Name2=value2 HTTP/1.1
Accept: application/json
Accept: */*
Date: Fri, 15 Jul 2022 14:24:55 GMT
Host: example.org
Signature-Input: transform=("@method" "@path" "@authority" \
  "accept");created=1618884473;keyid="test-key-ed25519"
Signature: transform=:ZT1kooQsEHpZ0I1IjCqtQppOmIqlJPeo7DHR3SoMn0s5J\
  Z1eRGS0A+vyYP9t/LXlh5QMFFQ6cpLt2m0pmj3NDA==:
```

The following message has been altered by changing the method to POST and the authority to "example.com" (inside the Host header field). Since both the method and authority are covered by the signature, the same signature is NOT still valid:

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

POST /demo?name1=Value1&Name2=value2 HTTP/1.1
Host: example.com
Date: Fri, 15 Jul 2022 14:24:55 GMT
Accept: application/json
Accept: */*
Signature-Input: transform=("@method" "@path" "@authority" \
  "accept");created=1618884473;keyid="test-key-ed25519"
Signature: transform=:ZT1kooQsEHpZ0I1IjCqtQppOmIqlJPeo7DHR3SoMn0s5J\
  Z1eRGS0A+vyYP9t/LXlh5QMFFQ6cpLt2m0pmj3NDA==:
```

The following message has been altered by changing the order of the two instances of the Accept header field. Since the order of fields with the same name is semantically significant in HTTP, this changes the value used in the signature base, and the same signature is NOT still valid:

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

GET /demo?name1=Value1&Name2=value2 HTTP/1.1
Host: example.org
Date: Fri, 15 Jul 2022 14:24:55 GMT
Accept: */*
Accept: application/json
Signature-Input: transform=("@method" "@path" "@authority" \
  "accept");created=1618884473;keyid="test-key-ed25519"
Signature: transform=:ZT1kooQsEHpZ0I1IjCqtQppOmIqlJPeo7DHR3SoMn0s5J\
  Z1eRGS0A+vyYP9t/LXlh5QMFFQ6cpLt2m0pmj3NDA==:
```

Acknowledgements

This specification was initially based on [SIGNING-HTTP-MESSAGES]. The editors would like to thank the authors of [SIGNING-HTTP-MESSAGES] -- Mark Cavage and Manu Sporny -- for their work on that Internet-Draft and their continuing contributions. This specification also includes contributions from [SIGNING-HTTP-REQS-OAUTH] and other similar efforts.

The editors would also like to thank the following individuals (listed in alphabetical order) for feedback, insight, and implementation of this document and its predecessors: 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, Ilari Liusvaara, Dave Longley, James H. Manger, Kathleen Moriarty, Yoav Nir, Mark Nottingham, 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.

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 (editor)
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