OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: January 22, 2016

J. Bradley
Ping Identity
A. Sanso, Ed.
Adobe Systems
H. Tschofenig
July 21, 2015

OAuth 2.0 Security: OAuth Open Redirector
draft-bradley-oauth-open-redirector-02.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 January 22, 2016.

Copyright Notice

Table of Contents

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.

  https://AUTHORIZATION_SERVER/authorize?response_type=code&client_id=s6BhdRkqt3&state=xyz&redirect_uri=https%3A%2F%2Fattacker%2Ecom&scope=INVALID_SCOPE

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

  https://AUTHORIZATION_SERVER/authorize?response_type=code&client_id=malicious_client&redirect_uri=https%3A%2F%2Fattacker%2Ecom&scope=INVALID_SCOPE

- 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

- 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:

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

- 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, a 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.

  Some

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

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

Figure 2

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

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

Figure 3

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

3. Acknowledgements

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

4. Normative References

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

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

[RFC6819] Lodderstedt, T., Ed., McGloin, M., and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations", RFC 6819, DOI 10.17487/RFC6819, January 2013, <http://www.rfc-editor.org/info/rfc6819>.

Appendix A. Document History

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

-01

- Added information on HTTP headders to include to set referrer to origin

-00

- Wrote the first draft.

- Changed Document name to conform to WG naming convention
- Added Section on redirect leaking security information
- Added Terminology section
- fixed file name
- 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

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 24, 2019

N. Sakimura
Nomura Research Institute
J. Bradley
Yubico

October 21, 2018

The OAuth 2.0 Authorization Framework: JWT Secured Authorization Request (JAR)

draft-ietf-oauth-jwsreq-17

Abstract

The authorization request in OAuth 2.0 described in RFC 6749 utilizes query parameter serialization, which means that Authorization Request parameters are encoded in the URI of the request and sent through user agents such as web browsers. While it is easy to implement, it means that (a) the communication through the user agents are not integrity protected and thus the parameters can be tainted, and (b) the source of the communication is not authenticated. Because of these weaknesses, several attacks to the protocol have now been put forward.

This document introduces the ability to send request parameters in a JSON Web Token (JWT) instead, which allows the request to be signed with JSON Web Signature (JWS) and encrypted with JSON Web Encryption (JWE) so that the integrity, source authentication and confidentiality property of the Authorization Request is attained. The request can be sent by value or by reference.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on April 24, 2019.

Table of Contents

1. Introduction

The Authorization Request in OAuth 2.0 [RFC6749] utilizes query parameter serialization and is typically sent through user agents such as web browsers.

For example, the parameters "response_type", "client_id", "state", and "redirect_uri" are encoded in the URI of the request:

```
GET /authorize?response_type=code&client_id=s6BhdRkqt3&state=xyz
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb HTTP/1.1
Host: server.example.com
```

While it is easy to implement, the encoding in the URI does not allow application layer security with confidentiality and integrity protection to be used. While TLS is used to offer communication security between the Client and the user-agent as well as the user-agent and the Authorization Server, TLS sessions are terminated in the user-agent. In addition, TLS sessions may be terminated prematurely at some middlebox (such as a load balancer).

As the result, the Authorization Request of [RFC6749] has shortcomings in that:

(a) the communication through the user agents are not integrity protected and thus the parameters can be tainted (integrity protection failure)

(b) the source of the communication is not authenticated (source authentication failure)

(c) the communication through the user agents can be monitored (containment / confidentiality failure).

Due to these inherent weaknesses, several attacks against the protocol, such as Redirection URI rewriting and Mix-up attack [FETT], have been identified.

The use of application layer security mitigates these issues.

The use of application layer security allows requests to be prepared by a third party so that a client application cannot request more permissions than previously agreed. This offers an additional degree of privacy protection.

Furthermore, the request by reference allows the reduction of over-the-wire overhead.

The JWT [RFC7519] encoding has been chosen because of

(1) its close relationship with JSON, which is used as OAuth's response format

(2) its developer friendliness due to its textual nature

(3) its relative compactness compared to XML

(4) its development status that it is an RFC and so is its associated signing and encryption methods as [RFC7515] and [RFC7516]

(5) the relative ease of JWS and JWE compared to XML Signature and Encryption.

The parameters "request" and "request_uri" are introduced as additional authorization request parameters for the OAuth 2.0 [RFC6749] flows. The "request" parameter is a JSON Web Token (JWT) [RFC7519] whose JWT Claims Set holds the JSON encoded OAuth 2.0 authorization request parameters. This JWT is integrity protected and source authenticated using JWS.

The JWT [RFC7519] can be passed to the authorization endpoint by reference, in which case the parameter "request_uri" is used instead of the "request".

Using JWT [RFC7519] as the request encoding instead of query parameters has several advantages:

(a) (integrity protection) The request can be signed so that the integrity of the request can be checked.

(b) (source authentication) The request can be signed so that the signer can be authenticated.

(c) (confidentiality protection) The request can be encrypted so that end-to-end confidentiality can be provided even if the TLS connection is terminated at one point or another.

(d) (collection minimization) The request can be signed by a third party attesting that the authorization request is compliant with a certain policy. For example, a request can be pre-examined by a third party that all the personal data requested is strictly necessary to perform the process that the end-user asked for, and statically signed by that third party. The authorization server then examines the signature and shows the conformance status to the end-user, who would have some assurance as to the legitimacy of the request when authorizing it. In some cases, it may even be desirable to skip the authorization dialogue under such circumstances.

There are a few cases that request by reference is useful such as:

1. When it is desirable to reduce the size of transmitted request. The use of application layer security increases the size of the request, particularly when public key cryptography is used.

2. When the client does not want to do the crypto. The Authorization Server may provide an endpoint to accept the Authorization Request through direct communication with the Client so that the Client is authenticated and the channel is TLS protected.

This capability is in use by OpenID Connect [OpenID.Core].

1.1. 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].

2. Terminology

For the purposes of this specification, the following terms and definitions in addition to what is defined in OAuth 2.0 Framework [RFC6749], JSON Web Signature [RFC7515], and JSON Web Encryption [RFC7519] apply.

2.1. Request Object

JWT [RFC7519] that holds an OAuth 2.0 authorization request as JWT Claims Set

2.2. Request Object URI

Absolute URI from which the Request Object (Section 2.1) can be obtained

3. Symbols and abbreviated terms

The following abbreviations are common to this specification.

JSON Javascript Object Notation

JWT JSON Web Token

JWS JSON Web Signature

JWE JSON Web Encryption

URI Uniform Resource Identifier

URL Uniform Resource Locator

4. Request Object

A Request Object (Section 2.1) is used to provide authorization request parameters for an OAuth 2.0 authorization request. It MUST contains all the OAuth 2.0 [RFC6749] authorization request parameters including extension parameters. The parameters are represented as the JWT claims. Parameter names and string values MUST be included as JSON strings. Since Request Objects are handled across domains and potentially outside of a closed ecosystem, per section 8.1 of [RFC8259], these JSON strings MUST be encoded using UTF-8 [RFC3629]. Numerical values MUST be included as JSON numbers. It MAY include any extension parameters. This JSON [RFC7159] constitutes the JWT Claims Set defined in JWT [RFC7519]. The JWT Claims Set is then signed or signed and encrypted.

To sign, JSON Web Signature (JWS) [RFC7515] is used. The result is a JWS signed JWT [RFC7519]. If signed, the Authorization Request Object SHOULD contain the Claims "iss" (issuer) and "aud" (audience) as members, with their semantics being the same as defined in the JWT [RFC7519] specification.

To encrypt, JWE [RFC7516] is used. When both signature and encryption are being applied, the JWT MUST be signed then encrypted as advised in the section 11.2 of [RFC7519]. The result is a Nested JWT, as defined in [RFC7519].

The Authorization Request Object MAY be sent by value as described in Section 5.1 or by reference as described in Section 5.2.

"request" and "request_uri" parameters MUST NOT be included in Request Objects.

The following is an example of the Claims in a Request Object before base64url encoding and signing. Note that it includes extension variables such as "nonce" and "max_age".

```
{
 "iss": "s6BhdRkqt3",
 "aud": "https://server.example.com",
 "response_type": "code id_token",
 "client_id": "s6BhdRkqt3",
 "redirect_uri": "https://client.example.org/cb",
 "scope": "openid",
 "state": "af0ifjsldkj",
 "nonce": "n-0S6_WzA2Mj",
 "max_age": 86400
}
```

Signing it with the "RS256" algorithm results in this Request Object value (with line wraps within values for display purposes only):

```
eyJhbGciOiJSUzI1NiIsImtpZCI6ImsyYmRjIn0.ew0KICJpc3MiOiAiczZCaGRSa3
F0MyIsDQogImF1ZCI6ICJodHRwczovL3NlcnZlci5leGFtcGxlLmNvbSIsDQogInJl
c3BvbnNlX3R5cGUiOiAiY29kZSBpZF90b2tlbiIsDQogImNsaWVudF9pZCI6ICJzNk
JoZFJrcXQzIiwNCiAicmVkaXJlY3RfdXJpIjogImh0dHBzOi8vY2xpZW50LmV4YW1w
bGUub3JnL2NiIiwNCiAic2NvcGUiOiAib3BlbmlkIiwNCiAic3RhdGUiOiAiYWYwaW
Zqc2xka2oiLA0KICJub25jZSI6ICJuLTBTNl9XekEyTWoiLA0KICJtYXhfYWdlIjog
ODY0MDAsDQogImNsYWltcyI6IA0KICB7DQogICAidXNlcmluZm8iOiANCiAgICB7DQ
ogICAgICJnaXZlbl9uYW1lIjogeyJlc3NlbnRpYWwiOiB0cnVlfSwNCiAgICAgIm5p
Y2tuYW1lIjogbnVsbCwNCiAgICAgImVtYWlsIjogeyJlc3NlbnRpYWwiOiB0cnVlfS
wNCiAgICAgImVtYWlsX3ZlcmlmaWVkIjogeyJlc3NlbnRpYWwiOiB0cnVlfSwNCiAg
ICAgInBpY3R1cmUiOiBudWxsDQogICAgfSwNCiAgICJpZF90b2tlbiI6IA0KICAgIH
sNCiAgICAgImdlbmRlciI6IG51bGwsDQogICAgICJiaXJ0aGRhdGUiOiB7ImVzc2Vu
dGlhbCI6IHRydWV9LA0KICAgICAiYWNyIjogeyJ2YWx1ZXMiOiBbInVybjptYWNlOm
luY29tbW9uOmlhcDpzaWx2ZXIiXX0NCiAgICB9DQogIH0NCn0.nwwnNsk1-Zkbmnvs
F6zTHm8CHERFMGQPhos-EJcaH4Hh-sMgk8ePrGhw_trPYs8KQxsn6R9Emo_wHwajyF
KzuMXZFSZ3p6Mb8dkxtVyjoy2GIzvuJT_u7PkY2t8QU9hjBcHs68PkgjDVTrG1uRTx
0GxFbuPbj96tVuj11pTnmFCUR6IEOXKYr7iGOCRB3btfJhM0_AKQUfqKnRlrRscc8K
ol-cSLWoYE9l5QqholImzjT_cMnNIznW9E7CDyWXTsO70xnB4SkG6pXfLSjLLlxmPG
iyon_-Te111V8uE83IlzCYIb_NMXvtTIVc1jpspnTSD7xMbpL-2QgwUsAlMGzw
```

The following RSA public key, represented in JWK format, can be used to validate the Request Object signature in this and subsequent Request Object examples (with line wraps within values for display purposes only):

```
{
 "kty":"RSA",
 "kid":"k2bdc",
 "n":"y9Lqv4fCp6Ei-u2-ZCKq83YvbFEk6JMs_pSj76eMkddWRuWX2aBKGHAtKlE5P
      7_vn__PCKZWePt3vGkB6ePgzAFu08NmKemwE5bQI0e6kIChtt_6KzT5OaaXDF
      I6qCLJmk51Cc4VYFaxgqevMncYrzaW_50mZ1yGSFIQzLYP8bijAHGVjdEFgZa
      ZEN9lsn_GdWLaJpHrB3ROlS50E45wxrlg9xMncVb8qDPuXZarvghLL0HzOuYR
      adBJVoWZowDNTpKpk2RklZ7QaBO7XDv3uR7s_sf2g-bAjSYxYUGsqkNA9b3xV
      W53am_UZZ3tZbFTIh557JICWKHlWj5uzeJXaw",
 "e":"AQAB"
}
```

5. Authorization Request

The client constructs the authorization request URI by adding one of the following parameters but not both to the query component of the authorization endpoint URI using the "application/x-www-form-urlencoded" format:

request The Request Object (Section 2.1) that holds authorization request parameters stated in section 4 of OAuth 2.0 [RFC6749].

request_uri The absolute URI as defined by RFC3986 [RFC3986] that points to the Request Object (Section 2.1) that holds authorization request parameters stated in section 4 of OAuth 2.0 [RFC6749].

The client directs the resource owner to the constructed URI using an HTTP redirection response, or by other means available to it via the user-agent.

For example, the client directs the end user’s user-agent to make the following HTTPS request:

```
GET /authz?request=eyJhbG..AlMGzw HTTP/1.1
Host: server.example.com
```

The value for the request parameter is abbreviated for brevity.

The authorization request object MUST be one of the following:

(a) JWS signed

(b) JWS signed and JWE encrypted

The client MAY send the parameters included in the request object duplicated in the query parameters as well for the backward compatibility etc. However, the authorization server supporting this specification MUST only use the parameters included in the request object.

5.1. Passing a Request Object by Value

The Client sends the Authorization Request as a Request Object to the Authorization Endpoint as the "request" parameter value.

The following is an example of an Authorization Request using the "request" parameter (with line wraps within values for display purposes only):

```
https://server.example.com/authorize?
  request=eyJhbGciOiJSUzI1NiIsImtpZCI6ImsyYmRjIn0.ew0KICJpc3MiOiA
  iczZCaGRSa3F0MyIsDQogImF1ZCI6ICJodHRwczovL3NlcnZlci5leGFtcGxlLmN
  vbSIsDQogInJlc3BvbnNlX3R5cGUiOiAiY29kZSBpZF90b2tlbiIsDQogImNsaWV
  udF9pZCI6ICJzNkJoZFJrcXQzIiwNCiAicmVkaXJlY3RfdXJpIjogImh0dHBzOi8
  vY2xpZW50LmV4YW1wbGUub3JnL2NiIiwNCiAic2NvcGUiOiAib3BlbmlkIiwNCiA
  ic3RhdGUiOiAiYWYwaWZqc2xka2oiLA0KICJub25jZSI6ICJuLTBTNl9XekEyTWo
  iLA0KICJtYXhfYWdlIjogODY0MDAsDQogImNsYWltcyI6IA0KICB7DQogICAidXN
  lcmluZm8iOiANCiAgICB7DQogICAgICJnaXZlbl9uYW1lIjogeyJlc3NlbnRpYWw
  iOiB0cnVlfSwNCiAgICAgIm5pY2tuYW1lIjogbnVsbCwNCiAgICAgImVtYWlsIjo
  geyJlc3NlbnRpYWwiOiB0cnVlfSwNCiAgICAgImVtYWlsX3ZlcmlmaWVkIjogeyJ
  lc3NlbnRpYWwiOiB0cnVlfSwNCiAgICAgInBpY3R1cmUiOiBudWxsDQogICAgfSw
  NCiAgICJpZF90b2tlbiI6IA0KICAgIHsNCiAgICAgImdlbmRlciI6IG51bGwsDQo
  gICAgICJiaXJ0aGRhdGUiOiB7ImVzc2VudGlhbCI6IHRydWV9LA0KICAgICAiYWN
  yIjogeyJ2YWx1ZXMiOiBbInVybjptYWNlOmluY29tbW9uOmlhcDpzaWx2ZXIiXX0
  NCiAgICB9DQogIH0NCn0.nwwnNsk1-ZkbmnvsF6zTHm8CHERFMGQPhos-EJcaH4H
  h-sMgk8ePrGhw_trPYs8KQxsn6R9Emo_wHwajyFKzuMXZFSZ3p6Mb8dkxtVyjoy2
  GIzvuJT_u7PkY2t8QU9hjBcHs68PkgjDVTrG1uRTx0GxFbuPbj96tVuj11pTnmFC
  UR6IEOXKYr7iGOCRB3btfJhM0_AKQUfqKnRlrRscc8Kol-cSLWoYE9l5QqholImz
  jT_cMnNIznW9E7CDyWXTsO70xnB4SkG6pXfLSjLLlxmPGiyon_-Te111V8uE83Il
  zCYIb_NMXvtTIVc1jpspnTSD7xMbpL-2QgwUsAlMGzw
```

5.2. Passing a Request Object by Reference

The "request_uri" Authorization Request parameter enables OAuth authorization requests to be passed by reference, rather than by value. This parameter is used identically to the "request" parameter, other than that the Request Object value is retrieved from the resource identified by the specified URI rather than passed by value.

The entire Request URI MUST NOT exceed 512 ASCII characters. There are three reasons for this restriction.

1. Many phones in the market as of this writing still do not accept large payloads. The restriction is typically either 512 or 1024 ASCII characters.

2. The maximum URL length supported by older versions of Internet Explorer is 2083 ASCII characters.

3. On a slow connection such as 2G mobile connection, a large URL would cause the slow response and therefore the use of such is not advisable from the user experience point of view.

The contents of the resource referenced by the URI MUST be a Request Object. The "request_uri" value MUST be either URN as defined in RFC8141 [RFC8141] or "https" URI, as defined in 2.7.2 of RFC7230 [RFC7230] . The "request_uri" value MUST be reachable by the Authorization Server.

The following is an example of the contents of a Request Object resource that can be referenced by a "request_uri" (with line wraps within values for display purposes only):

eyJhbGciOiJSUzI1NiIsImtpZCI6ImsyYmRjIn0.ew0KICJpc3MiOiAiczZCaGRSa3
F0MyIsDQogImF1ZCI6ICJodHRwczovL3NlcnZlci5leGFtcGxlLmNvbSIsDQogInJl
c3BvbnNlX3R5cGUiOiAiY29kZSBpZF90b2tlbiIsDQogImNsaWVudF9pZCI6ICJzNk
JoZFJrcXQzIiwNCiAicmVkaXJlY3RfdXJpIjogImh0dHBzOi8vY2xpZW50LmV4YW1w
bGUub3JnL2NiIiwNCiAic2NvcGUiOiAib3BlbmlkIiwNCiAic3RhdGUiOiAiYWYwaW
Zqc2xka2oiLA0KICJub25jZSI6ICJuLTBTNl9XekEyTWoiLA0KICJtYXhfYWdlIjog
ODY0MDAsDQogImNsYWltcyI6IA0KICB7DQogICAidXNlcmluZm8iOiANCiAgICB7DQ
ogICAgICJnaXZlbl9uYW1lIjogeyJlc3NlbnRpYWwiOiB0cnVlfSwNCiAgICAgIm5p
Y2tuYW1lIjogbnVsbCwNCiAgICAgImVtYWlsIjogeyJlc3NlbnRpYWwiOiB0cnVlfS
wNCiAgICAgImVtYWlsX3ZlcmlmaWVkIjogeyJlc3NlbnRpYWwiOiB0cnVlfSwNCiAg
ICAgInBpY3R1cmUiOiBudWxsDQogICAgfSwNCiAgICJpZF90b2tlbiI6IA0KICAgIH
sNCiAgICAgImdlbmRlciI6IG51bGwsDQogICAgICJiaXJ0aGRhdGUiOiB7ImVzc2Vu
dGlhbCI6IHRydWV9LA0KICAgICAiYWNyIjogeyJ2YWx1ZXMiOiBbInVybjptYWNlOm
luY29tbW9uOmlhcDpzaWx2ZXIiXX0NCiAgICB9DQogIH0NCn0.nwwnNsk1-Zkbmnvs
F6zTHm8CHERFMGQPhos-EJcaH4Hh-sMgk8ePrGhw_trPYs8KQxsn6R9Emo_wHwajyF
KzuMXZFSZ3p6Mb8dkxtVyjoy2GIzvuJT_u7PkY2t8QU9hjBcHs68PkgjDVTrG1uRTx
0GxFbuPbj96tVuj11pTnmFCUR6IEOXKYr7iGOCRB3btfJhM0_AKQUfqKnRlrRscc8K
ol-cSLWoYE9l5QqholImzjT_cMnNIznW9E7CDyWXTsO70xnB4SkG6pXfLSjLLlxmPG
iyon_-Te111V8uE83IlzCYIb_NMXvtTIVc1jpspnTSD7xMbpL-2QgwUsAlMGzw

5.2.1. URI Referencing the Request Object

The Client stores the Request Object resource either locally or remotely at a URI the Authorization Server can access. Such facility may be provided by the authorization server or a third party. For example, the authorization server may provide a URL to which the client POSTs the request object and obtains the Requiest URI. This URI is the Request Object URI, "request_uri".

It is possible for the Request Object to include values that are to be revealed only to the Authorization Server. As such, the "request_uri" MUST have appropriate entropy for its lifetime. For the guidance, refer to 5.1.4.2.2 of [RFC6819]. It is RECOMMENDED that it be removed after a reasonable timeout unless access control measures are taken.

The following is an example of a Request Object URI value (with line wraps within values for display purposes only):

```
https://tfp.example.org/request.jwt#
  GkurKxf5T0Y-mnPFCHqWOMiZi4VS138cQO_V7PZHAdM
```

5.2.2. Request using the "request_uri" Request Parameter

The Client sends the Authorization Request to the Authorization Endpoint.

The following is an example of an Authorization Request using the "request_uri" parameter (with line wraps within values for display purposes only):

```
https://server.example.com/authorize?
  response_type=code%20id_token
  &client_id=s6BhdRkqt3
  &request_uri=https%3A%2F%2Ftfp.example.org%2Frequest.jwt
  %23GkurKxf5T0Y-mnPFCHqWOMiZi4VS138cQO_V7PZHAdM
  &state=af0ifjsldkj
```

5.2.3. Authorization Server Fetches Request Object

Upon receipt of the Request, the Authorization Server MUST send an HTTP "GET" request to the "request_uri" to retrieve the referenced Request Object, unless it is stored in a way so that it can retrieve it through other mechanism securely, and parse it to recreate the Authorization Request parameters.

