Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 22, 2020

M. Cavage
Oracle
M. Sporny
Digital Bazaar
October 20, 2019

Signing HTTP Messages
draft-cavage-http-signatures-12

Abstract

When communicating over the Internet using the HTTP protocol, it can be desirable for a server or client to authenticate the sender of a particular message. It can also be desirable to ensure that the message was not tampered with during transit. This document describes a way for servers and clients to simultaneously add authentication and message integrity to HTTP messages by using a digital signature.

Feedback

This specification is a joint work product of the W3C Digital Verification Community Group [1] and the W3C Credentials Community Group [2]. Feedback related to this specification should logged in the issue tracker [3] or be sent to public-credentials@w3.org [4].

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on April 22, 2020.

Table of Contents

1. Introduction

This protocol extension is intended to provide a simple and standard way for clients to sign HTTP messages.

HTTP Authentication [RFC2617] defines Basic and Digest authentication mechanisms, TLS 1.2 [RFC5246] defines cryptographically strong transport layer security, and OAuth 2.0 [RFC6749] provides a fully-specified alternative for authorization of web service requests. Each of these approaches are employed on the Internet today with varying degrees of protection. However, none of these schemes are designed to cryptographically sign the HTTP messages themselves, which is required in order to ensure end-to-end message integrity. An added benefit of signing the HTTP message for the purposes of end-to-end message integrity is that the client can be authenticated using the same mechanism without the need for multiple round-trips.

Several web service providers have invented their own schemes for signing HTTP messages, but to date, none have been standardized. While there are no techniques in this proposal that are novel beyond the previous art, it is useful to standardize a simple and cryptographically strong mechanism for digitally signing HTTP messages.

This specification presents two mechanisms with distinct purposes:

1. The "Signature" scheme which is intended primarily to allow a sender to assert the contents of the message sent are correct and have not been altered during transmission or storage in a way that alters the meaning expressed in the original message as signed. Any party reading the message (the verifier) may independently confirm the validity of the message signature. This scheme is agnostic to the client/server direction and can be used to verify the contents of either HTTP requests, HTTP reponses, or both.

2. The "Authorization" scheme which is intended primarily to allow a sender to request access to a resource or resources by proving that they control a secret key. This specification allows for this both with a shared secret (using HMAC) or with public/private keys. The "Authorization" scheme is typically used in authentication processes and not directly for message signing.

As a consequence `Authorization` header is normally generated (and the message signed) by the HTTP client and the message verified by the HTTP server.

1.1. Using Signatures in HTTP Requests

It is common practice to protect sensitive website and API functionality via authentication mechanisms. Often, the entity accessing these APIs is a piece of automated software outside of an interactive human session. While there are mechanisms like OAuth and API secrets that are used to grant API access, each have their weaknesses such as unnecessary complexity for particular use cases or the use of shared secrets which may not be acceptable to an implementer. Shared secrets also prohibit any possibility for non-repudiation, while secure transports such as TLS do not provide for this at all.

Digital signatures are widely used to provide authentication and integrity assurances without the need for shared secrets. They also do not require a round-trip in order to authenticate the client, and allow the integrity of a message to be verified independently of the transport (e.g. TLS). A server need only have an understanding of the key (e.g. through a mapping between the key being used to sign the content and the authorized entity) to verify that a message was signed by that entity.

When optionally combined with asymmetric keys associated with an identity, this specification can also enable authentication of a client and server with or without prior knowledge of each other.

1.2. Using Signatures in HTTP Responses

HTTP messages are routinely altered as they traverse the infrastructure of the Internet, for mostly benign reasons. Gateways and proxies add, remove and alter headers for operational reasons, so a sender cannot rely on the recipient receiving exactly the message transmitted. By allowing a sender to sign specified headers, and recipient or intermediate system can confirm that the original intent of the sender is preserved, and including a Digest header can also verify the message body is not modified. This allows any recipient to easily confirm both the sender's identity, and any incidental or malicious changes that alter the content or meaning of the message.

2. The Components of a Signature

There are a number of components in a signature that are common between the 'Signature' HTTP Authentication Scheme and the

'Signature' HTTP Header. This section details the components of the digital signature paremeters common to both schemes.

2.1. Signature Parameters

The following section details the Signature Parameters.

