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

A. Backman, Ed.
Amazon

November 19, 2019

Signing HTTP Requests via JSON Web Signatures
draft-richanna-http-jwt-signature-00

Abstract

This document defines a method for generating and validating a digital signature or Message Authentication Code (MAC) over a set of protocol elements within an HTTP Request, using JSON Web Signatures (JWS).

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 May 22, 2020.

Table of Contents

1. Introduction

Digital signatures and MACs are popular cryptographic tools that can be used to address a variety of use cases, such as providing message integrity, or establishing proof of possession of a cryptographic key. While several digital signature algorithms exist, they generally share the constraint that any party wishing to validate a signature must have or be able to produce the exact byte sequence of the message that was signed. Consequently, it is non-trivial to create digital signatures over content that may undergo transformation, such as can occur with HTTP messages as they pass through proxies and software libraries in use by the sender or recipient.

This draft describes a method for generating and validating digital signatures or MACs over a set of protocol elements within an HTTP Request. This method consists of:

Mechanisms for identifying the protocol elements covered by the signature.

Mechanisms for creating canonical representations of protocol elements for the purpose of signing.

A mechanism creating and encoding a signature over those canonical representations using JSON Web Signatures (JWS) [RFC7515].

Many HTTP application frameworks reorder or insert extra headers, query parameters, and otherwise manipulate the HTTP request on its way from the web server into the application code itself. Such transformations may be applied by the sender and recipient, as well as any proxy through which the message passes. It is the goal of this draft to have a signature protection mechanism that is sufficiently robust against such deployment constraints while still providing sufficient security benefits.

This draft is concerned specifically with the generation, representation, and validation of signatures over elements within an HTTP request, with the expectation that this draft will be profiled by later drafts that seek to apply these signatures to address specific use cases within a larger application context. Consequently, key distribution, signing algorithm selection, and determination of which elements must be covered by the signature are all out of scope of this draft.

2. 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 RFC 2119 [RFC2119].

Other terms such as "client", "server", "HTTP request", and "protocol element" are inherited from HTTP [RFC7230].

This document uses the term 'sign' (or 'signature') to denote both a keyed message digest and a digital signature operation.

3. Generating a HTTP Request Signature Using JWS

This specification uses JSON Web Signature [RFC7515] to sign a set of protocol elements taken from an HTTP Request. When a JWS is created for this purpose, its ""typ"" header attribute MUST have the value ""http-sig"".

The JWS MUST be signed with a valid algorithm as defined in [RFC7518]. The "none" algorithm MUST NOT be used.

3.1. Generating the Payload of the JWS

The JWS Payload is a JSON object containing the data that will be covered by the signature. In order to include a protocol element within the covered data, its value must be represented within this JSON object. Some elements are represented directly, by setting the value of a member in the object to the element's value in the HTTP Request. Others are included indirectly, by setting the value of a member in the object to a cryptographic hash or other value derived from the element's value in the HTTP Request.

The below list defines the means of inclusion of various protocol elements, including the JSON object member that MUST be used when including the element, and how the element's value should be included. When present, each of these members MUST be a top-level member of the JSON object.

The JSON object MAY contain other top-level members. The syntax and semantics of members not listed below are out of scope of this specification. Implementations SHOULD consider a signature invalid if the JSON object contains members that the implementation does not understand.

ts RECOMMENDED. The timestamp. This integer provides replay protection of the signed JSON object. Its value MUST be a number containing an integer value representing number of whole integer seconds from midnight, January 1, 1970 GMT.

m OPTIONAL. The HTTP Method used to make this request. This MUST be the uppercase HTTP verb as a JSON string.

u OPTIONAL. The HTTP URL host component as a JSON string. This MAY include the port separated from the host by a colon in host:port format.

p OPTIONAL. The HTTP URL path component of the request as an HTTP string.

q OPTIONAL. The hashed HTTP URL query parameter map of the request as a two-part JSON array. The first part of this array is a JSON array listing all query parameters that were used in the calculation of the hash in the order that they were added to the hashed value as described below. The second part of this array is a JSON string containing the Base64URL encoded hash itself, calculated as described below.

h OPTIONAL. The hashed HTTP request headers as a two-part JSON array. The first part of this array is a JSON array listing all

headers that were used in the calculation of the hash in the order that they were added to the hashed value as described below. The second part of this array is a JSON string containing the Base64URL encoded hash itself, calculated as described below.

b OPTIONAL. The base64URL encoded hash of the HTTP Request body, calculated as the SHA256 of the byte array of the body

All hashes SHALL be calculated using the SHA256 algorithm.