The following is an example of this fetch process:

```
GET /request.jwt HTTP/1.1
Host: tfp.example.org
```

The following is an example of the fetch response:

```
HTTP/1.1 200 OK
Date: Thu, 16 Feb 2017 23:52:39 GMT
Server: Apache/2.2.22 (tfp.example.org)
Content-type: application/jwt
Content-Length: 1250
Last-Modified: Wed, 15 Feb 2017 23:52:32 GMT

eyJhbGciOiJSUzI1NiIsImtpZCI6ImsyYmRjIn0.ew0KICJpc3MiOiAiczZCaGRSa3
F0MyIsDQogImF1ZCI6ICJodHRwczovL3NlcnZlci5leGFtcGxlLmNvbSIsDQogInJl
c3BvbnNlX3R5cGUiOiAiY29kZSBpZF90b2tlbiIsDQogImNsaWVudF9pZCI6ICJzNk
JoZFJrcXQzIiwNCiAicmVkaXJlY3RfdXJpIjogImh0dHBzOi8vY2xpZW50LmV4YW1w
bGUub3JnL2NiIiwNCiAic2NvcGUiOiAib3BlbmlkIiwNCiAic3RhdGUiOiAiYWYwaW
Zqc2xka2oiLA0KICJub25jZSI6ICJuLTBTNl9XekEyTWoiLA0KICJtYXhfYWdlIjog
ODY0MDAsDQogImNsYWltcyI6IA0KICB7DQogICAidXNlcmluZm8iOiANCiAgICB7DQ
ogICAgICJnaXZlbl9uYW1lIjogeyJlc3NlbnRpYWwiOiB0cnVlfSwNCiAgICAgIm5p
Y2tuYW1lIjogbnVsbCwNCiAgICAgImVtYWlsIjogeyJlc3NlbnRpYWwiOiB0cnVlfS
wNCiAgICAgImVtYWlsX3ZlcmlmaWVkIjogeyJlc3NlbnRpYWwiOiB0cnVlfSwNCiAg
ICAgInBpY3R1cmUiOiBudWxsDQogICAgfSwNCiAgICJpZF90b2tlbiI6IA0KICAgIH
sNCiAgICAgImdlbmRlciI6IG51bGwsDQogICAgICJiaXJ0aGRhdGUiOiB7ImVzc2Vu
dGlhbCI6IHRydWV9LA0KICAgICAiYWNyIjogeyJ2YWx1ZXMiOiBbInVybjptYWNlOm
luY29tbW9uOmlhcDpzaWx2ZXIiXX0NCiAgICB9DQogIH0NCn0.nwwnNsk1-Zkbmnvs
F6zTHm8CHERFMGQPhos-EJcaH4Hh-sMgk8ePrGhw_trPYs8KQxsn6R9Emo_wHwajyF
KzuMXZFSZ3p6Mb8dkxtVyjoy2GIzvuJT_u7PkY2t8QU9hjBcHs68PkgjDVTrG1uRTx
0GxFbuPbj96tVuj11pTnmFCUR6IEOXKYr7iGOCRB3btfJhM0_AKQUfqKnRlrRscc8K
ol-cSLWoYE9l5QqholImzjT_cMnNIznW9E7CDyWXTsO70xnB4SkG6pXfLSjLLlxmPG
iyon_-Te111V8uE83IlzCYIb_NMXvtTIVc1jpspnTSD7xMbpL-2QgwUsAlMGzw
```

6. Validating JWT-Based Requests

6.1. Encrypted Request Object

If the request object is encrypted, the Authorization Server MUST decrypt the JWT in accordance with the JSON Web Encryption [RFC7516] specification.

The result is a signed request object.

If decryption fails, the Authorization Server MUST return an "invalid_request_object" error.

6.2. JWS Signed Request Object

The Authorization Server MUST perform the signature validation of the JSON Web Signature [RFC7515] signed request object. For this, the "alg" Header Parameter in its JOSE Header MUST match the value of the pre-registered algorithm. The signature MUST be validated against the appropriate key for that "client_id" and algorithm.

If signature validation fails, the Authorization Server MUST return an "invalid_request_object" error.

6.3. Request Parameter Assembly and Validation

The Authorization Server MUST extract the set of Authorization Request parameters from the Request Object value. The Authorization Server MUST only use the parameters in the Request Object even if the same parameter is provided in the query parameter. The Authorization Server then validates the request as specified in OAuth 2.0 [RFC6749].

If the validation fails, then the Authorization Server MUST return an error as specified in OAuth 2.0 [RFC6749].

7. Authorization Server Response

Authorization Server Response is created and sent to the client as in Section 4 of OAuth 2.0 [RFC6749] .

In addition, this document uses these additional error values:

invalid_request_uri The "request_uri" in the Authorization Request returns an error or contains invalid data.

invalid_request_object The request parameter contains an invalid Request Object.

request_not_supported The Authorization Server does not support the use of the "request" parameter.

request_uri_not_supported The Authorization Server does not support the use of the "request_uri" parameter.

8. TLS Requirements

Client implementations supporting the Request Object URI method MUST support TLS following Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) [BCP195].

To protect against information disclosure and tampering, confidentiality protection MUST be applied using TLS with a cipher suite that provides confidentiality and integrity protection.

HTTP clients MUST also verify the TLS server certificate, using subjectAltName dNSName identities as described in [RFC6125], to avoid man-in-the-middle attacks. The rules and guidelines defined in [RFC6125] apply here, with the following considerations:

- Support for DNS-ID identifier type (that is, the dNSName identity in the subjectAltName extension) is REQUIRED. Certification authorities which issue server certificates MUST support the DNS-ID identifier type, and the DNS-ID identifier type MUST be present in server certificates.

- DNS names in server certificates MAY contain the wildcard character "*".

- Clients MUST NOT use CN-ID identifiers; a CN field may be present in the server certificate's subject name, but MUST NOT be used for authentication within the rules described in [BCP195] .

- SRV-ID and URI-ID as described in Section 6.5 of [RFC6125] MUST NOT be used for comparison.

9. IANA Considerations

This specification requests no actions by IANA.

10. Security Considerations

In addition to the all the security considerations discussed in OAuth 2.0 [RFC6819], the security considerations in [RFC7515], [RFC7516], and [RFC7518] needs to be considered. Also, there are several academic papers such as [BASIN] that provide useful insight into the security properties of protocols like OAuth.

In consideration of the above, this document advises taking the following security considerations into account.

10.1. Choice of Algorithms

When sending the authorization request object through "request" parameter, it MUST either be signed using JWS [RFC7515] or encrypted using JWE [RFC7516] with then considered appropriate algorithm.

10.2. Request Source Authentication

The source of the Authorization Request MUST always be verified. There are several ways to do it in this specification.

(a) Verifying the JWS Signature of the Request Object.

(b) Verifying that the symmetric key for the JWE encryption is the correct one if the JWE is using symmetric encryption.

(c) Verifying the TLS Server Identity of the Request Object URI. In this case, the Authorization Server MUST know out-of-band that the Client uses Request Object URI and only the Client is covered by the TLS certificate. In general, it is not a reliable method.

(d) Authorization Server is providing an endpoint that provides a Request Object URI in exchange for a Request Object. In this case, the Authorization Server MUST perform Client Authentication to accept the Request Object and bind the Client Identifier to the Request Object URI it is providing. Since Request Object URI can be replayed, the lifetime of the Request Object URI MUST be short and preferably one-time use. The entropy of the Request Object URI MUST be sufficiently large. The adequate shortness of the validity and the entropy of the Request Object URI depends on the risk calculation based on the value of the resource being protected. A general guidance for the validity time would be less than a minute and the Request Object URI is to include a cryptographic random value of 128bit or more at the time of the writing of this specification.

(e) A third party, such as a Trust Framework Provider, provides an endpoint that provides a Request Object URI in exchange for a Request Object. The same requirements as (b) above apply. In addition, the Authorization Server MUST know out-of-band that the Client utilizes the Trust Framework Operator.

10.3. Explicit Endpoints

Although this specification does not require them, research such as [BASIN] points out that it is a good practice to explicitly state the intended interaction endpoints and the message position in the sequence in a tamper evident manner so that the intent of the initiator is unambiguous. The endpoints that come into question in this specification are :

(a) Protected Resources ("protected_resources")

(b) Authorization Endpoint ("authorization_endpoint")

(c) Redirection URI ("redirect_uri")

(d) Token Endpoint ("token_endpoint")

Further, if dynamic discovery is used, then the discovery related endpoints also come into question.

In [RFC6749], while Redirection URI is included, others are not included in the Authorization Request. As the result, the same applies to Authorization Request Object.

The lack of the link among those endpoints are sited as the cause of Cross-Phase Attacks introduced in [FETT]. An extension specification should be created as a measure to address the risk.

10.4. Risks Associated with request_uri

The introduction of "request_uri" introduces several attack possibilities.

10.4.1. DDoS Attack on the Authorization Server

A set of malicious client can launch a DoS attack to the authorization server by pointing the "request_uri" to a uri that returns extremely large content or extremely slow to respond. Under such an attack, the server may use up its resource and start failing.

Similarly, a malicious client can specify the "request_uri" value that itself points to an authorization request URI that uses "request_uri" to cause the recursive lookup.

To prevent such attack to succeed, the server should (a) check that the value of "request_uri" parameter does not point to an unexpected location, (b) check the content type of the response is "application/jwt" (c) implement a time-out for obtaining the content of "request_uri", and (d) do not perform recursive GET on the "request_uri".

10.4.2. Request URI Rewrite

The value of "request_uri" is not signed thus it can be tampered by Man-in-the-browser attacker. Several attack possibilities rise because of this, e.g., (a) attacker may create another file that the rewritten URI points to making it possible to request extra scope (b) attacker launches a DoS attack to a victim site by setting the value of "request_uri" to be that of the victim.

To prevent such attack to succeed, the server should (a) check that the value of "request_uri" parameter does not point to an unexpected location, (b) check the content type of the response is "application/jwt" (c) implement a time-out for obtaining the content of "request_uri".

11. TLS security considerations

Curent security considerations can be found in Recommendations for Secure Use of TLS and DTLS [BCP195]. This supersedes the TLS version recommendations in OAuth 2.0 [RFC6749].

12. Privacy Considerations

When the Client is being granted access to a protected resource containing personal data, both the Client and the Authorization Server need to adhere to Privacy Principles. RFC 6973 Privacy Considerations for Internet Protocols [RFC6973] gives excellent guidance on the enhancement of protocol design and implementation. The provision listed in it should be followed.

Most of the provision would apply to The OAuth 2.0 Authorization Framework [RFC6749] and The OAuth 2.0 Authorization Framework: Bearer Token Usage [RFC6750] and are not specific to this specification. In what follows, only the specific provisions to this specification are noted.

12.1. Collection limitation

When the Client is being granted access to a protected resource containing personal data, the Client SHOULD limit the collection of personal data to that which is within the bounds of applicable law and strictly necessary for the specified purpose(s).

It is often hard for the user to find out if the personal data asked for is strictly necessary. A Trust Framework Provider can help the user by examining the Client request and comparing to the proposed processing by the Client and certifying the request. After the certification, the Client, when making an Authorization Request, can submit Authorization Request to the Trust Framework Provider to obtain the Request Object URI.

Upon receiving such Request Object URI in the Authorization Request, the Authorization Server first verifies that the authority portion of the Request Object URI is a legitimate one for the Trust Framework Provider. Then, the Authorization Server issues HTTP GET request to the Request Object URI. Upon connecting, the Authorization Server MUST verify the server identity represented in the TLS certificate is

legitimate for the Request Object URI. Then, the Authorization Server can obtain the Request Object, which includes the "client_id" representing the Client.

The Consent screen MUST indicate the Client and SHOULD indicate that the request has been vetted by the Trust Framework Operator for the adherence to the Collection Limitation principle.

12.2. Disclosure Limitation

12.2.1. Request Disclosure

This specification allows extension parameters. These may include potentially sensitive information. Since URI query parameter may leak through various means but most notably through referrer and browser history, if the authorization request contains a potentially sensitive parameter, the Client SHOULD JWE [RFC7516] encrypt the request object.

Where Request Object URI method is being used, if the request object contains personally identifiable or sensitive information, the "request_uri" SHOULD be used only once, have a short validity period, and MUST have large enough entropy deemed necessary with applicable security policy unless the Request Object itself is JWE [RFC7516] Encrypted. The adequate shortness of the validity and the entropy of the Request Object URI depends on the risk calculation based on the value of the resource being protected. A general guidance for the validity time would be less than a minute and the Request Object URI is to include a cryptographic random value of 128bit or more at the time of the writing of this specification.

12.2.2. Tracking using Request Object URI

Even if the protected resource does not include a personally identifiable information, it is sometimes possible to identify the user through the Request Object URI if persistent per-user Request Object URI is used. A third party may observe it through browser history etc. and start correlating the user’s activity using it. In a way, it is a data disclosure as well and should be avoided.

Therefore, per-user Request Object URI should be avoided.

13. Acknowledgements

The following people contributed to the creation of this document in the OAuth WG. (Affiliations at the time of the contribution are used.)

Sergey Beryozkin, Brian Campbell (Ping Identity), Vladimir Dzhuvinov (Connect2id), Michael B. Jones (Microsoft), Torsten Lodderstedt (YES) Jim Manico, Axel Nenker(Deutsche Telecom), Hannes Tschofenig (ARM), Ben Campbell, Kathleen Moriarty (as AD), and Steve Kent (as SECDIR).

The following people contributed to creating this document through the OpenID Connect Core 1.0 [OpenID.Core].

Brian Campbell (Ping Identity), George Fletcher (AOL), Ryo Itou (Mixi), Edmund Jay (Illumila), Michael B. Jones (Microsoft), Breno de Medeiros (Google), Hideki Nara (TACT), Justin Richer (MITRE).

In addition, the following people contributed to this and previous versions through the OAuth Working Group.

Dirk Balfanz (Google), James H. Manger (Telstra), John Panzer (Google), David Recordon (Facebook), Marius Scurtescu (Google), Luke Shepard (Facebook).

14. Revision History

Note to the RFC Editor: Please remove this section from the final RFC.

-17

- #78 Typos in content-type

-16

- Treated remaining Ben Campbell comments.

-15

- Removed further duplication

-14

- #71 Reiterate dynamic params are included.
- #70 Made clear that AS must return error.
- #69 Inconsistency of the need to sign.
- Fixed Mimetype.
- #67 Incosistence in requiring HTTPS in request uri.

- #66 Dropped ISO 29100 reference.
- #25 Removed Encrypt only option.
- #59 Same with #25.

-13

- add TLS Security Consideration section
- replace RFC7525 reference with BCP195
- moved front tag in FETT reference to fix XML structure
- changes reference from SoK to FETT

-12

- fixes #62 - Alexey Melnikov Discuss
- fixes #48 - OPSDIR Review : General - delete semicolors after list items
- fixes #58 - DP Comments for the Last Call
- fixes #57 - GENART - Remove "non-normative ... " from examples.
- fixes #45 - OPSDIR Review : Introduction - are attacks discovered or already opened
- fixes #49 - OPSDIR Review : Introduction - Inconsistent colons after initial sentence of list items.
- fixes #53 - OPSDIR Review : 6.2 JWS Signed Request Object - Clarify JOSE Header
- fixes #42 - OPSDIR Review : Introduction - readability of 'and' is confusing
- fixes #50 - OPSDIR Review : Section 4 Request Object - Clarify 'signed, encrypted, or signed and encrypted'
- fixes #39 - OPSDIR Review : Abstract - Explain/Clarify JWS and JWE
- fixed #50 - OPSDIR Review : Section 4 Request Object - Clarify 'signed, encrypted, or signed and encrypted'

- fixes #43 - OPSDIR Review : Introduction - 'properties' sounds awkward and are not exactly 'properties'
- fixes #56 - OPSDIR Review : 12 Acknowledgements - 'contribution is' => 'contribution are'
- fixes #55 - OPSDIR Review : 11.2.2 Privacy Considerations - ' It is in a way' => 'In a way, it is'
- fixes #54 - OPSDIR Review : 11 Privacy Considerations - 'and not specific' => 'and are not specific'
- fixes #51 - OPSDIR Review : Section 4 Request Object - 'It is fine' => 'It is recommended'
- fixes #47 - OPSDIR Review : Introduction - 'over- the- wire' => 'over-the-wire'
- fixes #46 - OPSDIR Review : Introduction - 'It allows' => 'The use of application security' for
- fixes #44 - OPSDIR Review : Introduction - 'has' => 'have'
- fixes #41 - OPSDIR Review : Introduction - missing 'is' before 'typically sent'
- fixes #38 - OPSDIR Review : Section 11 - Delete 'freely accessible' regarding ISO 29100

-11

- s/bing/being/
- Added history for -10

-10

- #20: KM1 -- some wording that is awkward in the TLS section.
- #21: KM2 - the additional attacks against OAuth 2.0 should also have a pointer
- #22: KM3 -- Nit: in the first line of 10.4:
- #23: KM4 -- Mention RFC6973 in Section 11 in addition to ISO 29100
- #24: SECDIR review: Section 4 -- Confusing requirements for sign+encrypt

- #25: SECDIR review: Section 6 -- authentication and integrity need not be provided if the requestor encrypts the token?
- #26: SECDIR Review: Section 10 -- why no reference for JWS algorithms?
- #27: SECDIR Review: Section 10.2 - how to do the agreement between client and server "a priori"?
- #28: SECDIR Review: Section 10.3 - Indication on "large entropy" and "short lifetime" should be indicated
- #29: SECDIR Review: Section 10.3 - Typo
- #30: SECDIR Review: Section 10.4 - typos and missing articles
- #31: SECDIR Review: Section 10.4 - Clearer statement on the lack of endpoint identifiers needed
- #32: SECDIR Review: Section 11 - ISO29100 needs to be moved to normative reference
- #33: SECDIR Review: Section 11 - Better English and Entropy language needed
- #34: Section 4: Typo
- #35: More Acknowledgment
- #36: DP - More precise qualification on Encryption needed.

-09

- Minor Editorial Nits.
- Section 10.4 added.
- Explicit reference to Security consideration (10.2) added in section 5 and section 5.2.
- , (add yourself) removed from the acknowledgment.

-08

- Applied changes proposed by Hannes on 2016-06-29 on IETF OAuth list recorded as https://bitbucket.org/Nat/oauth-jwsreq/issues/12/.

- TLS requirements added.
- Security Consideration reinforced.
- Privacy Consideration added.
- Introduction improved.

-07

- Changed the abbrev to OAuth JAR from oauth-jar.
- Clarified sig and enc methods.
- Better English.
- Removed claims from one of the example.
- Re-worded the URI construction.
- Changed the example to use request instead of request_uri.
- Clarified that Request Object parameters take precedence regardless of request or request_uri parameters were used.
- Generalized the language in 4.2.1 to convey the intent more clearly.
- Changed "Server" to "Authorization Server" as a clarification.
- Stopped talking about request_object_signing_alg.
- IANA considerations now reflect the current status.
- Added Brian Campbell to the contributors list. Made the lists alphabetic order based on the last names. Clarified that the affiliation is at the time of the contribution.
- Added "older versions of " to the reference to IE uri length limitations.
- Stopped talking about signed or unsigned JWS etc.
- 1.Introduction improved.

-06

- Added explanation on the 512 chars URL restriction.

- Updated Acknowledgements.

-05

- More alignment with OpenID Connect.

-04

- Fixed typos in examples. (request_url -> request_uri, cliend_id -> client_id)
- Aligned the error messages with the OAuth IANA registry.
- Added another rationale for having request object.

-03

- Fixed the non-normative description about the advantage of static signature.
- Changed the requirement for the parameter values in the request itself and the request object from 'MUST MATCH" to 'Req Obj takes precedence.

-02

- Now that they are RFCs, replaced JWS, JWE, etc. with RFC numbers.

-01

- Copy Edits.

15. References

15.1. Normative References

[BCP195] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 7525, May 2015.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <https://www.rfc-editor.org/info/rfc3629>.

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

[RFC6125] Saint-Andre, P. and J. Hodges, "Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)", RFC 6125, DOI 10.17487/RFC6125, March 2011, <https://www.rfc-editor.org/info/rfc6125>.

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

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

[RFC6819] Lodderstedt, T., Ed., McGloin, M., and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations", RFC 6819, DOI 10.17487/RFC6819, January 2013, <https://www.rfc-editor.org/info/rfc6819>.

[RFC6973] Cooper, A., Tschofenig, H., Aboba, B., Peterson, J., Morris, J., Hansen, M., and R. Smith, "Privacy Considerations for Internet Protocols", RFC 6973, DOI 10.17487/RFC6973, July 2013, <https://www.rfc-editor.org/info/rfc6973>.

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

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <https://www.rfc-editor.org/info/rfc7230>.

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

[RFC7516] Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)", RFC 7516, DOI 10.17487/RFC7516, May 2015, <https://www.rfc-editor.org/info/rfc7516>.

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

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

[RFC8141] Saint-Andre, P. and J. Klensin, "Uniform Resource Names (URNs)", RFC 8141, DOI 10.17487/RFC8141, April 2017, <https://www.rfc-editor.org/info/rfc8141>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

15.2. Informative References

[BASIN] Basin, D., Cremers, C., and S. Meier, "Provably Repairing the ISO/IEC 9798 Standard for Entity Authentication", Journal of Computer Security - Security and Trust Principles Volume 21 Issue 6, Pages 817-846, November 2013, <https://www.cs.ox.ac.uk/people/cas.cremers/downloads/papers/BCM2012-iso9798.pdf>.

[FETT] Fett, D., Kusters, R., and G. Schmitz, "A Comprehensive Formal Security Analysis of OAuth 2.0", CCS '16 Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security Pages 1204-1215 , October 2016, <https://infsec.uni-trier.de/people/publications/paper/FettKuestersSchmitz-CCS-2016.pdf>.

[OpenID.Core] Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0", OpenID Foundation Standards, February 2014, <http://openid.net/specs/openid-connect-core-1_0.html>.

Authors' Addresses

Nat Sakimura
Nomura Research Institute
Otemachi Financial City Grand Cube, 1-9-2 Otemachi
Chiyoda-ku, Tokyo 100-0004
Japan

Phone: +81-3-5533-2111
Email: n-sakimura@nri.co.jp
URI: http://nat.sakimura.org/

John Bradley
Yubico
Casilla 177, Sucursal Talagante
Talagante, RM
Chile

Phone: +1.202.630.5272
Email: ve7jtb@ve7jtb.com
URI: http://www.thread-safe.com/

OAuth
Internet-Draft
Intended status: Informational
Expires: January 9, 2017

P. Hunt, Ed.
Oracle Corporation
J. Richer

W. Mills

P. Mishra
Oracle Corporation
H. Tschofenig
ARM Limited
July 8, 2016

OAuth 2.0 Proof-of-Possession (PoP) Security Architecture
draft-ietf-oauth-pop-architecture-08.txt

Abstract

The OAuth 2.0 bearer token specification, as defined in RFC 6750, allows any party in possession of a bearer token (a "bearer") to get access to the associated resources (without demonstrating possession of a cryptographic key). To prevent misuse, bearer tokens must be protected from disclosure in transit and at rest.

Some scenarios demand additional security protection whereby a client needs to demonstrate possession of cryptographic keying material when accessing a protected resource. This document motivates the development of the OAuth 2.0 proof-of-possession security mechanism.

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 January 9, 2017.

Table of Contents

1. Introduction

The OAuth 2.0 protocol family ([RFC6749], [RFC6750], and [RFC6819]) offer a single token type known as the "bearer" token to access protected resources. RFC 6750 [RFC6750] specifies the bearer token mechanism and defines it as follows:

> "A security token with the property that any party in possession of the token (a "bearer") can use the token in any way that any other party in possession of it can. Using a bearer token does not require a bearer to prove possession of cryptographic key material."