2.1.1. keyId

REQUIRED. The `keyId` field is an opaque string that the server can use to look up the component they need to validate the signature. It could be an SSH key fingerprint, a URL to machine-readable key data, an LDAP DN, etc. Management of keys and assignment of `keyId` is out of scope for this document. Implementations MUST be able to discover metadata about the key from the `keyId` such that they can determine the type of digital signature algorithm to employ when creating or verifying signatures.

2.1.2. signature

REQUIRED. The `signature` parameter is a base 64 encoded digital signature, as described in RFC 4648 [RFC4648], Section 4 [5]. The client uses the `algorithm` and `headers` Signature Parameters to form a canonicalized `signing string`. This `signing string` is then signed using the key associated with the `keyId` according to its digital signature algorithm. The `signature` parameter is then set to the base 64 encoding of the signature.

2.1.3. algorithm

RECOMMENDED. The `algorithm` parameter is used to specify the signature string construction mechanism. Valid values for this parameter can be found in the HTTP Signatures Algorithms Registry [6] and MUST NOT be marked "deprecated". Implementers SHOULD derive the digital signature algorithm used by an implementation from the key metadata identified by the `keyId` rather than from this field. If `algorithm` is provided and differs from the key metadata identified by the `keyId`, for example `rsa-sha256` but an EdDSA key is identified via `keyId`, then an implementation MUST produce an error. Implementers should note that previous versions of the `algorithm` parameter did not use the key information to derive the digital signature type and thus could be utilized by attackers to expose security vulnerabilities.

2.1.4. created

RECOMMENDED. The `created` field expresses when the signature was created. The value MUST be a Unix timestamp integer value. A signature with a `created` timestamp value that is in the future MUST NOT be processed. Using a Unix timestamp simplifies processing and avoids timezone management required by specifications such as RFC3339. Subsecond precision is not supported. This value is useful when clients are not capable of controlling the `Date` HTTP Header such as when operating in certain web browser environments.

2.1.5. expires

OPTIONAL. The `expires` field expresses when the signature ceases to be valid. The value MUST be a Unix timestamp integer value. A signature with an `expires` timestamp value that is in the past MUST NOT be processed. Using a Unix timestamp simplifies processing and avoid timezone management existing in RFC3339. Subsecod precision is allowed using decimal notation.

2.1.6. headers

OPTIONAL. The `headers` parameter is used to specify the list of HTTP headers included when generating the signature for the message. If specified, it SHOULD be a lowercased, quoted list of HTTP header fields, separated by a single space character. If not specified, implementations MUST operate as if the field were specified with a single value, `(created)`, in the list of HTTP headers. Note:

1. The list order is important, and MUST be specified in the order the HTTP header field-value pairs are concatenated together during Signature String Construction (Section 2.3) used during signing and verifying.

2. A zero-length `headers` parameter value MUST NOT be used, since it results in a signature of an empty string.

2.2. Ambiguous Parameters

If any of the parameters listed above are erroneously duplicated in the associated header field, then the the signature MUST NOT be processed. Any parameter that is not recognized as a parameter, or is not well-formed, MUST be ignored.

2.3. Signature String Construction

A signed HTTP message needs to be tolerant of some trivial alterations during transmission as it goes through gateways, proxies, and other entities. These changes are often of little consequence and very benign, but also often not visible to or detectable by either the sender or the recipient. Simply signing the entire message that was transmitted by the sender is therefore not feasible: Even very minor changes would result in a signature which cannot be verified.

This specification allows the sender to select which headers are meaningful by including their names in the `headers` Signature Parameter. The headers appearing in this parameter are then used to construct the intermediate Signature String, which is the data that is actually signed.

In order to generate the string that is signed with a key, the client MUST use the values of each HTTP header field in the `headers` Signature Parameter, in the order they appear in the `headers` Signature Parameter. It is out of scope for this document to dictate what header fields an application will want to enforce, but implementers SHOULD at minimum include the `(request-target)` and `(created)` header fields if `algorithm` does not start with `rsa`, `hmac`, or `ecdsa`. Otherwise, `(request-target)` and `date` SHOULD be included in the signature.

