Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 9, 2016

M. Cavage
Oracle
M. Sporny
Digital Bazaar

October 7, 2015

Signing HTTP Messages
draft-cavage-http-signatures-05

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 part of the Web Payments [1] work. Feedback related to this specification should be sent to public-webpayments@w3.org [2].

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 http://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 9, 2016.

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.

1.1. Using Signatures in HTTP Requests

It is common practice to protect sensitive website 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.

Digital signatures are widely used to provide authentication without the need for shared secrets. They also do not require a round-trip in order to authenticate the client. A server need only have 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.

This specification provides two mechanisms that can be used by a server to authenticate a client. The first is the 'Signature' HTTP Authentication Scheme, which may be used for interactive sessions. The second is the Signature HTTP Header, which is typically used by automated software agents.

1.2. Using Signatures in HTTP Responses

For high security transactions, having an additional signature on the HTTP header allows a client to ensure that even if the transport channel has been compromised, that the content of the messages have not been compromised. This specification provides a HTTP Signature Header mechanism that can be used by a client to authenticate the sender of a message and ensure that particular headers have not been modified in transit.

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 a digital signature.

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.

2.1.2. algorithm

REQUIRED. The `algorithm` parameter is used to specify the digital signature algorithm to use when generating the signature. Valid values for this parameter can be found in the Signature Algorithms registry located at http://www.iana.org/assignments/signature-algorithms and MUST NOT be marked "deprecated".

2.1.3. 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, the `Date` header, in the list of HTTP headers. Note that the list order is important, and MUST be specified in the order the HTTP header field-value pairs are concatenated together during signing.

2.1.4. signature

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

2.2. Ambiguous Parameters

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

2.3. Signature String Construction

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 Date header fields.

To include the HTTP request target in the signature calculation, use the special `(request-target)` 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 8.1.2.1 [5]).

2. 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 [6]). 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. Any other modification to the header field value MUST NOT be made.

3. 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) host date cache-control 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.
Cache-Control: max-age=60
Cache-Control: must-revalidate
```

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

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

2.4. Creating a Signature

In order to create a signature, a client MUST:

1. Use the contents of the HTTP message, the `headers` value, and the Signature String Construction algorithm to create the signature string.

2. The `algorithm` and key associated with `keyId` must then be used 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 was "rsa-sha256". This would signal to the application that the data associated with `keyId` is an RSA Private Key (as defined in RFC 3447 [RFC3447]), the signature string hashing function is SHA-256, and the signing algorithm is the one defined in RFC 3447 [RFC3447], Section Section 8.2.1 [7]. The result of the signature creation algorithm specified in RFC 3447 [RFC3447] 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 algorithm to recreate the signature string.

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.

For example, assume that the `algorithm` value was "rsa-sha256". This would signal to the application that the data associated with `keyId` is an RSA Public Key (as defined in RFC 3447 [RFC3447]), the signature string hashing function is SHA-256, and the `signature` verification algorithm to use to verify the signature is the one defined in RFC 3447 [RFC3447], Section Section 8.2.2 [8]. The result of the signature verification algorithm specified in RFC 3447 [RFC3447] 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. However, it does explicitly assume that clients can send an HTTP `Date` header.

3.1. Authorization Header

The client is expected to send an Authorization header (as defined in RFC 7235 [RFC7235], Section 4.1 [9]) 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 if 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 [10] 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 refers to a private key known to the client and a public key known to the server. The "hmac-key-1" keyId refers to key known to the client and server.

3.1.1. Initiating Signature Authorization

A server may notify a client when a protected resource could be accessed by authenticating itself to the server. To initiate this process, the server will request that the client authenticate itself via a 401 response 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) date"

...
```

3.1.2. RSA Example

The authorization header and signature would be generated as:

```
Authorization: Signature keyId="rsa-key-1",algorithm="rsa-sha256",
headers="(request-target) host date digest content-length",
signature="Base64(RSA-SHA256(signing string))"
```

