OAuth 2.0 Security: OAuth Open Redirector
draft-bradley-oauth-open-redirector-01.txt

Abstract

This document gives additional security considerations for OAuth, beyond those in the OAuth 2.0 specification and in the OAuth 2.0 Threat Model and Security Considerations.

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 24, 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
   1.1. Notational Conventions ................................. 2
   1.2. Terminology ............................................. 3
2. Authorization Server Error Response ......................... 3
   2.1. Abuse: The Authorization Server As Open Redirector . . . . 3
   2.2. Security Compromise: The Authorization Server As Open Redirector .............................................. 4
   2.3. Mitigation .................................................. 5
3. Acknowledgements ............................................... 6
4. Normative References .......................................... 6
Appendix A. Document History ................................. 7
Authors' Addresses .................................................... 7

1. Introduction

This document gives additional security considerations for OAuth, beyond those in the OAuth 2.0 specification [RFC6749] and in the OAuth 2.0 Threat Model and Security Considerations [RFC6819]. In particular focuses its attention on the risk of abuse the Authorization Server (AS) (Section 1.2) as an open redirector.

It contains the following content:

- Describes the Authorization Server Error Response as defined in [RFC6749].
- Describes the risk of abuse the Authorization Server as an open redirector.
- Gives some mitigation details on how to hinder the risk of open redirector in the AS.

1.1. Notational Conventions

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].

Unless otherwise noted, all the protocol parameter names and values are case sensitive.
1.2. Terminology

Authorization Server (AS)
The server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization.

Redirection endpoint
Used by the authorization server to return responses containing authorization credentials to the client via the resource owner user-agent.

2. Authorization Server Error Response

The OAuth 2.0 specification [RFC6749] defines the Error Response associated with the Authorization Code Grant flow and the Implicit Grant flow. Both flows use a redirection endpoint where the resource owner’s user agent is directed after the resource owner has completed interacting with the authorization server. The redirection endpoint is also used in the error response scenario. As per RFC6749 Section 4.1.2.1 and 4.2.2.1 [RFC6749] if the resource owner denies the access request or if the request fails for reasons other than a missing or invalid redirection URI, the ?AS? redirects the user-agent by sending the following HTTP response:

HTTP/1.1 302 Found Location: https://client.example.com/cb?error=access_denied

2.1. Abuse: The Authorization Server As Open Redirector

As described in [RFC6819] an attacker could utilize a user’s trust in an ?AS? to launch a phishing attack. The attack described here though is not mitigated using the countermeasures listed in [RFC6819]. In this scenario the attacker:

- Performs a client registration as per the core specification [RFC6749]. The provided redirection URI is a malicious one e.g. https://attacker.com (namely the one where the victim’s user agent will land without any validation)

- Prepare a forged URI using the assumption that the ?AS? complies with the OAuth 2.0 specification [RFC6749]. In particular with the ?AS? Error Response described in the previous section (Section 2). As an example he can use a wrong or not existing scope e.g.

- Attempt the phishing attack trying to have the victim clicking the forged URI prepared on the previous step. Should the attack succeeds the victim’s user agent is redirected to https://attacker.com (all with any user interaction) The HTTP referer header will be set to the AS domain perhaps allowing manipulation of the user.

2.2. Security Compromise: The Authorization Server As Open Redirector

The attacker can use a redirect error redirection to intercept redirect based protocol messages via the Referer header and URI fragment. In this scenario the attacker:

- Performs a registration of a malicious client as per the core specification [RFC6749]. The provided redirection URI is a malicious one e.g. https://attacker.com (This URI will capture the fragment and referer header sent as part of the error)

- Creates an invalid Authentication request URI for the malicious client. As an example he can use a wrong or not existing scope e.g.


- If the AS supports sticky grants (not re-prompting for consent based on a previous grant) a valid authentication request for the user may also be used to trigger a 30x redirect.

- Performs a OAuth Authorization request using the invalid Authorization request as the redirect_uri. This works if the AS is pattern matching redirect_uri and has a public client that shares the same domain as the AS.

(line breaks for display only)
https://AUTHORIZATION_SERVER/authorize?response_type=token
&client_id=good-client&scope=VALID_SCOPE
&redirect_uri=https%3A%2F%2AUTHORIZATION_SERVER%Fauthorize
%3Fresponse_type%3Dcode
%26client_id%3Dattacker-client-id
%26scope%3DINVALID_SCOPE
%26redirect_uri%3Dhttps%253A%252F%252Fattacker.com

Figure 1

o Receive the response redirected to https://attacker.com

The legitimate OAuth Authorization response will include an access token in the URI fragment.

Most web browsers will append the fragment to the URI sent in the location header of a 302 response if no fragment is included in the location URI.

If the Authorization request is code instead of token, the same technique is used, but the code is leaked by the browser in the referer header rather than the fragment.

This causes the access token from a successful authorization to be leaked across the redirect to the malicious client. This is due to browser behaviour and not because the AS has included any information in the redirect URI other than the error code.

Protocols other than OAuth may be particularly vulnerable to this if they are only verifying the domain of the redirect. Performing exact redirect URI matching in OAuth will protect the AS, but not other protocols.

It should be noted that a legitimate OAuth client registered with a AS might be compromised and used as a redirect target by an attacker, perhaps without the knowledge of the client site. This increases a the attack surface for a ?AS?.

2.3. Mitigation

In order to defend against the attacks described in Section 2.1 and Section 2.2 the ?AS? can either:

o Respond with an HTTP 400 (Bad Request) status code.

o Perform a redirect to an intermediate URI under the control of the AS to clear referer information in the browser that may contain
security token information. This page SHOULD provide notice to the resource owner that an error occurred, and request permission to redirect them to an external site.

If redirected, the fragment ":" MUST be appended to the error redirect URI. This prevents the browser from reattaching the fragment from a previous URI to the new location URI.

When redirecting via 30x a Content Security Policy header SHOULD be added:

```plaintext
Content-Security-Policy: referrer origin;
```

Figure 2

When redirecting via a form post the following tag SHOULD be included:

```html
<meta name="referrer" content="origin"/>
```

Figure 3

Only newer browsers support these headers, so users with older browsers will be vulnerable to leaking referer information unless an intermediate redirect is used.

3. Acknowledgements

We would like to thank all the people that participated to the discussion, namely Bill Burke, Hans Zandbelt, Justin P. Richer, Phil Hunt, Takahiko Kawasaki, Torsten Lodderstedt, Sergey Beryozkin.

4. Normative References


Appendix A. Document History

-01
  o Added information on HTTP headers to include to set referrer to origin

-00
  o Wrote the first draft.
  o Changed Document name to conform to WG naming convention
  o Added Section on redirect leaking security information
  o Added Terminology section
  o fixed file name
  o cleaned up mitigations a bit

Authors’ Addresses

John Bradley
Ping Identity
Email: ve7jtb@ve7jtb.com
URI: http://www.thread-safe.com/

Antonio Sanso (editor)
Adobe Systems
Email: asanso@adobe.com

Hannes Tschofenig
Email: Hannes.Tschofenig@gmx.net
URI: http://www.tschofenig.priv.at
Destination Claim for JSON Web Token
draft-campbell-oauth-dst4jwt-00

Abstract

The Destination Claim for JSON Web Token (JWT) provides a means of indicating the address to which the JWT is sent. The Claim can be used to prevent malicious forwarding or redirection of a JWT to unintended recipients.

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 August 27, 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.
1. Introduction

JWT [I-D.ietf-oauth-json-web-token] is a compact, URL-safe means of representing claims to be transferred between two parties. Oftentimes an HTTP 302 redirect or an auto-submitted HTML form, using the user agent as a intermediary, is employed as the method of transfer. The Destination Claim provides a standard way for the Issuer to indicate the address to which it instructed the user agent to deliver the JWT. The recipient of the JWT can detect and prevent malicious forwarding or redirection to unintended recipients by verifying that the address conveyed by the Destination Claim matches the actual location at which the JWT was received.

While the Destination Claim bears some seeming similarity to the Audience Claim already defined in JWT, the distinction is that the Audience identifies _who_ the JWT is intended for while the Destination identifies _where_ the JWT is sent.

1.1. Requirements Notation and Conventions

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].

1.2. Terminology

This specification uses the terms "JSON Web Token (JWT)", "Issuer" "Claim", "Claim Name", and "Claim Value" as defined in [I-D.ietf-oauth-json-web-token], and the term "user agent" as defined by RFC 7230 [RFC7230].
2. The Destination Claim

The Claim Name of the Destination Claim is "dst" and its Claim Value is a URI [RFC3986] indicating the address to which the JWT is sent. Use of this Claim is OPTIONAL but, if the Claim is present, the recipient MUST check that the URI identifies the location at which the JWT was received. If the JWT is received at a different location than the one conveyed by the value of the "dst" claim, then the JWT MUST be rejected.