To include the HTTP request target in the signature calculation, use the special `(request-target)` header field name. To include the signature creation time, use the special `(created)` header field name. To include the signature expiration time, use the special `(expires)` header field name.

1. If the header field name is `(request-target)` then generate the header field value by concatenating the lowercased :method, an ASCII space, and the :path pseudo-headers (as specified in HTTP/2, Section 8.1.2.3 [7]). Note: For the avoidance of doubt, lowercasing only applies to the :method pseudo-header and not to the :path pseudo-header.

2. If the header field name is `(created)` and the `algorithm` parameter starts with `rsa`, `hmac`, or `ecdsa` an implementation MUST produce an error. If the `created` Signature Parameter is not specified, or is not an integer, an implementation MUST produce an error. Otherwise, the header field value is the integer expressed by the `created` signature parameter.

3. If the header field name is `(expires)` and the `algorithm` parameter starts with `rsa`, `hmac`, or `ecdsa` an implementation MUST produce an error. If the `expires` Signature Parameter is not specified, or is not an integer, an implementation MUST produce an error. Otherwise, the header field value is the integer expressed by the `created` signature parameter.

4. Create the header field string by concatenating the lowercased header field name followed with an ASCII colon `:`, an ASCII space ` `, and the header field value. Leading and trailing optional whitespace (OWS) in the header field value MUST be omitted (as specified in RFC7230 [RFC7230], Section 3.2.4 [8]).

    1. If there are multiple instances of the same header field, all header field values associated with the header field MUST be concatenated, separated by a ASCII comma and an ASCII space `, `, and used in the order in which they will appear in the transmitted HTTP message.

    2. If the header value (after removing leading and trailing whitespace) is a zero-length string, the signature string line correlating with that header will simply be the (lowercased) header name, an ASCII colon `:`, and an ASCII space ` `.

    3. Any other modification to the header field value MUST NOT be made.

    4. If a header specified in the headers parameter is malformed or cannot be matched with a provided header in the message, the implementation MUST produce an error.

5. If value is not the last value then append an ASCII newline `\n`.

To illustrate the rules specified above, assume a `headers` parameter list with the value of `(request-target) (created) host date cache-control x-emptyheader x-example` with the following HTTP request headers:

```
GET /foo HTTP/1.1
Host: example.org
Date: Tue, 07 Jun 2014 20:51:35 GMT
X-Example: Example header
    with some whitespace.
X-EmptyHeader:
Cache-Control: max-age=60
Cache-Control: must-revalidate
```

For the HTTP request headers above, the corresponding signature string is:

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

2.4. Creating a Signature

In order to create a signature, a client MUST:

1. Use the `headers` and `algorithm` values as well as the contents of the HTTP message, to create the signature string.

2. Use the key associated with `keyId` to generate a digital signature on the signature string.

3. The `signature` is then generated by base 64 encoding the output of the digital signature algorithm.

For example, assume that the `algorithm` value is "hs2019" and the `keyId` refers to an EdDSA public key. This would signal to the application that the signature string construction mechanism is the one defined in Section 2.3: Signature String Construction [9], the signature string hashing function is SHA-512, and the signing algorithm is Ed25519 as defined in RFC 8032 [RFC8032], Section 5.1: Ed25519ph, Ed25519ctx, and Ed25519. The result of the signature creation algorithm should result in a binary string, which is then base 64 encoded and placed into the `signature` value.

2.5. Verifying a Signature

In order to verify a signature, a server MUST:

1. Use the received HTTP message, the `headers` value, and the Signature String Construction (Section 2.3) algorithm to recreate the signature.

2. The `algorithm`, `keyId`, and base 64 decoded `signature` listed in the Signature Parameters are then used to verify the authenticity of the digital signature. Note: The application verifying the signature MUST derive the digital signature algorithm from the metadata associated with the `keyId` and MUST NOT use the value of `algorithm` from the signed message.

If a header specified in the `headers` value of the Signature Parameters (or the default item `(created)` where the `headers` value is not supplied) is absent from the message, the implementation MUST produce an error.

For example, assume that the `algorithm` value was "hs2019" and and the `keyId` refers to an EdDSA public key. This would signal to the application that the signature string construction mechanism is the one defined in Section 2.3: Signature String Construction [10], the signature string hashing function is SHA-512, and the signing algorithm is Ed25519 as defined in RFC 8032 [RFC8032], Section 5.1: Ed25519ph, Ed25519ctx, and Ed25519. The result of the signature verification algorithm should result in a successful verification unless the headers protected by the signature were tampered with in transit.