3.2. Calculating the query parameter list and hash

To generate the query parameter list and hash, the signer creates two data objects: an ordered list of strings to hold the query parameter names and a string buffer to hold the data to be hashed.

The signer iterates through all query parameters in whatever order it chooses and for each query parameter it does the following:

1. Adds the name of the query parameter to the end of the list.

2. Percent-encodes the name and value of the parameter as specified in [RFC3986]. Note that if the name and value have already been percent-encoded for transit, they are not re-encoded for this step.

3. Encodes the name and value of the query parameter as "name=value" and appends it to the string buffer separated by the ampersand "&" character.

Repeated parameter names are processed separately with no special handling. Parameters MAY be skipped by the client if they are not required (or desired) to be covered by the signature.

The signer then calculates the hash over the resulting string buffer. The list and the hash result are added to a list as the value of the "q" member.

For example, the query parameter set of "b=bar", "a=foo", "c=duck" is concatenated into the string:

b=bar&a=foo&c=duck

When added to the JSON structure using this process, the results are:

"q": [["b", "a", "c"], "u4LgkGUWhP9MsKrEjA4dizIllDXluDku6ZqCeyuR-JY"]

3.3. Calculating the header list and hash

To generate the header list and hash, the signer creates two data objects: an ordered list of strings to hold the header names and a string buffer to hold the data to be hashed.

The signer iterates through all query parameters in whatever order it chooses and for each query parameter it does the following:

1. Lowercases the header name.

2. Adds the name of the header to the end of the list.

3. Encodes the name and value of the header as "name: value" and appends it to the string buffer separated by a newline "\n" character.

Repeated header names are processed separately with no special handling. Headers MAY be skipped by the client if they are not required (or desired) to be covered by the signature.

The signer then calculates the hash over the resulting string buffer. The list and the hash result are added to a list as the value of the "h" member.

For example, the headers "Content-Type: application/json" and "Etag: 742-3u8f34-3r2nvv3" are concatenated into the string:

```
content-type: application/json
etag: 742-3u8f34-3r2nvv3

"h": [["content-type", "etag"],
  "bZA981YJBrPlIzOvplbu3e7ueREXXr38vSkxIBYOaxI"]
```

4. Validating the HTTP Request Signature

Validation of the signature is done using normal JWS validation for the signature and key type. Additionally, in order to trust any of the hashed components of the HTTP request, the validator MUST re-create and verify a hash for each component as described below. This process is a mirror of the process used to create the hashes in the first place, with a mind toward the fact that order may have changed and that elements may have been added or deleted. The protected resource MUST similarly compare the replicated values included in various JSON fields with the corresponding actual values from the request. Failure to do so will allow an attacker to modify the underlying request while at the same time having the application layer verify the signature correctly.

4.1. Validating the query parameter list and hash

The validator has at its disposal a map that indexes the query parameter names to the values given. The validator creates a string buffer for calculating the hash. The validator then iterates through the "list" portion of the "p" parameter. For each item in the list (in the order of the list) it does the following:

1. Fetch the value of the parameter from the HTTP request query parameter map. If a parameter is found in the list of signed parameters but not in the map, the validation fails.

2. Percent-encodes the name and value of the parameter as specified in [RFC3986]. Note that if the name and value have already been percent-encoded for transit, they are not re-encoded for this step.

3. Encode the parameter as "name=value" and concatenate it to the end of the string buffer, separated by an ampersand character.

The validator calculates the hash of the string buffer and base64url encodes it. The protected resource compares that string to the string passed in as the hash. If the two match, the hash validates, and all named parameters and their values are considered covered by the signature.

There MAY be additional query parameters that are not listed in the list and are therefore not covered by the signature. The validator MUST decide whether or not to accept a request with these uncovered parameters.

4.2. Validating the header list and hash

The validator has at its disposal a map that indexes the header names to the values given. The validator creates a string buffer for calculating the hash. The validator then iterates through the "list" portion of the "h" parameter. For each item in the list (in the order of the list) it does the following:

1. Fetch the value of the header from the HTTP request header map. If a header is found in the list of signed parameters but not in the map, the validation fails.

2. Encode the parameter as "name: value" and concatenate it to the end of the string buffer, separated by a newline character.

The validator calculates the hash of the string buffer and base64url encodes it. The protected resource compares that string to the

string passed in as the hash. If the two match, the hash validates, and all named headers and their values are considered covered by the signature.

There MAY be additional headers that are not listed in the list and are therefore not covered by the signature. The validator MUST decide whether or not to accept a request with these uncovered headers.