The bearer token meets the security needs of a number of use cases the OAuth 2.0 protocol had originally been designed for. There are, however, other scenarios that require stronger security properties and ask for active participation of the OAuth client in form of cryptographic computations when presenting an access token to a resource server.

This document outlines additional use cases requiring stronger security protection in Section 3, identifies threats in Section 4, proposes different ways to mitigate those threats in Section 6, outlines an architecture for a solution that builds on top of the existing OAuth 2.0 framework in Section 7, and concludes with a requirements list 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 [RFC2119], with the important qualification that, unless otherwise stated, these terms apply to the design of the protocol, not its implementation or application.

3. Use Cases

The main use case that motivates improvement upon "bearer" token security is the desire of resource servers to obtain additional assurance that the client is indeed authorized to present an access token. The expectation is that the use of additional credentials (symmetric or asymmetric keying material) will encourage developers to take additional precautions when transferring and storing access token in combination with these credentials.

Additional use cases listed below provide further requirements for the solution development. Note that a single solution does not necessarily need to offer support for all use cases.

3.1. Preventing Access Token Re-Use by the Resource Server

In a scenario where a resource server receives a valid access token, the resource server then re-uses it with other resource server. The reason for re-use may be malicious or may well be legitimate. In a legitimate case, the intent is to support chaining of computations whereby a resource server needs to consult other third party resource servers to complete a requested operation. In both cases it may be assumed that the scope and audience of the access token is sufficiently defined that to allow such a re-use. For example, imagine a case where a company operates email services as well as picture sharing services and that company had decided to issue access tokens with a scope and audience that allows access to both services.

With this use case the desire is to prevent such access token re-use. This also implies that the legitimate use cases require additional enhancements for request chaining.

3.2. TLS and DTLS Channel Binding Support

In this use case we consider the scenario where an OAuth 2.0 request to a protected resource is secured using TLS or DTLS (see [RFC4347]), but the client and the resource server demand that the underlying TLS/DTLS exchange is bound to additional application layer security to prevent cases where the TLS/DTLS connection is terminated at a TLS/DTLS intermediary, which splits the TLS/DTLS connection into two separate connections.

In this use case additional information should be conveyed to the resource server to ensure that no entity entity has tampered with the TLS/DTLS connection.

3.3. Access to a Non-TLS Protected Resource

This use case is for a web client that needs to access a resource that makes data available (such as videos) without offering integrity and confidentiality protection using TLS. Still, the initial resource request using OAuth, which includes the access token, must be protected against various threats (e.g., token replay, token modification).

While it is possible to utilize bearer tokens in this scenario with TLS protection when the request to the protected resource is made, as described in [RFC6750], there may be the desire to avoid using TLS

between the client and the resource server at all. In such a case the bearer token approach is not possible since it relies on TLS for ensuring integrity and confidentiality protection of the access token exchange since otherwise replay attacks are possible: First, an eavesdropper may steal an access token and present it at a different resource server. Second, an eavesdropper may steal an access token and replay it against the same resource server at a later point in time. In both cases, if the attack is successful, the adversary gets access to the resource owners data or may perform an operation selected by the adversary (e.g., sending a message). Note that the adversary may obtain the access token (if the recommendations in [RFC6749] and [RFC6750] are not followed) using a number of ways, including eavesdropping the communication on the wireless link.

Consequently, the important assumption in this use case is that a resource server does not have TLS support and the security solution should work in such a scenario. Furthermore, it may not be necessary to provide authentication of the resource server towards the client.

3.4. Offering Application Layer End-to-End Security

In Web deployments resource servers are often placed behind load balancers, which are deployed by the same organization that operates the resource servers. These load balancers may terminate the TLS connection setup and HTTP traffic is transmitted without TLS protection from the load balancer to the resource server. With application layer security in addition to the underlying TLS security it is possible to allow application servers to perform cryptographic verification on an end-to-end basis.

The key aspect in this use case is therefore to offer end-to-end security in the presence of load balancers via application layer security. Enterprise networks also deploy proxies that inspect traffic and thereby break TLS.

4. Security and Privacy Threats

The following list presents several common threats against protocols utilizing some form of token. This list of threats is based on NIST Special Publication 800-63 [NIST800-63]. We exclude a discussion of threats related to any form of identity proofing and authentication of the resource owner to the authorization server since these procedures are not part of the OAuth 2.0 protocol specification itself.

Token manufacture/modification:

An attacker may generate a bogus token or modify the token content (such as authentication or attribute statements) of an existing token, causing resource server to grant inappropriate access to the client. For example, an attacker may modify the token to extend the validity period. A client, which MAY be a normal client or MAY be assumed to be constrained (see [RFC7252]), may modify the token to have access to information that they should not be able to view.

Token disclosure:

Tokens may contain personal data, such as real name, age or birthday, payment information, etc.

Token redirect:

An attacker uses the token generated for consumption by the resource server to obtain access to another resource server.

Token reuse:

An attacker attempts to use a token that has already been used once with a resource server. The attacker may be an eavesdropper who observes the communication exchange or, worse, one of the communication end points. A client may, for example, leak access tokens because it cannot keep secrets confidential. A client may also reuse access tokens for some other resource servers. Finally, a resource server may use a token it had obtained from a client and use it with another resource server that the client interacts with. A resource server, offering relatively unimportant application services, may attempt to use an access token obtained from a client to access a high-value service, such as a payment service, on behalf of the client using the same access token.

Token repudiation:

Token repudiation refers to a property whereby a resource server is given an assurance that the authorization server cannot deny to have created a token for the client.

5. Requirements

RFC 4962 [RFC4962] gives useful guidelines for designers of authentication and key management protocols. While RFC 4962 was written with the AAA framework used for network access authentication in mind the offered suggestions are useful for the design of other key management systems as well. The following requirements list

applies OAuth 2.0 terminology to the requirements outlined in RFC 4962.

These requirements include

Cryptographic Algorithm Independent:

The key management protocol MUST be cryptographic algorithm independent.

Strong, fresh session keys:

Session keys MUST be strong and fresh. Each session deserves an independent session key, i.e., one that is generated specifically for the intended use. In context of OAuth this means that keying material is created in such a way that can only be used by the combination of a client instance, protected resource, and authorization scope.

Limit Key Scope:

Following the principle of least privilege, parties MUST NOT have access to keying material that is not needed to perform their role. Any protocol that is used to establish session keys MUST specify the scope for session keys, clearly identifying the parties to whom the session key is available.

Replay Detection Mechanism:

The key management protocol exchanges MUST be replay protected. Replay protection allows a protocol message recipient to discard any message that was recorded during a previous legitimate dialogue and presented as though it belonged to the current dialogue.

Authenticate All Parties:

Each party in the key management protocol MUST be authenticated to the other parties with whom they communicate. Authentication mechanisms MUST maintain the confidentiality of any secret values used in the authentication process. Secrets MUST NOT be sent to another party without confidentiality protection.

Authorization:

Client and resource server authorization MUST be performed. These entities MUST demonstrate possession of the appropriate keying material, without disclosing it. Authorization is REQUIRED

whenever a client interacts with an authorization server. Authorization checking prevents an elevation of privilege attack.

Keying Material Confidentiality and Integrity:

While preserving algorithm independence, confidentiality and integrity of all keying material MUST be maintained.

Confirm Cryptographic Algorithm Selection:

The selection of the "best" cryptographic algorithms SHOULD be securely confirmed. The mechanism SHOULD detect attempted roll-back attacks.

Uniquely Named Keys:

Key management proposals require a robust key naming scheme, particularly where key caching is supported. The key name provides a way to refer to a key in a protocol so that it is clear to all parties which key is being referenced. Objects that cannot be named cannot be managed. All keys MUST be uniquely named, and the key name MUST NOT directly or indirectly disclose the keying material.

Prevent the Domino Effect:

Compromise of a single client MUST NOT compromise keying material held by any other client within the system, including session keys and long-term keys. Likewise, compromise of a single resource server MUST NOT compromise keying material held by any other Resource Server within the system. In the context of a key hierarchy, this means that the compromise of one node in the key hierarchy must not disclose the information necessary to compromise other branches in the key hierarchy. Obviously, the compromise of the root of the key hierarchy will compromise all of the keys; however, a compromise in one branch MUST NOT result in the compromise of other branches. There are many implications of this requirement; however, two implications deserve highlighting. First, the scope of the keying material must be defined and understood by all parties that communicate with a party that holds that keying material. Second, a party that holds keying material in a key hierarchy must not share that keying material with parties that are associated with other branches in the key hierarchy.

Bind Key to its Context:

Keying material MUST be bound to the appropriate context. The context includes the following.

* The manner in which the keying material is expected to be used.

* The other parties that are expected to have access to the keying material.

* The expected lifetime of the keying material. Lifetime of a child key SHOULD NOT be greater than the lifetime of its parent in the key hierarchy.

Any party with legitimate access to keying material can determine its context. In addition, the protocol MUST ensure that all parties with legitimate access to keying material have the same context for the keying material. This requires that the parties are properly identified and authenticated, so that all of the parties that have access to the keying material can be determined. The context will include the client and the resource server identities in more than one form.

Authorization Restriction:

If client authorization is restricted, then the client SHOULD be made aware of the restriction.

Client Identity Confidentiality:

A client has identity confidentiality when any party other than the resource server and the authorization server cannot sufficiently identify the client within the anonymity set. In comparison to anonymity and pseudonymity, identity confidentiality is concerned with eavesdroppers and intermediaries. A key management protocol SHOULD provide this property.

Resource Owner Identity Confidentiality:

Resource servers SHOULD be prevented from knowing the real or pseudonymous identity of the resource owner, since the authorization server is the only entity involved in verifying the resource owner’s identity.

Collusion:

Resource servers that collude can be prevented from using information related to the resource owner to track the individual. That is, two different resource servers can be prevented from determining that the same resource owner has authenticated to both

of them. Authorization servers MUST bind different keying material to access tokens used for resource servers from different origins (or similar concepts in the app world).

AS-to-RS Relationship Anonymity:

For solutions using asymmetric key cryptography the client MAY conceal information about the resource server it wants to interact with. The authorization server MAY reject such an attempt since it may not be able to enforce access control decisions.

Channel Binding:

A solution MUST enable support for channel bindings. The concept of channel binding, as defined in [RFC5056], allows applications to establish that the two end-points of a secure channel at one network layer are the same as at a higher layer by binding authentication at the higher layer to the channel at the lower layer.

There are performance concerns with the use of asymmetric cryptography. Although symmetric key cryptography offers better performance asymmetric cryptography offers additional security properties. A solution MUST therefore offer the capability to support both symmetric as well as asymmetric keys.

There are threats that relate to the experience of the software developer as well as operational practices. Verifying the servers identity in TLS is discussed at length in [RFC6125].

A number of the threats listed in Section 4 demand protection of the access token content and a standardized solution, for example, in the form of a JSON-based format, is available with the JWT [RFC7519].

6. Threat Mitigation

A large range of threats can be mitigated by protecting the content of the token, for example using a digital signature or a keyed message digest. Alternatively, the content of the token could be passed by reference rather than by value (requiring a separate message exchange to resolve the reference to the token content).

To simplify discussion in the following example we assume that the token itself cannot be modified by the client, either due to cryptographic protection (such as signature or encryption) or use of a reference value with sufficient entropy and associated secure lookup. The token remains opaque to the client. These are characteristics shared with bearer tokens and more information on

best practices can be found in [RFC6819] and in the security considerations section of [RFC6750].

To deal with token redirect it is important for the authorization server to include the identifier of the intended recipient - the resource server. A resource server must not be allowed to accept access tokens that are not meant for its consumption.

To provide protection against token disclosure two approaches are possible, namely (a) not to include sensitive information inside the token or (b) to ensure confidentiality protection. The latter approach requires at least the communication interaction between the client and the authorization server as well as the interaction between the client and the resource server to experience confidentiality protection. As an example, TLS with a ciphersuite that offers confidentiality protection has to be applied as per [RFC7525]. Encrypting the token content itself is another alternative. In our scenario the authorization server would, for example, encrypt the token content with a symmetric key shared with the resource server.

To deal with token reuse more choices are available.

6.1. Confidentiality Protection

In this approach confidentiality protection of the exchange is provided on the communication interfaces between the client and the resource server, and between the client and the authorization server. No eavesdropper on the wire is able to observe the token exchange. Consequently, a replay by a third party is not possible. An authorization server wants to ensure that it only hands out tokens to clients it has authenticated first and who are authorized. For this purpose, authentication of the client to the authorization server will be a requirement to ensure adequate protection against a range of attacks. This is, however, true for the description in Section 6.2 and Section 6.3 as well. Furthermore, the client has to make sure it does not distribute (or leak) the access token to entities other than the intended the resource server. For that purpose the client will have to authenticate the resource server before transmitting the access token.

6.2. Sender Constraint

Instead of providing confidentiality protection, the authorization server could also put the identifier of the client into the protected token with the following semantic: 'This token is only valid when presented by a client with the following identifier.' When the access token is then presented to the resource server how does it

know that it was provided by the client? It has to authenticate the client! There are many choices for authenticating the client to the resource server, for example by using client certificates in TLS [RFC5246], or pre-shared secrets within TLS [RFC4279]. The choice of the preferred authentication mechanism and credential type may depend on a number of factors, including

- security properties
- available infrastructure
- library support
- credential cost (financial)
- performance
- integration into the existing IT infrastructure
- operational overhead for configuration and distribution of credentials

This long list hints to the challenge of selecting at least one mandatory-to-implement client authentication mechanism.

6.3. Key Confirmation

A variation of the mechanism of sender authentication, described in Section 6.2, is to replace authentication with the proof-of-possession of a specific (session) key, i.e., key confirmation. In this model the resource server would not authenticate the client itself but would rather verify whether the client knows the session key associated with a specific access token. Examples of this approach can be found with the OAuth 1.0 MAC token [RFC5849], and Kerberos [RFC4120] when utilizing the AP_REQ/AP_REP exchange (see also [I-D.hardjono-oauth-kerberos] for a comparison between Kerberos and OAuth).

To illustrate key confirmation, the first example is borrowed from Kerberos and use symmetric key cryptography. Assume that the authorization server shares a long-term secret with the resource server, called K(Authorization Server-Resource Server). This secret would be established between them out-of-band. When the client requests an access token the authorization server creates a fresh and unique session key Ks and places it into the token encrypted with the long term key K(Authorization Server-Resource Server). Additionally, the authorization server attaches Ks to the response message to the client (in addition to the access token itself) over a

confidentiality protected channel. When the client sends a request to the resource server it has to use Ks to compute a keyed message digest for the request (in whatever form or whatever layer). The resource server, when receiving the message, retrieves the access token, verifies it and extracts K(Authorization Server-Resource Server) to obtain Ks. This key Ks is then used to verify the keyed message digest of the request message.

Note that in this example one could imagine that the mechanism to protect the token itself is based on a symmetric key based mechanism to avoid any form of public key infrastructure but this aspect is not further elaborated in the scenario.

A similar mechanism can also be designed using asymmetric cryptography. When the client requests an access token the authorization server creates an ephemeral public / privacy key pair (PK/SK) and places the public key PK into the protected token. When the authorization server returns the access token to the client it also provides the PK/SK key pair over a confidentiality protected channel. When the client sends a request to the resource server it has to use the privacy key SK to sign the request. The resource server, when receiving the message, retrieves the access token, verifies it and extracts the public key PK. It uses this ephemeral public key to verify the attached signature.

6.4. Summary

As a high level message, there are various ways the threats can be mitigated. While the details of each solution are somewhat different, they all accomplish the goal of mitigating the threats.

The three approaches are:

Confidentiality Protection:

The weak point with this approach, which is briefly described in Section 6.1, is that the client has to be careful to whom it discloses the access token. What can be done with the token entirely depends on what rights the token entitles the presenter and what constraints it contains. A token could encode the identifier of the client but there are scenarios where the client is not authenticated to the resource server or where the identifier of the client rather represents an application class rather than a single application instance. As such, it is possible that certain deployments choose a rather liberal approach to security and that everyone who is in possession of the access token is granted access to the data.

Sender Constraint:

The weak point with this approach, which is briefly described in Section 6.2, is to setup the authentication infrastructure such that clients can be authenticated towards resource servers. Additionally, the authorization server must encode the identifier of the client in the token for later verification by the resource server. Depending on the chosen layer for providing client-side authentication there may be additional challenges due to Web server load balancing, lack of API access to identity information, etc.

Key Confirmation:

The weak point with this approach, see Section 6.3, is the increased complexity: a complete key distribution protocol has to be defined.

In all cases above it has to be ensured that the client is able to keep the credentials secret.

7. Architecture

The proof-of-possession security concept assumes that the authorization server acts as a trusted third party that binds keys to access tokens. These keys are then used by the client to demonstrate the possession of the secret to the resource server when accessing the resource. The resource server, when receiving an access token, needs to verify that the key used by the client matches the one included in the access token.

There are slight differences between the use of symmetric keys and asymmetric keys when they are bound to the access token and the subsequent interaction between the client and the authorization server when demonstrating possession of these keys. Figure 1 shows the symmetric key procedure and Figure 2 illustrates how asymmetric keys are used. While symmetric cryptography provides better performance properties the use of asymmetric cryptography allows the client to keep the private key locally and never expose it to any other party.

For example, with the JSON Web Token (JWT) [RFC7519] a standardized format for access tokens is available. The necessary elements to bind symmetric or asymmetric keys to a JWT are described in [I-D.ietf-oauth-proof-of-possession].

Note: The negotiation of cryptographic algorithms between the client and the authorization server is not shown in the examples below and

assumed to be present in a protocol solution to meet the requirements for crypto-agility.

7.1. Client and Authorization Server Interaction

7.1.1. Symmetric Keys

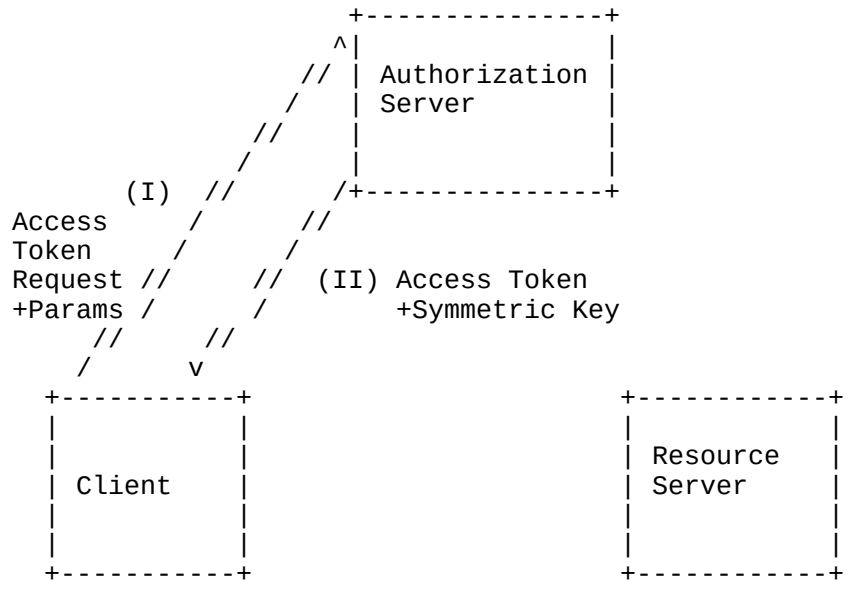

Figure 1: Interaction between the Client and the Authorization Server (Symmetric Keys).

In order to request an access token the client interacts with the authorization server as part of the a normal grant exchange, as shown in Figure 1. However, it needs to include additional information elements for use with the PoP security mechanism, as depicted in message (I). In message (II) the authorization server then returns the requested access token. In addition to the access token itself, the symmetric key is communicated to the client. This symmetric key is a unique and fresh session key with sufficient entropy for the given lifetime. Furthermore, information within the access token ties it to this specific symmetric key.

Note: For this security mechanism to work the client as well as the resource server need to have access to the session key. While the key transport mechanism from the authorization server to the client has been explained in the previous paragraph there are three ways for communicating this session key from the authorization server to the resource server, namely

Embedding the symmetric key inside the access token itself. This requires that the symmetric key is confidentiality protected.

The resource server queries the authorization server for the symmetric key. This is an approach envisioned by the token introspection endpoint [RFC7662].

The authorization server and the resource server both have access to the same back-end database. Smaller, tightly coupled systems might prefer such a deployment strategy.

7.1.2. Asymmetric Keys

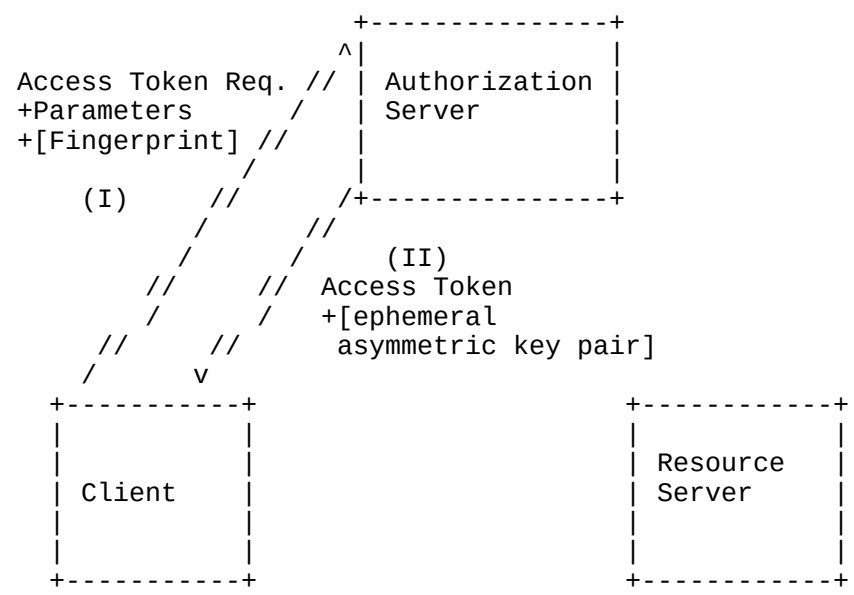

Figure 2: Interaction between the Client and the Authorization Server (Asymmetric Keys).

The use of asymmetric keys is slightly different since the client or the server could be involved in the generation of the ephemeral key pair. This exchange is shown in Figure 1. If the client generates the key pair it either includes a fingerprint of the public key or the public key in the request to the authorization server. The authorization server would include this fingerprint or public key in the confirmation claim inside the access token and thereby bind the asymmetric key pair to the token. If the client did not provide a fingerprint or a public key in the request then the authorization server is asked to create an ephemeral asymmetric key pair, binds the fingerprint of the public key to the access token, and returns the

asymmetric key pair (public and private key) to the client. Note that there is a strong preference for generating the private/public key pair locally at the client rather than at the server.

7.2. Client and Resource Server Interaction

The specification describing the interaction between the client and the authorization server, as shown in Figure 1 and in Figure 2, can be found in [I-D.ietf-oauth-pop-key-distribution].