3. IANA Considerations

3.1. JSON Web Token Claim Registration

This specification registers the Destination Claim defined herein in the IANA JSON Web Token Claims registry defined in [I-D.ietf-oauth-json-web-token].

3.1.1. Registry Request Contents

- Claim Name: "dst"
- Claim Description: Destination
- Change Controller: IESG
- Specification Document(s): Section 2 of this document

4. Security Considerations

The Destination Claim defined in Section 2 provides a means to assist in detecting and preventing malicious forwarding or redirection of a JWT to unintended recipients. If, for example, an Issuer can be tricked into sending a JWT to a malicious site (perhaps due to inadequate checking of the target URI combined with Cross-Site Request Forgery) the JWT would be unusable at the legitimate site because the "dst" would contain a URI of the malicious site.

5. References

5.1. Normative References

[I-D.ietf-oauth-json-web-token]


5.2. Informative References


Appendix A. Open Issues

- Is there compelling reason to allow the "dst" Claim to accommodate multiple values? A single value is sufficient for the cases envisioned and is certainly simpler.

Appendix B. Document History

[ ] to be removed by the RFC Editor before publication as an RFC]

-00

- Gotta start somewhere...

Authors' Addresses

Brian Campbell
Ping Identity
Email: brian.d.campbell@gmail.com

Guoping Liu
Ping Identity
Email: gliu@pingidentity.com
OAuth 2.0 Token Introspection
draft-ietf-oauth-introspection-06

Abstract

This specification defines a method for a protected resource to query an OAuth 2.0 authorization server to determine the active state of an OAuth 2.0 token and to determine meta-information about this token. OAuth 2.0 deployments can use this method to convey information about the authorization context of the token from the authorization server to the protected resource.

Requirements Language

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].

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 24, 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
1. Introduction

In OAuth 2.0, the contents of tokens are opaque to clients. This means that the client does not need to know anything about the content or structure of the token itself, if there is any. However, there is still a large amount of metadata that may be attached to a token, such as its current validity, approved scopes, and information about the context in which the token was issued. These pieces of information are often vital to protected resources making authorization decisions based on the tokens being presented. Since OAuth 2.0 [RFC6749] does not define a protocol for the resource server to learn meta-information about a token that has received from an authorization server, several different approaches have been developed to bridge this gap. These include using structured token formats such as JWT [JWT] or proprietary inter-service communication mechanisms (such as shared databases and protected enterprise service buses) that convey token information.

This specification defines an interoperable web API that allows authorized protected resources to query the authorization server to determine the set of metadata for a given token that was presented to...
them by an OAuth 2.0 client. This metadata includes whether or not
the token is currently active (or if it has expired or otherwise been
revoked), what rights of access the token carries (usually conveyed
through OAuth 2.0 scopes), and the authorization context in which the
token was granted (including who authorized the token and which
client it was issued to). Token introspection allows a protected
resource to query this information regardless of whether or not it is
carried in the token itself, allowing this method to be used along
with or independently of structured token values. Additionally, a
protected resource can use the mechanism described in this
specification to introspect the token in a particular authorization
decision context and ascertain the relevant metadata about the token
in order to make this authorization decision appropriately.

1.1. Terminology

This section defines the terminology used by this specification.
This section is a normative portion of this specification, imposing
requirements upon implementations.

This specification uses the terms "access token", "authorization
endpoint", "authorization grant", "authorization server", "client",
"client identifier", "protected resource", "refresh token", "resource
owner", "resource server", and "token endpoint" defined by OAuth 2.0
[RFC6749], and the terms "claim names" and "claim values" defined by
JSON Web Token (JWT) [JWT].

2. Introspection Endpoint

The introspection endpoint is an OAuth 2.0 endpoint that takes a
parameter representing an OAuth 2.0 token and returns a JSON
document representing the meta information surrounding the
token, including whether this token is currently active. The
definition of an active token is up to the authorization server, but
this is commonly a token that has been issued by this authorization
server, is not expired, has not been revoked, and is within the
purview of the protected resource making the introspection call.

The introspection endpoint MUST be protected by a transport-layer
security mechanism as described in Section 4.

2.1. Introspection Request

The protected resource calls the introspection endpoint using an HTTP
POST [RFC7231] request with parameters sent as "application/x-www-
form-urlencoded" data as defined in [W3C.REC-httplib5-20141028]. The
authorization server MAY allow an HTTP GET [RFC7231] request with
parameters passed in the query string as defined in
The protected resource sends a parameter representing the token along with optional parameters representing additional context that is known by the protected resource to aid the authorization server in its response.

token  REQUIRED. The string value of the token. For access tokens, this is the "access_token" value returned from the token endpoint defined in OAuth 2.0 [RFC6749] section 5.1. For refresh tokens, this is the "refresh_token" value returned from the token endpoint as defined in OAuth 2.0 [RFC6749] section 5.1. Other token types are outside the scope of this specification.

token_type_hint  OPTIONAL. A hint about the type of the token submitted for introspection. The protected resource MAY pass this parameter in order to help the authorization server to optimize the token lookup. If the server is unable to locate the token using the given hint, it MUST extend its search across all of its supported token types. An authorization server MAY ignore this parameter, particularly if it is able to detect the token type automatically. Values for this field are defined in OAuth Token Revocation [RFC7009].

The endpoint MAY allow other parameters to provide further context to the query. For instance, an authorization service may need to know the IP address of the client accessing the protected resource in order to determine the appropriateness of the token being presented.

To prevent unauthorized token scanning attacks, the endpoint MUST also require some form of authorization to access this endpoint, such as client authentication as described in OAuth 2.0 [RFC6749] or a separate OAuth 2.0 access token such as the bearer token described in OAuth 2.0 Bearer Token Usage [RFC6750]. The methods of managing and validating these authentication credentials are out of scope of this specification.

For example, the following example shows a protected resource calling the token introspection endpoint to query about an OAuth 2.0 bearer. The protected resource is using a separate OAuth 2.0 bearer token to authorize this call.
Following is a non-normative example request:

POST /introspect HTTP/1.1
Host: server.example.com
Accept: application/json
Content-Type: application/x-www-form-urlencoded
Authorization: Bearer 23410913-abewfq.123483
token=2YotnFZFEjr1zCsicMWpAA

In this example, the protected resource uses a client identifier and client secret to authenticate itself to the introspection endpoint as well as send a token type hint.

Following is a non-normative example request:

POST /introspect HTTP/1.1
Host: server.example.com
Accept: application/json
Content-Type: application/x-www-form-urlencoded
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
token=mF_9.B5f-4.1JqM&token_type_hint=access_token

2.2. Introspection Response

The server responds with a JSON object [RFC7159] in "application/json" format with the following top-level members.

active
REQUIRED. Boolean indicator of whether or not the presented token is currently active. The specifics of a token’s "active" state will vary depending on the implementation of the authorization server, and the information it keeps about its tokens, but a "true" value return for the "active" property will generally indicate that a given token has been issued by this authorization server, has not been revoked by the resource owner, and is within its given time window of validity (e.g. After its issuance time but not yet expired). See Section 4 for information on implementation of such checks.

scope
OPTIONAL. A space-separated list of strings representing the scopes associated with this token, in the format described in section 3.3 of OAuth 2.0 [RFC6749].
client_id
    OPTIONAL. Client identifier for the OAuth 2.0 client that requested this token.

username
    OPTIONAL. Human-readable identifier for the resource owner who authorized this token.

token_type
    OPTIONAL. Type of the token as defined in section 5.1 of OAuth 2.0 [RFC6749].

The response MAY include any claims defined as JWT [JWT] claim names and carry the same semantics and syntax, such as the following:

exp
    OPTIONAL. Integer timestamp, measured in the number of seconds since January 1 1970 UTC, indicating when this token will expire.

iat
    OPTIONAL. Integer timestamp, measured in the number of seconds since January 1 1970 UTC, indicating when this token was originally issued.

nbf
    OPTIONAL. Integer timestamp, measured in the number of seconds since January 1 1970 UTC, indicating when this token is not to be used before.

sub
    OPTIONAL. Subject of the token. Usually a machine-readable identifier of the resource owner who authorized this token.

aud
    OPTIONAL. Service-specific string identifier or list of string identifiers representing the intended audience for this token.

iss
    OPTIONAL. String representing the issuer of this token.

jti
    OPTIONAL. String identifier for the token.

Specific implementations MAY extend this structure with their own service-specific pieces of information as top-level members of this JSON object.
The authorization server MAY respond differently to different protected resources making the same request. For instance, an authorization server MAY limit which scopes from a given token for each protected resource in order to prevent protected resources from learning more about the larger network than is necessary for their function.

The response MAY be cached by the protected resource to improve performance and reduce load on the introspection endpoint, but at the cost of liveness of the information used by the protected resource. See Section 4 for more information regarding the trade off when the response is cached.