The client would compose the signing string as:

```
(request-target): post /foo\n
host: example.org\n
date: Tue, 07 Jun 2014 20:51:35 GMT\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.

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

```
Authorization: Signature keyId="rsa-key-1",algorithm="rsa-sha256",
headers="(request-target) host date digest content-length",
signature="Base64(RSA-SHA256(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="hmac-sha256",
headers="(request-target) host date digest content-length",
signature="Base64(HMAC-SHA256(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
host: example.org\n
date: Tue, 07 Jun 2014 20:51:35 GMT\n
digest: SHA-256=X48E9qOokqqrvdts8nOJRJN3OWDUoyWxBf7kbu9DBPE=\n
content-length: 18
```

4. The 'Signature' HTTP Header

The "signature" HTTP Header is based on the model that the sender 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. However, it does explicitly assume that senders can send an HTTP `Date` header.

4.1. Signature Header

The sender is expected to transmit a header (as defined in RFC 7230 [RFC7230], Section 3.2 [11]) 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 [12]) where the "auth-param" parameters meet the requirements listed in Section 2: The Components of a Signature.

The rest if 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 refers to a private key known to the client and a public key known to the server. The "hmac-key-1" keyId refers to key known to the client and server.

4.1.1. RSA Example

The signature header and signature would be generated as:

```
Signature: keyId="rsa-key-1",algorithm="rsa-sha256",
headers="(request-target) host date digest content-length",
signature="Base64(RSA-SHA256(signing string))"
```

The client would compose the signing string as:

```
(request-target): post /foo\n
host: example.org\n
date: Tue, 07 Jun 2014 20:51:35 GMT\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.

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

```
Signature: keyId="rsa-key-1",algorithm="rsa-sha256",
headers="(request-target) host date digest content-length",
signature="Base64(RSA-SHA256(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="hmac-sha256",
headers="(request-target) host date digest content-length",
signature="Base64(HMAC-SHA256(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
host: example.org\n
date: Tue, 07 Jun 2014 20:51:35 GMT\n
digest: SHA-256=X48E9qOokqqrvdts8nOJRJN3OWDUoyWxBf7kbu9DBPE=\n
content-length: 18
```

5. References

5.1. Normative References

[I-D.ietf-jose-json-web-algorithms]
Jones, M., "JSON Web Algorithms (JWA)", draft-ietf-jose-json-web-algorithms-20 (work in progress), January 2014.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.

[RFC6376] Crocker, D., Hansen, T., and M. Kucherawy, "DomainKeys Identified Mail (DKIM) Signatures", STD 76, RFC 6376, September 2011.

[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, <http://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, <http://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, June 1999.

[RFC3230] Mogul, J. and A. Van Hoff, "Instance Digests in HTTP", RFC 3230, January 2002.

[RFC3447] Jonsson, J. and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1", RFC 3447, DOI 10.17487/RFC3447, February 2003, <http://www.rfc-editor.org/info/rfc3447>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

[RFC6749] Hardt, D., "The OAuth 2.0 Authorization Framework", RFC 6749, October 2012.

5.3. URIs

[2] http://tools.ietf.org/html/rfc4648#section-4

[3] http://tools.ietf.org/html/draft-ietf-httpbis-http2-13#section-8.1.2.1

[4] http://tools.ietf.org/html/rfc7230#section-3.2.4

[5] http://tools.ietf.org/html/rfc3447#section-8.2.1

[6] http://tools.ietf.org/html/rfc3447#section-8.2.2

[7] http://tools.ietf.org/html/draft-ietf-rfc7235-auth-25#section-4.1

[8] http://tools.ietf.org/html/rfc3230#section-4.3.2

[9] http://tools.ietf.org/html/rfc7230#section-3.2

[10] http://tools.ietf.org/html/rfc7235#section-4.1

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

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

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

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 [13].

