

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 24, 2015

B. Campbell
Ping Identity
C. Mortimore
Salesforce
M. Jones
Y. Goland
Microsoft
October 21, 2014

Assertion Framework for OAuth 2.0 Client Authentication and
Authorization Grants
draft-ietf-oauth-assertions-18

Abstract

This specification provides a framework for the use of assertions with OAuth 2.0 in the form of a new client authentication mechanism and a new authorization grant type. Mechanisms are specified for transporting assertions during interactions with a token endpoint, as well as general processing rules.

The intent of this specification is to provide a common framework for OAuth 2.0 to interwork with other identity systems using assertions, and to provide alternative client authentication mechanisms.

Note that this specification only defines abstract message flows and processing rules. In order to be implementable, companion specifications are necessary to provide the corresponding concrete instantiations.

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 24, 2015.

Copyright Notice

Copyright (c) 2014 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction	3
2. Notational Conventions	4
3. Framework	4
4. Transporting Assertions	7
4.1. Using Assertions as Authorization Grants	7
4.1.1. Error Responses	8
4.2. Using Assertions for Client Authentication	8
4.2.1. Error Responses	9
5. Assertion Content and Processing	10
5.1. Assertion Metamodel	10
5.2. General Assertion Format and Processing Rules	11
6. Common Scenarios	12
6.1. Client Authentication	12
6.2. Client Acting on Behalf of Itself	12
6.3. Client Acting on Behalf of a User	13
6.3.1. Client Acting on Behalf of an Anonymous User	13
7. Interoperability Considerations	14
8. Security Considerations	14
8.1. Forged Assertion	15
8.2. Stolen Assertion	15
8.3. Unauthorized Disclosure of Personal Information	16
8.4. Privacy Considerations	16
9. IANA Considerations	17
9.1. assertion Parameter Registration	17
9.2. client_assertion Parameter Registration	17
9.3. client_assertion_type Parameter Registration	17
10. References	18
10.1. Normative References	18
10.2. Informative References	18
Appendix A. Acknowledgements	19
Appendix B. Document History	19

Authors' Addresses	23
--------------------	----

1. Introduction

An assertion is a package of information that facilitates the sharing of identity and security information across security domains. Section 3 provides a more detailed description of the concept of an assertion for the purpose of this specification.

OAuth 2.0 [RFC6749] is an authorization framework that enables a third-party application to obtain limited access to a protected HTTP resource. In OAuth, those third-party applications are called clients; they access protected resources by presenting an access token to the HTTP resource. Access tokens are issued to clients by an authorization server with the (sometimes implicit) approval of the resource owner. These access tokens are typically obtained by exchanging an authorization grant, which represents the authorization granted by the resource owner (or by a privileged administrator). Several authorization grant types are defined to support a wide range of client types and user experiences. OAuth also provides an extensibility mechanism for defining additional grant types, which can serve as a bridge between OAuth and other protocol frameworks.

This specification provides a general framework for the use of assertions as authorization grants with OAuth 2.0. It also provides a framework for assertions to be used for client authentication. It provides generic mechanisms for transporting assertions during interactions with an authorization server's token endpoint, as well as general rules for the content and processing of those assertions. The intent is to provide an alternative client authentication mechanism (one that doesn't send client secrets), as well as to facilitate the use of OAuth 2.0 in client-server integration scenarios, where the end-user may not be present.

This specification only defines abstract message flows and processing rules. In order to be implementable, companion specifications are necessary to provide the corresponding concrete instantiations. For instance, SAML 2.0 Profile for OAuth 2.0 Client Authentication and Authorization Grants [I-D.ietf-oauth-saml2-bearer] defines a concrete instantiation for SAML 2.0 assertions and JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants [I-D.ietf-oauth-jwt-bearer] defines a concrete instantiation for JWTs.

Note: The use of assertions for client authentication is orthogonal to and separable from using assertions as an authorization grant. They can be used either in combination or separately. Client assertion authentication is nothing more than an alternative way for

a client to authenticate to the token endpoint and must be used in conjunction with some grant type to form a complete and meaningful protocol request. Assertion authorization grants may be used with or without client authentication or identification. Whether or not client authentication is needed in conjunction with an assertion authorization grant, as well as the supported types of client authentication, are policy decisions at the discretion of the authorization server.

2. 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 [RFC2119] .

Throughout this document, values are quoted to indicate that they are to be taken literally. When using these values in protocol messages, the quotes must not be used as part of the value.

3. Framework

An assertion is a package of information that allows identity and security information to be shared across security domains. An assertion typically contains information about a subject or principal, information about the party that issued the assertion and when was it issued, as well as the conditions under which the assertion is to be considered valid, such as when and where it can be used.

The entity that creates and signs or integrity protects the assertion is typically known as the "Issuer" and the entity that consumes the assertion and relies on its information is typically known as the "Relying Party". In the context of this document, the authorization server acts as a relying party.

Assertions used in the protocol exchanges defined by this specification MUST always be integrity protected using a digital signature or Message Authentication Code applied by the issuer, which authenticates the issuer and ensures integrity of the assertion content. In many cases, the assertion is issued by a third party and it must be protected against tampering by the client that presents it. An assertion MAY additionally be encrypted, preventing unauthorized parties (such as the client) from inspecting the content.

Although this document does not define the processes by which the client obtains the assertion (prior to sending it to the authorization server), there are two common patterns described below.

In the first pattern, depicted in Figure 1, the client obtains an assertion from a third party entity capable of issuing, renewing, transforming, and validating security tokens. Typically such an entity is known as a "Security Token Service" (STS) or just "Token Service" and a trust relationship (usually manifested in the exchange of some kind of key material) exists between the token service and the relying party. The token service is the assertion issuer; its role is to fulfill requests from clients, which present various credentials, and mint assertions as requested, fill them with appropriate information, and integrity protect them with a signature or message authentication code. WS-Trust [OASIS.WS-Trust] is one available standard for requesting security tokens (assertions).