For example, the following response contains a set of information about an active token:

Following is a non-normative example response:

HTTP/1.1 200 OK
Content-Type: application/json

{  
  "active": true,
  "client_id": "l238j323ds-23ij4",
  "username": "jdoe",
  "scope": "read write dolphin",
  "sub": "2503upPC88QrAjx00dis",
  "iss": "https://protected.example.net/resource",
  "aud": "https://protected.example.net/resource",
  "iss": "https://server.example.com/",
  "exp": 1419356238,
  "iat": 1419350238,
  "extension_field": "twenty-seven"
}

If the introspection call is properly authorized but the token is not active, does not exist on this server, or the protected resource is not allowed to introspect this particular token, the authorization server MUST return an introspection response with the active field set to false. Note that in order to avoid disclosing too much of the authorization server’s state to a third party, the authorization server SHOULD NOT include any additional information about an inactive token, including why the token is inactive. For example, the response for a token that has been revoked or is otherwise invalid would look like the following:
Following is a non-normative example response:

HTTP/1.1 200 OK
Content-Type: application/json

{  
  "active": false
}

2.3. Error Response

If the protected resource uses OAuth 2.0 client credentials to authenticate to the introspection endpoint and its credentials are invalid, the authorization server responds with an HTTP 401 (Unauthorized) as described in section 5.2 of OAuth 2.0 [RFC6749].

If the protected resource uses an OAuth 2.0 bearer token to authorize its call to the introspection endpoint and the token used for authorization does not contain sufficient privileges or is otherwise invalid for this request, the authorization server responds with an HTTP 401 code as described in section 3 of OAuth 2.0 Bearer Token Usage [RFC6750].

Note that a properly formed and authorized query for an inactive or otherwise invalid token (or a token the protected resource is not allowed to know about) is not considered an error response by this specification. In these cases, the authorization server MUST instead respond with an introspection response with the "active" field set to "false" as described in Section 2.2.

3. IANA Considerations

This specification requests IANA to register the following values into the IANA JSON Web Token Claims registry for JWT Claim Names.

- Claim Name: "active"
  - Claim Description: Token active status
  - Change Controller: IESG
  - Specification Document(s): Section 2.2 of [[ this document ]].

- Claim Name: "username"
  - Claim Description: User identifier of the resource owner
  - Change Controller: IESG
  - Specification Document(s): Section 2.2 of [[ this document ]].

- Claim Name: "client_id"
  - Claim Description: Client identifier of the client
4. Security Considerations

Since there are many different and valid ways to implement an OAuth 2.0 system, there are consequently many ways for an authorization server to determine whether or not a token is currently "active" or not. However, since resource servers using token introspection rely on the authorization server to determine the state of a token, the authorization server MUST perform all applicable checks against a token’s state. For instance:

- If the token can expire, the authorization server MUST determine whether or not the token has expired.
- If the token can be issued before it is able to be used, the authorization server MUST determine whether or not a token’s valid period has started yet.
- If the token can be revoked after it was issued, the authorization server MUST determine whether or not such a revocation has taken place.
- If the token has been signed, the authorization server MUST validate the signature.

If an authorization server fails to perform any applicable check, the resource server could make an errant security decision based on that response. Note that not all of these checks will be applicable to all OAuth 2.0 deployments and it is up to the authorization server to determine which of these checks (and any other checks) apply.

If left unprotected and un-throttled, the introspection endpoint could present a means for an attacker to poll a series of possible token values, fishing for a valid token. To prevent this, the authorization server MUST require authentication of protected resources that need to access the introspection endpoint and SHOULD
require protected resources to be specifically authorized to call the introspection endpoint. The specifics of this authentication credentials are out of scope of this specification, but commonly these credentials could take the form of any valid client authentication mechanism used with the token endpoint, an OAuth 2.0 access token, or other HTTP authorization or authentication mechanism. A single piece of software acting as both a client and a protected resource MAY re-use the same credentials between the token endpoint and the introspection endpoint, though doing so potentially conflates the activities of the client and protected resource portions of the software and the authorization server MAY require separate credentials for each mode.

Since the introspection endpoint takes in OAuth 2.0 tokens as parameters and responds with information used to make authorization decisions, the server MUST support TLS 1.2 RFC 5246 [RFC5246] and MAY support additional transport-layer mechanisms meeting its security requirements. When using TLS, the client or protected resource MUST perform a TLS/SSL server certificate check, per RFC 6125 [RFC6125]. Implementation security considerations can be found in Recommendations for Secure Use of TLS and DTLS [TLS.BCP].

In order to prevent the values of access tokens being from leaking into server-side logs via query parameters, an authorization server offering token introspection MAY disallow HTTP GET and instead require an HTTP POST method to be used at the introspection endpoint.

In order to avoid disclosing internal server state, an introspection response for an inactive token SHOULD NOT contain any additional claims beyond the required "active" claim (with its value set to "false").

Since a protected resource MAY cache the response of the introspection endpoint, designers of an OAuth 2.0 system using this protocol MUST consider the performance and security trade-offs inherent in caching security information such as this. A less aggressive cache with a short timeout will provide the protected resource with more up to date information (due to it needing to query the introspection endpoint more often) at the cost of increased network traffic and load on the introspection endpoint. A more aggressive cache with a longer duration will minimize network traffic and load on the introspection endpoint, but at the cost of liveness of information about the token. For example, the token may be revoked while the protected resource is relying on the value of the cached response to make authorization decisions. This creates a window during which a revoked token could be used at the protected resource. Consequently, an acceptable cache validity duration needs to be carefully considered given the concerns and sensitivities of
the protected resource being accessed and the likelihood of a token
being revoked or invalidated in the interim period. Highly sensitive
environments can opt to disable caching entirely on the protected
resource in order to maximize liveness of information.

An authorization server offering token introspection MUST be able
to understand the token values being presented to it during this call.
The exact means by which this happens is an implementation detail and
outside the scope of this specification. For unstructured tokens,
this could take the form of a simple server-side database query
against a data store containing the context information for the
token. For structured tokens, this could take the form of the server
parsing the token, validating its signature or other protection
mechanisms, and returning the information contained in the token back
to the protected resource (allowing the protected resource to be
unaware of the token’s contents, much like the client).

Note that for tokens carrying encrypted information that is needed
during the introspection process, the authorization server MUST be
able to decrypt and validate the token in order to access this
information. Also note that in cases where the authorization server
stores no information about the token and has no means of accessing
information about the token by parsing the token itself, it can not
likely offer an introspection service.

5. Privacy Considerations

The introspection response may contain privacy-sensitive information
such as user identifiers for resource owners. When this is the case,
measures MUST be taken to prevent disclosure of this information to
unintended parties. One way to limit disclosure is to require
authorization to call the introspection endpoint and to limit calls
to only registered and trusted protected resource servers. Another
method is to transmit user identifiers as opaque service-specific
strings, potentially returning different identifiers to each
protected resource. Omitting privacy-sensitive information from an
introspection response is the simplest way of minimizing privacy
issues.

6. Acknowledgements

Thanks to the OAuth Working Group and the UMA Working Group for
feedback.
7. References

7.1. Normative References


7.2. Informative References


Appendix A. Use with Proof of Posession Tokens

With bearer tokens such as those defined by OAuth 2.0 Bearer Token Usage [RFC6750], the protected resource will have in its possession the entire secret portion of the token for submission to the introspection service. However, for proof-of-possession style tokens, the protected resource will have only a token identifier used during the request, along with the cryptographic signature on the request. The protected resource would be able to submit the token identifier to the authorization server’s token endpoint in order to obtain the necessary key information needed to validate the signature on the request. The details of this usage are outside the scope of this specification and will be defined in an extension to this specification.

Appendix B. Document History

[[ To be removed by the RFC Editor. ]]
-06
  o Clarified relationship between AS and RS in introduction.
  o Used updated TLS text imported from Dyn-Reg drafts.
  o Clarified definition of active state.
  o Added some advice on caching responses.
  o Added security considerations on active state implementation.
  o Changed user_id to username based on WG feedback.
-05
  o Typo fix.
  o Updated author information.
  o Removed extraneous "linewrap" note from examples.
  - 04
    o Removed "resource_id" from request.
  o Added examples.
  - 03
updated HTML and HTTP references.

- 02

- Call for registration of parameters in the JWT registry.

- Removed SAML pointer.

- Clarified what an "active" token could be.

- Explicitly declare introspection request as x-www-form-urlencoded format.

- Added extended example.

- Made protected resource authentication a MUST.

- 01

- Fixed casing and consistent term usage.

- Incorporated working group comments.

- Clarified that authorization servers need to be able to understand the token if they’re to introspect it.

- Various editorial cleanups.

- 00

- Created initial IETF drafted based on draft-richer-oauth-introspection-06 with no normative changes.

Author’s Address

Justin Richer (editor)

Email: ietf@justin.richer.org
Abstract

RFC 6750 specified the bearer token concept for securing access to protected resources. Bearer tokens need to be protected in transit as well as at rest. When a client requests access to a protected resource it hands-over the bearer token to the resource server.