5. IANA Considerations

5.1. JSON Web Signature and Encryption Type Values Registration

This specification registers the "http-sig" type value in the IANA JSON Web Signature and Encryption Type Values registry [RFC7515]:

- "typ" Header Parameter Value: "http-sig"
- Abbreviation for MIME Type: None
- Change Controller: IETF
- Specification Document(s): [[this document]]

6. Security Considerations

6.1. Offering Confidentiality Protection for Access to Protected Resources

This specification can be used with and without Transport Layer Security (TLS).

Without TLS this protocol provides a mechanism for verifying the integrity of requests, it provides no confidentiality protection. Consequently, eavesdroppers will have full access to communication content and any further messages exchanged between the client and the server. This could be problematic when data is exchanged that requires care, such as personal data.

When TLS is used then confidentiality of the transmission can be ensured between endpoints, including both the request and the response. The use of TLS in combination with the signed HTTP request mechanism is highly recommended to ensure the confidentiality of the data returned from the protected resource.

6.2. Plaintext Storage of Credentials

The mechanism described in this document works in a similar way to many three-party authentication and key exchange mechanisms. In order to compute the signature over the HTTP request, the client must have access to the decryption key in plaintext form. If an attacker were to gain access to these stored secrets at the client or (in case of symmetric keys) at the server they would be able to forge signatures for any HTTP request they wished, effectively allowing them to impersonate the client.

It is therefore paramount to the security of the protocol that any private or symmetric keys used to sign HTTP requests are protected from unauthorized access.

6.3. Entropy of Keys

Unless TLS is used between the client and the resource server, eavesdroppers will have full access to requests sent by the client. They will thus be able to mount off-line brute-force attacks to attempt recovery of the session key or private key used to compute the keyed message digest or digital signature, respectively.

Key generation and distribution is out of scope for this document. It is the responsibility of users of this specification to ensure that keys are generated with sufficient entropy and rotated at an appropriate frequency to sufficiently mitigate the risk of such attacks, as appropriate for their use case.

6.4. Denial of Service

This specification includes a number of features which may make resource exhaustion attacks against servers possible. For example, server may need to consult back-end databases or other servers in order to verify a signature, or the cryptographic overhead may present a significant burden on the server. An attacker could leverage this overhead to attempt a denial of service attack by sending a large number of invalid requests to the server, causing the server to expend significant resources checking invalid signatures. This attack vector must be taken into consideration when implementing or deploying this specification.

6.5. Validating the integrity of HTTP message

This specification provides flexibility for selectively validating the integrity of the HTTP request, including header fields, query parameters, and message bodies. Since all components of the HTTP request are only optionally validated by this method, and even some

components may be validated only in part (e.g., some headers but not others) it is up to developers to verify that any vital parameters in a request are actually covered by the signature. Failure to do so could allow an attacker to inject vital parameters or headers into the request, ouside of the protection of the signature.

The application verifying this signature MUST NOT assume that any particular parameter is appropriately covered by the signature unless it is included in the signed structure and the hash is verified. Any applications that are sensitive of header or query parameter order MUST verify the order of the parameters on their own. The application MUST also compare the values in the JSON container with the actual parameters received with the HTTP request (using a direct comparison or a hash calculation, as appropriate). Failure to make this comparison will render the signature mechanism useless for protecting these elements.

The behavior of repeated query parameters or repeated HTTP headers is undefined by this specification. If a header or query parameter is repeated on either the outgoing request from the client or the incoming request to the protected resource, that query parameter or header name MUST NOT be covered by the hash and signature.

This specification records the order in which query parameters and headers are hashed, but it does not guarantee that order is preserved between the client and protected resource. If the order of parameters or headers are significant to the underlying application, it MUST confirm their order on its own, apart from the signature and HTTP message validation.

7. Privacy Considerations

This specification addresses machine to machine communications and raises no privacy considerations beyond existing HTTP interactions.

8. Acknowledgements

The authors thank the OAuth Working Group for input into this work.

In particular, the authors thank Justin Richer for his work on [I-D.ietf-oauth-signed-http-request], on which this specification is based.

9. Normative References

[I-D.ietf-oauth-signed-http-request]
Richer, J., Bradley, J., and H. Tschofenig, "A Method for Signing HTTP Requests for OAuth", draft-ietf-oauth-signed-http-request-03 (work in progress), August 2016.

[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>.

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

[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>.

[RFC7515] 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>.

[RFC7518] Jones, M., "JSON Web Algorithms (JWA)", RFC 7518, DOI 10.17487/RFC7518, May 2015, <https://www.rfc-editor.org/info/rfc7518>.

Author's Address

Annabelle Backman (editor)
Amazon

Email: richanna@amazon.com