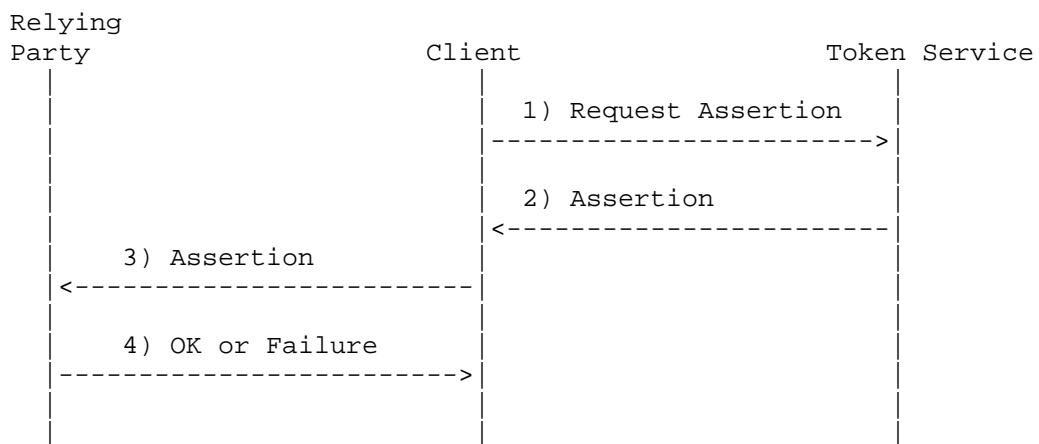


Figure 1: Third Party Created Assertion

In the second pattern, depicted in Figure 2, the client creates assertions locally. To apply the signatures or message authentication codes to assertions, it has to obtain key material: either symmetric keys or asymmetric key pairs. The mechanisms for obtaining this key material are beyond the scope of this specification.

Although assertions are usually used to convey identity and security information, self-issued assertions can also serve a different purpose. They can be used to demonstrate knowledge of some secret, such as a client secret, without actually communicating the secret directly in the transaction. In that case, additional information included in the assertion by the client itself will be of limited value to the relying party and, for this reason, only a bare minimum of information is typically included in such an assertion, such as information about issuing and usage conditions.

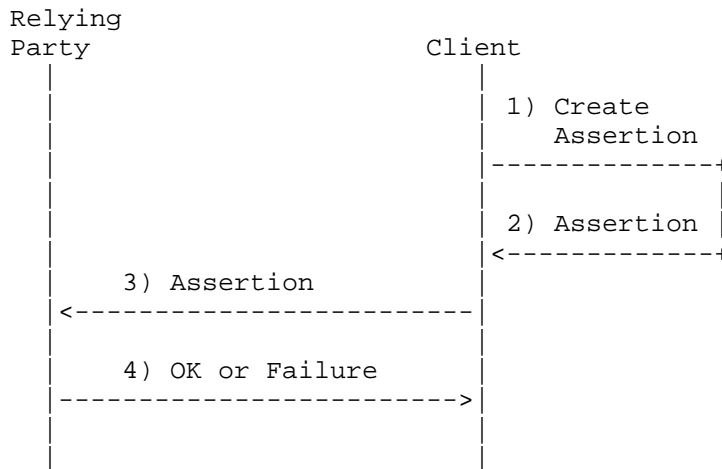


Figure 2: Self-Issued Assertion

Deployments need to determine the appropriate variant to use based on the required level of security, the trust relationship between the entities, and other factors.

From the perspective of what must be done by the entity presenting the assertion, there are two general types of assertions:

1. **Bearer Assertions:** Any entity in possession of a bearer assertion (the bearer) can use it to get access to the associated resources (without demonstrating possession of a cryptographic key). To prevent misuse, bearer assertions need to be protected from disclosure in storage and in transport. Secure communication channels are required between all entities to avoid leaking the assertion to unauthorized parties.
2. **Holder-of-Key Assertions:** To access the associated resources, the entity presenting the assertion must demonstrate possession of additional cryptographic material. The token service thereby binds a key identifier to the assertion and the client has to demonstrate to the relying party that it knows the key corresponding to that identifier when presenting the assertion.

The protocol parameters and processing rules defined in this document are intended to support a client presenting a bearer assertion to an authorization server. They are not directly suitable for use with holder-of-key assertions. While they could be used as a baseline for a holder-of-key assertion system, there would be a need for additional mechanisms (to support proof-of-possession of the secret

key), and possibly changes to the security model (e.g., to relax the requirement for an Audience).

4. Transporting Assertions

This section defines HTTP parameters for transporting assertions during interactions with a token endpoint of an OAuth authorization server. Because requests to the token endpoint result in the transmission of clear-text credentials (in both the HTTP request and response), all requests to the token endpoint MUST use TLS, as mandated in Section 3.2 of OAuth 2.0 [RFC6749].

4.1. Using Assertions as Authorization Grants

This section defines the use of assertions as authorization grants, based on the definition provided in Section 4.5 of OAuth 2.0 [RFC6749]. When using assertions as authorization grants, the client includes the assertion and related information using the following HTTP request parameters:

grant_type

REQUIRED. The format of the assertion as defined by the authorization server. The value will be an absolute URI.

assertion

REQUIRED. The assertion being used as an authorization grant. Specific serialization of the assertion is defined by profile documents.

scope

OPTIONAL. The requested scope as described in Section 3.3 of OAuth 2.0 [RFC6749]. When exchanging assertions for access tokens, the authorization for the token has been previously granted through some out-of-band mechanism. As such, the requested scope MUST be equal or lesser than the scope originally granted to the authorized accessor. The Authorization Server MUST limit the scope of the issued access token to be equal or lesser than the scope originally granted to the authorized accessor.