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 [14] specification details how to use HTTP Signatures over a non-secured channel like HTTP and the HTTP Signature Trailers [15] 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) http://www.iana.org/assignments/http-auth-scheme-signature . 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

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: Thu, 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. The string to sign would be:

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

The Authorization header would be:

```
Authorization: Signature keyId="Test",algorithm="rsa-sha256",
signature="ATp0r26dbMIxOopqw0OfABDT7CKMIoENumuruOtarj8n/97Q3htH
FYpH8yOSQk3Z5zh8UxUym6FYTb5+A0Nz3NRsXJibnYi7brE/4tx5But9kkFGzG+
xpUmimN4c3TMN7OFH//+r8hBf7BT9/GmHDUVZT2JzWGLZES2xDOUuMtA="
```

The Signature header would be:

```
Signature: keyId="Test",algorithm="rsa-sha256",
signature="ATp0r26dbMIxOopqw0OfABDT7CKMIoENumuruOtarj8n/97Q3htH
FYpH8yOSQk3Z5zh8UxUym6FYTb5+A0Nz3NRsXJibnYi7brE/4tx5But9kkFGzG+
xpUmimN4c3TMN7OFH//+r8hBf7BT9/GmHDUVZT2JzWGLZES2xDOUuMtA="
```

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: Thu, 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="KcLSABBj/m3v2Dhxi
CKJmzYJvnx74tDO1SaURD8Dr8XpugN5wpy8iBVJtpkHUIp4qBYpzx2QvD16t8X
0BUMiKc53Age+baQFWwb2iYYJzvuUL+krrl/Q7H6fPBADBsHqEZ7IE8rR0Ys3l
b7J5A6VB9J/4yVTRiBcxTypW/mpr5w="
```

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: Thu, 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",
headers="(request-target) host date content-type digest content-length",
signature="jgSqYK0yKclIHfF9zdApVEbDp5eqj8C4i4X76pE+XHoxugXv7q
nVrGR+30bmBgtpR39I4utq17s9ghz/2QFVxlnToYAvbSVZJ9ulLd1HQBugO0j
Oyn9sXOtcN7uNHBjqNCqUsnt0sw/cJA6B6nJZpyNqNyAXKdxZZItOuhIs78w="
```

The Signature header would be:

```
Signature: keyId="Test",algorithm="rsa-sha256",
headers="(request-target) host date content-type digest content-length",
signature="jgSqYK0yKclIHfF9zdApVEbDp5eqj8C4i4X76pE+XHoxugXv7q
nVrGR+30bmBgtpR39I4utq17s9ghz/2QFVxlnToYAvbSVZJ9ulLd1HQBugO0j
Oyn9sXOtcN7uNHBjqNCqUsnt0sw/cJA6B6nJZpyNqNyAXKdxZZItOuhIs78w="
```

Appendix D. Acknowledgements

The editor would like to thank the following individuals for feedback on and implementations of the specification (in alphabetical order): Stephen Farrell, Phillip Hallam-Baker, Dave Lehn, Dave Longley, James H. Manger, Mark Nottingham, Yoav Nir, Julian Reschke, and Michael Richardson.

Appendix E. IANA Considerations

E.1. Signature Authentication Scheme

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

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. Signature Algorithm Registry

The following initial entries should be added to the Signature Algorithm Registry to be created and maintained at (the suggested URI) http://www.iana.org/assignments/signature-algorithms :

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: rsa-sha1
Reference: RFC 6376 [RFC6376], Section 3.3.1
Status: deprecated

Algorithm Name: rsa-sha256
Reference: RFC 6376 [RFC6376], Section 3.3.2
Status: active

Algorithm Name: hmac-sha256
Reference: HS256 in JOSE JSON Web Algorithms
[I-D.ietf-jose-json-web-algorithms], Section 3.2
Status: active

Algorithm Name: ecdsa-sha256

Reference: ES256 in JOSE JSON Web Algorithms
[I-D.ietf-jose-json-web-algorithms], Section 3.4
Status: active

Authors' Addresses

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

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

Manu Sporny
Digital Bazaar
1700 Kraft Drive
Suite 2408
Blacksburg, VA 24060
US

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