Once the client has obtained the necessary access token and keying material it can start to interact with the resource server. To demonstrate possession of the key bound to the access token it needs to apply this key to the request by computing a keyed message digest (i.e., a symmetric key-based cryptographic primitive) or a digital signature (i.e., an asymmetric cryptographic computation). When the resource server receives the request it verifies it and decides whether access to the protected resource can be granted. This exchange is shown in Figure 3.

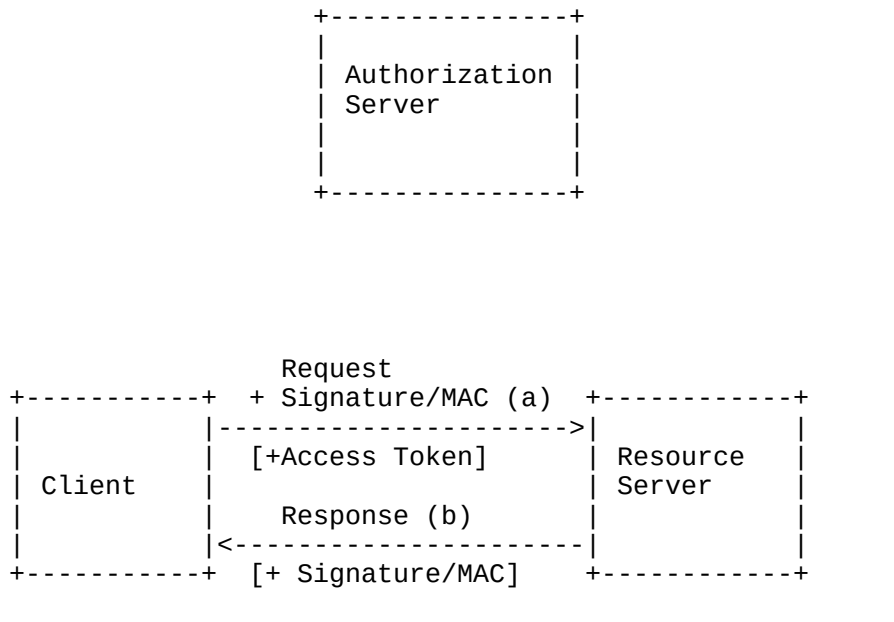

Figure 3: Client Demonstrates PoP.

The specification describing the ability to sign the HTTP request from the client to the resource server can be found in [I-D.ietf-oauth-signed-http-request].

7.3. Resource and Authorization Server Interaction (Token Introspection)

So far the examples talked about access tokens that are passed by value and allow the resource server to make authorization decisions immediately after verifying the request from the client. In some deployments a real-time interaction between the authorization server and the resource server is envisioned that lowers the need to pass self-contained access tokens around. In that case the access token merely serves as a handle or a reference to state stored at the authorization server. As a consequence, the resource server cannot autonomously make an authorization decision when receiving a request

from a client but has to consult the authorization server. This can, for example, be done using the token introspection endpoint (see [RFC7662]). Figure 4 shows the protocol interaction graphically. Despite the additional token exchange previous descriptions about associating symmetric and asymmetric keys to the access token are still applicable to this scenario.

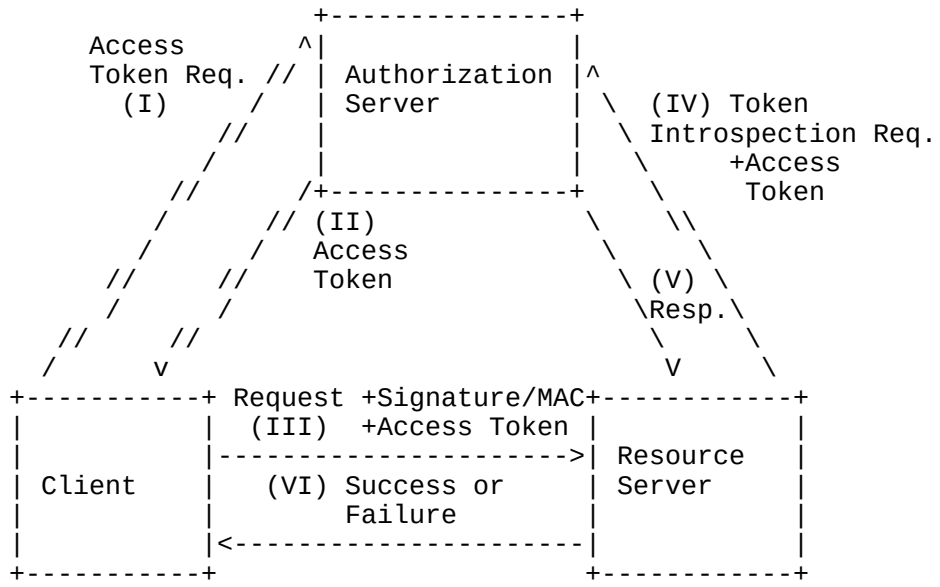

Figure 4: Token Introspection and Access Token Handles.

8. Security Considerations

The purpose of this document is to provide use cases, requirements, and motivation for developing an OAuth security solution extending Bearer Tokens. As such, this document is only about security.

9. IANA Considerations

This document does not require actions by IANA.

10. Acknowledgments

This document is the result of conference calls late 2012/early 2013 and in design team conference calls February 2013 of the IETF OAuth working group. The following persons (in addition to the OAuth WG chairs, Hannes Tschofenig, and Derek Atkins) provided their input during these calls: Bill Mills, Justin Richer, Phil Hunt, Prateek Mishra, Mike Jones, George Fletcher, Leif Johansson, Lucy Lynch, John

Bradley, Tony Nadalin, Klaas Wierenga, Thomas Hardjono, Brian Campbell

In the appendix of this document we reuse content from [RFC4962] and the authors would like thank Russ Housely and Bernard Aboba for their work on RFC 4962.

We would like to thank Reddy Tirumaleswar for his review.

11. References

11.1. Normative References

[I-D.ietf-oauth-pop-key-distribution]
Bradley, J., Hunt, P., Jones, M., and H. Tschofenig, "OAuth 2.0 Proof-of-Possession: Authorization Server to Client Key Distribution", draft-ietf-oauth-pop-key-distribution-02 (work in progress), October 2015.

[I-D.ietf-oauth-proof-of-possession]
Jones, M., Bradley, J., and H. Tschofenig, "Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs)", draft-ietf-oauth-proof-of-possession-11 (work in progress), December 2015.

[I-D.ietf-oauth-signed-http-request]
Richer, J., Bradley, J., and H. Tschofenig, "A Method for Signing HTTP Requests for OAuth", draft-ietf-oauth-signed-http-request-02 (work in progress), February 2016.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <http://www.rfc-editor.org/info/rfc5246>.

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

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

[RFC7525] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 7525, DOI 10.17487/RFC7525, May 2015, <http://www.rfc-editor.org/info/rfc7525>.

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

11.2. Informative References

[I-D.hardjono-oauth-kerberos]
Hardjono, T., "OAuth 2.0 support for the Kerberos V5 Authentication Protocol", draft-hardjono-oauth-kerberos-01 (work in progress), December 2010.

[NIST800-63]
Burr, W., Dodson, D., Perlner, R., Polk, T., Gupta, S., and E. Nabbus, "NIST Special Publication 800-63-1, INFORMATION SECURITY", December 2008.

[RFC4120] Neuman, C., Yu, T., Hartman, S., and K. Raeburn, "The Kerberos Network Authentication Service (V5)", RFC 4120, DOI 10.17487/RFC4120, July 2005, <http://www.rfc-editor.org/info/rfc4120>.

[RFC4279] Eronen, P., Ed. and H. Tschofenig, Ed., "Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)", RFC 4279, DOI 10.17487/RFC4279, December 2005, <http://www.rfc-editor.org/info/rfc4279>.

[RFC4347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security", RFC 4347, DOI 10.17487/RFC4347, April 2006, <http://www.rfc-editor.org/info/rfc4347>.

[RFC4962] Housley, R. and B. Aboba, "Guidance for Authentication, Authorization, and Accounting (AAA) Key Management", BCP 132, RFC 4962, DOI 10.17487/RFC4962, July 2007, <http://www.rfc-editor.org/info/rfc4962>.

[RFC5056] Williams, N., "On the Use of Channel Bindings to Secure Channels", RFC 5056, DOI 10.17487/RFC5056, November 2007, <http://www.rfc-editor.org/info/rfc5056>.

[RFC5849] Hammer-Lahav, E., Ed., "The OAuth 1.0 Protocol", RFC 5849, DOI 10.17487/RFC5849, April 2010, <http://www.rfc-editor.org/info/rfc5849>.

[RFC6125] Saint-Andre, P. and J. Hodges, "Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)", RFC 6125, DOI 10.17487/RFC6125, March 2011, <http://www.rfc-editor.org/info/rfc6125>.

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

[RFC6819] Lodderstedt, T., Ed., McGloin, M., and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations", RFC 6819, DOI 10.17487/RFC6819, January 2013, <http://www.rfc-editor.org/info/rfc6819>.

[RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, DOI 10.17487/RFC7252, June 2014, <http://www.rfc-editor.org/info/rfc7252>.

Authors' Addresses

Phil Hunt (editor)
Oracle Corporation

Email: phil.hunt@yahoo.com

Justin Richer

Email: ietf@justin.richer.org

William Mills

Email: wmills@yahoo-inc.com

Prateek Mishra
Oracle Corporation

Email: prateek.mishra@oracle.com

Hannes Tschofenig
ARM Limited
Hall in Tirol 6060
Austria

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

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 26, 2019

J. Bradley
Ping Identity
P. Hunt
Oracle Corporation
M. Jones
Microsoft
H. Tschofenig
Arm Ltd.
M. Mihaly
NIIF Institute

October 23, 2018

OAuth 2.0 Proof-of-Possession: Authorization Server to Client Key Distribution
draft-ietf-oauth-pop-key-distribution-04

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.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on April 26, 2019.

Table of Contents

1. Introduction

The work on proof-of-possession tokens, an extended token security mechanisms for OAuth 2.0, is motivated in [21]. This document defines the ability for the client request and to obtain PoP tokens from the authorization server. After successfully completing the exchange the client is in possession of a PoP token and the keying material bound to it. Clients that access protected resources then need to demonstrate knowledge of the secret key that is bound to the PoP token.

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). To demonstrate possession of the private/secret key to the resource server protocol mechanisms outside the scope of this document are used.

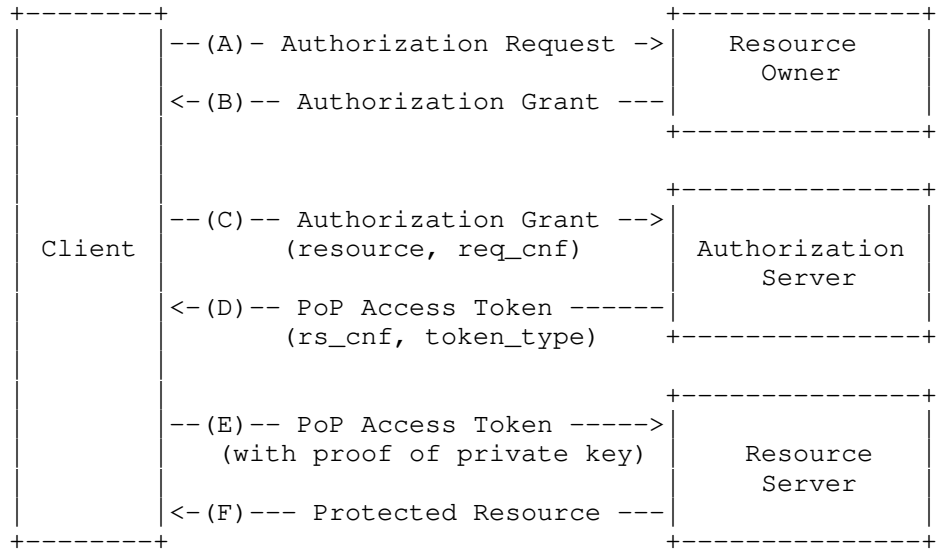

Figure 1: Augmented OAuth 2.0 Protocol Flow

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 [18] 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 specification 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 4.1 and the procedure for dealing with asymmetric keys is described in Section 4.2.

This document describes how the client requests and obtains a PoP access token from the authorization server for use with HTTPS-based

transport. The use of alternative transports, such as Constrained Application Protocol (CoAP), is described in [23].

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:

> In the context of this specification '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[7]

JWT: JSON Web Token[9]

JWS: JSON Web Signature[6]

JWK: JSON Web Key[5]

JWE: JSON Web Encryption[8]

CWT: CBOR Web Token[13]

COSE: CBOR Object Signing and Encryption[14]

3. 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 use the resource identicator defined in [15] when symmetric PoP tokens are used. For asymmetric PoP tokens the use of resource indicators is optional but recommended.

Step (2): The authorization server parses the request from the server and determines the suitable response based on OAuth 2.0 and the PoP token credential procedures.

Note that PoP access tokens may be encoded in a variety of ways:

JWT The access token may be encoded using the JSON Web Token (JWT) format [9]. The proof-of-possession token functionality is described in [10]. A JWT encoded PoP token MUST be protected against modification by either using a digital signature or a keyed message digest, as described in [6]. The JWT may also be encrypted using [8].

CWT [13] defines an alternative token format based on CBOR. The proof-of-possession token functionality is defined in [12]. A CWT encoded PoP token MUST be protected against modification by either using a digital signature or a keyed message digest, as described in [12].

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 [22].

Note that the OAuth 2.0 framework nor this specification does not mandate a specific PoP token format but using a standardized format will improve interoperability and will lead to better code re-use.

Application layer interactions between the client and the resource server are beyond the scope of this document.

4. Examples

This section provides a number of examples.

4.1. Symmetric Key Transport

4.1.1. Client-to-AS Request

The client starts with a request to the authorization server indicating that it is interested to obtain a token for https://www.example.com

```
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
&resource=https://www.example.com
```

Example Request to the Authorization Server

4.1.2. 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.

Figure 2 shows a response containing a token and a "cnf" parameter with a symmetric proof-of-possession key both encoded in a JSON-based serialization format. The "cnf" parameter contains the RFC 7517 [5] encoded key element.

```
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",
  "cnf":{
    {"keys":
     [
       {"kty":"oct",
         "alg":"A128KW",
         "k":"GawgguFyGrWKav7AX4VKUg"
               }
        ]
      }
    }
}
```

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

Note that the cnf payload in Figure 2 is not encrypted at the application layer since Transport Layer Security is used between the AS and the client and the content of the cnf payload is consumed by the client itself. Alternatively, a JWE could be used to encrypt the key distribution, as shown in Figure 3.

```
{
  "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",
  "cnf":{
      "jwe":
        "eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkExMjhDQkMtSFMyNTYifQ.
        (remainder of JWE omitted for brevity)"
      }
      }
}
```

Figure 3: Example: Encrypted Symmmetric Key

The content of the 'access_token' in JWT format contains the 'cnf' (confirmation) claim. 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 has to be present in a real deployment to mitigate a number of security threats.

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).

```
 {
    "iss": "https://server.example.com",
    "sub": "24400320",
    "aud": "s6BhdRkqt3",
    "nonce": "n-0S6_WzA2Mj",
    "exp": 1311281970,
    "iat": 1311280970,
    "cnf":{
      "jwe":
        "eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkExMjhDQkMtSFMyNTYifQ.
        (remainder of 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 since content is meant for consumption by the selected resource server only. The details are described in [21].

4.2. Asymmetric Key Transport

4.2.1. Client-to-AS Request

This example illustrates the case where an asymmetric key shall be bound to an access token. The client makes the following HTTPS request shown in Figure 5. Extra line breaks are for display purposes only.

```
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
&req_cnf=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 'req_cnf' parameter contains the ECC public key the client would like to associate with the access token (in JSON format).

```
"jwk":{
  "kty": "EC",
  "use": "sig",
  "crv": "P-256",
  "x": "18wHLeIgW9wVN6VD1Txgpqy2LszYkMf6J8njVAibvhM",
  "y": "-V4dS4UaLMgP_4fY4j8ir7cl1TXlFdAgcx55o7TkcSA"
 }
```

Figure 6: Client Providing Public Key to Authorization Server

4.2.2. 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. 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 "pop" 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",
  "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, as shown in Figure 8. The digital signature covering the access token offering authenticity and integrity protection is not shown below (but must be present).

```
{
  "iss":"xas.example.com",
  "aud":"http://auth.example.com",
  "exp":"1361398824",
  "nbf":"1360189224",
  "cnf":{
     "jwk" : {
       "kty" : "EC",
       "kid" : h'11',
       "crv" : "P-256",
       "x" : b64'usWxHK2PmfnHKwXPS54m0kTcGJ90UiglWiGahtagnv8',
       "y" : b64'IBOL+C3BttVivg+lSreASjpkttcsz+1rb7btKLv8EX4'
     }
   }
 }
```

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 key (as well as the public key).

5. Security Considerations

[21] 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 [16] 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.

6. IANA Considerations

6.1. OAuth Access Token Types

This specification registers the following error in the IANA "OAuth Access Token Types" [24] established by [16].

- Name: pop
- Change controller: IESG
- Specification document(s): [[this specification]]

6.2. OAuth Parameters Registration

This specification registers the following value in the IANA "OAuth Parameters" registry [24] established by [2].

- Parameter name: cnf_req
- Parameter usage location: authorization request, token request
- Change controller: IESG
- Specification document(s): [[this specification]]

- Parameter name: cnf
- Parameter usage location: authorization response, token response
- Change controller: IESG
- Specification document(s): [[this specification]]

- Parameter name: rs_cnf
- Parameter usage location: token response
- Change controller: IESG
- Specification document(s): [[this specification]]

6.3. OAuth Extensions Error Registration

This specification registers the following error in the IANA "OAuth Extensions Error Registry" [24] established by [2].

- Error name: invalid_token_type
- Error usage location: implicit grant error response, token error response
- Related protocol extension: token_type parameter
- Change controller: IESG
- Specification document(s): [[this specification]]

7. Acknowledgements

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

8. References

8.1. Normative References

[1] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

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

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

[4] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <https://www.rfc-editor.org/info/rfc5246>.

[5] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, <https://www.rfc-editor.org/info/rfc7517>.

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

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

[8] Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)", RFC 7516, DOI 10.17487/RFC7516, May 2015, <https://www.rfc-editor.org/info/rfc7516>.

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

[10] Jones, M., Bradley, J., and H. Tschofenig, "Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs)", RFC 7800, DOI 10.17487/RFC7800, April 2016, <https://www.rfc-editor.org/info/rfc7800>.

[11] Jones, M. and N. Sakimura, "JSON Web Key (JWK) Thumbprint", RFC 7638, DOI 10.17487/RFC7638, September 2015, <https://www.rfc-editor.org/info/rfc7638>.

[12] Jones, M., Seitz, L., Selander, G., Erdtman, S., and H. Tschofenig, "Proof-of-Possession Key Semantics for CBOR Web Tokens (CWTs)", draft-ietf-ace-cwt-proof-of-possession-03 (work in progress), June 2018.

[13] Jones, M., Wahlstroem, E., Erdtman, S., and H. Tschofenig, "CBOR Web Token (CWT)", RFC 8392, DOI 10.17487/RFC8392, May 2018, <https://www.rfc-editor.org/info/rfc8392>.

[14] Schaad, J., "CBOR Object Signing and Encryption (COSE)", RFC 8152, DOI 10.17487/RFC8152, July 2017, <https://www.rfc-editor.org/info/rfc8152>.

[15] Campbell, B., Bradley, J., and H. Tschofenig, "Resource Indicators for OAuth 2.0", draft-ietf-oauth-resource-indicators-01 (work in progress), October 2018.

8.2. Informative References

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

[17] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[18] Campbell, B., Mortimore, C., Jones, M., and Y. Goland, "Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants", RFC 7521, DOI 10.17487/RFC7521, May 2015, <https://www.rfc-editor.org/info/rfc7521>.

[19] Sakimura, N., Ed., Bradley, J., and N. Agarwal, "Proof Key for Code Exchange by OAuth Public Clients", RFC 7636, DOI 10.17487/RFC7636, September 2015, <https://www.rfc-editor.org/info/rfc7636>.

[20] Richer, J., Ed., Jones, M., Bradley, J., Machulak, M., and P. Hunt, "OAuth 2.0 Dynamic Client Registration Protocol", RFC 7591, DOI 10.17487/RFC7591, July 2015, <https://www.rfc-editor.org/info/rfc7591>.

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

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

[23] Seitz, L., Selander, G., Wahlstroem, E., Erdtman, S., and H. Tschofenig, "Authentication and Authorization for Constrained Environments (ACE) using the OAuth 2.0 Framework (ACE-OAuth)", draft-ietf-ace-oauth-authz-16 (work in progress), October 2018.

[24] IANA, "OAuth Parameters", October 2018.

[25] IANA, "JSON Web Token Claims", June 2018.

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 Ltd.
Absam 6067
Austria

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

Meszaros Mihaly
NIIF Institute
Hungary

Email: misi@niif.hu

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: June 20, 2016

M. Jones
Microsoft
J. Bradley
Ping Identity
H. Tschofenig
ARM Limited

December 18, 2015

Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs)
draft-ietf-oauth-proof-of-possession-11

Abstract

This specification defines how to declare in a JSON Web Token (JWT) that the presenter of the JWT possesses a particular proof-of-possession key and that the recipient can cryptographically confirm proof-of-possession of the key by the presenter. Being able to prove possession of a key 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 June 20, 2016.

Copyright Notice

Table of Contents

1. Introduction

This specification defines how a JSON Web Token [JWT] can declare that the presenter of the JWT possesses a particular proof-of-possession (PoP) key and that the recipient can cryptographically confirm proof-of-possession of the key by the presenter. Proof-of-possession of a key is also sometimes described as the presenter being a holder-of-key. The [I-D.ietf-oauth-pop-architecture] specification describes key confirmation, among other confirmation mechanisms. This specification defines how to communicate key confirmation key information in JWTs.