Authentication of the client is optional, as described in Section 3.2.1 of OAuth 2.0 [RFC6749] and consequently, the "client_id" is only needed when a form of client authentication that relies on the parameter is used.

The following example demonstrates an assertion being used as an authorization grant (with extra line breaks for display purposes only):

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

grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Asaml2-bearer&
assertion=PHNhbWxwOl...[omitted for brevity]...ZT4
```

An assertion used in this context is generally a short lived representation of the authorization grant and authorization servers SHOULD NOT issue access tokens with a lifetime that exceeds the validity period of the assertion by a significant period. In practice, that will usually mean that refresh tokens are not issued in response to assertion grant requests and access tokens will be issued with a reasonably short lifetime. Clients can refresh an expired access token by requesting a new one using the same assertion, if it is still valid, or with a new assertion.

An IETF URN for use as the "grant_type" value can be requested using the template in [RFC6755]. A URN of the form urn:ietf:params:oauth:grant-type:* is suggested.

4.1.1. Error Responses

If an assertion is not valid or has expired, the Authorization Server constructs an error response as defined in OAuth 2.0 [RFC6749]. The value of the "error" parameter MUST be the "invalid_grant" error code. The authorization server MAY include additional information regarding the reasons the assertion was considered invalid using the "error_description" or "error_uri" parameters.

For example:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Cache-Control: no-store

{
  "error": "invalid_grant",
  "error_description": "Audience validation failed"
}
```

4.2. Using Assertions for Client Authentication

The following section defines the use of assertions as client credentials as an extension of Section 2.3 of OAuth 2.0 [RFC6749]. When using assertions as client credentials, the client includes the assertion and related information using the following HTTP request parameters:

client_assertion_type

REQUIRED. The format of the assertion as defined by the authorization server. The value will be an absolute URI.

client_assertion

REQUIRED. The assertion being used to authenticate the client. Specific serialization of the assertion is defined by profile documents.

client_id

OPTIONAL. The client identifier as described in Section 2.2 of OAuth 2.0 [RFC6749]. The "client_id" is unnecessary for client assertion authentication because the client is identified by the subject of the assertion. If present, the value of the "client_id" parameter MUST identify the same client as is identified by the client assertion.

The following example demonstrates a client authenticating using an assertion during an Access Token Request, as defined in Section 4.1.3 of OAuth 2.0 [RFC6749] (with extra line breaks for display purposes only):

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

grant_type=authorization_code&
code=i1WsRnluB1&
client_assertion_type=urn%3Aietf%3Aparams%3Aoauth
%3Aclient-assertion-type%3Asaml2-bearer&
client_assertion=PHNhbW...[omitted for brevity]...ZT
```

Token endpoints can differentiate between assertion based credentials and other client credential types by looking for the presence of the "client_assertion" and "client_assertion_type" parameters, which will only be present when using assertions for client authentication.

An IETF URN for use as the "client_assertion_type" value may be requested using the template in [RFC6755]. A URN of the form urn:ietf:params:oauth:client-assertion-type:* is suggested.

4.2.1. Error Responses

If an assertion is invalid for any reason or if more than one client authentication mechanism is used, the Authorization Server constructs an error response as defined in OAuth 2.0 [RFC6749]. The value of the "error" parameter MUST be the "invalid_client" error code. The authorization server MAY include additional information regarding the

reasons the client assertion was considered invalid using the "error_description" or "error_uri" parameters.

For example:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Cache-Control: no-store

{
  "error": "invalid_client"
  "error_description": "assertion has expired"
}
```

5. Assertion Content and Processing

This section provides a general content and processing model for the use of assertions in OAuth 2.0 [RFC6749].

5.1. Assertion Metamodel

The following are entities and metadata involved in the issuance, exchange, and processing of assertions in OAuth 2.0. These are general terms, abstract from any particular assertion format. Mappings of these terms into specific representations are provided by profiles of this specification.

Issuer

A unique identifier for the entity that issued the assertion. Generally this is the entity that holds the key material used to sign or integrity protect the assertion. Examples of issuers are OAuth clients (when assertions are self-issued) and third party security token services. If the assertion is self-issued, the Issuer value is the client identifier. If the assertion was issued by a Security Token Service (STS), the Issuer should identify the STS in a manner recognized by the Authorization Server. In the absence of an application profile specifying otherwise, compliant applications MUST compare Issuer values using the Simple String Comparison method defined in Section 6.2.1 of RFC 3986 [RFC3986].

Subject

A unique identifier for the principal that is the subject of the assertion.

- * When using assertions for client authentication, the Subject identifies the client to the authorization server using the value of the "client_id" of the OAuth client.

- * When using assertions as an authorization grant, the Subject identifies an authorized accessor for which the access token is being requested (typically the resource owner, or an authorized delegate).

Audience

A value that identifies the party or parties intended to process the assertion. The URL of the Token Endpoint, as defined in Section 3.2 of OAuth 2.0 [RFC6749], can be used to indicate that the authorization server as a valid intended audience of the assertion. In the absence of an application profile specifying otherwise, compliant applications MUST compare the audience values using the Simple String Comparison method defined in Section 6.2.1 of RFC 3986 [RFC3986].

Issued At

The time at which the assertion was issued. While the serialization may differ by assertion format, it is REQUIRED that the time be expressed in UTC with no time zone component.

Expires At

The time at which the assertion expires. While the serialization may differ by assertion format, it is REQUIRED that the time be expressed in UTC with no time zone component.

Assertion ID

A nonce or unique identifier for the assertion. The Assertion ID may be used by implementations requiring message de-duplication for one-time use assertions. Any entity that assigns an identifier MUST ensure that there is negligible probability that that entity or any other entity will accidentally assign the same identifier to a different data object.