3. The 'Signature' HTTP Authentication Scheme

The "Signature" authentication scheme is based on the model that the client must authenticate itself with a digital signature produced by either a private asymmetric key (e.g., RSA) or a shared symmetric key (e.g., HMAC).

The scheme is parameterized enough such that it is not bound to any particular key type or signing algorithm.

3.1. Authorization Header

The client is expected to send an Authorization header (as defined in RFC 7235 [RFC7235], Section 4.1 [11]) where the "auth-scheme" is "Signature" and the "auth-param" parameters meet the requirements listed in Section 2: The Components of a Signature.

The rest of this section uses the following HTTP request as an example.

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

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

Note that the use of the `Digest` header field is per RFC 3230 [RFC3230], Section 4.3.2 [12] and is included merely as a demonstration of how an implementer could include information about

the body of the message in the signature. The following sections also assume that the "rsa-key-1" keyId asserted by the client is an identifier meaningful to the server.

3.1.1. Initiating Signature Authorization

A server may notify a client when a resource is protected by requiring a signature. To initiate this process, the server will request that the client authenticate itself via a 401 response [13] code. The server may optionally specify which HTTP headers it expects to be signed by specifying the `headers` parameter in the WWW-Authenticate header. For example:

```
HTTP/1.1 401 Unauthorized
Date: Thu, 08 Jun 2014 18:32:30 GMT
Content-Length: 1234
Content-Type: text/html
WWW-Authenticate: Signature
  realm="Example",headers="(request-target) (created)"

...
```

3.1.2. RSA Example

The authorization header and signature would be generated as:

```
Authorization: Signature keyId="rsa-key-1",algorithm="hs2019",
  headers="(request-target) (created) host digest content-length",
  signature="Base64(RSA-SHA512(signing string))"
```

The client would compose the signing string as:

```
(request-target): post /foo\n
(created): 1402174295
host: example.org\n
digest: SHA-256=X48E9qOokqqrvdts8nOJRJN3OWDUoyWxBf7kbu9DBPE=\n
content-length: 18
```

Note that the '\n' symbols above are included to demonstrate where the new line character should be inserted. There is no new line on the final line of the signing string. Each HTTP header above is displayed on a new line to provide better readability of the example.

For an RSA-based signature, the authorization header and signature would then be generated as:

```
Authorization: Signature keyId="rsa-key-1",algorithm="hs2019",
headers="(request-target) (created) host digest content-length",
signature="Base64(RSA-SHA512(signing string))"
```

3.1.3. HMAC Example

For an HMAC-based signature without a list of headers specified, the authorization header and signature would be generated as:

```
Authorization: Signature keyId="hmac-key-1",algorithm="hs2019",
headers="(request-target) (created) host digest content-length",
signature="Base64(HMAC-SHA512(signing string))"
```

The only difference between the RSA Example and the HMAC Example is the digital signature algorithm that is used. The client would compose the signing string in the same way as the RSA Example above:

```
(request-target): post /foo\n
(created): 1402174295
host: example.org\n
digest: SHA-256=X48E9qOokqqrvdts8nOJRJN3OWDUoyWxBf7kbu9DBPE=\n
content-length: 18
```

4. The 'Signature' HTTP Header

The "Signature" HTTP Header provides a mechanism to link the headers of a message (client request or server response) to a digital signature. By including the "Digest" header with a properly formatted digest, the message body can also be linked to the signature. The signature is generated and verified either using a shared secret (e.g. HMAC) or public/private keys (e.g. RSA, EC). This allows the receiver and/or any intermediate system to immediately or later verify the integrity of the message. When the signature is generated with a private key it can also provide a measure of non-repudiation, though a full implementation of a non-repudiatable statement is beyond the scope of this specification and highly dependent on implementation.

The "Signature" scheme can also be used for authentication similar to the purpose of the 'Signature' HTTP Authentication Scheme (Section 3). The scheme is parameterized enough such that it is not bound to any particular key type or signing algorithm.

