OAuth Working Group J. Richer, Ed.
Internet-Draft
Intended status: Experimental J. Bradley
Expires: September 10, 2015 Ping Identity
H. Tschofenig
ARM Limited
March 09, 2015

A Method for Signing an HTTP Requests for OAuth
draft-ietf-oauth-signed-http-request-01.txt

Abstract

This document a method for offering data origin authentication and integrity protection of HTTP requests. To convey the relevant data items in the request a JSON-based encapsulation is used and the JSON Web Signature (JWS) technique is re-used. JWS offers integrity protection using symmetric as well as asymmetric cryptography.

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 September 10, 2015.

Copyright Notice

Copyright (c) 2015 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction 2
2. Terminology 3
3. Generating a JSON Object from an HTTP Request 3
  3.1. Selection of a hashing algorithm and size 4
  3.2. Calculating the query parameter list and hash 4
  3.3. Calculating the header list and hash 5
4. Verifying the Hashes 5
  4.1. Validating the query parameter list and hash 6
  4.2. Validating the header list and hash 6
5. Example 7
6. IANA Considerations 7
  6.1. The 'pop' OAuth Access Token Type 7
  6.2. JSON Web Signature and Encryption Type Values Registration 8
7. Security Considerations 8
  7.1. Offering Confidentiality Protection for Access to Protected Resources 8
  7.2. Authentication of Resource Servers 8
  7.3. Plaintext Storage of Credentials 9
  7.4. Entropy of Keys 9
  7.5. Denial of Service 9
  7.6. Protecting HTTP Header Fields 10
8. Acknowledgements 10
9. References 10
  9.1. Normative References 10
  9.2. Informative References 11
Authors' Addresses 11

1. Introduction

In order to protect an HTTP request with a signature, a method for conveying various parameters and to compute a signature is needed. Ideally, this should be done without replicating the information already present in the HTTP request. This version of the document still replicates most of the headers though.

The keying material required for this signature calculation is distributed via mechanisms described in companion documents (see [I-D.bradley-oauth-pop-key-distribution] and [I-D.hunt-oauth-pop-architecture]). The JSON Web Signature (JWS) specification [I-D.ietf-jose-json-web-signature] is re-used for

computing a digital signature (which uses asymmetric cryptography) or a keyed message digest (in case of symmetric cryptography).

The scope of the mechanism described in this document is shown in Figure 1 where a client in possession of keying material that is tied to the access token creates a JSON object, signs it, and issues an request to a resource server for access to a protected resource.

```
 +-----------+                                    +------------+
 |           |--(1)- HTTP Request               ->| Resource   |
 | Client    |       (+Signature, +Access Token)->| Server     |
 |           |                                    |            |
 |           |<-(2)- HTTP Response ---------------|            |
 +-----------+                                    +------------+
```

Figure 1: Message Flow.

Many HTTP application frameworks insert extra headers, query parameters, and otherwise manipulate the HTTP request on its way from the web server into the application code itself. 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).

The method of conveying the token and signed request to the protected resource server is undefined by this document, but [RFC6750] could be re-used.

The mechanism described in this document does not provide authentication of the resource server to the client. This version of the document does not provide a cryptographic binding to Transport Layer Security (TLS) used underneath the an HTTPS request.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

We use the term 'sign' (or 'signature') to denote both a keyed message digest and a digital signature operation.

3. Generating a JSON Object from an HTTP Request

This section describes how to generate a JSON object below is included as a member of the JSON object. All members are OPTIONAL.

m	The HTTP Method used to make this request. This MUST be the uppercase HTTP verb as a JSON string.
u	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	The HTTP URL path component of the request as an HTTP string.
q	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	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	The base64URL encoded hash of the HTTP Request body, calculated as the HMAC of the byte array of the body.
ts	The "ts" (timestamp) element provides replay protection of the JSON object. Its value MUST be a number containing an IntDate value representing number of whole integer seconds from midnight, January 1, 1970 GMT.

3.1. Selection of a hashing algorithm and size

The hashes SHALL be calculated using the HMAC algorithm using a hash size equal to the size of the surrounding JWT's alg header field. That is, if the JWT uses HS256 or RS256, the HMAC here uses a 256-bit HMAC. If the JWT uses RS512, the HMAC here uses 512-bit HMAC, and so forth.

3.2. Calculating the query parameter list and hash

To generate the query parameter list and hash, the client 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 client 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. Encodes the name and value of the query parameter as "name=value" and appends it to the string buffer. [[Separated by an ampersand? Alternatively we could have this also pulled into an ordered list and post-process the concatenation, but that might be too deep into the weeds.]]

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 client then calculates the HMAC hash over the resulting string buffer. The list and the hash result are added as the value of the "p" member.

3.3. Calculating the header list and hash

To generate the header list and hash, the client 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 client 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 header to the end of the list.
2. Encodes the name and value of the header as "name: value" and appends it to the string buffer. [[Separated by a newline? Alternatively we could have this also pulled into an ordered list and post-process the concatenation, but that might be too deep into the weeds.]]

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

4. Verifying the Hashes

Validation of the overall signature is done using the standard JWS mechanisms for JSON structures. However, in order to trust any of the hashed mechanisms above, an application MUST re-create and verify a hash for each component. Additionally, an application MUST compare the replicated values included in various JSON fields with the actual

header fields of the request. Failure to-do so will allow an attacker to modify the underlying request, connect do different resources while at the same time having the application layer verify the signature correctly.

4.1. Validating the query parameter list and hash

The client has at its disposal a map that indexes the query parameter names to the values given. The client creates a string buffer for calculating the hash. The client 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 parameter map. If a parameter 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. [[same separator issue as above.]]