5.2. General Assertion Format and Processing Rules

The following are general format and processing rules for the use of assertions in OAuth:

- o The assertion MUST contain an Issuer. The Issuer identifies the entity that issued the assertion as recognized by the Authorization Server. If an assertion is self-issued, the Issuer MUST be the value of the client's "client_id".
- o The assertion MUST contain a Subject. The Subject typically identifies an authorized accessor for which the access token is being requested (i.e., the resource owner or an authorized delegate), but in some cases, may be a pseudonymous identifier or other value denoting an anonymous user. When the client is acting

on behalf of itself, the Subject MUST be the value of the client's "client_id".

- o The assertion MUST contain an Audience that identifies the Authorization Server as the intended audience. The Authorization Server MUST reject any assertion that does not contain the its own identity as the intended audience.
- o The assertion MUST contain an Expires At entity that limits the time window during which the assertion can be used. The authorization server MUST reject assertions that have expired (subject to allowable clock skew between systems). Note that the authorization server may reject assertions with an Expires At attribute value that is unreasonably far in the future.
- o The assertion MAY contain an Issued At entity containing the UTC time at which the assertion was issued.
- o The Authorization Server MUST reject assertions with an invalid signature or Message Authentication Code. The algorithm used to validate the signature or message authentication code and the mechanism for designating the secret used to generate the signature or message authentication code over the assertion are beyond the scope of this specification.

6. Common Scenarios

The following provides additional guidance, beyond the format and processing rules defined in Section 4 and Section 5, on assertion use for a number of common use cases.

6.1. Client Authentication

A client uses an assertion to authenticate to the authorization server's token endpoint by using the "client_assertion_type" and "client_assertion" parameters as defined in Section 4.2. The Subject of the assertion identifies the client. If the assertion is self-issued by the client, the Issuer of the assertion also identifies the client.

The example in Section 4.2 shows a client authenticating using an assertion during an Access Token Request.

6.2. Client Acting on Behalf of Itself

When a client is accessing resources on behalf of itself, it does so in a manner analogous to the Client Credentials Grant defined in Section 4.4 of OAuth 2.0 [RFC6749]. This is a special case that

combines both the authentication and authorization grant usage patterns. In this case, the interactions with the authorization server should be treated as using an assertion for Client Authentication according to Section 4.2, while using the `grant_type` parameter with the value "client_credentials" to indicate that the client is requesting an access token using only its client credentials.

The following example demonstrates an assertion being used for a Client Credentials Access Token Request, as defined in Section 4.4.2 of OAuth 2.0 [RFC6749] (with extra line breaks for display purposes only):