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

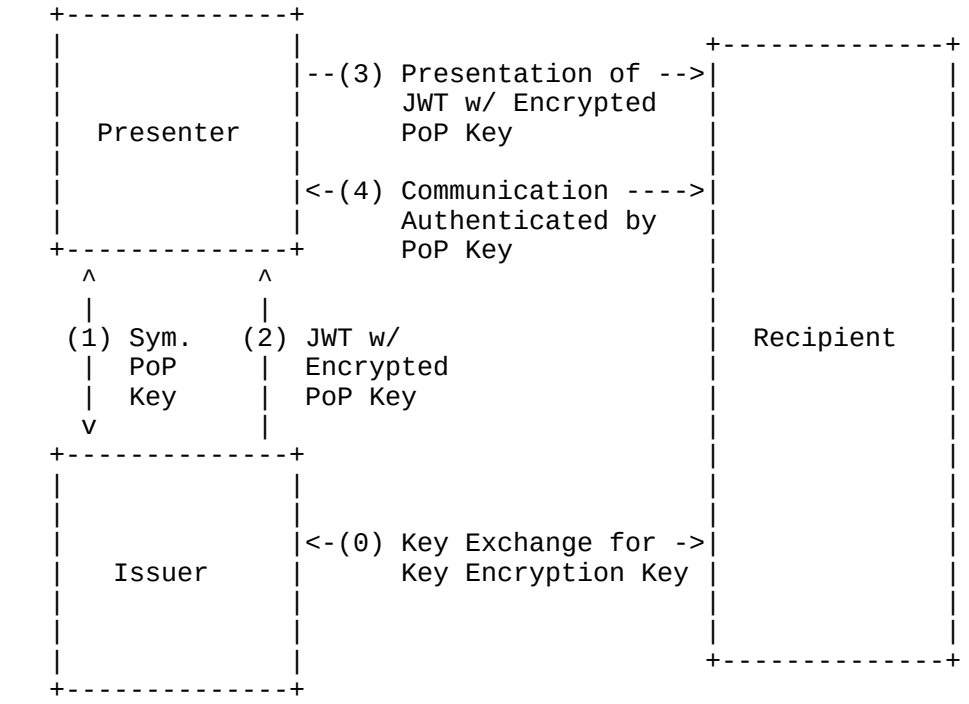

Figure 1: Proof-of-Possession with a Symmetric Key

In the case illustrated in Figure 1, either the presenter generates a symmetric key and privately sends it to the issuer (1) or the issuer generates a symmetric key and privately sends it to the presenter (1). The issuer generates a JWT with an encrypted copy of this symmetric key in the confirmation claim. This symmetric key is

encrypted with a key known only to the issuer and the recipient, which was previously established in step (0). The entire JWT is integrity protected by the issuer. The JWT is then (2) sent to the presenter. Now, the presenter is in possession of the symmetric key as well as the JWT (which includes the confirmation claim). When the presenter (3) presents the JWT to the recipient, it also needs to demonstrate possession of the symmetric key; the presenter, for example, (4) 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 key in 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 (4). This symmetric key mechanism described above is conceptually similar to the use of Kerberos tickets.

Note that for simplicity, the diagram above and associated text describe the direct use of symmetric keys without the use of derived keys. A more secure practice is to derive the symmetric keys actually used from secrets exchanged, such as the key exchanged in step (0), using a Key Derivation Function (KDF) and use the derived keys, rather than directly using the secrets exchanged.

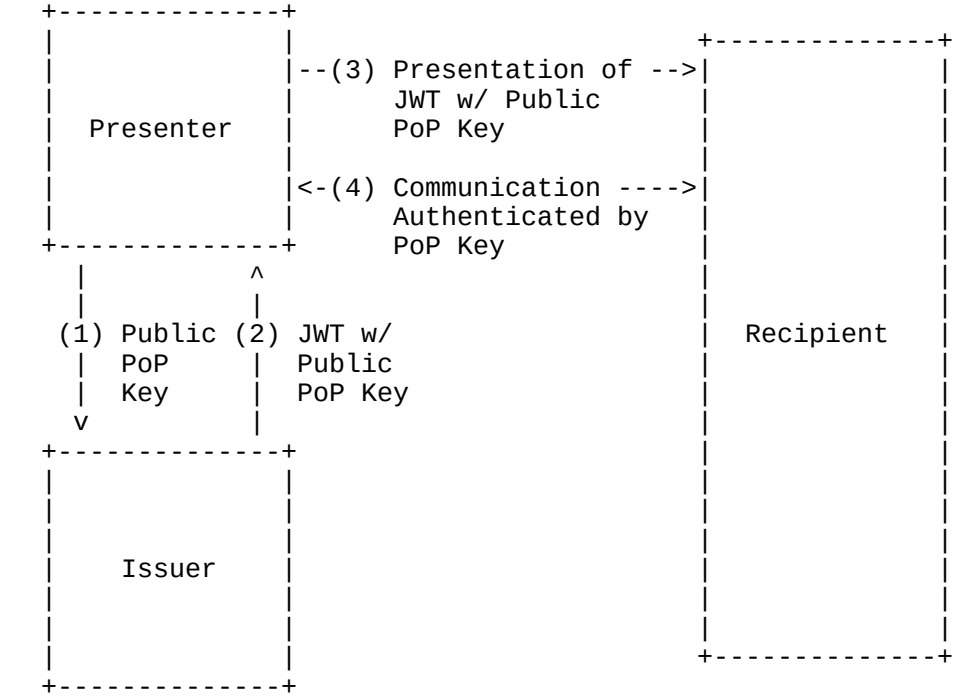

Figure 2: Proof-of-Possession with an Asymmetric Key

In the case illustrated in Figure 2, the presenter generates a public/private key pair and (1) sends the public key to the issuer, which creates a JWT that contains the public key (or an identifier for it) in the confirmation claim. The entire JWT is integrity protected using a digital signature to protect it against modifications. The JWT is then (2) sent to the presenter. When the presenter (3) presents the JWT to the recipient, it also needs to demonstrate possession of the private key. The presenter, for example, (4) uses the private key in a TLS exchange with the recipient or (4) signs a nonce with the private key. 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 or by checking the nonce signature.

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], JSON Web Key [JWK], and JSON Web Encryption [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.

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

3. Representations for Proof-of-Possession Keys

By including a "cnf" (confirmation) claim in a JWT, the issuer of the JWT declares that the presenter possesses a particular key, and that the recipient can cryptographically confirm that the presenter has possession of that key. The value of the "cnf" claim is a JSON object and the members of that object identify the proof-of-possession key.

The presenter can be identified in one of several ways by the JWT, depending upon the application requirements. If the JWT contains a "sub" (subject) claim [JWT], the presenter is normally the subject identified by the JWT. (In some applications, the subject identifier will be relative to the issuer identified by the "iss" (issuer) claim [JWT].) If the JWT contains no "sub" (subject) claim, the presenter is normally 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. Some use cases may require that both be present.

Another means used by some applications to identify the presenter is an explicit claim, such as the "azp" (authorized party) claim defined by OpenID Connect [OpenID.Core]. Ultimately, the means of identifying the presenter is application-specific, as is the means of confirming possession of the key that is communicated.

3.1. Confirmation Claim

The "cnf" (confirmation) claim is used in the JWT to contain members used to identify the proof-of-possession key. Other members of the "cnf" object may be defined 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.

The set of confirmation members that a JWT must contain to be considered valid is context dependent and is outside the scope of this specification. Specific applications of JWTs will require implementations to understand and process some confirmation members in particular ways. However, in the absence of such requirements,

all confirmation members that are not understood by implementations MUST be ignored.

This specification establishes the IANA "JWT Confirmation Methods" registry for these members in Section 6.2 and registers the members defined by this specification. Other specifications can register other members used for confirmation, including other members for conveying proof-of-possession keys, possibly using different key representations.

The "cnf" claim value MUST represent only a single proof-of-possession key; thus, at most one of the "jwk", "jwe", and "jku" confirmation values defined below may be present. Note that if an application needs to represent multiple proof-of-possession keys in the same JWT, one way for it to achieve this is to use other claim names, in addition to "cnf", to hold the additional proof-of-possession key information. These claims could use the same syntax and semantics as the "cnf" claim. Those claims would be defined by applications or other specifications and could be registered in the IANA "JSON Web Token Claims" registry [IANA.JWT.Claims].

3.2. Representation of an Asymmetric Proof-of-Possession Key

When the key held by the presenter is an asymmetric private key, the "jwk" member is a JSON Web Key [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,
 "cnf":{
   "jwk":{
     "kty": "EC",
     "use": "sig",
     "crv": "P-256",
     "x": "18wHLeIgW9wVN6VD1Txgpqy2LszYkMf6J8njVAibvhM",
     "y": "-V4dS4UaLMgP_4fY4j8ir7cl1TXlFdAgcx55o7TkcSA"
    }
  }
}
```

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.

The "jwk" member MAY also be used for a JWK representing a symmetric

key, provided that the JWT is encrypted so that the key is not revealed to unintended parties. If the JWT is not encrypted, the symmetric key MUST be encrypted as described below.

3.3. Representation of an Encrypted Symmetric Proof-of-Possession Key

When the key held by the presenter is a symmetric key, the "jwe" member is an encrypted JSON Web Key [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 7 of the JSON Web Key [JWK] specification.

The following example illustrates a symmetric key that could subsequently be encrypted for use in the "jwe" member:

```
{
 "kty": "oct",
 "alg": "HS256",
 "k": "ZoRSOrFzN_FzUA5XKMYoVHyzff5oRJxl-IXRtztJ6uE"
}
```

The UTF-8 [RFC3629] encoding of this JWK is 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": "RSA-OAEP",
 "enc": "A128CBC-HS256"
}
```

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

```
{
 "iss": "https://server.example.com",
 "sub": "24400320",
 "aud": "s6BhdRkqt3",
 "nonce": "n-0S6_WzA2Mj",
 "exp": 1311281970,
 "iat": 1311280970,
 "cnf":{
   "jwe":
     "eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkExMjhDQkMtSFMyNTYifQ.
     (remainder of JWE omitted for brevity)"
   }
}
```

3.4. Representation of a Key ID for a Proof-of-Possession Key

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 issuer of a JWT declares that the presenter 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:

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

The content of the "kid" value is application specific. For instance, some applications may choose to use a JWK Thumbprint [JWK.Thumbprint] value as the "kid" value.

3.5. Representation of a URL for a Proof-of-Possession Key

The proof-of-possession key can be passed by reference instead of being passed by value. This is done using the "jku" (JWK Set URL) member. Its value is a URI [RFC3986] that refers to a resource for a set of JSON-encoded public keys represented as a JWK Set [JWK], one of which is the proof-of-possession key. If there are multiple keys in the referenced JWK Set document, a "kid" member MUST also be included, with the referenced key’s JWK also containing the same "kid" value.

The protocol used to acquire the resource MUST provide integrity protection. An HTTP GET request to retrieve the JWK Set MUST use Transport Layer Security (TLS) [RFC5246] and the identity of the server MUST be validated, as per Section 6 of RFC 6125 [RFC6125].

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

```
{
 "iss": "https://server.example.com",
 "sub": "17760704",
 "aud": "https://client.example.org",
 "exp": 1440804813,
 "cnf":{
   "jku": "https://keys.example.net/pop-keys.json",
   "kid": "2015-08-28"
  }
}
```

3.6. 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 examples using the mechanisms defined in this specification, see [I-D.ietf-oauth-pop-architecture].

4. Security Considerations

All of the security considerations that are discussed in [JWT] also apply here. In addition, proof-of-possession introduces its own unique security issues. Possessing a key is only valuable if it is kept secret. Appropriate means must be used to ensure that unintended parties do not learn private key or symmetric key values.

Applications utilizing proof-of-possession should also utilize audience restriction, as described in Section 4.1.3 of [JWT], as it provides different protections. Proof-of-possession can be used by recipients to reject messages from unauthorized senders. Audience restriction can be used by recipients to reject messages intended for different recipients.

A recipient might not understand the "cnf" claim. Applications that

require the proof-of-possession keys communicated with it to be understood and processed must ensure that the parts of this specification that they use are implemented.

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 challenge for each interaction. Replay can also be avoided if a sub-key is derived from a shared secret that is specific to the instance of the PoP demonstration.

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.

5. Privacy Considerations

A proof-of-possession key can be used as a correlation handle if the same key is used with multiple parties. Thus, for privacy reasons, it is recommended that different proof-of-possession keys be used when interacting with different parties.

6. IANA Considerations

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

Values are registered on a Specification Required [RFC5226] basis after a three-week review period on the oauth-pop-reg-review@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 Experts may approve registration once they are satisfied that such a specification will be published. [[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.]]

Registration requests sent to the mailing list for review should use an appropriate subject (e.g., "Request to register JWT Confirmation

Method: example"). 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 Experts include 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, evaluating the security properties of the item being registered, and whether the registration makes sense.

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 Experts.

6.1. JSON Web Token Claims Registration

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

6.1.1. Registry Contents

- Claim Name: "cnf"
- Claim Description: Confirmation
- Change Controller: IESG
- Specification Document(s): Section 3.1 of [[this document]]

6.2. 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.

6.2.1. Registration Template

Confirmation Method Value:
: The name requested (e.g., "kid"). 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 Experts state that there is a compelling reason to allow an exception.

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

Change Controller:
For Standards Track RFCs, list the "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 or documents that specify the parameter, preferably including URIs that can be used to retrieve copies of the documents. An indication of the relevant sections may also be included but is not required.

6.2.2. Initial Registry Contents

- Confirmation Method Value: "jwk"
- Confirmation Method Description: JSON Web Key Representing Public Key
- Change Controller: IESG
- Specification Document(s): Section 3.2 of [[this document]]

- Confirmation Method Value: "jwe"
- Confirmation Method Description: Encrypted JSON Web Key
- Change Controller: IESG
- Specification Document(s): Section 3.3 of [[this document]]

- Confirmation Method Value: "kid"
- Confirmation Method Description: Key Identifier
- Change Controller: IESG
- Specification Document(s): Section 3.4 of [[this document]]

- Confirmation Method Value: "jku"
- Confirmation Method Description: JWK Set URL
- Change Controller: IESG
- Specification Document(s): Section 3.5 of [[this document]]

7. References

7.1. Normative References

[IANA.JWT.Claims]
IANA, "JSON Web Token Claims",
<http://www.iana.org/assignments/jwt>.

[JWE] Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)",

RFC 7516, DOI 10.17487/RFC7156, May 2015, <http://www.rfc-editor.org/info/rfc7516>.

[JWK] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7157, May 2015, <http://www.rfc-editor.org/info/rfc7517>.

[JWT] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7159, May 2015, <http://www.rfc-editor.org/info/rfc7519>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <http://www.rfc-editor.org/info/rfc3629>.

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

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, DOI 10.17487/RFC5226, May 2008, <http://www.rfc-editor.org/info/rfc5226>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <http://www.rfc-editor.org/info/rfc5246>.

[RFC6125] Saint-Andre, P. and J. Hodges, "Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)", RFC 6125, DOI 10.17487/RFC6125, March 2011, <http://www.rfc-editor.org/info/rfc6125>.

7.2. Informative References

[I-D.ietf-oauth-pop-architecture] Hunt, P., Richer, J., Mills, W., Mishra, P., and H. Tschofenig, "OAuth 2.0 Proof-of-Possession (PoP) Security Architecture", draft-ietf-oauth-pop-architecture-05 (work

in progress), October 2015.

[JWK.Thumbprint]
Jones, M. and N. Sakimura, "JSON Web Key (JWK) Thumbprint", RFC 7638, DOI 10.17487/RFC7638, September 2015, <http://www.rfc-editor.org/info/rfc7638>.

[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-2.0-os, March 2005.

[OpenID.Core]
Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0", November 2014, <http://openid.net/specs/openid-connect-core-1_0.html>.

Appendix A. Acknowledgements

The authors wish to thank Brian Campbell, Stephen Farrell, Barry Leiba, Kepeng Li, Chris Lonvick, James Manger, Kathleen Moriarty, Justin Richer, and Nat Sakimura for their reviews of the specification.

Appendix B. Document History

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

-11

- Addressed Sec-Dir review comments by Chris Lonvick and ballot comments by Stephen Farrell.

-10

- Addressed ballot comments by Barry Leiba.

-09

- Removed erroneous quotation marks around numeric "exp" claim values in examples.

-08

- Added security consideration about also utilizing audience restriction.

-07

- Addressed review comments by Hannes Tschofenig, Kathleen Moriarty, and Justin Richer. Changes were:

- Clarified that symmetric proof-of-possession keys can be generated by either the presenter or the issuer.

- Clarified that confirmation members that are not understood must be ignored unless otherwise specified by the application.

-06

- Added diagrams to the introduction.

-05

- Addressed review comments by Kepeng Li.

-04

- Allowed the use of "jwk" for symmetric keys when the JWT is encrypted.

- Added the "jku" (JWK Set URL) member.

- Added privacy considerations.

- Reordered sections so that the "cnf" (confirmation) claim is defined before it is used.

- Noted that applications can define new claim names, in addition to "cnf", to represent additional proof-of-possession keys, using the same representation as "cnf".

- Applied wording clarifications suggested by Nat Sakimura.

-03

- Separated the "jwk" and "jwe" confirmation members; the former represents a public key as a JWK and the latter represents a symmetric key as a JWE encrypted JWK.

- Changed the title to indicate that a proof-of-possession key is being communicated.

- Updated language that formerly assumed that the issuer was an OAuth 2.0 authorization server.
- Described ways that applications can choose to identify the presenter, including use of the "iss", "sub", and "azp" claims.
- Harmonized the registry language with that used in JWT [RFC 7519].
- Addressed other issues identified during working group last call.
- Referenced the JWT and JOSE RFCs.

-02

- Defined the terms Issuer, Presenter, and Recipient and updated their usage within the document.
- Added a description of a use case using an asymmetric proof-of-possession key to the introduction.
- Added the "kid" (key ID) confirmation method.
- These changes address the open issues identified in the previous draft.

-01

- Updated references.

-00

- 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

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 22, 2019

M. Jones
A. Nadalin
Microsoft
B. Campbell, Ed.
J. Bradley
Ping Identity
C. Mortimore
Salesforce

October 19, 2018

OAuth 2.0 Token Exchange
draft-ietf-oauth-token-exchange-16

Abstract

This specification defines a protocol for an HTTP- and JSON- based Security Token Service (STS) by defining how to request and obtain security tokens from OAuth 2.0 authorization servers, including security tokens employing impersonation and delegation.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on April 22, 2019.

Copyright Notice

Table of Contents

1. Introduction

A security token is a set of information that facilitates the sharing of identity and security information in heterogeneous environments or across security domains. Examples of security tokens include JSON Web Tokens (JWTs) [JWT] and SAML 2.0 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 sometimes described as Assertions, such as in [RFC7521].

A Security Token Service (STS) is a service capable of validating and issuing security tokens, which enables clients to obtain appropriate access credentials for resources in heterogeneous environments or across security domains. Web Service clients have used WS-Trust [WS-Trust] as the protocol to interact with an STS for token exchange. While WS-Trust uses XML and SOAP, the trend in modern Web development has been towards RESTful patterns and JSON. The OAuth 2.0 Authorization Framework [RFC6749] and OAuth 2.0 Bearer Tokens [RFC6750] have emerged as popular standards for authorizing third-party applications access to HTTP and RESTful resources. The conventional OAuth 2.0 interaction involves the exchange of some representation of resource owner authorization for an access token, which has proven to be an extremely useful pattern in practice, however, its input and output are somewhat too constrained as is to fully accommodate a security token exchange framework.

This specification defines a protocol extending OAuth 2.0 that enables clients to request and obtain security tokens from authorization servers acting in the role of an STS. Similar to OAuth 2.0, this specification focuses on client developer simplicity and requires only an HTTP client and JSON parser, which are nearly universally available in modern development environments. The STS protocol defined in this specification is not itself RESTful (an STS doesn't lend itself particularly well to a REST approach) but does utilize communication patterns and data formats that should be familiar to developers accustomed to working with RESTful systems.

A new grant type for a token exchange request and the associated specific parameters for such a request to the token endpoint are defined by this specification. A token exchange response is a normal OAuth 2.0 response from the token endpoint with a few additional parameters defined herein to provide information to the client.

The entity that makes the request to exchange tokens is considered the client in the context of the token exchange interaction. However, that does not restrict usage of this profile to traditional OAuth clients. An OAuth resource server, for example, might assume the role of the client during token exchange in order to trade an access token, which it received in a protected resource request, for a new token that is appropriate to include in a call to a backend service. The new token might be an access token that is more narrowly scoped for the downstream service or it could be an entirely different kind of token.

The scope of this specification is limited to the definition of a basic request and response protocol for an STS-style token exchange utilizing OAuth 2.0. Although a few new JWT claims are defined that enable delegation semantics to be expressed, the specific syntax, semantics and security characteristics of the tokens themselves (both those presented to the authorization server and those obtained by the client) are explicitly out of scope and no requirements are placed on the trust model in which an implementation might be deployed. Additional profiles may provide more detailed requirements around the specific nature of the parties and trust involved, such as whether signing and/or encryption of tokens is needed or if proof-of-possession style tokens will be required or issued; however, such details will often be policy decisions made with respect to the specific needs of individual deployments and will be configured or implemented accordingly.

The security tokens obtained may be used in a number of contexts, the specifics of which are also beyond the scope of this specification.

1.1. Delegation vs. Impersonation Semantics

When principal A impersonates principal B, A is given all the rights that B has within some defined rights context and is indistinguishable from B in that context. Thus, when principal A impersonates principal B, then in so far as any entity receiving such a token 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 impersonating B, A is B.

Delegation semantics are different than impersonation semantics, though the two are closely related. With delegation semantics, principal A still has its own identity separate from B and it is explicitly understood that while B may have delegated some of its rights to A, any actions taken are being taken by A representing B. In a sense, A is an agent for B.

Delegation and impersonation are not inclusive of all situations. When a principal is acting directly on its own behalf, for example, neither delegation nor impersonation are in play. They are, however, the more common semantics operating for token exchange and, as such, are given more direct treatment in this specification.

Delegation semantics are typically expressed in a token by including information about both the primary subject of the token as well as the actor to whom that subject has delegated some of its rights. Such a token is sometimes referred to as a composite token because it is composed of information about multiple subjects. Typically, in the request, the "subject_token" represents the identity of the party on behalf of whom the token is being requested while the "actor_token" represents the identity of the party to whom the access rights of the issued token are being delegated. A composite token issued by the authorization server will contain information about both parties. When and if a composite token is issued is at the discretion of the authorization server and applicable policy and configuration.

The specifics of representing a composite token and even whether or not such a token will be issued depend on the details of the implementation and the kind of token. The representations of composite tokens that are not JWTs are beyond the scope of this specification. The "actor_token" request parameter, however, does provide a means for providing information about the desired actor and the JWT "act" claim can provide a representation of a chain of delegation.

1.2. 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 BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

1.3. Terminology

This specification uses the terms "access token type", "authorization server", "client", "client identifier", "resource server", "token endpoint", "token request", and "token response" defined by OAuth 2.0 [RFC6749], and the terms "Base64url Encoding", "Claim", and "JWT Claims Set" defined by JSON Web Token (JWT) [JWT].

2. Token Exchange Request and Response

2.1. Request

A client requests a security token by making a token request to the authorization server's token endpoint using the extension grant type mechanism defined in Section 4.5 of OAuth 2.0 [RFC6749].

Client authentication to the authorization server is done using the normal mechanisms provided by OAuth 2.0. Section 2.3.1 of The OAuth 2.0 Authorization Framework [RFC6749] defines password-based authentication of the client, however, client authentication is extensible and other mechanisms are possible. For example, [RFC7523] defines client authentication using JSON Web Tokens (JWTs) [JWT]. The supported methods of client authentication and whether or not to allow unauthenticated or unidentified clients are deployment decisions that are at the discretion of the authorization server.