4.1. Signature Header

The sender is expected to transmit a header (as defined in RFC 7230 [RFC7230], Section 3.2 [14]) where the "field-name" is "Signature", and the "field-value" contains one or more "auth-param"s (as defined

in RFC 7235 [RFC7235], Section 4.1 [15]) where the "auth-param" parameters meet the requirements listed in Section 2: The Components of a Signature.

The rest of this section uses the following HTTP request as an example.

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

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

The following sections assume that the "rsa-key-1" keyId provided by the signer is an identifier meaningful to the server.

4.1.1. RSA Example

The signature header and signature would be generated as:

```
Signature: keyId="rsa-key-1",algorithm="hs2019",
  created=1402170695, expires=1402170995,
  headers="(request-target) (created) (expires)
    host date digest content-length",
  signature="Base64(RSA-SHA256(signing string))"
```

The client would compose the signing string as:

```
(request-target): post /foo\n
(created): 1402170695
(expires): 1402170995
host: example.org\n
digest: SHA-256=X48E9qOokqqrvdts8nOJRJN3OWDUoyWxBf7kbu9DBPE=\n
content-length: 18
```

Note that the '\n' symbols above are included to demonstrate where the new line character should be inserted. There is no new line on the final line of the signing string. Each HTTP header above is displayed on a new line to provide better readability of the example.

For an RSA-based signature, the authorization header and signature would then be generated as:

```
Signature: keyId="rsa-key-1",algorithm="hs2019",created=1402170695,
  headers="(request-target) (created) host digest content-length",
  signature="Base64(RSA-SHA512(signing string))"
```

4.1.2. HMAC Example

For an HMAC-based signature without a list of headers specified, the authorization header and signature would be generated as:

```
Signature: keyId="hmac-key-1",algorithm="hs2019",created=1402170695,
  headers="(request-target) (created) host digest content-length",
  signature="Base64(HMAC-SHA512(signing string))"
```

The only difference between the RSA Example and the HMAC Example is the signature algorithm that is used. The client would compose the signing string in the same way as the RSA Example above:

```
(request-target): post /foo\n
(created): 1402170695
host: example.org\n
digest: SHA-256=X48E9qOokqqrvdts8nOJRJN3OWDUoyWxBf7kbu9DBPE=\n
content-length: 18
```

5. References

5.1. Normative References

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <https://www.rfc-editor.org/info/rfc7230>.

[RFC7235] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Authentication", RFC 7235, DOI 10.17487/RFC7235, June 2014, <https://www.rfc-editor.org/info/rfc7235>.

5.2. Informative References

[RFC2617] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication", RFC 2617, DOI 10.17487/RFC2617, June 1999, <https://www.rfc-editor.org/info/rfc2617>.

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

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <https://www.rfc-editor.org/info/rfc5246>.

[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>.

[RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <https://www.rfc-editor.org/info/rfc6749>.

[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>.

5.3. URIs

[1] https://w3c-dvcg.github.io/

[2] https://w3c-ccg.github.io/

[3] https://github.com/w3c-dvcg/http-signatures/issues

[4] mailto:public-credentials@w3.org

[5] https://tools.ietf.org/html/rfc4648#section-4

[6] #hsa-registry

[7] https://tools.ietf.org/html/rfc7540#section-8.1.2.3

[8] https://tools.ietf.org/html/rfc7230#section-3.2.4

[9] #canonicalization

[10] #canonicalization

[11] https://tools.ietf.org/html/rfc7235#section-2.1

[12] https://tools.ietf.org/html/rfc3230#section-4.3.2

[13] https://tools.ietf.org/html/rfc7235#section-3.1

[14] https://tools.ietf.org/html/rfc7230#section-3.2

[15] https://tools.ietf.org/html/rfc7235#section-4.1

[16] https://web-payments.org/specs/source/http-signatures-audit/

[17] https://web-payments.org/specs/source/http-signature-nonces/

[18] https://web-payments.org/specs/source/http-signature-trailers/

[19] https://www.iana.org/assignments/http-auth-scheme-signature

[20] https://www.iana.org/assignments/http-authschemes

[21] https://www.iana.org/assignments/shm-algorithms

[22] #canonicalization

[23] #canonicalization

[24] #canonicalization

[25] #canonicalization

[26] #canonicalization

Appendix A. Security Considerations

There are a number of security considerations to take into account when implementing or utilizing this specification. A thorough security analysis of this protocol, including its strengths and weaknesses, can be found in Security Considerations for HTTP Signatures [16].