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

grant_type=client_credentials&
client_assertion_type=urn%3Aietf%3Aparams%3Aoauth
%3Aclient-assertion-type%3Asaml2-bearer&
client_assertion=PHNhbW...[omitted for brevity]...ZT
```

6.3. Client Acting on Behalf of a User

When a client is accessing resources on behalf of a user, it does so by using the "grant_type" and "assertion" parameters as defined in Section 4.1. The Subject identifies an authorized accessor for which the access token is being requested (typically the resource owner, or an authorized delegate).

The example in Section 4.1 shows a client making an Access Token Request using an assertion as an Authorization Grant.

6.3.1. Client Acting on Behalf of an Anonymous User

When a client is accessing resources on behalf of an anonymous user, a mutually agreed upon Subject identifier indicating anonymity is used. The Subject value might be an opaque persistent or transient pseudonymous identifier for the user or be an agreed upon static value indicating an anonymous user (e.g., "anonymous"). The authorization may be based upon additional criteria, such as additional attributes or claims provided in the assertion. For example, a client might present an assertion from a trusted issuer asserting that the bearer is over 18 via an included claim. In this case, no additional information about the user's identity is included, yet all the data needed to issue an access token is present.

More information about anonymity, pseudonymity, and privacy considerations in general can be found in [RFC6973].

7. Interoperability Considerations

This specification defines a framework for using assertions with OAuth 2.0. However, as an abstract framework in which the data formats used for representing many values are not defined, on its own, this specification is not sufficient to produce interoperable implementations.

Two other specifications that profile this framework for specific assertion have been developed: one [I-D.ietf-oauth-saml2-bearer] uses SAML 2.0-based assertions and the other [I-D.ietf-oauth-jwt-bearer] uses JSON Web Tokens (JWTs). These two instantiations of this framework specify additional details about the assertion encoding and processing rules for using those kinds of assertions with OAuth 2.0.

However, even when profiled for specific assertion types, agreements between system entities regarding identifiers, keys, and endpoints are required in order to achieve interoperable deployments. Specific items that require agreement are as follows: values for the issuer and audience identifiers, supported assertion and client authentication types, the location of the token endpoint, the key used to apply and verify the digital signature or Message Authentication Code over the assertion, one-time use restrictions on assertions, maximum assertion lifetime allowed, and the specific subject and attribute requirements of the assertion. The exchange of such information is explicitly out of scope for this specification. Deployments for particular trust frameworks, circles of trust, or other uses cases will need to agree among the participants on the kinds of values to be used for some abstract fields defined by this specification. In some cases, additional profiles may be created that constrain or prescribe these values or specify how they are to be exchanged. The OAuth 2.0 Dynamic Client Registration Core Protocol [I-D.ietf-oauth-dyn-reg] is one such profile that enables OAuth Clients to register metadata about themselves at an Authorization Server.

8. Security Considerations

This section discusses security considerations that apply when using assertions with OAuth 2.0 as described in this document. As discussed in Section 3, there are two different ways to obtain assertions: either as self-issued or obtained from a third party token service. While the actual interactions for obtaining an assertion are outside the scope of this document, the details are important from a security perspective. Section 3 discusses the high

level architectural aspects. Many of the security considerations discussed in this section are applicable to both the OAuth exchange as well as the client obtaining the assertion.

The remainder of this section focuses on the exchanges that concern presenting an assertion for client authentication and for the authorization grant.

8.1. Forged Assertion

Threat:

An adversary could forge or alter an assertion in order to obtain an access token (in case of the authorization grant) or to impersonate a client (in case of the client authentication mechanism).

Countermeasures:

To avoid this kind of attack, the entities must assure that proper mechanisms for protecting the integrity of the assertion are employed. This includes the issuer digitally signing the assertion or computing a keyed message digest over the assertion.

8.2. Stolen Assertion

Threat:

An adversary may be able obtain an assertion (e.g., by eavesdropping) and then reuse it (replay it) at a later point in time.

Countermeasures:

The primary mitigation for this threat is the use of secure communication channels with server authentication for all network exchanges.

An assertion may also contain several elements to prevent replay attacks. There is, however, a clear tradeoff between reusing an assertion for multiple exchanges and obtaining and creating new fresh assertions.

Authorization Servers and Resource Servers may use a combination of the Assertion ID and Issued At/Expires At attributes for replay protection. Previously processed assertions may be rejected based on the Assertion ID. The addition of the validity window relieves the authorization server from maintaining an infinite state table of processed Assertion IDs.

8.3. Unauthorized Disclosure of Personal Information

Threat:

The ability for other entities to obtain information about an individual, such as authentication information, role in an organization, or other authorization relevant information, raises privacy concerns.

Countermeasures:

To address the threats, two cases need to be differentiated:

First, a third party that did not participate in any of the exchange is prevented from eavesdropping on the content of the assertion by employing confidentiality protection of the exchange using TLS. This ensures that an eavesdropper on the wire is unable to obtain information. However, this does not prevent legitimate protocol entities from obtaining information that they are not allowed to possess from assertions. Some assertion formats allow for the assertion to be encrypted, preventing unauthorized parties from inspecting the content.

Second, an Authorization Server may obtain an assertion that was created by a third party token service and that token service may have placed attributes into the assertion. To mitigate potential privacy problems, prior consent for the release of such attribute information from the resource owner should be obtained. OAuth itself does not directly provide such capabilities, but this consent approval may be obtained using other identity management protocols, user consent interactions, or in an out-of-band fashion.

For the cases where a third party token service creates assertions to be used for client authentication, privacy concerns are typically lower, since many of these clients are Web servers rather than individual devices operated by humans. If the assertions are used for client authentication of devices or software that can be closely linked to end users, then privacy protection safeguards need to be taken into consideration.