The OAuth 2.0 Proof-of-Possession security concept extends bearer token security and requires the client to demonstrate possession of a key when accessing a protected resource.

This document describes how the client obtains this keying material from the authorization server.

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 6, 2015.
1. Introduction

The work on additional security mechanisms beyond OAuth 2.0 bearer tokens [12] is motivated in [17], which also outlines use cases, requirements and an architecture. This document defines the ability for the client to indicate support for this functionality and to obtain keying material from the authorization server. As an outcome of the
To best describe the scope of this specification, the OAuth 2.0 protocol exchange sequence is shown in Figure 1. The extension defined in this document piggybacks on the message exchange marked with (C) and (D).

![Figure 1: Abstract OAuth 2.0 Protocol Flow](image)

In OAuth 2.0 [2] access tokens can be obtained via authorization grants and using refresh tokens. The core OAuth specification defines four authorization grants, see Section 1.3 of [2], and [14] adds an assertion-based authorization grant to that list. The token endpoint, which is described in Section 3.2 of [2], is used with every authorization grant except for the implicit grant type. In the implicit grant type the access token is issued directly.

This document extends the functionality of the token endpoint, i.e., the protocol exchange between the client and the authorization server, to allow keying material to be bound to an access token. Two types of keying material can be bound to an access token, namely symmetric keys and asymmetric keys. Conveying symmetric keys from the authorization server to the client is described in Section 4 and the procedure for dealing with asymmetric keys is described in Section 5.
2. Terminology

The key words 'MUST', 'MUST NOT', 'REQUIRED', 'SHALL', 'SHALL NOT', 'SHOULD', 'SHOULD NOT', 'RECOMMENDED', 'MAY', and 'OPTIONAL' in this specification are to be interpreted as described in [1].

Session Key:

The term session key refers to fresh and unique keying material established between the client and the resource server. This session key has a lifetime that corresponds to the lifetime of the access token, is generated by the authorization server and bound to the access token.

This document uses the following abbreviations:

JWA: JSON Web Algorithms (JWA) [7]
JWT: JSON Web Token (JWT) [9]
JWS: JSON Web Signature (JWS) [6]
JWK: JSON Web Key (JWK) [5]
JWE: JSON Web Encryption (JWE) [8]

3. Audience

When an authorization server creates an access token, according to the PoP security architecture [17], it may need to know which resource server will process it. This information is necessary when the authorization server applies integrity protection to the JWT using a symmetric key and has to selected the key of the resource server that has to verify it. The authorization server also requires this audience information if it has to encrypt a symmetric session key inside the access token using a long-term symmetric key.

This section defines a new header that is used by the client to indicate what protected resource at which resource server it wants to access. This information may subsequently also communicated by the authorization server securely to the resource server, for example within the audience field of the access token.

QUESTION: A benefit of asymmetric cryptography is to allow clients to request a PoP token for use with multiple resource servers. The downside of that approach is linkability since different resource servers will be able to link individual requests to the same client.
(The same is true if the a single public key is linked with PoP tokens used with different resource servers.) Nevertheless, to support the functionality the audience parameter could carry an array of values. Is this desirable?

3.1. Audience Parameter

The client constructs the access token request to the token endpoint by adding the ‘aud’ parameter using the "application/x-www-form-urlencoded" format with a character encoding of UTF-8 in the HTTP request entity-body.

The URI included in the aud parameter MUST be an absolute URI as defined by Section 4.3 of [3]. It MAY include an "application/x-www-form-urlencoded" formatted query component (Section 3.4 of [3]). The URI MUST NOT include a fragment component.

The ABNF syntax for the ‘aud’ element is defined in Appendix A.

3.2. Processing Instructions

Step (0): As an initial step the client typically determines the resource server it wants to interact with. This may, for example, happen as part of a discovery procedure or via manual configuration.

Step (1): The client starts the OAuth 2.0 protocol interaction based on the selected grant type.

Step (2): When the client interacts with the token endpoint to obtain an access token it MUST populate the newly defined ‘audience’ parameter with the information obtained in step (0).

Step (2): The authorization server who obtains the request from the client needs to parse it to determine whether the provided audience value matches any of the resource servers it has a relationship with. If the authorization server fails to parse the provided value it MUST reject the request using an error response with the error code "invalid_request". If the authorization server does not consider the resource server acceptable it MUST return an error response with the error code "access_denied". In both cases additional error information may be provided via the error_description, and the error_uri parameters. If the request has, however, been verified successfully then the authorization server MUST include the audience claim into the access token with the value copied from the audience field provided by the client. In case the access token is encoded using the JSON Web Token format [9] the "aud" claim MUST be used. The access token, if
passed per value, MUST be protected against modification by either using a digital signature or a keyed message digest. Access tokens can also be passed by reference, which then requires the token introspection endpoint (or a similar, proprietary protocol mechanism) to be used. The authorization server returns the access token to the client, as specified in [2].

Subsequent steps for the interaction between the client and the resource server are beyond the scope of this document.

4. Symmetric Key Transport

4.1. Client-to-AS Request

In case a symmetric key shall be bound to an PoP token the following procedure is applicable. In the request message from the OAuth client to the OAuth authorization server the following parameters MAY be included:

token_type: OPTIONAL. See Section 6 for more details.

alg: OPTIONAL. See Section 6 for more details.

These two new parameters are optional in the case where the authorization server has prior knowledge of the capabilities of the client otherwise these two parameters are required. This prior knowledge may, for example, be set by the use of a dynamic client registration protocol exchange.

QUESTION: Should we register these two parameters for use with the dynamic client registration protocol?

For example, the client makes the following HTTP request using TLS (extra line breaks are for display purposes only).
Example Request to the Authorization Server

4.2. Client-to-AS Response

If the access token request has been successfully verified by the authorization server and the client is authorized to obtain a PoP token for the indicated resource server, the authorization server issues an access token and optionally a refresh token. If client authentication failed or is invalid, the authorization server returns an error response as described in Section 5.2 of [2].

The authorization server MUST include an access token and a ‘key’ element in a successful response. The ‘key’ parameter either contains a plain JWK structure or a JWK encrypted with a JWE. The difference between the two approaches is the following:

Plain JWK: If the JWK container is placed in the ‘key’ element then the security of the overall PoP architecture relies on Transport Layer Security (TLS) between the authorization server and the client. Figure 2 illustrates an example response using a plain JWK for key transport from the authorization server to the client.

JWK protected by a JWE: If the JWK container is protected by a JWE then additional security protection at the application layer is provided between the authorization server and the client beyond the use of TLS. This approach is a reasonable choice, for example, when a hardware security module is available on the client device and confidentiality protection can be offered directly to this hardware security module.

Note that there are potentially two JSON-encoded structures in the response, namely the access token (with the recommended JWT encoding) and the actual key transport mechanism itself. Note, however, that the two structures serve a different purpose and are consumed by different parites. The access token is created by the authorization server and processed by the resource server (and opaque to the
client) whereas the key transport payload is created by the authorization server and processed by the client; it is never forwarded to the resource server.

HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
  "access_token": "SlAV32hkKG ...
  (remainder of JWT omitted for brevity;
  JWT contains JWK in the cnf claim)",
  "token_type": "pop",
  "expires_in": 3600,
  "refresh_token": "8xLOxBtZp8",
  "key": "eyJhbGciOiJSU0ExXzUi ...
  (remainder of plain JWK omitted for brevity)"
}

Figure 2: Example: Response from the Authorization Server (Symmetric Variant)

The content of the key parameter, which is a JWK in our example, is shown in Figure 3.

{
  "kty": "oct",
  "kid": "id123",
  "alg": "HS256",
  "k": "ZoRSOrFzN_FzUA5XKMYoVHyzff5oRJx1-IXRtztJ6uE"
}

Figure 3: Example: Key Transport to Client via a JWK

The content of the ‘access_token’ in JWT format contains the ‘cnf’ (confirmation) claim, as shown in Figure 4. The confirmation claim is defined in [10]. The digital signature or the keyed message digest offering integrity protection is not shown in this example but MUST be present in a real deployment to mitigate a number of security threats. Those security threats are described in [17].

The JWK in the key element of the response from the authorization server, as shown in Figure 2, contains the same session key as the JWK inside the access token, as shown in Figure 4. It is, in this
example, protected by TLS and transmitted from the authorization server to the client (for processing by the client).

```json
{
  "iss": "https://server.example.com",
  "sub": "24400320",
  "aud": "s6BhdRkqt3",
  "nonce": "n-0S6_WzA2Mj",
  "exp": 1311281970,
  "iat": 1311280970,
  "cnf": {
    "jwk": "JDLUhTMjU2IiwiY3R5Ijoi ... (remainder of JWK protected by JWE omitted for brevity)"
  }
}
```

**Figure 4: Example: Access Token in JWT Format**

Note: When the JWK inside the access token contains a symmetric key it MUST be confidentiality protected using a JWE to maintain the security goals of the PoP architecture, as described in [17] since content is meant for consumption by the selected resource server only.