The client calculates the hash of the string buffer and base64url encodes it. The client 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 client MUST decide whether or not to accept a request with these uncovered parameters.

4.2. Validating the header list and hash

The client has at its disposal a map that indexes the header names to the values given. The client creates a string buffer for calculating the hash. The client 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. [[same separator issue as above.]]

The client calculates the hash of the string buffer and base64url encodes it. The client 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 client MUST decide whether or not to accept a request with these uncovered headers.

5. Example

Example goes in here but will look like something like this (symmetric key case).

1) HTTP Request (plain)

```
POST /request?b5=%3D%253D&a3=a&c%40=&a2=r%20b&c2 HTTP/1.1
Host: example.com
```

2) JWS protected JSON object

```
{"typ":"pop",
 "alg":"HS256",
 "kid":"client12345@example.com"}
.
{"m":"POST",
 "u":"example.com",
 "p":"request",
 "q":[["a3", "b5", "a2"], "m2398f32i2o3roiu2313aa"],
 "ts":1300819380
}
.
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

Figure 2: Message Flow.

6. IANA Considerations

6.1. The 'pop' OAuth Access Token Type

Section 11.1 of [RFC6749] defines the OAuth Access Token Type Registry and this document adds another token type to this registry.

Type name: pop

Additional Token Endpoint Response Parameters: (none)

HTTP Authentication Scheme(s): Proof-of-possession access token for use with OAuth 2.0

Change controller: IETF

Specification document(s): [[this document]]

6.2. JSON Web Signature and Encryption Type Values Registration

This specification registers the "pop" type value in the IANA JSON Web Signature and Encryption Type Values registry [I-D.ietf-jose-json-web-signature]:

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

7. Security Considerations

7.1. 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 resource server. This could be problematic when data is exchanged that requires care, such as personal data.

When TLS is used then confidentiality can be ensured; this version of the specification does, however, not provide the TLS channel binding feature, which ensures that the TLS channel is cryptographically bound to the application layer protocol authentication defined in this document.

The use of TLS in combination with the signed HTTP request mechanism is highly recommended to ensure the confidentiality of the user's data.

7.2. Authentication of Resource Servers

This protocol allows clients to verify the authenticity of resource servers only when TLS is used. With TLS the resource server is authenticated as part of the TLS handshake. The mechanism described

in this document does not provide any mechanism for the client to authenticate the resource server at the application layer.

7.3. Plaintext Storage of Credentials

The mechanism described in this document works similar 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 a key bound to the access token (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 resource server he or she would be able to perform any action on behalf of any client.

It is therefore paramount to the security of the protocol that the private keys associated with the access tokens are protected from unauthorized access.

7.4. 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 recover the session key or private key used to compute the keyed message digest or digital signature, respectively.

This specification assumes that the keying material for use with the described HTTP signing mechanism has been distributed via other mechanisms, such as [I-D.bradley-oauth-pop-key-distribution]. Hence, it is the responsibility of the authorization server and or the client to be careful when generating fresh and unique keys with sufficient entropy to resist such attacks for at least the length of time that the session keys (and the access tokens) are valid.

For example, if the key bound to the access token is valid for one day, authorization servers must ensure that it is not possible to mount a brute force attack that recovers that key in less than one day. Of course, servers are urged to err on the side of caution, and use the longest key length reasonable.

7.5. Denial of Service

This specification includes a number of features which may make resource exhaustion attacks against resource servers possible. For example, a resource server may need to need to the resource server has to process the incoming request, verify the access token, perform signature verification, and might have (in certain circumstances)

consult back-end databases or the authorization server before granting access to the protected resource.

An attacker may exploit this to perform a denial of service attack by sending a large number of invalid requests to the server. The computational overhead of verifying the keyed message digest alone is, however, not sufficient to mount a denial of service attack since keyed message digest functions belong to the computationally fastest cryptographic algorithms. The situation may, however, be different when using asymmetric cryptography, which is also supported by the JWS.

7.6. Protecting HTTP Header Fields

This specification provides flexibility for selectively protecting header fields and even the body of the message. Since all components of the HTTP request are only optionally protected by this method, and even some components may be protected only in part (e.g., some headers but not others) it is up to application developers to verify that any parameters in a request are actually covered by the signature.

The application verifying this signature MUST NOT assume that any particular parameter is appropriately covered by the signature. 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. Failure to make this comparison will render the signature mechanism useless.

8. Acknowledgements

The authors acknowledge the OAuth Working Group and submit this draft for feedback and input into the ongoing work of signed HTTP requests for the interaction between clients and resource servers.

9. References

9.1. Normative References

[I-D.ietf-jose-json-web-signature]
Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", draft-ietf-jose-json-web-signature-41 (work in progress), January 2015.

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

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

[RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, October 2012.

9.2. Informative References

[I-D.bradley-oauth-pop-key-distribution]
Bradley, J., Hunt, P., Jones, M., and H. Tschofenig, "OAuth 2.0 Proof-of-Possession: Authorization Server to Client Key Distribution", draft-bradley-oauth-pop-key-distribution-01 (work in progress), June 2014.

[I-D.hunt-oauth-pop-architecture]
Hunt, P., Richer, J., Mills, W., Mishra, P., and H. Tschofenig, "OAuth 2.0 Proof-of-Possession (PoP) Security Architecture", draft-hunt-oauth-pop-architecture-02 (work in progress), June 2014.

Authors' Addresses

Justin Richer (editor)

Email: ietf@justin.richer.org

John Bradley
Ping Identity

Email: ve7jtb@ve7jtb.com
URI: http://www.thread-safe.com/

Hannes Tschofenig
ARM Limited
Austria

Email: Hannes.Tschofenig@gmx.net
URI: http://www.tschofenig.priv.at