Appendix B. Extensions

This specification was designed to be simple, modular, and extensible. There are a number of other specifications that build on this one. For example, the HTTP Signature Nonces [17] specification details how to use HTTP Signatures over a non-secured channel like

HTTP and the HTTP Signature Trailers [18] specification explains how to apply HTTP Signatures to streaming content. Developers that desire more functionality than this specification provides are urged to ensure that an extension specification doesn't already exist before implementing a proprietary extension.

If extensions to this specification are made by adding new Signature Parameters, those extension parameters MUST be registered in the Signature Authentication Scheme Registry. The registry will be created and maintained at (the suggested URI) https://www.iana.org/assignments/http-auth-scheme-signature [19]. An example entry in this registry is included below:

Signature Parameter: nonce
Reference to specification: [HTTP_AUTH_SIGNATURE_NONCE], Section XYZ.
Notes (optional): The HTTP Signature Nonces specification details how to use HTTP Signatures over a unsecured channel like HTTP.

Appendix C. Test Values

WARNING: THESE TEST VECTORS ARE OLD AND POSSIBLY WRONG. THE NEXT VERSION OF THIS SPECIFICATION WILL CONTAIN THE PROPER TEST VECTORS.

The following test data uses the following RSA 2048-bit keys, which we will refer to as `keyId=Test` in the following samples:

```
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDCFENGw33yGihy92pDjZQhl0C3
6rPJj+CvfSC8+q28hxA161QFNUd13wuCTUcq0Qd2qsBe/2hFyc2DCJJg0h1L78+6
Z4UMR7EOcpfdUE9Hf3m/hs+FUR45uBJeDK1HSFHD8bHKD6kv8FPGfJTotc+2xjJw
oYi+1hqp1fIekaxsyQIDAQAB
-----END PUBLIC KEY-----
```

```
-----BEGIN RSA PRIVATE KEY-----
MIICXgIBAAKBgQDCFENGw33yGihy92pDjZQhl0C36rPJj+CvfSC8+q28hxA161QF
NUd13wuCTUcq0Qd2qsBe/2hFyc2DCJJg0h1L78+6Z4UMR7EOcpfdUE9Hf3m/hs+F
UR45uBJeDK1HSFHD8bHKD6kv8FPGfJTotc+2xjJwoYi+1hqp1fIekaxsyQIDAQAB
AoGBAJR8ZkCUvx5kzv+utdl7T5MnordT1TvoXXJGXK7ZZ+UuvMNUCdN2QPc4sBiA
QWvLw1cSKt5DsKZ8UETpYPy8pPYnnDEz2dDYiaew9+xEpubyeW2oH4Zx71wqBtOK
kqwrXa/pzdpiucRRjk6vE6YY7EBBs/g7uanVpGibOVAEsqH1AkEA7DkjVH28WDUg
f1nqvfn2Kj6CT7nIcE3jGJsZZ7zlZmBmHFDONMLUrXR/Zm3pR5m0tCmBqa5RK95u
412jt1dPIwJBANJT3v8pnkth48bQo/fKel6uEYyboRtA5/uHuHkZ6FQF7OUkGogc
mSJluOdc5t6hI1VsLn0QZEjQZMEOWr+wKSMCQQCC4kXJEsHAve77oP6HtG/IiEn7
kpyUXRNvFsDE0czpJJBvL/aRFUJxuRK91jhjC68sA7NsKMGg5OXb5I5Jj36xAkEA
gIT7aFOYBFwGgQAQkWNKLvySgKbAZRTeLBacpHMuQdl1DfdntvAyqpAZ0lY0RKmW
G6aFKaqQfOXKCyWoUiVknQJAXrlgySFci/2ueKlIE1QqIiLSZ8V8OlpFLRnb1pzI
7U1yQXnTAEFYM560yJlzUpOb1V4cScGd365tiSMvxLOvTA==
-----END RSA PRIVATE KEY-----
```