The client makes a token exchange request to the token endpoint with an extension grant type by including the following parameters using the "application/x-www-form-urlencoded" format with a character encoding of UTF-8 in the HTTP request entity-body:

grant_type
: REQUIRED. The value "urn:ietf:params:oauth:grant-type:token-exchange" indicates that a token exchange is being performed.

resource
: OPTIONAL. Indicates the location of the target service or resource where the client intends to use the requested security token. This enables the authorization server to apply policy as appropriate for the target, such as determining the type and content of the token to be issued or if and how the token is to be encrypted. In many cases, a client will not have knowledge of the logical organization of the systems with which it interacts and will only know the location of the service where it intends to use the token. The "resource" parameter allows the client to indicate to the authorization server where it intends to use the issued token by providing the location, typically as an https URL, in the token exchange request in the same form that will be used to

access that resource. The authorization server will typically have the capability to map from a resource URI value to an appropriate policy. The value of the "resource" parameter MUST be an absolute URI, as specified by Section 4.3 of [RFC3986], which MAY include a query component and MUST NOT include a fragment component. Multiple "resource" parameters may be used to indicate that the issued token is intended to be used at the multiple resources listed.

audience
OPTIONAL. The logical name of the target service where the client intends to use the requested security token. This serves a purpose similar to the "resource" parameter, but with the client providing a logical name rather than a location. Interpretation of the name requires that the value be something that both the client and the authorization server understand. An OAuth client identifier, a SAML entity identifier [OASIS.saml-core-2.0-os], an OpenID Connect Issuer Identifier [OpenID.Core], or a URI are examples of things that might be used as "audience" parameter values. Multiple "audience" parameters may be used to indicate that the issued token is intended to be used at the multiple audiences listed. The "audience" and "resource" parameters may be used together to indicate multiple target services with a mix of logical names and locations.

scope
OPTIONAL. A list of space-delimited, case-sensitive strings, as defined in Section 3.3 of [RFC6749], that allow the client to specify the desired scope of the requested security token in the context of the service or resource where the token will be used. The values and associated semantics of scope are service specific and expected to be described in the relevant service documentation.

requested_token_type
OPTIONAL. An identifier, as described in Section 3, for the type of the requested security token. If the requested type is unspecified, the issued token type is at the discretion of the authorization server and may be dictated by knowledge of the requirements of the service or resource indicated by the "resource" or "audience" parameter.

subject_token
REQUIRED. A security token that represents the identity of the party on behalf of whom the request is being made. Typically, the subject of this token will be the subject of the security token issued in response to this request.

subject_token_type
: REQUIRED. An identifier, as described in Section 3, that indicates the type of the security token in the "subject_token" parameter.

actor_token
: OPTIONAL. A security token that represents the identity of the acting party. Typically, this will be the party that is authorized to use the requested security token and act on behalf of the subject.

actor_token_type
: An identifier, as described in Section 3, that indicates the type of the security token in the "actor_token" parameter. This is REQUIRED when the "actor_token" parameter is present in the request but MUST NOT be included otherwise.

In processing the request, the authorization sever MUST validate the subject token as appropriate for the indicated token type and, if the actor token is present, also validate it according to its token type. The validity criteria and details of any particular token are beyond the scope of this document and are specific to the respective type of token and its content.

In the absence of one-time-use or other semantics specific to the token type, the act of performing a token exchange has no impact on the validity of the subject token or actor token. Furthermore, the validity of the subject token or actor token have no impact on the validity of the issued token after the exchange has occurred.

2.1.1. Relationship Between Resource, Audience and Scope

When requesting a token, the client can indicate the desired target service(s) where it intends to use that token by way of the "audience" and "resource" parameters, as well as indicating the desired scope of the requested token using the "scope" parameter. The semantics of such a request are that the client is asking for a token with the requested scope that is usable at all the requested target services. Effectively, the requested access rights of the token are the cartesian product of all the scopes at all the target services.

An authorization server may be unwilling or unable to fulfill any token request but the likelihood of an unfulfillable request is significantly higher when very broad access rights are being solicited. As such, in the absence of specific knowledge about the relationship of systems in a deployment, clients should exercise discretion in the breadth of the access requested, particularly the

number of target services. An authorization server can use the "invalid_target" error code, defined in Section 2.2.2, to inform a client that it requested access to too many target services simultaneously.

2.2. Response

The authorization server responds to a token exchange request with a normal OAuth 2.0 response from the token endpoint, as specified in Section 5 of [RFC6749]. Additional details and explanation are provided in the following subsections.

2.2.1. Successful Response

If the request is valid and meets all policy and other criteria of the authorization server, a successful token response is constructed by adding the following parameters to the entity-body of the HTTP response using the "application/json" media type, as specified by [RFC7159], and an HTTP 200 status code. The parameters are serialized into a JavaScript Object Notation (JSON) structure by adding each parameter at the top level. Parameter names and string values are included as JSON strings. Numerical values are included as JSON numbers. The order of parameters does not matter and can vary.

access_token
: REQUIRED. The security token issued by the authorization server in response to the token exchange request. The "access_token" parameter from Section 5.1 of [RFC6749] is used here to carry the requested token, which allows this token exchange protocol to use the existing OAuth 2.0 request and response constructs defined for the token endpoint. The identifier "access_token" is used for historical reasons and the issued token need not be an OAuth access token.

issued_token_type
: REQUIRED. An identifier, as described in Section 3, for the representation of the issued security token.

token_type
: REQUIRED. A case-insensitive value specifying the method of using the access token issued, as specified in Section 7.1 of [RFC6749]. It provides the client with information about how to utilize the access token to access protected resources. For example, a value of "Bearer", as specified in [RFC6750], indicates that the security token is a bearer token and the client can simply present it as is without any additional proof of eligibility beyond the contents of the token itself. Note that the meaning of this

parameter is different from the meaning of the "issued_token_type" parameter, which declares the representation of the issued security token; the term "token type" is typically used with this meaning, as it is in all "*_token_type" parameters in this specification. If the issued token is not an access token or usable as an access token, then the "token_type" value "N_A" is used to indicate that an OAuth 2.0 "token_type" identifier is not applicable in that context.

expires_in
RECOMMENDED. The validity lifetime, in seconds, of the token issued by the authorization server. Oftentimes the client will not have the inclination or capability to inspect the content of the token and this parameter provides a consistent and token type agnostic indication of how long the token can be expected to be valid. For example, the value 1800 denotes that the token will expire in thirty minutes from the time the response was generated.

scope
OPTIONAL, if the scope of the issued security token is identical to the scope requested by the client; otherwise, REQUIRED.

refresh_token
OPTIONAL. A refresh token will typically not be issued when the exchange is of one temporary credential (the subject_token) for a different temporary credential (the issued token) for use in some other context. A refresh token can be issued in cases where the client of the token exchange needs the ability to access a resource even when the original credential is no longer valid (e.g., user-not-present or offline scenarios where there is no longer any user entertaining an active session with the client). Profiles or deployments of this specification should clearly document the conditions under which a client should expect a refresh token in response to "urn:ietf:params:oauth:grant-type:token-exchange" grant type requests.

2.2.2. Error Response

If the request itself is not valid or if either the "subject_token" or "actor_token" are invalid for any reason, or are unacceptable based on policy, the authorization server MUST construct an error response, as specified in Section 5.2 of [RFC6749]. The value of the "error" parameter MUST be the "invalid_request" error code.

If the authorization server is unwilling or unable to issue a token for all the target services indicated by the "resource" or "audience" parameters, the "invalid_target" error code SHOULD be used in the error response.

The authorization server MAY include additional information regarding the reasons for the error using the "error_description" and/or "error_uri" parameters.

Other error codes may also be used, as appropriate.

2.3. Example Token Exchange

The following example demonstrates a hypothetical token exchange in which an OAuth resource server assumes the role of the client during token exchange in order to trade an access token that it received in a protected resource request for a token that it will use to call to a backend service (extra line breaks and indentation in the examples are for display purposes only).

The resource server receives the following request containing an OAuth access token in the Authorization request header, as specified in Section 2.1 of [RFC6750].

```
GET /resource HTTP/1.1
Host: frontend.example.com
Authorization: Bearer accVkjcJyb4BWCxGsndESCJQbdFMogUC5PbRDqceLTC
```

Figure 1: Protected Resource Request

The resource server assumes the role of the client for the token exchange and the access token from the request above is sent to the authorization server using a request as specified in Section 2.1. The value of the "subject_token" parameter carries the access token and the value of the "subject_token_type" parameter indicates that it is an OAuth 2.0 access token. The resource server, acting in the role of the client, uses its identifier and secret to authenticate to the authorization server using the HTTP Basic authentication scheme. The "resource" parameter indicates the location of the backend service, https://backend.example.com/api, where the issued token will be used.

```
POST /as/token.oauth2 HTTP/1.1
Host: as.example.com
Authorization: Basic cnMwODpsb25nLXNlY3VyZS1yYW5kb20tc2VjcmV0
Content-Type: application/x-www-form-urlencoded

grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Atoken-exchange
&resource=https%3A%2F%2Fbackend.example.com%2Fapi%20
&subject_token=accVkjcJyb4BWCxGsndESCJQbdFMogUC5PbRDqceLTC
&subject_token_type=
 urn%3Aietf%3Aparams%3Aoauth%3Atoken-type%3Aaccess_token
```

Figure 2: Token Exchange Request

The authorization server validates the client credentials and the "subject_token" presented in the token exchange request. From the "resource" parameter, the authorization server is able to determine the appropriate policy to apply to the request and issues a token suitable for use at https://backend.example.com. The "access_token" parameter of the response contains the new token, which is itself a bearer OAuth access token that is valid for one minute. The token happens to be a JWT; however, its structure and format are opaque to the client so the "issued_token_type" indicates only that it is an access token.

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

{
 "access_token":"eyJhbGciOiJFUzI1NiIsImtpZCI6IjllciJ9.eyJhdWQiOiJo
   dHRwczovL2JhY2tlbmQuZXhhbXBsZS5jb20iLCJpc3MiOiJodHRwczovL2FzLmV
   4YW1wbGUuY29tIiwiZXhwIjoxNDQxOTE3NTkzLCJpYXQiOjE0NDE5MTc1MzMsIn
   N1YiI6ImJjQGV4YW1wbGUuY29tIiwic2NvcGUiOiJhcGkifQ.K4Ik-igqOKi_4C
   nBu4dG3-gGUObfgv-rJhgXVDNOWW_MHgVwddhgVLLQf_bm3xlpQM6wHrLbMaZC4
   LicsQC23g",
 "issued_token_type":
     "urn:ietf:params:oauth:token-type:access_token",
 "token_type":"Bearer",
 "expires_in":60
}
```

Figure 3: Token Exchange Response

The resource server can then use the newly acquired access token in making a request to the backend server.

```
GET /api HTTP/1.1
Host: backend.example.com
Authorization: Bearer eyJhbGciOiJFUzI1NiIsImtpZCI6IjllciJ9.eyJhdWQ
   iOiJodHRwczovL2JhY2tlbmQuZXhhbXBsZS5jb20iLCJpc3MiOiJodHRwczovL2
   FzLmV4YW1wbGUuY29tIiwiZXhwIjoxNDQxOTE3NTkzLCJpYXQiOjE0NDE5MTc1M
   zMsInN1YiI6ImJjQGV4YW1wbGUuY29tIiwic2NwIjpbImFwaSJdfQ.MXgnpvPMo
   0nhcePwnQbunD2gw_pDyCFA-Saobl6gyLAdyPbaALFuAOyFc4XTWaPEnHV_LGmX
   klSTpz0yC7hlSQ
```

Figure 4: Backend Protected Resource Request

Additional examples can be found in Appendix A.

3. Token Type Identifiers

Several parameters in this specification utilize an identifier as the value to describe the token in question. Specifically, they are the "requested_token_type", "subject_token_type", "actor_token_type" parameters of the request and the "issued_token_type" member of the response. Token type identifiers are URIs. Token Exchange can work with both tokens issued by other parties and tokens from the given authorization server. For the former the token type identifier indicates the syntax (e.g., JWT or SAML 2.0) so the authorization server can parse it; for the latter it indicates what the given authorization server issued it for (e.g., access_token or refresh_token).

The following token type identifiers are defined by this specification. Other URIs MAY be used to indicate other token types.

urn:ietf:params:oauth:token-type:access_token
: Indicates that the token is an OAuth 2.0 access token issued by the given authorization server.

urn:ietf:params:oauth:token-type:refresh_token
: Indicates that the token is an OAuth 2.0 refreshe token issued by the given authorization server.

urn:ietf:params:oauth:token-type:id_token
: Indicates that the token is an ID Token, as defined in Section 2 of [OpenID.Core].

urn:ietf:params:oauth:token-type:saml1
: Indicates that the token is a base64url-encoded SAML 1.1 [OASIS.saml-core-1.1] assertion.

urn:ietf:params:oauth:token-type:saml2

Indicates that the token is a base64url-encoded SAML 2.0 [OASIS.saml-core-2.0-os] assertion.

The value "urn:ietf:params:oauth:token-type:jwt", which is defined in Section 9 of [JWT], indicates that the token is a JWT.

The distinction between an access token and a JWT is subtle. An access token represents a delegated authorization decision, whereas JWT is a token format. An access token can be formatted as a JWT but doesn't necessarily have to be. And a JWT might well be an access token but not all JWTs are access tokens. The intent of this specification is that "urn:ietf:params:oauth:token-type:access_token" be an indicator that the token is a typical OAuth access token issued by the authorization server in question, opaque to the client, and usable the same manner as any other access token obtained from that authorization server. (It could well be a JWT, but the client isn't and needn't be aware of that fact.) Whereas, "urn:ietf:params:oauth:token-type:jwt" is to indicate specifically that a JWT is being requested or sent (perhaps in a cross-domain use-case where the JWT is used as an authorization grant to obtain an access token from a different authorization server as is facilitated by [RFC7523]).

4. JSON Web Token Claims and Introspection Response Parameters

It is useful to have defined mechanisms to express delegation within a token as well as to express authorization to delegate or impersonate. Although the token exchange protocol described herein can be used with any type of token, this section defines claims to express such semantics specifically for JWTs and in an OAuth 2.0 Token Introspection [RFC7662] response. Similar definitions for other types of tokens are possible but beyond the scope of this specification.

Note that the claims not established herein but used in examples and descriptions, such as "iss", "sub", "exp", etc., are defined by [JWT].

4.1. "act" (Actor) Claim

The "act" (actor) claim provides a means within a JWT to express that delegation has occurred and identify the acting party to whom authority has been delegated. The "act" claim value is a JSON object and members in the JSON object are claims that identify the actor. The claims that make up the "act" claim identify and possibly provide additional information about the actor. For example, the combination of the two claims "iss" and "sub" might be necessary to uniquely identify an actor.

However, claims within the "act" claim pertain only to the identity of the actor and are not relevant to the validity of the containing JWT in the same manner as the top-level claims. Consequently, non-identity claims (e.g., "exp", "nbf", and "aud") are not meaningful when used within an "act" claim, and therefore must not be used.

The following example illustrates the "act" (actor) claim within a JWT Claims Set. The claims of the token itself are about user@example.com while the "act" claim indicates that admin@example.com is the current actor.

```
 {
   "aud":"https://consumer.example.com",
   "iss":"https://issuer.example.com",
   "exp":1443904177,
   "nbf":1443904077,
   "sub":"user@example.com",
   "act":
   {
     "sub":"admin@example.com"
   }
 }
```

Figure 5: Actor Claim

A chain of delegation can be expressed by nesting one "act" claim within another. The outermost "act" claim represents the current actor while nested "act" claims represent prior actors. The least recent actor is the most deeply nested.

For the purpose of applying access control policy, the consumer of a token MUST only consider the token’s top-level claims and the party identified as the current actor by the "act" claim. Prior actors identified by any nested "act" claims are informational only and are not to be considered in access control decisions.

The following example illustrates nested "act" (actor) claims within a JWT Claims Set. The claims of the token itself are about user@example.com while the "act" claim indicates that the system https://service16.example.com is the current actor and https://service77.example.com was a prior actor. Such a token might come about as the result of service16 receiving a token in a call from service77 and exchanging it for a token suitable to call service26 while the authorization server notes the situation in the newly issued token.

```
 {
   "aud":"https://service26.example.com",
   "iss":"https://issuer.example.com",
   "exp":1443904100,
   "nbf":1443904000,
   "sub":"user@example.com",
   "act":
   {
     "sub":"https://service16.example.com",
     "act":
     {
       "sub":"https://service77.example.com",
     }
   }
 }
```

Figure 6: Nested Actor Claim

When included as a top-level member of an OAuth token introspection response, "act" has the same semantics and format as the claim of the same name.

4.2. "scope" (Scopes) Claim

The value of the "scope" claim is a JSON string containing a space-separated list of scopes associated with the token, in the format described in Section 3.3 of OAuth 2.0 [RFC6749].

The following example illustrates the "scope" claim within a JWT Claims Set.

```
 {
   "aud":"https://consumer.example.com",
   "iss":"https://issuer.example.com",
   "exp":1443904177,
   "nbf":1443904077,
   "sub":"dgaf4mvfs75Fci_FL3heQA",
   "scope":"email profile phone address"
 }
```

Figure 7: Scopes Claim

OAuth 2.0 Token Introspection [RFC7662] already defines the "scope" parameter to convey the scopes associated with the token.

4.3. "client_id" (Client Identifier) Claim

The "client_id" claim carries the client identifier of the OAuth 2.0 [RFC6749] client that requested the token.

The following example illustrates the "client_id" claim within a JWT Claims Set indicating an OAuth 2.0 client with "s6BhdRkqt3" as its identifier.

```
 {
   "aud":"https://consumer.example.com",
   "iss":"https://issuer.example.com",
   "exp":1443904177,
   "sub":"user@example.com",
   "client_id":"s6BhdRkqt3"
 }
```

Figure 8: Client Identifier Claim

OAuth 2.0 Token Introspection [RFC7662] already defines the "client_id" parameter as the client identifier for the OAuth 2.0 client that requested the token.

4.4. "may_act" (May Act For) Claim

The "may_act" claim makes a statement that one party is authorized to become the actor and act on behalf of another party. The claim value is a JSON object and members in the JSON object are claims that identify the party that is asserted as being eligible to act for the party identified by the JWT containing the claim. The claims that make up the "may_act" claim identify and possibly provide additional

information about the authorized actor. For example, the combination of the two claims "iss" and "sub" are sometimes necessary to uniquely identify an authorized actor, while the "email" claim might be used to provide additional useful information about that party.

However, claims within the "may_act" claim pertain only to the identity of that party and are not relevant to the validity of the containing JWT in the same manner as top-level claims. Consequently, claims such as "exp", "nbf", and "aud" are not meaningful when used within a "may_act" claim, and therefore should not be used.

The following example illustrates the "may_act" claim within a JWT Claims Set. The claims of the token itself are about user@example.com while the "may_act" claim indicates that admin@example.com is authorized to act on behalf of user@example.com.

```
 {
   "aud":"https://consumer.example.com",
   "iss":"https://issuer.example.com",
   "exp":1443904177,
   "nbf":1443904077,
   "sub":"user@example.com",
   "may_act":
   {
     "sub":"admin@example.com"
   }
 }
```

Figure 9: May Act For Claim

When included as a top-level member of an OAuth token introspection response, "may_act" has the same semantics and format as the claim of the same name.

5. Security Considerations

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

In addition, both delegation and impersonation introduce unique security issues. Any time one principal is delegated the rights of another principal, the potential for abuse is a concern. The use of the "scope" claim is suggested to mitigate potential for such abuse, as it restricts the contexts in which the delegated rights can be exercised.

6. Privacy Considerations

Tokens employed in the context of the functionality described herein may contain privacy-sensitive information and, to prevent disclosure of such information to unintended parties, should only be transmitted over encrypted channels, such as Transport Layer Security (TLS). In cases where it is desirable to prevent disclosure of certain information to the client, the token should be encrypted to its intended recipient. Deployments should determine the minimally necessary amount of data and only include such information in issued tokens. In some cases, data minimization may include representing only an anonymous or pseudonymous user.

7. IANA Considerations

7.1. OAuth URI Registration

This specification registers the following values in the IANA "OAuth URI" registry [IANA.OAuth.Parameters] established by [RFC6755].

7.1.1. Registry Contents

- URN: urn:ietf:params:oauth:grant-type:token-exchange
- Common Name: Token exchange grant type for OAuth 2.0
- Change controller: IESG
- Specification Document: Section 2.1 of [[this specification]]

- URN: urn:ietf:params:oauth:token-type:access_token
- Common Name: Token type URI for an OAuth 2.0 access token
- Change controller: IESG
- Specification Document: Section 3 of [[this specification]]

- URN: urn:ietf:params:oauth:token-type:refresh_token
- Common Name: Token type URI for an OAuth 2.0 refresh token
- Change controller: IESG
- Specification Document: Section 3 of [[this specification]]

- URN: urn:ietf:params:oauth:token-type:id_token
- Common Name: Token type URI for an ID Token
- Change controller: IESG
- Specification Document: Section 3 of [[this specification]]

- URN: urn:ietf:params:oauth:token-type:saml1
- Common Name: Token type URI for a base64url-encoded SAML 1.1 assertion
- Change Controller: IESG
- Specification Document: Section 3 of [[this specification]]

- URN: urn:ietf:params:oauth:token-type:saml2
- Common Name: Token type URI for a base64url-encoded SAML 2.0 assertion
- Change Controller: IESG
- Specification Document: Section 3 of [[this specification]]

7.2. OAuth Parameters Registration

This specification registers the following values in the IANA "OAuth Parameters" registry [IANA.OAuth.Parameters] established by [RFC6749].

7.2.1. Registry Contents

- Parameter name: resource
- Parameter usage location: token request
- Change controller: IESG
- Specification document(s): Section 2.1 of [[this specification]]

- Parameter name: audience
- Parameter usage location: token request
- Change controller: IESG
- Specification document(s): Section 2.1 of [[this specification]]

- Parameter name: requested_token_type
- Parameter usage location: token request
- Change controller: IESG
- Specification document(s): Section 2.1 of [[this specification]]

- Parameter name: subject_token
- Parameter usage location: token request
- Change controller: IESG
- Specification document(s): Section 2.1 of [[this specification]]

- Parameter name: subject_token_type
- Parameter usage location: token request
- Change controller: IESG
- Specification document(s): Section 2.1 of [[this specification]]

- Parameter name: actor_token
- Parameter usage location: token request
- Change controller: IESG
- Specification document(s): Section 2.1 of [[this specification]]

- Parameter name: actor_token_type
- Parameter usage location: token request
- Change controller: IESG
- Specification document(s): Section 2.1 of [[this specification]]

- Parameter name: issued_token_type
- Parameter usage location: token response
- Change controller: IESG
- Specification document(s): Section 2.2.1 of [[this specification]]

7.3. OAuth Access Token Type Registration

This specification registers the following access token type in the IANA "OAuth Access Token Types" registry [IANA.OAuth.Parameters] established by [RFC6749].

7.3.1. Registry Contents

- Type name: N_A
- Additional Token Endpoint Response Parameters: (none)
- HTTP Authentication Scheme(s): (none)
- Change controller: IESG
- Specification document(s): Section 2.2.1 of [[this specification]]

7.4. JSON Web Token Claims Registration

This specification registers the following Claims in the IANA "JSON Web Token Claims" registry [IANA.JWT.Claims] established by [JWT].

7.4.1. Registry Contents

- Claim Name: "act"
- Claim Description: Actor
- Change Controller: IESG
- Specification Document(s): Section 4.1 of [[this specification]]

- Claim Name: "scope"
- Claim Description: Scope Values
- Change Controller: IESG
- Specification Document(s): Section 4.2 of [[this specification]]

- Claim Name: "client_id"
- Claim Description: Client Identifier
- Change Controller: IESG
- Specification Document(s): Section 4.3 of [[this specification]]

- Claim Name: "may_act"
- Claim Description: May Act For
- Change Controller: IESG
- Specification Document(s): Section 4.4 of [[this specification]]

7.5. OAuth Token Introspection Response Registration

This specification registers the following values in the IANA "OAuth Token Introspection Response" registry [IANA.OAuth.Parameters] established by [RFC7662].

7.5.1. Registry Contents

- Claim Name: "act"
- Claim Description: Actor
- Change Controller: IESG
- Specification Document(s): Section 4.1 of [[this specification]]

- Claim Name: "may_act"
- Claim Description: May Act For
- Change Controller: IESG
- Specification Document(s): Section 4.4 of [[this specification]]

7.6. OAuth Extensions Error Registration

This specification registers the following values in the IANA "OAuth Extensions Error" registry [IANA.OAuth.Parameters] established by [RFC6749].

7.6.1. Registry Contents

- Error Name: "invalid_target"
- Error Usage Location: token error response
- Related Protocol Extension: OAuth 2.0 Token Exchange
- Change Controller: IETF
- Specification Document(s): Section 2.2.2 of [[this specification]]

8. References

8.1. Normative References

[IANA.JWT.Claims]
IANA, "JSON Web Token Claims",
<http://www.iana.org/assignments/jwt>.

[IANA.OAuth.Parameters]
IANA, "OAuth Parameters",
<http://www.iana.org/assignments/oauth-parameters>.

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

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

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

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

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

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

8.2. Informative References

[OASIS.saml-core-1.1]
Maler, E., Mishra, P., and R. Philpott, "Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V1.1", OASIS Standard oasis-sstc-saml-core-1.1, September 2003.

[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-2.0-os, March 2005.

[OpenID.Core]
Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0", August 2015, <http://openid.net/specs/openid-connect-core-1_0.html>.

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

[RFC6755] Campbell, B. and H. Tschofenig, "An IETF URN Sub-Namespace for OAuth", RFC 6755, DOI 10.17487/RFC6755, October 2012, <https://www.rfc-editor.org/info/rfc6755>.

[RFC7521] Campbell, B., Mortimore, C., Jones, M., and Y. Goland, "Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants", RFC 7521, DOI 10.17487/RFC7521, May 2015, <https://www.rfc-editor.org/info/rfc7521>.

[RFC7523] Jones, M., Campbell, B., and C. Mortimore, "JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants", RFC 7523, DOI 10.17487/RFC7523, May 2015, <https://www.rfc-editor.org/info/rfc7523>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[WS-Trust] Nadalin, A., Goodner, M., Gudgin, M., Barbir, A., and H. Granqvist, "WS-Trust 1.4", February 2012, <http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/ws-trust.html>.

Appendix A. Additional Token Exchange Examples

Two example token exchanges are provided in the following sections illustrating impersonation and delegation, respectively (with extra line breaks and indentation for display purposes only).

A.1. Impersonation Token Exchange Example

A.1.1. Token Exchange Request

In the following token exchange request, a client is requesting a token with impersonation semantics. The client tells the authorization server that it needs a token for use at the target service with the logical name "urn:example:cooperation-context".