Further guidance on privacy friendly protocol design can be found in [RFC6973].

8.4. Privacy Considerations

An assertion may contain privacy-sensitive information and, to prevent disclosure of such information to unintended parties, should only be transmitted over encrypted channels, such as TLS. In cases

where it is desirable to prevent disclosure of certain information the client, the assertion, or portions of it, should be encrypted to the authorization server.

Deployments should determine the minimum amount of information necessary to complete the exchange and include only such information in the assertion. In some cases, the subject identifier can be a value representing an anonymous or pseudonymous user, as described in Section 6.3.1.

9. IANA Considerations

This is a request to add three values, as listed in the sub-sections below, to the "OAuth Parameters" registry established by RFC 6749 [RFC6749].

9.1. assertion Parameter Registration

- o Parameter name: assertion
- o Parameter usage location: token request
- o Change controller: IESG
- o Specification document(s): [[this document]]

9.2. client_assertion Parameter Registration

- o Parameter name: client_assertion
- o Parameter usage location: token request
- o Change controller: IESG
- o Specification document(s): [[this document]]

9.3. client_assertion_type Parameter Registration

- o Parameter name: client_assertion_type
- o Parameter usage location: token request
- o Change controller: IESG
- o Specification document(s): [[this document]]

10. References

10.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.
- [RFC6749] Hardt, D., "The OAuth 2.0 Authorization Framework", RFC 6749, October 2012.

10.2. Informative References

- [I-D.ietf-oauth-dyn-reg]
Richer, J., Jones, M., Bradley, J., Machulak, M., and P. Hunt, "OAuth 2.0 Dynamic Client Registration Protocol", draft-ietf-oauth-dyn-reg-20 (work in progress), August 2014.
- [I-D.ietf-oauth-jwt-bearer]
Jones, M., Campbell, B., and C. Mortimore, "JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants", draft-ietf-oauth-jwt-bearer (work in progress), October 2014.
- [I-D.ietf-oauth-saml2-bearer]
Campbell, B., Mortimore, C., and M. Jones, "SAML 2.0 Profile for OAuth 2.0 Client Authentication and Authorization Grants", draft-ietf-oauth-saml2-bearer (work in progress), October 2014.
- [OASIS.WS-Trust]
Nadalin, A., Ed., Goodner, M., Ed., Gudgin, M., Ed., Barbir, A., Ed., and H. Granqvist, Ed., "WS-Trust", Feb 2009.
- [RFC6755] Campbell, B. and H. Tschofenig, "An IETF URN Sub-Namespace for OAuth", RFC 6755, October 2012.
- [RFC6973] Cooper, A., Tschofenig, H., Aboba, B., Peterson, J., Morris, J., Hansen, M., and R. Smith, "Privacy Considerations for Internet Protocols", RFC 6973, July 2013.

Appendix A. Acknowledgements

The authors wish to thank the following people that have influenced or contributed this specification: Paul Madsen, Eric Sachs, Jian Cai, Tony Nadalin, Hannes Tschofenig, the authors of the OAuth WRAP specification, and the members of the OAuth working group.

Appendix B. Document History

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

draft-ietf-oauth-assertions-18

- o Changes/suggestions from IESG reviews.

draft-ietf-oauth-assertions-17

- o Added Privacy Considerations section per AD review discussion
<http://www.ietf.org/mail-archive/web/oauth/current/msg13148.html>
and <http://www.ietf.org/mail-archive/web/oauth/current/msg13144.html>

draft-ietf-oauth-assertions-16

- o Clarified some text around the treatment of subject based on the rough rough consensus from the thread staring at
<http://www.ietf.org/mail-archive/web/oauth/current/msg12630.html>

draft-ietf-oauth-assertions-15

- o Updated references.
- o Improved formatting of hanging lists.

draft-ietf-oauth-assertions-14

- o Update reference: draft-iab-privacy-considerations is now RFC 6973
- o Update reference: draft-ietf-oauth-dyn-reg from -13 to -15

draft-ietf-oauth-assertions-13

- o Clean up language around subject per the subject part of
<http://www.ietf.org/mail-archive/web/oauth/current/msg12155.html>
- o Replace "Client Credentials flow" by "Client Credentials _Grant_" as suggested in <http://www.ietf.org/mail-archive/web/oauth/current/msg12155.html>

- o For consistency with SAML and JWT per <http://www.ietf.org/mail-archive/web/oauth/current/msg12251.html> and <http://www.ietf.org/mail-archive/web/oauth/current/msg12253.html> Stated that "In the absence of an application profile specifying otherwise, compliant applications MUST compare the audience values using the Simple String Comparison method defined in Section 6.2.1 of RFC 3986."
- o Added one-time use, maximum lifetime, and specific subject and attribute requirements to Interoperability Considerations.

draft-ietf-oauth-assertions-12

- o Stated that issuer and audience values SHOULD be compared using the Simple String Comparison method defined in Section 6.2.1 of RFC 3986 unless otherwise specified by the application.

draft-ietf-oauth-assertions-11

- o Addressed comments from IESG evaluation
<https://datatracker.ietf.org/doc/draft-ietf-oauth-assertions/ballot/>.
- o Reworded Interoperability Considerations to state what identifiers, keys, endpoints, etc. need to be exchanged/agreed upon.
- o Added brief description of assertion to the intro and included a reference to Section 3 (Framework) where it's described more.
- o Changed such that a self-issued assertion must (was should) have the client id as the issuer.
- o Changed "Specific Assertion Format and Processing Rules" to "Common Scenarios" and reworded to be more suggestive of common practices, rather than trying to be normative. Also removed lots of repetitive text in that section.
- o Refined language around audience, subject, client identifiers, etc. to hopefully be clearer and less redundant.
- o Changed title from "Assertion Framework for OAuth 2.0" to "Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants" to be more explicit about the scope of the document per <http://www.ietf.org/mail-archive/web/oauth/current/msg11063.html>.
- o Noted that authentication of the client per Section 3.2.1 of OAuth is optional for an access token request with an assertion as an

authorization grant and removed `client_id` from the associated example.

draft-ietf-oauth-assertions-10

- o Changed term "Principal" to "Subject".
- o Added Interoperability Considerations section.
- o Applied Shawn Emery's comments from the security directorate review, including correcting `urn:ietf:params:oauth:grant_type:*` to `urn:ietf:params:oauth:grant-type:*`.

draft-ietf-oauth-assertions-09

- o Allow audience values to not be URIs.
- o Added informative references to draft-ietf-oauth-saml2-bearer and draft-ietf-oauth-jwt-bearer.
- o Clarified that the statements about possible issuers are non-normative by using the language "Examples of issuers".

draft-ietf-oauth-assertions-08

- o Update reference to RFC 6755 from draft-ietf-oauth-urn-sub-ns
- o Tidy up IANA consideration section

draft-ietf-oauth-assertions-07

- o Reference RFC 6749.
- o Remove extraneous word per <http://www.ietf.org/mail-archive/web/oauth/current/msg10029.html>

draft-ietf-oauth-assertions-06

- o Add more text to intro explaining that an assertion grant type can be used with or without client authentication/identification and that client assertion authentication is nothing more than an alternative way for a client to authenticate to the token endpoint