All examples use this request:

```
POST /foo?param=value&pet=dog HTTP/1.1
Host: example.com
Date: Sun, 05 Jan 2014 21:31:40 GMT
Content-Type: application/json
Digest: SHA-256=X48E9qOokqqrvdts8nOJRJN3OWDUoyWxBf7kbu9DBPE=
Content-Length: 18

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

C.1. Default Test

If a list of headers is not included, the date is the only header that is signed by default for rsa-sha256. The string to sign would be:

```
date: Sun, 05 Jan 2014 21:31:40 GMT
```

The Authorization header would be:

```
Authorization: Signature keyId="Test",algorithm="rsa-sha256",
signature="SjWJWbWN7i0wzBvtPl8rbASWz5xQW6mcJmn+ibttBqtifLN7Sazz
6m79cNfwwb8DMJ5cou1s7uEGKKCs+FLEEaDV5lp7q25WqS+lavg7T8hc0GppauB
6hbgEKTwblDHYGEtbGmtdHgVCk9SuS13F0hZ8FD0k/5OxEPXe5WozsbM="
```

The Signature header would be:

```
Signature: keyId="Test",algorithm="rsa-sha256",
signature="SjWJWbWN7i0wzBvtPl8rbASWz5xQW6mcJmn+ibttBqtifLN7Sazz
6m79cNfwwb8DMJ5cou1s7uEGKKCs+FLEEaDV5lp7q25WqS+lavg7T8hc0GppauB
6hbgEKTwblDHYGEtbGmtdHgVCk9SuS13F0hZ8FD0k/5OxEPXe5WozsbM="
```

C.2. Basic Test

The minimum recommended data to sign is the (request-target), host, and date. In this case, the string to sign would be:

```
(request-target): post /foo?param=value&pet=dog
host: example.com
date: Sun, 05 Jan 2014 21:31:40 GMT
```

The Authorization header would be:

```
Authorization: Signature keyId="Test",algorithm="rsa-sha256",
  headers="(request-target) host date",
  signature="qdx+H7PHHDZgy4y/Ahn9Tny9V3GP6YgBPyUXMmoxWtLbHpUnXS
  2mg2+SbrQDMCJypxBLSPQR2aAjn7ndmw2iicw3HMbe8VfEdKFYRqzic+efkb3
  nndiv/x1xSHDJWeSWkx3ButlYSuBskLu6kd9Fswtemr3lgdDEmn04swr2Os0="
```

C.3. All Headers Test

A strong signature including all of the headers and a digest of the body of the HTTP request would result in the following signing string:

```
(request-target): post /foo?param=value&pet=dog
host: example.com
date: Sun, 05 Jan 2014 21:31:40 GMT
content-type: application/json
digest: SHA-256=X48E9qOokqqrvdts8nOJRJN3OWDUoyWxBf7kbu9DBPE=
content-length: 18
```

The Authorization header would be:

```
Authorization: Signature keyId="Test",algorithm="rsa-sha256",
  created=1402170695, expires=1402170699,
  headers="(request-target) (created) (expires)
    host date content-type digest content-length",
  signature="vSdrb+dS3EceC9bcwHSo4MlyKS59iFIrhgYkz8+oVLEEzmYZZvRs
    8rgOp+63LEM3v+MFHB32NfpB2bEKBIvB1q52LaEUHFv120V01IL+TAD48XaERZF
    ukWgHoBTLMhYS2Gb51gWxpeIq8knRmPnYePbF5MOkR0Zkly4zKH7s1dE="
```

The Signature header would be:

```
Signature: keyId="Test",algorithm="rsa-sha256",
  created=1402170695, expires=1402170699,
  headers="(request-target) (created) (expires)
    host date content-type digest content-length",
  signature="vSdrb+dS3EceC9bcwHSo4MlyKS59iFIrhgYkz8+oVLEEzmYZZvRs
    8rgOp+63LEM3v+MFHB32NfpB2bEKBIvB1q52LaEUHFv120V01IL+TAD48XaERZF
    ukWgHoBTLMhYS2Gb51gWxpeIq8knRmPnYePbF5MOkR0Zkly4zKH7s1dE="
```

Appendix D. Acknowledgements

The editor would like to thank the following individuals for feedback on and implementations of the specification (in alphabetical order): Mark Adamcin, Mark Allen, Paul Annesley, Karl Boehlmark, Stephane Bortzmeyer, Sarven Capadisli, Liam Dennehy, ductm54, Stephen Farrell, Phillip Hallam-Baker, Eric Holmes, Andrey Kislyuk, Adam Knight, Dave Lehn, Dave Longley, James H. Manger, Ilari Liusvaara, Mark Nottingham, Yoav Nir, Adrian Palmer, Lucas Pardue, Roberto Polli,

Julian Reschke, Michael Richardson, Wojciech Rygielski, Adam Scarr, Cory J. Slep, Dirk Stein, Henry Story, Lukasz Szewc, Chris Webber, and Jeffrey Yasskin

Appendix E. IANA Considerations

E.1. Signature Authentication Scheme

The following entry should be added to the Authentication Scheme Registry located at https://www.iana.org/assignments/http-authschemes [20]

Authentication Scheme Name: Signature
Reference: [RFC_THIS_DOCUMENT], Section 2.
Notes (optional): The Signature scheme is designed for clients to authenticate themselves with a server.

E.2. HTTP Signatures Algorithms Registry

The following initial entries should be added to the Canonicalization Algorithms Registry to be created and maintained at (the suggested URI) https://www.iana.org/assignments/shm-algorithms [21]:

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

Algorithm Name: hs2019
Status: active
Canonicalization Algorithm: [RFC_THIS_DOCUMENT], Section 2.3:
Signature String Construction [22]
Hash Algorithm: RFC 6234 [RFC6234], SHA-512 (SHA-2 with 512-bits of digest output)
Digital Signature Algorithm: Derived from metadata associated with `keyId`. Recommend support for RFC 8017 [RFC8017], Section 8.1: RSASSA-PSS, RFC 6234 [RFC6234], Section 7.1: SHA-Based HMACs, ANSI X9.62-2005 ECDSA, P-256, and RFC 8032 [RFC8032], Section 5.1: Ed25519ph, Ed25519ctx, and Ed25519.

Algorithm Name: rsa-sha1
Status: deprecated, SHA-1 not secure.
Canonicalization Algorithm: [RFC_THIS_DOCUMENT], Section 2.3:
Signature String Construction [23]

Hash Algorithm: RFC 6234 [RFC6234], SHA-1 (SHA-1 with 160-bits of digest output)
Digital Signature Algorithm: RFC 8017 [RFC8017], Section 8.2: RSASSA-PKCS1-v1_5

Algorithm Name: rsa-sha256
Status: deprecated, specifying signature algorithm enables attack vector.
Canonicalization Algorithm: [RFC_THIS_DOCUMENT], Section 2.3: Signature String Construction [24]
Hash Algorithm: RFC 6234 [RFC6234], SHA-256 (SHA-2 with 256-bits of digest output)
Digital Signature Algorithm: RFC 8017 [RFC8017], Section 8.2: RSASSA-PKCS1-v1_5

Algorithm Name: hmac-sha256
Status: deprecated, specifying signature algorithm enables attack vector.
Canonicalization Algorithm: [RFC_THIS_DOCUMENT], Section 2.3: Signature String Construction [25]
Hash Algorithm: RFC 6234 [RFC6234], SHA-256 (SHA-2 with 256-bits of digest output)
Message Authentication Code Algorithm: RFC 6234 [RFC6234], Section 7.1: SHA-Based HMACs

Algorithm Name: ecdsa-sha256
Status: deprecated, specifying signature algorithm enables attack vector.
Canonicalization Algorithm: [RFC_THIS_DOCUMENT], Section 2.3: Signature String Construction [26]
Hash Algorithm: RFC 6234 [RFC6234], SHA-256 (SHA-2 with 256-bits of digest output)
Digital Signature Algorithm: ANSI X9.62-2005 ECDSA, P-256

Authors' Addresses

Mark Cavage
Oracle
500 Oracle Parkway
Redwood Shores, CA 94065
US

Phone: +1 415 400 0626
Email: mcavage@gmail.com
URI: https://www.oracle.com/

Manu Sporny
Digital Bazaar
203 Roanoke Street W.
Blacksburg, VA 24060
US

Phone: +1 540 961 4469
Email: msporny@digitalbazaar.com
URI: https://manu.sporny.org/