```
POST /as/token.oauth2 HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded

grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Atoken-exchange
&audience=urn%3Aexample%3Acooperation-context
&subject_token=eyJhbGciOiJFUzI1NiIsImtpZCI6IjE2In0.eyJhdWQiOiJodHRwc
  zovL2FzLmV4YW1wbGUuY29tIiwiaXNzIjoiaHR0cHM6Ly9vcmlnaW5hbC1pc3N1ZXI
  uZXhhbXBsZS5uZXQiLCJleHAiOjE0NDE5MTA2MDAsIm5iZiI6MTQ0MTkwOTAwMCwic
  3ViIjoiYmNAZXhhbXBsZS5uZXQiLCJzY29wZSI6Im9yZGVycyBwcm9maWxlIGhpc3R
  vcnkifQ.u0slqvbnqU43EvI_itGdFJ11StrAwXlxczYfMYsaR5p4J_gBp019mxljSx
  xmD3FfbrjTGyZ4eDh1JKJVpsnnPg
&subject_token_type=urn%3Aietf%3Aparams%3Aoauth%3Atoken-type%3Ajwt
```

Figure 10: Token Exchange Request

A.1.2. Subject Token Claims

The "subject_token" in the prior request is a JWT and the decoded JWT Claims Set is shown here. The JWT is intended for consumption by the authorization server within a specific time window. The subject of the JWT ("bc@example.net") is the party on behalf of whom the new token is being requested.

```
{
  "aud":"https://as.example.com",
  "iss":"https://original-issuer.example.net",
  "exp":1441910600,
  "nbf":1441909000,
  "sub":"bc@example.net",
  "scope":"orders profile history"
}
```

Figure 11: Subject Token Claims

A.1.3. Token Exchange Response

The "access_token" parameter of the token exchange response shown below contains the new token that the client requested. The other parameters of the response indicate that the token is a bearer access token that expires in an hour.

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

{
 "access_token":"eyJhbGciOiJFUzI1NiIsImtpZCI6IjcyIn0.eyJhdWQiOiJ1cm4
   6ZXhhbXBsZTpjb29wZXJhdGlvbi1jb250ZXh0IiwiaXNzIjoiaHR0cHM6Ly9hcy5l
   eGFtcGxlLmNvbSIsImV4cCI6MTQ0MTkxMzYxMCwic3ViIjoiYmNAZXhhbXBsZS5uZ
   XQiLCJzY29wZSI6Im9yZGVycyBwcm9maWxlIGhpc3RvcnkifQ._OnH9oHT2cd0-Sz
   OfBrNkVYlRdn48X8kI4_Is3LHeQmtkd-nDdR63IuuQ_GeZd7UafMV3bk8jqUDgi-l
   rTfSwA",
 "issued_token_type":
   "urn:ietf:params:oauth:token-type:access_token",
 "token_type":"Bearer",
 "expires_in":3600
}
```

Figure 12: Token Exchange Response

A.1.4. Issued Token Claims

The decoded JWT Claims Set of the issued token is shown below. The new JWT is issued by the authorization server and intended for consumption by a system entity known by the logical name "urn:example:cooperation-context" any time before its expiration. The subject ("sub") of the JWT is the same as the subject the token used to make the request, which effectively enables the client to impersonate that subject at the system entity known by the logical name of "urn:example:cooperation-context" by using the token.

```
 {
   "aud":"urn:example:cooperation-context",
   "iss":"https://as.example.com",
   "exp":1441913610,
   "sub":"bc@example.net",
   "scope":"orders profile history"
 }
```

Figure 13: Issued Token Claims

A.2. Delegation Token Exchange Example

A.2.1. Token Exchange Request

In the following token exchange request, a client is requesting a token and providing both a "subject_token" and an "actor_token". The client tells the authorization server that it needs a token for use at the target service with the logical name "urn:example:cooperation-

context". Policy at the authorization server dictates that the issued token be a composite.

```
POST /as/token.oauth2 HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded

grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Atoken-exchange
&audience=urn%3Aexample%3Acooperation-context
&subject_token=eyJhbGciOiJFUzI1NiIsImtpZCI6IjE2In0.eyJhdWQiOiJodHRwc
  zovL2FzLmV4YW1wbGUuY29tIiwiaXNzIjoiaHR0cHM6Ly9vcmlnaW5hbC1pc3N1ZXI
  uZXhhbXBsZS5uZXQiLCJleHAiOjE0NDE5MTAwNjAsInNjb3BlIjoic3RhdHVzIGZlZ
  WQiLCJzdWIiOiJ1c2VyQGV4YW1wbGUubmV0IiwibWF5X2FjdCI6eyJzdWIiOiJhZG1
  pbkBleGFtcGxlLm5ldCJ9fQ.4rPRSWihQbpMIgAmAoqaJojAxj-p2X8_fAtAGTXrvM
  xU-eEZHnXqY0_AOZgLdxw5DyLzua8H_I10MCcckF-Q_g
&subject_token_type=urn%3Aietf%3Aparams%3Aoauth%3Atoken-type%3Ajwt
&actor_token=eyJhbGciOiJFUzI1NiIsImtpZCI6IjE2In0.eyJhdWQiOiJodHRwczo
  vL2FzLmV4YW1wbGUuY29tIiwiaXNzIjoiaHR0cHM6Ly9vcmlnaW5hbC1pc3N1ZXIuZ
  XhhbXBsZS5uZXQiLCJleHAiOjE0NDE5MTAwNjAsInN1YiI6ImFkbWluQGV4YW1wbGU
  ubmV0In0.7YQ-3zPfhUvzje5oqw8COCvN5uP6NsKik9CVV6cAOf4QKgM-tKfiOwcgZ
  oUuDL2tEs6tqPlcBlMjiSzEjm3yBg
&actor_token_type=urn%3Aietf%3Aparams%3Aoauth%3Atoken-type%3Ajwt
```

Figure 14: Token Exchange Request

A.2.2. Subject Token Claims

The "subject_token" in the prior request is a JWT and the decoded JWT Claims Set is shown here. The JWT is intended for consumption by the authorization server before a specific expiration time. The subject of the JWT ("user@example.net") is the party on behalf of whom the new token is being requested.

```
{
  "aud":"https://as.example.com",
  "iss":"https://original-issuer.example.net",
  "exp":1441910060,
  "scope":"status feed",
  "sub":"user@example.net",
  "may_act":
  {
    "sub":"admin@example.net"
  }
}
```

Figure 15: Subject Token Claims

A.2.3. Actor Token Claims

The "actor_token" in the prior request is a JWT and the decoded JWT Claims Set is shown here. This JWT is also intended for consumption by the authorization server before a specific expiration time. The subject of the JWT ("admin@example.net") is the actor that will wield the security token being requested.

```
{
  "aud":"https://as.example.com",
  "iss":"https://original-issuer.example.net",
  "exp":1441910060,
  "sub":"admin@example.net"
}
```

Figure 16: Actor Token Claims

A.2.4. Token Exchange Response

The "access_token" parameter of the token exchange response shown below contains the new token that the client requested. The other parameters of the response indicate that the token is a JWT that expires in an hour and that the access token type is not applicable since the issued token is not an access token.

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

{
 "access_token":"eyJhbGciOiJFUzI1NiIsImtpZCI6IjcyIn0.eyJhdWQiOiJ1cm4
   6ZXhhbXBsZTpjb29wZXJhdGlvbi1jb250ZXh0IiwiaXNzIjoiaHR0cHM6Ly9hcy5l
   eGFtcGxlLmNvbSIsImV4cCI6MTQ0MTkxMzYxMCwic2NvcGUiOiJzdGF0dXMgZmVlZ
   CIsInN1YiI6InVzZXJAZXhhbXBsZS5uZXQiLCJhY3QiOnsic3ViIjoiYWRtaW5AZX
   hhbXBsZS5uZXQifX0.3paKl9UySKYB5ng6_cUtQ2qlO8Rc_y7Mea7IwEXTcYbNdwG
   9-G1EKCFe5fW3H0hwX-MSZ49Wpcb1SiAZaOQBtw",
 "issued_token_type":"urn:ietf:params:oauth:token-type:jwt",
 "token_type":"N_A",
 "expires_in":3600
}
```

Figure 17: Token Exchange Response

A.2.5. Issued Token Claims

The decoded JWT Claims Set of the issued token is shown below. The new JWT is issued by the authorization server and intended for consumption by a system entity known by the logical name

"urn:example:cooperation-context" any time before its expiration. The subject ("sub") of the JWT is the same as the subject of the "subject_token" used to make the request. The actor ("act") of the JWT is the same as the subject of the "actor_token" used to make the request. This indicates delegation and identifies "admin@example.net" as the current actor to whom authority has been delegated to act on behalf of "user@example.net".

```
{
  "aud":"urn:example:cooperation-context",
  "iss":"https://as.example.com",
  "exp":1441913610,
  "scope":"status feed",
  "sub":"user@example.net",
  "act":
  {
    "sub":"admin@example.net"
  }
}
```

Figure 18: Issued Token Claims

Appendix B. Acknowledgements

This specification was developed within the OAuth Working Group, which includes dozens of active and dedicated participants. It was produced under the chairmanship of Hannes Tschofenig, Derek Atkins, and Rifaat Shekh-Yusef with Kathleen Moriarty, Stephen Farrell, Eric Rescorla, and Benjamin Kaduk serving as Security Area Directors. The following individuals contributed ideas, feedback, and wording to this specification:

Caleb Baker, Vittorio Bertocci, Thomas Broyer, William Denniss, Vladimir Dzhuvinov, Phil Hunt, Benjamin Kaduk, Jason Keglovitz, Torsten Lodderstedt, Adam Lewis, James Manger, Nov Matake, Matt Miller, Hilarie Orman, Matthew Perry, Eric Rescorla, Justin Richer, Rifaat Shekh-Yusef, Scott Tomilson, and Hannes Tschofenig.

Appendix C. Document History

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

-16

o Fixed typo and added an AD to Acknowledgements.

-15

- Updated the nested actor claim example to (hopefully) be more straightforward.
- Reworked Privacy Considerations to say to use TLS in transit, minimize the amount of information in the token, and encrypt the token if disclosure of its information to the client is a concern per https://mailarchive.ietf.org/arch/msg/secdir/KJhx4aq_U5uk3k6zpYP-CEHbpVM
- Moved the Security and Privacy Considerations sections to before the IANA Considerations.

-14

- Added text in Section 4.1 about the "act" claim stating that only the top-level claims and the current actor are to be considered in applying access control decisions.

-13

- Updated the claim name and value syntax for scope to be consistent with the treatment of scope in RFC 7662 OAuth 2.0 Token Introspection.
- Updated the client identifier claim name to be consistent with the treatment of client id in RFC 7662 OAuth 2.0 Token Introspection.

-12

- Updated to use the boilerplate from RFC 8174.

-11

- Added new WG chair and AD to the Acknowledgements.
- Applied clarifications suggested during AD review by EKR.

-10

- Defined token type URIs for base64url-encoded SAML 1.1 and SAML 2.0 assertions.
- Applied editorial fixes.

-09

- Changed "security tokens obtained could be used in a number of contexts" to "security tokens obtained may be used in a number of contexts" per a WGLC suggestion.
- Clarified that the validity of the subject or actor token have no impact on the validity of the issued token after the exchange has occurred per a WGLC comment.

- Changed use of invalid_target error code to a SHOULD per a WGLC comment.
- Clarified text about non-identity claims within the "act" claim being meaningless per a WGLC comment.
- Added brief Privacy Considerations section per WGLC comments.

-08

- Use the bibxml reference for OpenID.Core rather than defining it inline.
- Added editor role for Campbell.
- Minor clarification of the text for actor_token.

-07

- Fixed typo (desecration -> discretion).
- Added an explanation of the relationship between scope, audience and resource in the request and added an "invalid_target" error code enabling the AS to tell the client that the requested audiences/resources were too broad.

-06

- Drop "An STS for the REST of Us" from the title.
- Drop "heavyweight" and "lightweight" from the abstract and introduction.
- Clarifications on the language around xxxxxx_token_type.
- Remove the want_composite parameter.
- Add a short mention of proof-of-possession style tokens to the introduction and remove the respective open issue.

-05

- Defined the JWT claim "cid" to express the OAuth 2.0 client identifier of the client that requested the token.
- Defined and requested registration for "act" and "may_act" as Token introspection response parameters (in addition to being JWT claims).
- Loosen up the language about refresh_token in the response to OPTIONAL from NOT RECOMMENDED based on feedback form real world deployment experience.
- Add clarifying text about the distinction between JWT and access token URIs.
- Close out (remove) some of the Open Issues bullets that have been resolved.

-04

- Clarified that the "resource" and "audience" request parameters can be used at the same time (via http://www.ietf.org/mail-archive/web/oauth/current/msg15335.html).
- Clarified subject/actor token validity after token exchange and explained a bit more about the recommendation to not issue refresh tokens (via http://www.ietf.org/mail-archive/web/oauth/current/msg15318.html).
- Updated the examples appendix to use an issuer value that doesn't imply that the client issued and signed the tokens and used "Bearer" and "urn:ietf:params:oauth:token-type:access_token" in one of the responses (via http://www.ietf.org/mail-archive/web/oauth/current/msg15335.html).
- Defined and registered urn:ietf:params:oauth:token-type:id_token, since some use cases perform token exchanges for ID Tokens and no URI to indicate that a token is an ID Token had previously been defined.

-03

- Updated the document editors (adding Campbell, Bradley, and Mortimore).
- Added to the title.
- Added to the abstract and introduction.
- Updated the format of the request to use application/x-www-form-urlencoded request parameters and the response to use the existing token endpoint JSON parameters defined in OAuth 2.0.
- Changed the grant type identifier to urn:ietf:params:oauth:grant-type:token-exchange.
- Added RFC 6755 registration requests for urn:ietf:params:oauth:token-type:refresh_token, urn:ietf:params:oauth:token-type:access_token, and urn:ietf:params:oauth:grant-type:token-exchange.
- Added RFC 6749 registration requests for request/response parameters.
- Removed the Implementation Considerations and the requirement to support JWTs.
- Clarified many aspects of the text.
- Changed "on_behalf_of" to "subject_token", "on_behalf_of_token_type" to "subject_token_type", "act_as" to "actor_token", and "act_as_token_type" to "actor_token_type".
- Added an "audience" request parameter used to indicate the logical names of the target services at which the client intends to use the requested security token.
- Added a "want_composite" request parameter used to indicate the desire for a composite token rather than trying to infer it from the presence/absence of token(s) in the request.

- Added a "resource" request parameter used to indicate the URLs of resources at which the client intends to use the requested security token.
- Specified that multiple "audience" and "resource" request parameter values may be used.
- Defined the JWT claim "act" (actor) to express the current actor or delegation principal.
- Defined the JWT claim "may_act" to express that one party is authorized to act on behalf of another party.
- Defined the JWT claim "scp" (scopes) to express OAuth 2.0 scope-token values.
- Added the "N_A" (not applicable) OAuth Access Token Type definition for use in contexts in which the token exchange syntax requires a "token_type" value, but in which the token being issued is not an access token.
- Added examples.

-02

- Enabled use of Security Token types other than JWTs for "act_as" and "on_behalf_of" request values.
- Referenced the JWT and OAuth Assertions RFCs.

-01

- Updated references.

-00

- 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

Brian Campbell (editor)
Ping Identity

Email: brian.d.campbell@gmail.com

John Bradley
Ping Identity

Email: ve7jtb@ve7jtb.com

Chuck Mortimore
Salesforce

Email: cmortimore@salesforce.com

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 20, 2016

N. Sakimura
Nomura Research Institute
K. Li
Alibaba Group

October 18, 2015

Sender Constrained JWT for OAuth 2.0
draft-sakimura-oauth-rjwtprof-06

Abstract

This discussion document describes a method to indicate a sender constraint within JWT. It could potentially be incorporated into Proof-Of-Possession Semantics for JSON Web Tokens(JWTs) [POPS]. This document was created in response to the WGLC of it.

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 April 20, 2016.

Copyright Notice

Table of Contents

1. Introduction

OAuth 2.0 Proof-of-Possession (PoP) Security Architecture [POPA] identifies Sender Constraint and Key Confirmation as possible threat mitigation methods against the use of token by an unauthorized presenter. While Proof-Of-Possession Semantics for JSON Web Tokens (JWTs) [POPS] touches briefly on the Sender Constraint, it is only one paragraph within a introductory text and does not discuss it in detail. Instead, it devotes much of the discussion to the Key Confirmation method. It also is making the usage of such token against the resource server out of scope.

This discussion draft describes a way to express the Sender Constraint in the JWT, as well as one possible way of using it to access a protected resource.

The initial draft of this document was created in response to the WGLC of the Proof-Of-Possession Semantics for JSON Web Tokens(JWTs) [POPS].

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. Terms and definitions

For the purpose of this document, the terms defined in RFC6749 [RFC6749] is used. In addition, following term is defined.

Authorized Presenter Party that the token is intended to be used by.

3. Justification

There are scenarios that the bearer token may be stolen, modified, reused or replayed. To prevent these threats, resource servers need to obtain additional assurance that the client is indeed authorized to present an access token. The detailed use cases can be found in OAuth 2.0 Proof-of-Possession (PoP) Security Architecture [POPA] specification that sites token reuse by the resource server and eavesdropping of the resource request among others. Some additional use-cases such as token leaking from the client’s database or authorization server’s database is also conceivable.

As described in OAuth 2.0 Proof-of-Possession (PoP) Security Architecture [POPA] specification, there are several ways to prevent these bearer token threats: Confidentiality Protection, Sender Constraint and Key Confirmation. Key Confirmation mechanism is described in OAuth 2.0 Proof-Of-Possession Semantics for JSON Web Tokens (JWTs) [POPS] specification in detail, but Sender Constraint mechanism is not explained in detail.

Sender confirmation mechanism has some advantage in some cases over the general key confirmation mechanism explained in [POPS] in cases such as:

(1) The client’s public key is published in a known way in the ecosystem, e.g., in .well-known/jwks and the private key is stored in a HSM.

(2) The resource server wishes to have some non-repudiation of the client.

These can be achieved with relative ease with sender confirmation.

Key Confirmation mechanism is more general in nature. It is applicable even in the case where client's privacy is sought or the client is a public client using OAuth PKCE [PKCE]. As the downside of it, it requires a complete key distribution protocol and can become more complicated. Sender Confirmation mechanism should also be specified, and it can work as an alternative mechanism to mitigate the bearer token threats.

4. Sender Constraint Representation

Sender Constraint is expressed by including the following member at the top level of JWT payload.

azp The Client ID of the Authorized Presenter.

Following is an example of such JWT payload.

```
{
 "iss": "https://server.example.com",
 "sub": "joe@example.com",
 "azp": "https://client.example.org",
 "aud": "https://resource.example.org",
 "exp": "1361398824",
 "nbf": "1360189224",
}
```

Figure 1 Example of Sender Constrained JWT.

5. Client Authentication

The resource server that supports this specification MUST authenticate the Client. In this document a possible method is proposed as follows:

(1). The authorized presenter issues a HEAD or GET request to the resource server.