draft-ietf-oauth-assertions-05

- o Non-normative editorial cleanups

draft-ietf-oauth-assertions-04

- o Updated document to incorporate the review comments from the shepherd - thread and alternative draft at <http://www.ietf.org/mail-archive/web/oauth/current/msg09437.html>
- o Added reference to draft-ietf-oauth-urn-sub-ns and include suggestions on `urn:ietf:params:oauth:[grant-type|client-assertion-type]:* URNs`

draft-ietf-oauth-assertions-03

- o updated reference to draft-ietf-oauth-v2 from -25 to -26

draft-ietf-oauth-assertions-02

- o Added text about limited lifetime ATs and RTs per <http://www.ietf.org/mail-archive/web/oauth/current/msg08298.html>.
- o Changed the line breaks in some examples to avoid awkward rendering to text format. Also removed encoded '=' padding from a few examples because both known derivative specs, SAML and JWT, omit the padding char in serialization/encoding.
- o Remove section 7 on error responses and move that (somewhat modified) content into subsections of section 4 broken up by authn/authz per <http://www.ietf.org/mail-archive/web/oauth/current/msg08735.html>.
- o Rework the text about "MUST validate ... in order to establish a mapping between ..." per <http://www.ietf.org/mail-archive/web/oauth/current/msg08872.html> and <http://www.ietf.org/mail-archive/web/oauth/current/msg08749.html>.
- o Change "The Principal MUST identify an authorized accessor. If the assertion is self-issued, the Principal SHOULD be the client_id" in 6.1 per <http://www.ietf.org/mail-archive/web/oauth/current/msg08873.html>.
- o Update reference in 4.1 to point to 2.3 (rather than 3.2) of oauth-v2 (rather than self) <http://www.ietf.org/mail-archive/web/oauth/current/msg08874.html>.
- o Move the "Section 3 of" out of the xref to hopefully fix the link in 4.1 and remove the client_id bullet from 4.2 per <http://www.ietf.org/mail-archive/web/oauth/current/msg08875.html>.
- o Add ref to Section 3.3 of oauth-v2 for scope definition and remove some then redundant text per <http://www.ietf.org/mail-archive/web/oauth/current/msg08890.html>.

- o Change "The following format and processing rules SHOULD be applied" to "The following format and processing rules apply" in sections 6.x to remove conflicting normative qualification of other normative statements per <http://www.ietf.org/mail-archive/web/oauth/current/msg08892.html>.
- o Add text the client_id must id the client to 4.1 and remove similar text from other places per <http://www.ietf.org/mail-archive/web/oauth/current/msg08893.html>.
- o Remove the MUST from the text prior to the HTTP parameter definitions per <http://www.ietf.org/mail-archive/web/oauth/current/msg08920.html>.
- o Updated examples to use grant_type and client_assertion_type values from the OAuth SAML Assertion Profiles spec.

Authors' Addresses

Brian Campbell
Ping Identity

Email: brian.d.campbell@gmail.com

Chuck Mortimore
Salesforce.com

Email: cmortimore@salesforce.com

Michael B. Jones
Microsoft

Email: mbj@microsoft.com

Yaron Y. Goland
Microsoft

Email: yarong@microsoft.com

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: November 29, 2015

J. Richer, Ed.

M. Jones
Microsoft
J. Bradley
Ping Identity
M. Machulak
Newcastle University
P. Hunt
Oracle Corporation
May 28, 2015

OAuth 2.0 Dynamic Client Registration Protocol
draft-ietf-oauth-dyn-reg-30

Abstract

This specification defines mechanisms for dynamically registering OAuth 2.0 clients with authorization servers. Registration requests send a set of desired client metadata values to the authorization server. The resulting registration responses return a client identifier to use at the authorization server and the client metadata values registered for the client. The client can then use this registration information to communicate with the authorization server using the OAuth 2.0 protocol. This specification also defines a set of common client metadata fields and values for clients to use during registration.

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 November 29, 2015.

Copyright Notice

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

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

Table of Contents

1. Introduction	3
1.1. Notational Conventions	4
1.2. Terminology	4
1.3. Protocol Flow	6
2. Client Metadata	7
2.1. Relationship between Grant Types and Response Types . . .	11
2.2. Human-Readable Client Metadata	12
2.3. Software Statement	13
3. Client Registration Endpoint	14
3.1. Client Registration Request	15
3.1.1. Client Registration Request Using a Software Statement	17
3.2. Responses	18
3.2.1. Client Information Response	18
3.2.2. Client Registration Error Response	20
4. IANA Considerations	22
4.1. OAuth Dynamic Client Registration Metadata Registry . . .	22
4.1.1. Registration Template	23
4.1.2. Initial Registry Contents	23
4.2. OAuth Token Endpoint Authentication Methods Registry . .	26
4.2.1. Registration Template	26
4.2.2. Initial Registry Contents	27
5. Security Considerations	27
6. Privacy Considerations	30
7. References	31
7.1. Normative References	31
7.2. Informative References	33
Appendix A. Use Cases	33
A.1. Open versus Protected Dynamic Client Registration	33
A.1.1. Open Dynamic Client Registration	33
A.1.2. Protected Dynamic Client Registration	33

A.2. Registration Without or With Software Statements	34
A.2.1. Registration Without a Software Statement	34
A.2.2. Registration With a Software Statement	34
A.3. Registration by the Client or Developer	34
A.3.1. Registration by the Client	34
A.3.2. Registration by the Developer	34
A.4. Client ID per Client Instance or per Client Software . .	34
A.4.1. Client ID per Client Software Instance	34
A.4.2. Client ID Shared Among All Instances of Client Software	35
A.5. Stateful or Stateless Registration	35
A.5.1. Stateful Client Registration	35
A.5.2. Stateless Client Registration	35
Appendix B. Acknowledgments	35
Appendix C. Document History	36
Authors' Addresses	42

1. Introduction

In order for an OAuth 2.0 [RFC6749] client to utilize an OAuth 2.0 authorization server, the client needs specific information to interact with the server, including an OAuth 2.0 client identifier to use at that server. This specification describes how an OAuth 2.0 client can be dynamically registered with an authorization server to obtain this information.

As part of the registration process, this specification also defines a mechanism for the client to present the authorization server with a set of metadata, such as a set of valid redirection URIs. This metadata can either be communicated in a self-asserted fashion or as a set of metadata called a software statement, which is digitally signed or MACed; in the case of a software statement, the issuer is vouching for the validity of the data about the client.

Traditionally, registration of a client with an authorization server is performed manually. The mechanisms defined in this specification can be used either for a client to dynamically register itself with authorization servers or for a client developer to programmatically register the client with authorization servers. Multiple applications using OAuth 2.0 have previously developed mechanisms for accomplishing such registrations. This specification generalizes the registration mechanisms defined by the OpenID Connect Dynamic Client Registration 1.0 [OpenID.Registration] specification and used by the User Managed Access (UMA) Profile of OAuth 2.0 [I-D.hardjono-oauth-umacore] specification in a way that is compatible with both, while being applicable to a wider set of OAuth 2.0 use cases.