Note: This document does not impose requirements on the encoding of the access token. The examples used in this document make use of the JWT structure since this is the only standardized format.

If the access token is only a reference then a look-up by the resource server is needed, as described in the token introspection specification [18].

5. Asymmetric Key Transport

5.1. Client-to-AS Request

In case an asymmetric key shall be bound to an access token then the following procedure is applicable. In the request message from the OAuth client to the OAuth authorization server the request MAY include the following parameters:

- **token_type**: OPTIONAL. See Section 6 for more details.
- **alg**: OPTIONAL. See Section 6 for more details.
key: OPTIONAL. This field contains information about the public key the client would like to bind to the access token in the JWK format. If the client does not provide a public key then the authorization server MUST create an ephemeral key pair (considering the information provided by the client) or alternatively respond with an error message. The client may also convey the fingerprint of the public key to the authorization server instead of passing the entire public key along (to conserve bandwidth). [11] defines a way to compute a thumbprint for a JWK and to embed it within the JWK format.

The ‘token_type’ and the ‘alg’ parameters are optional in the case where the authorization server has prior knowledge of the capabilities of the client otherwise these two parameters are required.

For example, the client makes the following HTTP request using TLS (extra line breaks are for display purposes only) shown in Figure 5.

```
POST /token HTTP/1.1
Host: server.example.com
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded;charset=UTF-8

grant_type=authorization_code
&code=SplxlOBeZQQYbYS6WxSbIA
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
&token_type=pop
&alg=RS256
&key=eyJhbGciOiJSU0ExXzUi ...
(remainder of JWK omitted for brevity)
```

Figure 5: Example Request to the Authorization Server (Asymmetric Key Variant)

As shown in Figure 6 the content of the ‘key’ parameter contains the RSA public key the client would like to associate with the access token.
5.2. Client-to-AS Response

If the access token request is valid and authorized, the authorization server issues an access token and optionally a refresh token. If the request client authentication failed or is invalid, the authorization server returns an error response as described in Section 5.2 of [2].

The authorization server also places information about the public key used by the client into the access token to create the binding between the two. The new token type "public_key" is placed into the 'token_type' parameter.

An example of a successful response is shown in Figure 7.

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{  
  "access_token":"2YotnFZFE....jr1zCsicMWpAA",
  "token_type":"pop",
  "alg":"RS256",
  "expires_in":3600,
  "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA"
}
```

Figure 7: Example: Response from the Authorization Server (Asymmetric Variant)

The content of the 'access_token' field contains an encoded JWT with the following structure, as shown in Figure 8. The digital signature
or the keyed message digest offering integrity protection is not shown (but must be present).

```json
{
  "iss": "xas.example.com",
  "aud": "http://auth.example.com",
  "exp": "1361398824",
  "nbf": "1360189224",
  "cnf": {
    "jwk": {
      "kty": "RSA",
      "n": "0vx7agoebGcQSuuPiLJXZptN9nndrQmbXEps2aiAFbWhM78LhWx4cbbfAAAVT86zwu1RK7aPPFxuhDR1L6tScO_BJECFeWKRXjBZCiFV4n3oknjhMs
         tn64tZ-2W-5JSY4Yc5n9yB9Arl931qtT_RN5w6CF0qYQ5v-65YGjQ5R_OBZ2
         QvzqY368QMGCtdaS5qzs8KJZgnYb9c7d0zgdAZHzu6gMQvRL5hajrnI91CbQpbI
         SD08qNLyrdkt-bFTWhAI4QvMQFh6Wezu0fN41Fw3Rr3XPksINHaQ-G_xBn1Iqb
         w0Ls1jF44-csFCur-kEgU8awapJzKQqgKuw",
      "e": "AQAB",
      "alg": "RS256",
      "kid": "id123"
    }
  }
}
```

Figure 8: Example: Access Token Structure (Asymmetric Variant)

Note: In this example there is no need for the authorization server to convey further keying material to the client since the client is already in possession of the private RSA key.

6. Token Types and Algorithms

To allow clients to indicate support for specific token types and respective algorithms they need to interact with authorization servers. They can either provide this information out-of-band, for example, via pre-configuration or up-front via the dynamic client registration protocol [16].

The value in the ‘alg’ parameter together with value from the ‘token_type’ parameter allow the client to indicate the supported algorithms for a given token type. The token type refers to the specification used by the client to interact with the resource server to demonstrate possession of the key. The ‘alg’ parameter provides further information about the algorithm, such as whether a symmetric or an asymmetric crypto-system is used. Hence, a client supporting a specific token type also knows how to populate the values to the ‘alg’ parameter.
The value for the ‘token_type’ MUST be taken from the ‘OAuth Access Token Types’ registry created by [2].

This document does not register a new value for the OAuth Access Token Types registry nor does it define values to be used for the ‘alg’ parameter since this is the responsibility of specifications defining the mechanism for clients interacting with resource servers. An example of such specification can be found in [19].

The values in the ‘alg’ parameter are case-sensitive. If the client supports more than one algorithm then each individual value MUST be separated by a space.

7. Security Considerations

[17] describes the architecture for the OAuth 2.0 proof-of-possession security architecture, including use cases, threats, and requirements. This requirements describes one solution component of that architecture, namely the mechanism for the client to interact with the authorization server to either obtain a symmetric key from the authorization server, to obtain an asymmetric key pair, or to offer a public key to the authorization. In any case, these keys are then bound to the access token by the authorization server.

To summarize the main security recommendations: A large range of threats can be mitigated by protecting the contents of the access token by using a digital signature or a keyed message digest. Consequently, the token integrity protection MUST be applied to prevent the token from being modified, particularly since it contains a reference to the symmetric key or the asymmetric key. If the access token contains the symmetric key (see Section 2.2 of [10] for a description about how symmetric keys can be securely conveyed within the access token) this symmetric key MUST be encrypted by the authorization server with a long-term key shared with the resource server.

To deal with token redirect, it is important for the authorization server to include the identity of the intended recipient (the audience), typically a single resource server (or a list of resource servers), in the token. Using a single shared secret with multiple authorization server to simplify key management is NOT RECOMMENDED since the benefit from using the proof-of-possession concept is significantly reduced.

Token replay is also not possible since an eavesdropper will also have to obtain the corresponding private key or shared secret that is bound to the access token. Nevertheless, it is good practice to
limit the lifetime of the access token and therefore the lifetime of associated key.

The authorization server MUST offer confidentiality protection for any interactions with the client. This step is extremely important since the client will obtain the session key from the authorization server for use with a specific access token. Not using confidentiality protection exposes this secret (and the access token) to an eavesdropper thereby making the OAuth 2.0 proof-of-possession security model completely insecure. OAuth 2.0 [2] relies on TLS to offer confidentiality protection and additional protection can be applied using the JWK [5] offered security mechanism, which would add an additional layer of protection on top of TLS for cases where the keying material is conveyed, for example, to a hardware security module. Which version(s) of TLS ought to be implemented will vary over time, and depend on the widespread deployment and known security vulnerabilities at the time of implementation. At the time of this writing, TLS version 1.2 [4] is the most recent version. The client MUST validate the TLS certificate chain when making requests to protected resources, including checking the validity of the certificate.

Similarly to the security recommendations for the bearer token specification [12] developers MUST ensure that the ephemeral credentials (i.e., the private key or the session key) is not leaked to third parties. An adversary in possession of the ephemeral credentials bound to the access token will be able to impersonate the client. Be aware that this is a real risk with many smart phone app and Web development environments.

Clients can at any time request a new proof-of-possession capable access token. Using a refresh token to regularly request new access tokens that are bound to fresh and unique keys is important. Keeping the lifetime of the access token short allows the authorization server to use shorter key sizes, which translate to a performance benefit for the client and for the resource server. Shorter keys also lead to shorter messages (particularly with asymmetric keying material).

When authorization servers bind symmetric keys to access tokens then they SHOULD scope these access tokens to a specific permissions.

8. IANA Considerations

This specification registers the following parameters in the OAuth Parameters Registry established by [2].

Parameter name:  alg
9. Acknowledgements

We would like to thank Chuck Mortimore for his review comments.

10. References

10.1. Normative References


10.2. Informative References


Appendix A. Augmented Backus-Naur Form (ABNF) Syntax

This section provides Augmented Backus-Naur Form (ABNF) syntax descriptions for the elements defined in this specification using the notation of [13].

A.1. ‘aud’ Syntax

The ABNF syntax is defined as follows where by the "URI-reference" definition is taken from [3]:

\[
\text{aud} = \text{URI-reference}
\]

A.2. ‘key’ Syntax

The "key" element is defined in Section 4 and Section 5:

\[
\text{key} = 1^*VSCCHAR
\]

A.3. ‘alg’ Syntax

The "alg" element is defined in Section 6:

\[
\text{alg} = \text{alg-token} *( \text{SP alg-token} )
\]
\[
\text{alg-token} = 1^*NQCHAR
\]
Authors’ Addresses

John Bradley
Ping Identity

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

Phil Hunt
Oracle Corporation

Email: phil.hunt@yahoo.com
URI: http://www.indepdentid.com

Michael B. Jones
Microsoft

Email: mbj@microsoft.com
URI: http://self-issued.info/

Hannes Tschofenig
ARM Limited
Austria

Email: Hannes.Tschofenig@gmx.net
URI: http://www.tschofenig.priv.at
Proof-Of-Possession Semantics for JSON Web Tokens (JWTs)
draft-ietf-oauth-proof-of-possession-02

Abstract

This specification defines how to express a declaration in a JSON Web Token (JWT) that the presenter of the JWT possesses a particular key and that the recipient can cryptographically confirm proof-of-possession of the key by the presenter. This property is also sometimes described as the presenter being a holder-of-key.

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 ................................................. 3
  1.1. Notational Conventions ............................... 4
2. Terminology .................................................. 4
3. Proof-Of-Possession Representation ......................... 4
  3.1. Proof-of-Possession of an Asymmetric Key ............... 5
  3.2. Proof-of-Possession of a Symmetric Key ................. 5
  3.3. Proof-of-Possession Using a Key ID ..................... 6
  3.4. Confirmation ........................................... 7
  3.5. Specifics Intentionally Not Specified .................. 7
4. Security Considerations ....................................... 8
5. IANA Considerations ........................................... 8
  5.1. JSON Web Token Claims Registration ..................... 9
    5.1.1. Registry Contents ................................. 9
  5.2. JWT Confirmation Methods Registry ...................... 10
    5.2.1. Registration Template ............................. 10
    5.2.2. Initial Registry Contents ......................... 10
6. References ..................................................... 11
  6.1. Normative References ..................................... 11
  6.2. Informative References ................................... 11
Appendix A. Acknowledgements ................................... 12
Appendix B. Document History .................................... 12
Authors’ Addresses ............................................... 12
1. Introduction

This specification defines how to express a declaration in a JSON Web Token (JWT) [JWT] that the presenter of the JWT possesses a particular key and that the recipient can cryptographically confirm proof-of-possession of the key by the presenter. This property is also sometimes described as the presenter being a holder-of-key.

Envision the following two use cases. The first use case describes the use of a symmetric proof-of-possession key and the second use case uses an asymmetric proof-of-possession key.

An OAuth 2.0 authorization server generates a JWT and places an encrypted symmetric key inside the newly introduced confirmation claim. This symmetric key is encrypted with a key known only to the authorization server and the recipient. The entire JWT is then integrity protected by the issuer (the authorization server). The JWT is then sent to the presenter. Since the presenter is unable to obtain the encrypted symmetric key from the JWT itself, the authorization server conveys that symmetric key separately to the presenter. Now, the presenter is in possession of the symmetric key as well as the JWT (which includes the confirmation claim member). When the presenter needs to present the JWT to the recipient, it also needs to demonstrate possession of the symmetric key; the presenter, for example, uses the symmetric key in a challenge/response protocol with the recipient. The recipient is then able to verify that it is interacting with the genuine presenter by decrypting the JWK contained inside the confirmation claim of the JWT. By doing this, the recipient obtains the symmetric key, which it then uses to verify cryptographically protected messages exchanged with the presenter. This symmetric key mechanism described above is conceptually similar to the use of Kerberos tickets.

In the second case, consider a presenter that generates a public/private key pair. It then sends the public key to an OAuth 2.0 authorization server (the issuer), which creates a JWT and places a public key (or an identifier for it) inside the newly introduced confirmation claim. The entire JWT is integrity protected using a digital signature to protect it against modifications. The JWT is then sent to the presenter. When the presenter needs to present the JWT to the recipient, it also needs to demonstrate possession of the private key. The presenter, for example, uses the private key in a TLS exchange with the recipient. The recipient is able to verify that it is interacting with the genuine presenter by extracting the public key from the confirmation claim of the JWT (after verifying the digital signature of the JWT) and utilizing it with the private key in the TLS exchange. The asymmetric key mechanism described above is conceptually similar to a certificate.
In both cases the JWT may contain other claims that are needed by the application.

1.1. Notational Conventions

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].

Unless otherwise noted, all the protocol parameter names and values are case sensitive.

2. Terminology

This specification uses terms defined in the JSON Web Token (JWT) [JWT], JSON Web Key (JWK) [JWK], and JSON Web Encryption (JWE) [JWE] specifications.

These terms are defined by this specification:

Issuer
Party that creates the JWT and binds the proof-of-possession key to it.

Presenter
Party that proves possession of a private key (for asymmetric key cryptography) or secret key (for symmetric key cryptography) to a recipient. The presenter may be the issuer or a party different from the issuer.

Recipient
Party that receives the JWT containing the proof-of-possession key information from the presenter.

3. Proof-Of-Possession Representation

The presenter of a JWT declares that it possesses a particular key and that the recipient can cryptographically confirm proof-of-possession of the key by the presenter by including a "cnf" (confirmation) claim in the JWT whose value is a JSON object, with the JSON object containing a "jwk" (JSON Web Key) or "kid" (key ID) member identifying the key.

The presenter can be identified in one of two ways by the JWT, depending upon the application requirements. If the JWT contains a
"sub" (subject) claim, the presenter is the subject identified by the JWT. (In some applications, the subject identifier will be relative to the issuer identified by the "iss" (issuer) claim.) If the JWT contains no "sub" (subject) claim, the presenter is the issuer identified by the JWT using the "iss" (issuer) claim. The case in which the presenter is the subject of the JWT is analogous to SAML 2.0 [OASIS.saml-core-2.0-os] SubjectConfirmation usage. At least one of the "sub" and "iss" claims MUST be present in the JWT, and in some use cases, both MUST be present.

3.1. Proof-of-Possession of an Asymmetric Key

When the key held by the presenter is an asymmetric private key, the value of the "jwk" member is a JSON Web Key (JWK) [JWK] representing the corresponding asymmetric public key. The following example demonstrates such a declaration in the JWT Claims Set of a JWT:

```
{
  "iss": "https://server.example.com",
  "aud": "https://client.example.org",
  "exp": "1361398824",
  "nbf": "1360189224",
  "cnf":{
    "jwk":{
      "kty": "EC",
      "use": "sig",
      "crv": "P-256",
      "x": "18wHLe1g9wVN6VD1Txpqy2LszYkMf6J8njVAibvhM",
      "y": "-V4dS4uLMgP_4fY4j8ir7c1lTX1FdAgcx55o7TkCSA"
    }
  }
}
```

The JWK MUST contain the required key members for a JWK of that key type and MAY contain other JWK members, including the "kid" (key ID) member.

3.2. Proof-of-Possession of a Symmetric Key

When the key held by the presenter is a symmetric key, the value of the "jwk" member is an encrypted JSON Web Key (JWK) [JWK] encrypted to a key known to the recipient using the JWE Compact Serialization containing the symmetric key. The rules for encrypting a JWK are found in Section 6 of the JSON Web Key [JWK] specification.

The following example illustrates a symmetric key that could subsequently be encrypted for use in the "jwk" member:
The UTF-8 [RFC3629] encoding of this JWK would be used as the JWE Plaintext when encrypting the key.

The following example is a JWE Header that could be used when encrypting this key:

```
{
    "alg": "RSA1_5",
    "enc": "A128CBC-HS256",
    "cty": "jwk+json"
}
```

The following example JWT Claims Set of a JWT illustrates the use of an encrypted symmetric key as the "jwk" claim value:

```
{
    "iss": "https://server.example.com",
    "sub": "24400320",
    "aud": "s6BhdRkqt3",
    "nonce": "n-0S6_WzA2Mj",
    "exp": 1311281970,
    "iat": 1311280970,
    "cnf": {
        "jwk":
            "eyJhbGciOiJSU0ExXzUiLCJlbmMiOiJBMTI4Q0JDLUhTMjU2IiwiY3R5Ijoi
            andrK2pzb24ifQ. ... (remainder of JWE omitted for brevity)"
    }
}
```

Note that the case in which the "jwk" claim contains an unencoded JWK value and the case in which it contains an encrypted JWK value can be distinguished by the type of the member value. In the first case, the value is a JSON object containing the JWK and in the second case, the value is a string containing the JWE JSON Serialization of the encrypted JWK representation.

### 3.3. Proof-of-Possession Using a Key ID

The proof-of-possession key can also be identified by the use of a Key ID instead of communicating the actual key, provided the recipient is able to obtain the identified key using the Key ID. In this case, the presenter of a JWT declares that it possesses a
particular key and that the recipient can cryptographically confirm proof-of-possession of the key by the presenter by including a "cnf" (confirmation) claim in the JWT whose value is a JSON object, with the JSON object containing a "kid" (key ID) member identifying the key.

The following example demonstrates such a declaration in the JWT Claims Set of a JWT:

```json
{
  "iss": "https://server.example.com",
  "aud": "https://client.example.org",
  "exp": "1361398824",
  "nbf": "1360189224",
  "cnf": {
    "kid": "dfd1aa97-6d8d-4575-a0fe-34b96de2bfad"
  }
}
```

3.4. Confirmation

The "cnf" (confirmation) claim is used in the JWT to contain the "jwk" or "kid" member because a proof-of-possession key may not be the only means of confirming the authenticity of the token. This is analogous to the SAML 2.0 [OASIS.saml-core-2.0-os] SubjectConfirmation element, in which a number of different subject confirmation methods can be included, including proof-of-possession key information. When a recipient receives a "cnf" claim with a member that it does not understand, it MUST ignore that member.

This specification defines a registry for these members in Section 5.2 and registers the "jwk" and "kid" members within the registry.

3.5. Specifics Intentionally Not Specified

Proof-of-possession is typically demonstrated by having the presenter sign a value determined by the recipient using the key possessed by the presenter. This value is sometimes called a "nonce" or a "challenge".

The means of communicating the nonce and the nature of its contents are intentionally not described in this specification, as different protocols will communicate this information in different ways. Likewise, the means of communicating the signed nonce is also not specified, as this is also protocol-specific.

Note that another means of proving possession of the key when it is a
symmetric key is to encrypt the key to the recipient. The means of obtaining a key for the recipient is likewise protocol-specific.

For an example specification that uses the mechanisms defined in this document, see [I-D.ietf-oauth-pop-architecture].

4. Security Considerations

All of the normal security issues, especially in relationship to comparing URIs and dealing with unrecognized values, that are discussed in JWT [JWT] also apply here.

In addition, proof-of-possession introduces its own unique security issues. Possessing the key is only valuable if it is kept secret. Appropriate means must be used to ensure that unintended parties do not learn the private key or symmetric key value.

Proof-of-possession via encrypted symmetric secrets is subject to replay attacks. This attack can be avoided when a signed nonce or challenge is used, since the recipient can use a distinct nonce or challenged for each interaction.

Similarly to other information included in a JWT, it is necessary to apply data origin authentication and integrity protection (via a keyed message digest or a digital signature). Data origin authentication ensures that the recipient of the JWT learns about the entity that created the JWT, since this will be important for any policy decisions. Integrity protection prevents an adversary from changing any elements conveyed within the JWT payload. Special care has to be applied when carrying symmetric keys inside the JWT, since those not only require integrity protection, but also confidentiality protection.

A recipient may not understand the newly introduced "cnf" claim and may consequently treat it as a bearer token. While this is a legitimate concern, it is outside the scope of this specification, since demonstration the possession of the key associated with the "cnf" claim is not covered by this specification. For more details, please consult [I-D.ietf-oauth-pop-architecture].

5. IANA Considerations

The following registration procedure is used for all the registries established by this specification.

Values are registered with a Specification Required [RFC5226] after a
three-week review period on the [TBD]@ietf.org mailing list, on the
advice of one or more Designated Experts. However, to allow for the
allocation of values prior to publication, the Designated Expert(s)
may approve registration once they are satisfied that such a
specification will be published.

Registration requests must be sent to the [TBD]@ietf.org mailing list
for review and comment, with an appropriate subject (e.g., "Request
for access token type: example"). [[ Note to the RFC Editor: The name
of the mailing list should be determined in consultation with the
IESG and IANA. Suggested name: oauth-pop-reg-review@ietf.org. ]]

Within the review period, the Designated Expert(s) will either
approve or deny the registration request, communicating this decision
to the review list and IANA. Denials should include an explanation
and, if applicable, suggestions as to how to make the request
successful. Registration requests that are undetermined for a period
longer than 21 days can be brought to the IESG’s attention (using the
iesg@ietf.org mailing list) for resolution.

Criteria that should be applied by the Designated Expert(s) includes
determining whether the proposed registration duplicates existing
functionality, determining whether it is likely to be of general
applicability or whether it is useful only for a single application,
and whether the registration makes sense.

IANA must only accept registry updates from the Designated Expert(s)
and should direct all requests for registration to the review mailing
list.

It is suggested that multiple Designated Experts be appointed who are
able to represent the perspectives of different applications using
this specification, in order to enable broadly-informed review of
registration decisions. In cases where a registration decision could
be perceived as creating a conflict of interest for a particular
Expert, that Expert should defer to the judgment of the other
Expert(s).

5.1. JSON Web Token Claims Registration

This specification registers the "cnf" claim in the IANA JSON Web
Token Claims registry defined in [JWT].

5.1.1. Registry Contents

- Claim Name: "cnf"
5.2. JWT Confirmation Methods Registry

This specification establishes the IANA JWT Confirmation Methods registry for JWT "cnf" member values. The registry records the confirmation method member and a reference to the specification that defines it.

5.2.1. Registration Template

Confirmation Method Value:
The name requested (e.g., "example"). Because a core goal of this specification is for the resulting representations to be compact, it is RECOMMENDED that the name be short -- not to exceed 8 characters without a compelling reason to do so. This name is case-sensitive. Names may not match other registered names in a case-insensitive manner unless the Designated Expert(s) state that there is a compelling reason to allow an exception in this particular case.

Confirmation Method Description:
Brief description of the confirmation method (e.g., "Example description").

Change Controller:
For Standards Track RFCs, state "IESG". For others, give the name of the responsible party. Other details (e.g., postal address, email address, home page URI) may also be included.

Specification Document(s):
Reference to the document(s) that specify the parameter, preferably including URI(s) that can be used to retrieve copies of the document(s). An indication of the relevant sections may also be included but is not required.

5.2.2. Initial Registry Contents

- Confirmation Method Value: "jwk"
- Confirmation Method Description: JSON Web Key or Encrypted JSON Web Key
- Change Controller: IESG
- Specification Document(s): Section 3.1 of [[this document]]
6. References

6.1. Normative References

draft-ietf-jose-json-web-encryption (work in progress),
January 2015.

[JWK] Jones, M., "JSON Web Key (JWK)",
draft-ietf-jose-json-web-key (work in progress),
January 2015.

[JWT] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token
(JWT)", draft-ietf-oauth-json-web-token (work in progress), December 2014.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an
IANA Considerations Section in RFCs", BCP 26, RFC 5226,
May 2008.

6.2. Informative References

[I-D.ietf-oauth-pop-architecture]
Hunt, P., ietf@justin.richer.org, i., Mills, W., Mishra,
P., and H. Tschofenig, "OAuth 2.0 Proof-of-Possession
(PoP) Security Architecture",
draft-ietf-oauth-pop-architecture-01 (work in progress),
March 2015.

[OASIS.saml-core-2.0-os]
Cantor, S., Kemp, J., Philpott, R., and E. Maler,
"Assertions and Protocol for the OASIS Security Assertion
Markup Language (SAML) V2.0", OASIS Standard saml-core-
Appendix A. Acknowledgements

The authors wish to thank James Manger for his review of the specification.

Appendix B. Document History

[[ to be removed by the RFC Editor before publication as an RFC ]]

-02

o Defined the terms Issuer, Presenter, and Recipient and updated their usage within the document.

o Added a description of a use case using an asymmetric proof-of-possession key to the introduction.

o Added the "kid" (key ID) confirmation method.

o These changes address the open issues identified in the previous draft.

-01

o Updated references.

-00

o Created the initial working group draft from draft-jones-oauth-proof-of-possession-02.

Authors’ Addresses

Michael B. Jones
Microsoft

Email: mbj@microsoft.com
URI: http://self-issued.info/

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
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 Registra
   tion ....................................................... 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 to 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_wW1gFWF0EjXk

   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
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]


9.2. Informative References

[I-D.bradley-oauth-pop-key-distribution]

[I-D.hunt-oauth-pop-architecture]

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
Abstract

This specification defines how to request and obtain Security Tokens from OAuth Authorization Servers, including enabling one party to act on behalf of another or enabling one party to delegate authority to another.

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 August 27, 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 .................................................... 3  
   1.1. Requirements Notation and Conventions .......................... 3  
   1.2. Terminology .................................................. 3  
   1.3. On-Behalf-Of vs. Impersonation Semantics .................... 3  
2. Security Token Request ........................................... 4  
   2.1. Act-As Security Token Requests ............................... 5  
   2.2. On-Behalf-Of Security Token Requests ......................... 6  
3. Security Token Response .......................................... 6  
4. Conveying Eligibility to Act As Another Party .................. 7  
5. Implementation Considerations .................................... 8  
6. IANA Considerations ............................................. 8  
7. Security Considerations .......................................... 8  
8. References ....................................................... 8  
   8.1. Normative References ......................................... 8  
   8.2. Informative References ...................................... 8  
Appendix A. Open Issues ............................................. 9  
Appendix B. Acknowledgements ....................................... 9  
Appendix C. Document History ...................................... 9  
Authors’ Addresses .................................................. 10
1. Introduction

This specification defines how to request and obtain Security Tokens from OAuth Authorization Servers [RFC6749], including enabling one party to act on behalf of another or enabling one party to delegate authority to another. The functionality defined and the terminology used in this specification are intentionally parallel to the functionality and terminology defined by [WS-Trust], including On-Behalf-Of and Act-As.

A Security Token is a set of information that facilitates the sharing of identity and security information across security domains. Examples of Security Tokens include JSON Web Tokens (JWTs) [JWT] and SAML Assertions [OASIS.saml-core-2.0-os]. Security Tokens are typically signed to achieve integrity and sometimes also encrypted to achieve confidentiality. Security Tokens are also described as Assertions in [I-D.ietf-oauth-assertions].

This specification defines a new Security Token Request Grant Type used at the Token Endpoint to convey the parameters for a Security Token request and Security Token response parameter used in responses to these requests. The Security Token Request is a JSON Web Token (JWT) [JWT] that is issued by the requesting party that contains parameters of the request as Claims.

The Security Tokens obtained could be used in a number of contexts, the specifics of which are beyond the scope of this specification. Examples include using them with the

1.1. Requirements Notation and Conventions

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].

1.2. Terminology

This specification uses the terms "Authorization Server", "Token Endpoint", "Token Request", and "Token Response" defined by OAuth 2.0 [RFC6749], and the terms "Claim" and "JWT Claims Set" defined by JSON Web Token (JWT) [JWT].

1.3. On-Behalf-Of vs. Impersonation Semantics

When principal A acts on behalf of principal B, A is given all the rights that B has within some defined rights context. Whereas, with on-behalf-of semantics, principal A still has its own identity separate from B and it is explicitly understood that while B may have
delegated its rights to A, any actions taken are being taken by A and not B. In a sense, A is an agent for B.

On-behalf-of semantics are therefore different than impersonation semantics, with which it is sometimes confused. When principal A impersonates principal B, then in so far as any entity receiving Claims is concerned, they are actually dealing with B. It is true that some members of the identity system might have awareness that impersonation is going on but it is not a requirement. For all intents and purposes, when A is acting for B, A is B.

2. Security Token Request

A Security Token Request is sent to the Token Endpoint as a Token Request message using this Grant Type value:

\texttt{urn:ietf:params:oauth:grant-type:security-token-request}

Grant Type value indicating that this Token Request is a Security Token Request.

A Token Request parameter with a related name is used to convey the information contained in Security Token Request as a JWT:

\texttt{security_token_request}

Token Request parameter whose value is a JWT containing the Security Token Request information.

An example Security Token Request (with extra line breaks for display purposes only) follows:

\begin{verbatim}
POST /token HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded

grant_type=urn:ietf:params:oauth:grant-type:security-token-request &security_token_request=eyJhbGciOiJSUzI1NiJ9....
[omitted for brevity]
\end{verbatim}

The "security_token_request" parameter value is a JWT with the following members:

\texttt{iss}

REQUIRED. Issuer of the principal requesting the Security Token.
sub
   REQUIRED. Identifier of the principal requesting the Security Token at the issuer.

security_token_type
   OPTIONAL. Identifier for the type of the requested Security Token. If not present, the default is that a JWT is being requested. A JWT can also be requested with the identifier "urn:ietf:params:oauth:token-type:jwt".

scopes
   OPTIONAL. Array of strings, each of which represents a service context that the requested Security Token is being requested to be used for. The array MUST contain at least one scope value. The definition of these contexts is outside the scope of this specification. (Note: This request element serves the same purpose as the WS-Trust AppliesTo RST element.)

The request JWT MUST be signed by the issuer so the identity of the requesting party can be validated unless the identity of the requesting party is known to the Authorization Server by other means; in that case, the JWT can use the "alg" value "none".

The following is an example of a JWT Claims Set for a Security Token Request:

{  "iss": "https://server.example.com",  "sub": "24400320",  "scopes": ["example"]}

2.1. Act-As Security Token Requests

This specification defines the ability to request a Security Token for the requesting party to use to act as the specified party. This is accomplished using this Token Request parameter:

act_as
   This OPTIONAL request parameter indicates that the requested Security Token is expected to contain information about the identity represented by the Security Token that is the value of this parameter, enabling the requesting party to use the returned Security Token to act as this identity.
The following is an example of a JWT Claims Set for a Security Token Request using an "act_as" Claim:

```
{
  "iss": "https://server.example.com",
  "sub": "24400320",
  "scopes": ["example"],
  "act_as": "eyJhbGciOiJSUzI1NiJ9.eyJpc3MiOiJ ..."
}
```

2.2. On-Behalf-Of Security Token Requests

This specification defines the ability to request a Security Token on behalf of another party. This is accomplished using this Token Request parameter:

```
on_behalf_of
```

This OPTIONAL request parameter indicates that the Security Token is being requested on behalf of another party. The identity of the party upon whose behalf the request is being made is represented by the Security Token that is the value of this parameter. Proof of eligibility to act on behalf of that identity MAY be conveyed by including an "actor" Claim identifying the requesting party in the Security Token, per Section 4, provided the Security Token is a JWT.

The following is an example of a JWT Claims Set for a Security Token Request using an "on_behalf_of" Claim:

```
{
  "iss": "https://server.example.com",
  "sub": "24400320",
  "scopes": ["example"],
  "on_behalf_of": "eyJhbGciOiJSUzI1NiJ9.eyJpc3MiOiJ ..."
}
```

3. Security Token Response

A Security Token Response is returned from the Token Endpoint as a Token Response message containing these members:

```
security_token
```

Returned Security Token.
security_token_type
  Identifier for the type of the returned Security Token. If the
  Security Token is a JWT, this identifier is
  "urn:ietf:params:oauth:token-type:jwt".

An example successful response is as follows:

HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "security_token": "eyJhbGciOiJSUzI1NiJ9.eyJpc3MiOiJ ...",
  "security_token_type": "urn:ietf:params:oauth:token-type:jwt"
}

4. Conveying Eligibility to Act As Another Party

It is useful to be able to make a statement that one party is
authorized to act on behalf of another party. This can be done by
having the party being acted for issue a Security Token containing a
Claim identifying the party that will act for it as an authorized
actor. This statement can also optionally identify scopes in which
the actor is eligible to act through another Claim. The following
Claims are defined for use in JWTs for these purposes:

actor
  Security Token that identifies a party who is asserted as being
  eligible to act for the party identified by the JWT containing
  this Claim.

scopes
  OPTIONAL. Array of strings, each of which represents a service
  context for which the actor is asserted as being eligible to act
  for the party identified by the JWT containing this Claim. The
  array MUST contain at least one scope value. The definition of
  these contexts is outside the scope of this specification.

The JWT issued by the party being acted for MUST be signed so the
identity of the party being acted for can be validated unless the
identity of the party being acted for is known to the Authorization
Server by other means; in that case, the JWT can use the "alg" value
"none".
5. Implementation Considerations

Implementations of the specification MUST implement support for using JWTs as the Security Tokens. Other Security Token types MAY be supported.

6. IANA Considerations

The "urn:ietf:params:oauth:grant-type:security-token-request" Grant Type is to be registered in the IANA urn:ietf:params:oauth registry established in An IETF URN Sub-Namespace for OAuth [RFC6755].

The "scopes", "act_as", and "on_behalf_of" Claims are to be registered in the JSON Web Token Claims registry.

7. Security Considerations

All of the normal security issues, especially in relationship to comparing URIs and dealing with unrecognized values, that are discussed in JWT [JWT] also apply here.

In addition, on-behalf-of introduces its own unique security issues. Any time one principal is delegated the rights of another principal, the potential for abuse is always a concern. That is why use of the "scopes" member is suggested. The scope values restrict the contexts in which the delegated rights can be exercised.

8. References

8.1. Normative References


8.2. Informative References

[I-D.ietf-oauth-assertions] Campbell, B., Mortimore, C., Jones, M., and Y. Goland,


Appendix A. Open Issues

The following decisions need to be made and updates on this spec performed:

- Should we say anything about proof of possession of the target party’s key in the On-Behalf-Of case beyond specifying the use of the "actor" Claim?

- Revise the text in the On-Behalf-Of vs. Impersonation Semantics section to better align the terminology used with the semantics specified.

- Address the sources of potential terminological confusion discussed in John Bradley’s review comments.

- Add examples illustrating concrete uses of act-as and on-behalf-of.

Appendix B. Acknowledgements

The authors wish to thank Brian Campbell and John Bradley for reviews of the specification.

Appendix C. Document History

[[ to be removed by the RFC Editor before publication as an RFC ]]

Jones & Nadalin Expires August 27, 2015 [Page 9]
-01
  o Updated references.
-00
  o Created initial working group draft from draft-jones-oauth-token-exchange-01.

Authors’ Addresses

Michael B. Jones
Microsoft
Email: mbj@microsoft.com
URI: http://self-issued.info/

Anthony Nadalin
Microsoft
Email: tonynad@microsoft.com