```
GET /resource/1234 HTTP/1.0
Host: server.example.com
```

(2). The resource server returns a HTTP 401 response with "WWW-Authenticate" header with "Named" scheme, which includes nonce.

```
HTTP/1.0 401 Unauthorized
Server: HTTPd/0.9
Date: Wed, 14 March 2015 09:26:53 GMT
WWW-Authenticate: Named nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093"
```

(3). The client creates JWS compact serialization over the nonce.

(4). The client sends the request to the resource server, this time with Authorization: header with Named scheme and access token and the JWS.

```
GET /resource/1234 HTTP/1.0
Host: server.example.com
Authorization: Named at="access.token.jwt", s="jws.of.nonce"
```

(5). The resource server finds the client key corresponding to the value of "azp" in the access token. It may have been obtained through client registration at the Issuer or through .well-known/jwk etc.

(6). The resource server creates the JWS of the nonce and compares it with the value of "s" of the Authorization header. If it fails, the process stops here and the resource access MUST be denied.

(7). The resource server MUST verify the access token. If it is valid, the resource SHOULD be returned as HTTP response.

6. Finding the client key

When the resource server authenticates the client, it has to find out the keys that corresponds to the signing key of the client. There are several possible ways to do this.

6.1. URI client ID

When the Client ID is a URI, then the key can be found from the .well-known/jwk URI.

6.2. pre-shared key tables

Alternatively, the collection of the keys can be pre-shared among the participants in advance as a key table that lists the client ID - public key pair.

6.3. Via client metadata API of the authorization server

Client Metadata can be exposed through a client metadata API at the Authorization Server, which can be defined by the authorization server in a way similar to OAuth 2.0 Token Introspection [TINTRO].

7. IANA Considerations

7.1. Named Authentication Scheme

A new scheme has been registered in the HTTP Authentication Scheme Registry as follows:

Authentication Scheme Name: Named

Reference: Section 5 of this specification

Notes (optional): The Named Authentication scheme is intended to be used only with OAuth Resource Access, and thus does not support proxy authentication.

8. Security Considerations

To avoid the situation that the client identifier is fake, the resource server that supports this specification MUST authenticate the client.

Integrity protection SHOULD be applied via a keyed message digest or a digital signature, to prevent an adversary from changing any elements conveyed within the JWT payload. Special care MUST be applied when carrying client's secret key inside the JWT, since those not only require integrity protection, but also confidentiality protection. The client's secret key must be encrypted and kept securely.

A client identifier may be used as a correlation handle if it has relationship with the user, e.g. mobile phone number. Thus, for privacy reasons, it is recommended to keep client identifier confidentially protected.

9. Acknowledgements

Thanks Mike Jones for the reviews, and thanks Brian Campbell, John Bradley for the discussions.

10. References

10.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

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

10.2. Informative References

[PKCE] Sakimura, N., "Proof Key for Code Exchange by OAuth Public Clients", July 2015.

[POPA] Hunt, P., Ed., "OAuth 2.0 Proof-of-Possession (PoP) Security Architecture", March 2015.

[POPS] Jones, M., "Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs)", March 2015.

[TINTRO] Richer, J., "OAuth 2.0 Token Introspection", July 2015.

Appendix A. Document History

-05 Added more justification. Also, added "Finding the client key" section.

-04 Added justification section

-03 Removed most of the duplication with [POPS]

-02 Included key confirmation method etc. The first version on the tools.ietf.org. (Previous versions were sent just as email attachments.)

Authors' Addresses

Nat Sakimura
Nomura Research Institute

Email: sakimura@gmail.com

Kepeng Li
Alibaba Group

Email: kepeng.lkp@alibaba-inc.com

OAuth Working Group
Internet-Draft
Intended status: Best Current Practice
Expires: January 23, 2016

W. Denniss
Google
J. Bradley
Ping Identity

July 22, 2015

OAuth 2.0 for Native Apps
draft-wdenniss-oauth-native-apps-00

Abstract

OAuth 2.0 authorization requests from native apps should only be made through external user-agents such as the system browser. This specification details the security and usability reasons why this is the case, and how native apps and authorization servers can implement this best practice.

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 January 23, 2016.

Copyright Notice

Table of Contents

1. Introduction

The OAuth 2.0 [RFC6749] authorization framework, documents two ways in Section 9 for native apps to interact with the authorization endpoint: via an embedded user-agent, or an external user-agent.

This document recommends external user-agents (such as the system browser) as the only secure and usable choice for OAuth2. It documents how native apps can implement authorization flows with such agents, and the additional requirements of authorization servers needed to support such usage.

Many native apps today are using an embedded user-agent in the form of a web-view. This approach suffers from several security and usability issues including allowing the client app to eavesdrop user credentials, and forcing users to sign-in to each app separately.

OAuth flows between a native app and the system browser (or another external user-agent) are more secure, and take advantage of the shared authentication state. Operating systems are increasingly making the system browser even more viable for OAuth by allowing apps

to show a browser window within the active app, removing the only usability benefit of using embedded browsers in the first place (not wanting to send the user to another app).

Inter-app communication (such as that between a native OAuth client and the system browser) can be achieved through app-specific custom URI schemes and/or claimed HTTPS URLs. For example, an app can launch the system browser with a HTTPS request (such as an OAuth request), the browser can process the request and return control to the app by simply following a URI using a scheme that the app registered (for example "com.example.app:/oauth2callback?code=..."), or a HTTPS path that the app claimed. Parameters can be passed through these URIs, allowing complete use of OAuth flows, while minimizing the added complexity for authorization servers to support native apps.

1.1. Authorization Flow for Native Apps

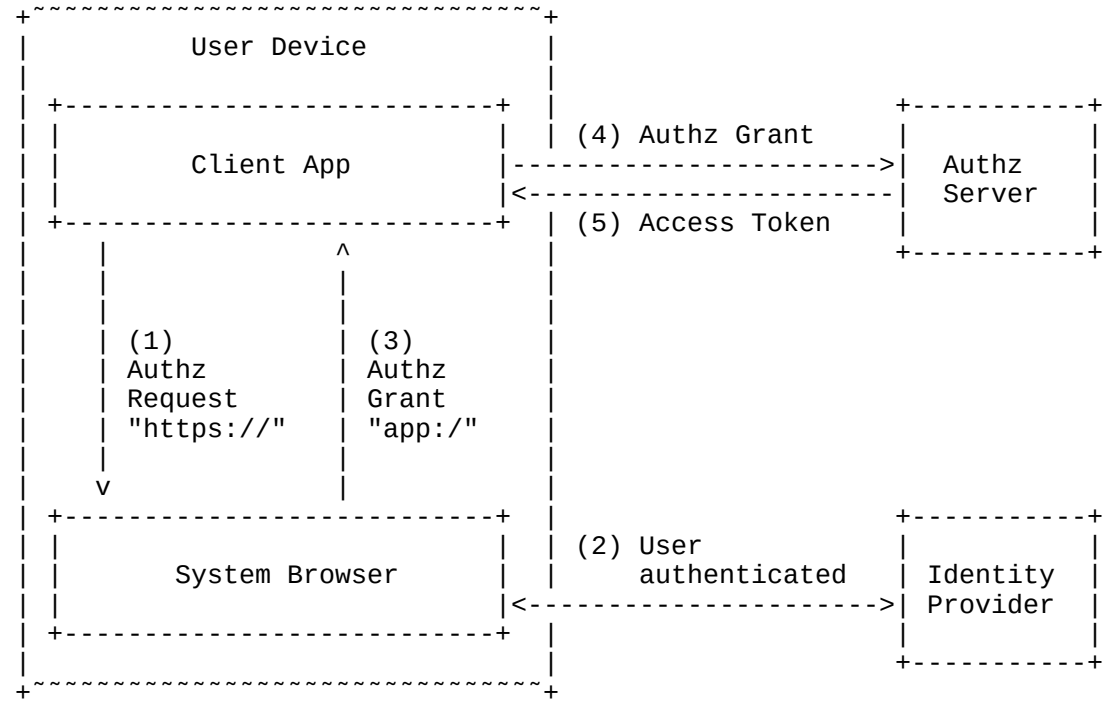

Figure 1: Native App Authorization via External User-agent

Figure 1 illustrates the interaction of the native app with the system browser to achieve authorization via an external user-agent.

1) The client app launches the system browser or browser-view with the authorization request (e.g. https://idp.example.com/oauth2/auth...)

2) Server authenticates the end-user, potentially chaining to another authentication system, and issues Authorization Code Grant on success

3) Browser switches focus back to the client app using a URI with a custom scheme or claimed HTTPS URL, passing the code as a URI parameter.

4) Client presents the OAuth 2.0 authorization code and PKCE [PKCE] proof of possession verifier

5) Server issues the tokens requested

2. 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 Key words for use in RFCs to Indicate Requirement Levels [RFC2119]. If these words are used without being spelled in uppercase then they are to be interpreted with their normal natural language meanings.

3. Terminology

In addition to the terms defined in referenced specifications, this document uses the following terms:

"app" A native application, such as one on a mobile device or desktop operating system.

"app store" An ecommerce store where users can download and purchase apps. Typically with quality-control measures to product users.

"system browser" The operating system’s native default browser, typically pre-installed as part of the operating system, or installed and set as default by the user. For example mobile Safari on iOS, and Chrome on Android.

"web-view" A web browser UI component that can be embedded in apps to render web pages, used to create embedded user-agents.

"browser-view" A full page browser with limited navigation capabilities that is displayed inside a host app, but retains the full security properties and authentication state of the system

browser. Goes by different names on different platforms, such as SFSafariViewController on iOS 9, and Chrome Custom Tab in Chrome for Android.

"reverse domain name notation" A naming convention based on the domain name system, but where where the domain components are reversed, for example "app.example.com" becomes "com.example.app".

"custom URI scheme" A URI scheme (as defined by [RFC3986]) that the app creates and registers with the OS (and is not a standard URI scheme like "https:" or "tel:"). Requests to such a scheme results in the app which registered it being launched by the OS. For example, "myapp:", "com.example.myapp:" are both custom URI schemes.

"inter-app communication" Communication between two apps on a device.

4. The External User-Agent

The external user-agent for native apps can be the system browser, or a native app provided by the authorization server.

Both the system browser and authorization server app affords several advantages for OAuth over embedded web-view based user-agents, including the security of a separate process, and usability of a shared authentication session.

The system browser is the RECOMMENDED external user-agent choice for most authorization servers, as it reduces implementation complexity by reusing the web authorization endpoint, and is often needed as a fallback even when an authorization server app is available.

5. Redirection URIs for Native Apps

5.1. App-claimed HTTPS URI Redirection

Several operating systems support a method for an app to claim a regular HTTPS URL. When such a URL is loaded in the browser, instead of the request being made and the page loaded, the native app is launched instead.

On operating systems that support app-claimed HTTPS URIs, these URIs SHOULD be used with OAuth, as they allow the identity of the destination app to be guaranteed by the operating system.

Apps on platforms that allow the user to disable this functionality, or lack it altogether MUST fallback to using custom URI schemes.

The authorization server MUST allow the registration of HTTPS redirect URIs for non-confidential native clients to support app-claimed HTTPS redirect URIs.

5.2. App-declared Custom URI Scheme Redirection

Most major mobile and desktop computing platforms support inter-app communication via URIs by allowing apps to register custom URI schemes. When the system browser or another app attempts to follow a URI with a custom scheme, the app that registered it is launched to handle the request. This document is only relevant on platforms that support this pattern.

In particular, the custom URI scheme pattern is supported on the mobile platforms Android [Android.URIScheme], iOS [iOS.URIScheme], and Windows Phone [WindowsPhone.URIScheme]. Desktop operating systems Windows [Windows.URIScheme] and OS X [OSX.URIScheme] also support custom URI schemes.

5.2.1. Using Custom URI Schemes for Redirection

To perform an OAuth 2.0 Authorization Request on a supported platform, the native app launches the system browser with a normal OAuth 2.0 Authorization Request, but provides a redirection URI that utilizes a custom URI scheme that is registered by the calling app.

When the authentication server completes the request, it redirects to the client’s redirection URI like it would any redirect URI, but as the redirection URI uses a custom scheme, this results in the OS launching the native app passing in the URI. The native app extracts the code from the query parameters from the URI just like a web client would, and exchanges the Authorization Code like a regular OAuth 2.0 client.

5.2.2. Custom URI Scheme Namespace Considerations

When selecting which URI scheme to associate with the app, apps SHOULD pick a scheme that is globally unique, and which they can assert ownership over.

To avoid clashing with existing schemes in use, using a scheme that follows the reverse domain name pattern applied to a domain under the app publishers control is RECOMMENDED. Such a scheme can be based on a domain they control, or the OAuth client identifier in cases where the authorization server issues client identifiers that are also valid DNS subdomains. The chosen scheme MUST NOT clash with any IANA registered scheme [IANA.URISchemes]. You SHOULD also ensure that no other app by the same publisher uses the same scheme.

Schemes using reverse domain name notation are hardened against collision. They are unlikely to clash with an officially registered scheme [IANA.URISchemes] or unregistered de-facto scheme, as these generally don't include a period character, and are unlikely to match your domain name in any case. They are guaranteed not to clash with any OAuth client following these naming guidelines in full.

Some platforms use globally unique bundle or package names that follow the reverse domain name notation pattern. In these cases, the app SHOULD register that bundle id as the custom scheme. If an app has a bundle id or package name that doesn't match a domain name under the control of the app, the app SHOULD NOT register that as a scheme, and instead create a URI scheme based off one of their domain names.

For example, an app whose publisher owns the top level domain name "example.com" can register "com.example.app:/" as their custom scheme. An app whose authorization server issues client identifiers that are also valid domain names, for example "client1234.usercontent.idp.com", can use the reverse domain name notation of that domain as the scheme, i.e. "com.idp.usercontent.client1234:/". Each of these examples are URI schemes which are likely to be unique, and where the publisher can assert ownership.

As a counter-example, using a simple custom scheme like "myapp:/" is not guaranteed to be unique and is NOT RECOMMENDED.

In addition to uniqueness, basing the URI scheme off a name that is under the control of the app's publisher can help to prove ownership in the event of a dispute where two apps register the same custom scheme (such as if an app is acting maliciously). For example, if two apps registered "com.example.app:", the true owner of "example.com" could petition the app store operator to remove the counterfeit app. This petition is harder to prove if a generic URI scheme was chosen.

5.2.3. Registration of App Redirection URIs

As recommended in Section 3.1.2.2 of [RFC6749], the authorization server SHOULD require the client to pre-register the redirection URI. This remains true for app redirection URIs that use custom schemes.

Additionally, authorization servers MAY request the inclusion of other platform-specific information, such as the app package or bundle name, or other information used to associate the app that may be useful for verifying the calling app's identity, on operating systems that support such functions.

Authorizations servers SHOULD support the ability for native apps to register Redirection URIs that utilize custom URI schemes. Authorization servers SHOULD enforce the recommendation in Section 5.2.2 that apps follow naming guidelines for URI schemes.

6. Security Considerations

6.1. Embedded User-Agents

Embedded user-agents, commonly implemented with web-views, are an alternative method for authorizing native apps. They are however unsafe for use by third-parties by definition. They involve the user signing in with their full login credentials, only to have them downscoped to less powerful OAuth credentials.

Even when used by trusted first-party apps, embedded user-agents violate the principle of least privilege by obtaining more powerful credentials than they need, potentially increasing the attack surface.

In typical web-view based implementations of embedded user-agents, the host application can: log every keystroke entered in the form to capture usernames and passwords; automatically submit forms and bypass user-consent; copy session cookies and use them to perform authenticated actions as the user.

Encouraging users to enter credentials in an embedded web-view without the usual address bar and other identity features that browsers have makes it impossible for the user to know if they are signing in to the legitimate site, and even when they are, it trains them that it’s OK to enter credentials without validating the site first.

Aside from the security concerns, web-views do not share the authentication state with other apps or the system browser, requiring the user to login for every authorization request and leading to a poor user experience.

The only use-case where it is reasonable to use an embedded user-agent is when the app itself is a trusted and secure first-party app that acts as the external user-agent for other apps. Use of embedded user-agents by first party apps other than those that act as an external user-agent themselves is NOT RECOMMENDED, as it increases development complexity and the potential to introduce security issues, and hampers the potential for usability improvements through taking advantage of the shared authentication context.

Authorization servers SHOULD consider taking steps to detect and block logins via embedded user-agents that are not their own, where possible.

6.2. Protecting the Authorization Code

A limitation of custom URI schemes is that multiple apps can typically register the same scheme, which makes it indeterminate as to which app will receive the Authorization Code Grant. This is not an issue for HTTPS redirection URIs (i.e. standard web URLs) due to the fact the HTTPS URI scheme is enforced by the authority (as defined by [RFC3986]), being the domain name system, which does not allow multiple entities to own a single domain.

If multiple apps register the same scheme, it is possible that the authorization code will be sent to the wrong app (generally the operating system makes no guarantee of which app will handle the URI when multiple register the same scheme). Figure 1 of [PKCE] demonstrates the code interception attack. This attack vector applies to public clients (clients that are unable to maintain a client secret) which is typical of most installed apps.

While Section 5.2.2 mentions ways that this can be mitigated through policy enforcement (by being able to request that the offending app is removed), we can also protect the authorization code grant from being used in cases where it was intercepted.

The Proof Key for Code Exchange by OAuth Public Clients (PKCE) [PKCE] standard was created specifically to mitigate against this attack. It is a Proof of Possession extension to OAuth 2.0 that protects the code grant from being used if it is intercepted.

Both the client and the Authorization Server MUST support PKCE [PKCE] to use custom URI schemes. Authorization Servers SHOULD reject requests that use a custom scheme in the redirection URI if the required PKCE parameters are not also present, returning the error message as defined in Section 4.4.1 of [PKCE]

PKCE provides proof of possession by the client generating a secret verifier which it passes in the initial authorization request, and which it must present later when redeeming the authorization code grant. An app that intercepted the authorization code would not be in possession of this secret, rendering the code useless.

6.3. Claimed URLs and Phishing

While using a claimed HTTPS URI for redirection in the system browser guarantees the identity of the receiving app, it is still possible for a bad app to put the user through an authentication flow in an embedded user-agent of their own, and observe the redirect URI.

We can't directly prevent this, however it can be mitigated through user contextual awareness. Such an attack necessarily starts with no authentication state, meaning that the user will be prompted to sign-in. If all native apps are using the techniques described here, users should not be signing-in frequently, and thus should treat any password request event with more suspicion. Sophisticated users will be able to recognise the UI treatment of the browser-view or full system browser, and shouldn't sign-in anywhere else. Users who are particularly security conscious can also use the "open in browser" functionality from the browser-view to gain even more assurances about where they are entering their credentials.

6.4. Always Prompting for User Interaction

Due to the fact that the identity of non-confidential clients cannot be assured, tokens SHOULD NOT be issued to such clients without user consent or interaction, even if the the user has consented to the scopes and approved the client previously.

7. References

7.1. Normative References

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

[PKCE] Sakimura, N., Ed., Bradley, J., and N. Agarwal, "The Proof Key for Code Exchange by OAuth Public Clients", February 2015, <https://tools.ietf.org/html/draft-ietf-oauth-spop>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

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

7.2. Informative References

[RFC6819] Lodderstedt, T., Ed., McGloin, M., and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations", RFC 6819, DOI 10.17487/RFC6819, January 2013, <http://www.rfc-editor.org/info/rfc6819>.

[iOS.URIScheme] "Inter-App Communication", February 2015, <https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/Inter-AppCommunication/Inter-AppCommunication.html>.

[OSX.URIScheme] "Launch Services Concepts", February 2015, <https://developer.apple.com/library/mac/documentation/Carbon/Conceptual/LaunchServicesConcepts/LSCConcepts/LSCConcepts.html#//apple_ref/doc/uid/TP30000999-CH202-CIHFEEAD>.

[Android.URIScheme] "Intents and Intent Filters", February 2015, <http://developer.android.com/guide/components/intents-filters.html#ires>.

[WindowsPhone.URIScheme] "Auto-launching apps using file and URI associations for Windows Phone 8", February 2015, <https://msdn.microsoft.com/en-us/library/windows/apps/jj206987(v=vs.105).aspx>.

[Windows.URIScheme] "Registering an Application to a URI Scheme", February 2015, <https://msdn.microsoft.com/en-us/library/ie/aa767914%28v=vs.85%29.aspx>.

[IANA.URISchemes] "Uniform Resource Identifier (URI) Schemes", February 2015, <http://www.iana.org/assignments/uri-schemes/uri-schemes.xhtml >.

[ChromeCustomTab] "Chrome Custom Tabs", July 2015, <https://developer.chrome.com/multidevice/android/customtabs>.

[SFSafariViewController]
"SafariServices Changes", July 2015, <https://developer.apple.com/library/prerelease/ios/releasenotes/General/iOS90APIDiffs/frameworks/SafariServices.html>.

[Android.AppLinks]
"App Links", July 2015, <https://developer.android.com/preview/features/app-linking.html>.

Appendix A. Operating System Specific Implementation Details

Most of this document attempts to lay out best practices in an generic manner, referencing technology available on most operating systems. This non-normative section contains OS-specific implementation details valid at the time of authorship.

It is expected that this OS-specific information will change, but that the overall principles described in this document for using external user-agents will remain valid for longer.

A.1. iOS Implementation Details

From iOS 9, apps can invoke the system browser without the user leaving the app through SFSafariViewController [SFSafariViewController], which implements the browser-view pattern. This class has all the properties of the system browser, and is considered an 'external user-agent', even though it is presented within the host app. Regardless of whether the system browser is opened, or SFSafariViewController, the return of the token goes through the same system.

A.2. Android Implementation Details

Chrome 45 introduced the concept of Chrome Custom Tab [ChromeCustomTab], which follows the browser-view pattern and allows authentication without the user leaving the app.

The return of the token can go through the custom URI scheme or claimed HTTPS URI (including those registered with the App Link [Android.AppLinks] system), or the navigation events can be observed by the host app. It is RECOMMENDED that the custom URI, or claimed HTTPS URI options be used for better portability, to allow the user to open the authorization request in the Chrome app, and to prevent accidental observation of intermediate tokens on URI parameters.

Appendix B. Acknowledgements

The author would like to acknowledge the work of Marius Scurtescu, and Ben Wiley Sittler whose design for using custom URI schemes in native OAuth 2.0 clients formed the basis of Section 5.2.

The following individuals contributed ideas, feedback, and wording that shaped and formed the final specification:

Naveen Agarwal, John Bradley, Brian Campbell, Adam Dawes, Ashish Jain, Paul Madsen, Breno de Medeiros, Eric Sachs, Nat Sakimura, Steve Wright.

Authors' Addresses

William Denniss
Google
1600 Amphitheatre Pkwy
Mountain View, CA 94043
USA

Phone: +1 650-253-0000
Email: wdenniss@google.com
URI: http://google.com/

John Bradley
Ping Identity

Phone: +44 20 8133 3718
Email: ve7jtb@ve7jtb.com
URI: http://www.thread-safe.com/