OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: February 9, 2017

J. Richer, Ed.

J. Bradley
Ping Identity

H. Tschofenig
ARM Limited

August 08, 2016

A Method for Signing HTTP Requests for OAuth
draft-ietf-oauth-signed-http-request-03

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 February 9, 2017.

Table of Contents

1. Introduction 2
2. Terminology 3
3. Generating a JSON Object from an HTTP Request 3
3.1. Calculating the query parameter list and hash 4
3.2. Calculating the header list and hash 5
4. Sending the signed object 6
4.1. HTTP Authorization header 6
4.2. HTTP Form body 6
4.3. HTTP Query parameter 7
5. Validating the request 7
5.1. Validating the query parameter list and hash 7
5.2. Validating the header list and hash 8
6. IANA Considerations 9
6.1. The 'pop' OAuth Access Token Type 9
6.2. JSON Web Signature and Encryption Type Values Registration 9
7. Security Considerations 9
7.1. Offering Confidentiality Protection for Access to Protected Resources 9
7.2. Plaintext Storage of Credentials 10
7.3. Entropy of Keys 10
7.4. Denial of Service 10
7.5. Validating the integrity of HTTP message 11
8. Privacy Considerations 12
9. Acknowledgements 12
10. Normative References 12
Authors' Addresses 13

1. Introduction

In order to prove possession of an access token and its associated key, an OAuth 2.0 client needs to compute some cryptographic function and present the results to the protected resource as a signature. The protected resource then needs to verify the signature and compare that to the expected keys associated with the access token. This is in addition to the normal token protections provided by a bearer token [RFC6750] and transport layer security (TLS).

Furthermore, it is desirable to bind the signature to the HTTP request. Ideally, this should be done without replicating the information already present in the HTTP request more than required. However, 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 key required for this signature calculation is distributed via mechanisms described in companion documents (see [I-D.ietf-oauth-pop-key-distribution] and [I-D.ietf-oauth-pop-architecture]). The JSON Web Signature (JWS) specification [RFC7515] is used for computing a digital signature (which uses asymmetric cryptography) or a keyed message digest (in case of symmetric cryptography).

The mechanism described in this document assumes that a client is in possession of an access token and asociated key. That client then creates a JSON object including the access token, signs the JSON object using JWS, and issues an request to a resource server for access to a protected resource using the signed object as its authorization. The protected resource validates the JWS signature and parses the JSON object to obtain token information.

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", "authorization server", "access token", and "protected resource" are inherited from OAuth 2.0 [RFC6749].

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 specification uses JSON Web Signatures [RFC7515] to protect the access token and, optionally, parts of the request.

This section describes how to generate a JSON [RFC7159] object from the HTTP request. Each value below is included as a member of the JSON object at the top level.

at REQUIRED. The access token value. This string is assumed to have no particular format or structure and remains opaque to the client.

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. [[Note to WG: do we want crypto agility here? If so how do we signal this]]

The JSON object is signed using the algorithm appropriate to the associated access token key, usually communicated as part of key distribution [I-D.ietf-oauth-pop-key-distribution].

3.1. 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. 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 client 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"]

3.2. 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. 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 client 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. Sending the signed object

In order to send the signed object to the protected resource, the client includes it in one of the following three places.

4.1. HTTP Authorization header

The client SHOULD send the signed object to the protected resource in the Authorization header. The value of the signed object in JWS compact form is appended to the Authorization header as a PoP value. This is the preferred method. Note that if this method is used, the Authorization header MUST NOT be included in the protected elements of the signed object.

```
GET /resource/foo
Authorization: PoP eyJ....omitted for brevity...
```

4.2. HTTP Form body

If the client is sending the request as a form-encoded HTTP message with parameters in the body, the client MAY send the signed object as part of that form body. The value of the signed object in JWS compact form is sent as the form parameter pop_access_token. Note that if this method is used, the body hash cannot be included in the protected elements of the signed object.

```
POST /resource
Content-type: application/www-form-encoded

pop_access_token=eyJ....omitted for brevity...
```

4.3. HTTP Query parameter

If neither the Authorization header nor the form-encoded body parameter are available to the client, the client MAY send the signed object as a query parameter. The value of the signed object in JWS compact form is sent as the query parameter pop_access_token. Note that if this method is used, the pop_access_token parameter MUST NOT be included in the protected elements of the signed object.

GET /resource?pop_access_token=eyJ....

5. Validating the request

Just like with a bearer token [RFC6750], while the access token value included in the signed object is opaque to the client, it MUST be understood by the protected resource in order to fulfill the request. Also like a bearer token, the protected resource traditionally has several methods at its disposal for understanding the access token. It can look up the token locally (such as in a database), it can parse a structured token (such as JWT [RFC7519]), or it can use a service to look up token information (such as introspection [RFC7662]). Whatever method is used to look up token information, the protected resource MUST have access to the key associated with the access token, as this key is required to validate the signature of the incoming request. 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 protected resource 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.

5.1. 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 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 client 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 client MUST decide whether or not to accept a request with these uncovered parameters.

5.2. 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, separated by a newline character.

The client 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 client MUST decide whether or not to accept a request with these uncovered headers.

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 [RFC7515]:

- "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 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.

7.2. 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 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 they would be able to perform any action on behalf of any client just as if they had stolen a bearer token.

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

7.3. 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.

This specification assumes that the key used herein has been distributed via other mechanisms, such as [I-D.ietf-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 possible within reason.

7.4. 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 process the incoming request, verify the access token, perform signature verification, and might (in certain circumstances) have to consult back-end databases or the

authorization server before granting access to the protected resource. Many of these actions are shared with bearer tokens, but the additional cryptographic overhead of validating the signed request needs to be taken into consideration with deployment of this specification.

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 not likely sufficient to mount a denial of service attack. To help combat this, it is RECOMMENDED that the protected resource validate the access token (contained in the "at" member of the signed structure) before performing any cryptographic verification calculations.

7.5. 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 protected resource 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.

8. Privacy Considerations

This specification addresses machine to machine communications and raises no privacy considerations beyond existing OAuth transactions.

9. Acknowledgements

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

10. Normative References

[I-D.ietf-oauth-pop-architecture]
Hunt, P., Richer, J., Mills, W., Mishra, P., and H. Tschofenig, "OAuth 2.0 Proof-of-Possession (PoP) Security Architecture", draft-ietf-oauth-pop-architecture-08 (work in progress), July 2016.

[I-D.ietf-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-ietf-oauth-pop-key-distribution-02 (work in progress), October 2015.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://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, <http://www.rfc-editor.org/info/rfc3986>.

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

[RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, <http://www.rfc-editor.org/info/rfc6750>.

[RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, <http://www.rfc-editor.org/info/rfc7159>.

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

[RFC7519] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <http://www.rfc-editor.org/info/rfc7519>.

[RFC7662] Richer, J., Ed., "OAuth 2.0 Token Introspection", RFC 7662, DOI 10.17487/RFC7662, October 2015, <http://www.rfc-editor.org/info/rfc7662>.

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