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 [RFC2119].

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

1.2. Terminology

This specification uses the terms "access token", "authorization code", "authorization endpoint", "authorization grant", "authorization server", "client", "client identifier", "client secret", "grant type", "protected resource", "redirection URI", "refresh token", "resource owner", "resource server", "response type", and "token endpoint" defined by OAuth 2.0 [RFC6749] and uses the term "Claim" defined by JSON Web Token (JWT) [RFC7519].

This specification defines the following terms:

Client Software

Software implementing an OAuth 2.0 client.

Client Instance

A deployed instance of a piece of client software.

Client Developer

The person or organization that builds a client software package and prepares it for distribution. At the time of building the client, the developer is often not aware of who the deploying service provider organizations will be. Client developers will need to use dynamic registration when they are unable to predict aspects of the software, such as the deployment URLs, at compile time. For instance, this can occur when the software API publisher and the deploying organization are not the same.

Client Registration Endpoint

OAuth 2.0 endpoint through which a client can be registered at an authorization server. The means by which the URL for this endpoint is obtained are out of scope for this specification.

Initial Access Token

OAuth 2.0 access token optionally issued by an authorization server to a developer or client and used to authorize calls to the client registration endpoint. The type and format of this token are likely service-specific and are out of scope for this specification. The means by which the authorization server issues

this token as well as the means by which the registration endpoint validates this token are out of scope for this specification. Use of an initial access token is required when the authorization server limits the parties that can register a client.

Deployment Organization

An administrative security domain under which a software API (service) is deployed and protected by an OAuth 2.0 framework. In some OAuth scenarios, the deployment organization and the software API publisher are the same. In these cases, the deploying organization will often have a close relationship with client software developers. In many other cases, the definer of the service may be an independent third-party publisher or a standards organization. When working to a published specification for an API, the client software developer is unable to have a prior relationship with the potentially many deployment organizations deploying the software API (service).

Software API Deployment

A deployed instance of a software API that is protected by OAuth 2.0 (a protected resource) in a particular deployment organization domain. For any particular software API, there may be one or more deployments. A software API deployment typically has an associated OAuth 2.0 authorization server as well as a client registration endpoint. The means by which endpoints are obtained are out of scope for this specification.

Software API Publisher

The organization that defines a particular web accessible API that may be deployed in one or more deployment environments. A publisher may be any standards body, commercial, public, private, or open source organization that is responsible for publishing and distributing software and API specifications that may be protected via OAuth 2.0. In some cases, a software API publisher and a client developer may be the same organization. At the time of publication of a web accessible API, the software publisher often does not have a prior relationship with the deploying organizations.

Software Statement

Digitally signed or MACed JSON Web Token (JWT) [RFC7519] that asserts metadata values about the client software. In some cases, a software statement will be issued directly by the client developer. In other cases, a software statement will be issued by a third party organization for use by the client developer. In both cases, the trust relationship the authorization server has with the issuer of the software statement is intended to be used as an input to the evaluation of whether the registration request

is accepted. A software statement can be presented to an authorization server as part of a client registration request.

1.3. Protocol Flow

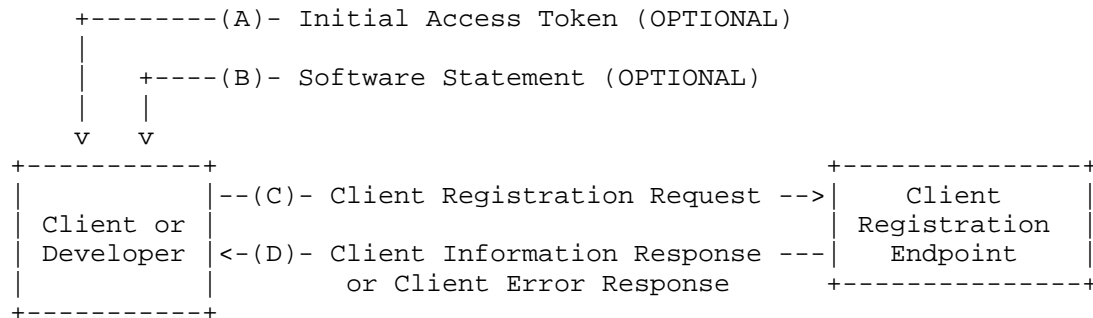


Figure 1: Abstract Dynamic Client Registration Flow

The abstract OAuth 2.0 client dynamic registration flow illustrated in Figure 1 describes the interaction between the client or developer and the endpoint defined in this specification. This figure does not demonstrate error conditions. This flow includes the following steps:

- (A) Optionally, the client or developer is issued an initial access token giving access to the client registration endpoint. The method by which the initial access token is issued to the client or developer is out of scope for this specification.
- (B) Optionally, the client or developer is issued a software statement for use with the client registration endpoint. The method by which the software statement is issued to the client or developer is out of scope for this specification.
- (C) The client or developer calls the client registration endpoint with the client's desired registration metadata, optionally including the initial access token from (A) if one is required by the authorization server.
- (D) The authorization server registers the client and returns:
 - * the client's registered metadata,
 - * a client identifier that is unique at the server, and

- * a set of client credentials such as a client secret, if applicable for this client.

Examples of different configurations and usages are included in Appendix A.

2. Client Metadata

Registered clients have a set of metadata values associated with their client identifier at an authorization server, such as the list of valid redirection URIs or a display name.

These client metadata values are used in two ways:

- o as input values to registration requests, and
- o as output values in registration responses.

The following client metadata fields are defined by this specification. The implementation and use of all client metadata fields is OPTIONAL, unless stated otherwise. All data member types (strings, arrays, numbers) are defined in terms of their JSON [RFC7159] representations.

`redirect_uris`

Array of redirection URI strings for use in redirect-based flows such as the authorization code and implicit flows. As required by Section 2 of OAuth 2.0 [RFC6749], clients using flows with redirection MUST register their redirection URI values. Authorization servers that support dynamic registration for redirect-based flows MUST implement support for this metadata value.

`token_endpoint_auth_method`

String indicator of the requested authentication method for the token endpoint. Values defined by this specification are:

- * "none": The client is a public client as defined in OAuth 2.0 and does not have a client secret.
- * "client_secret_post": The client uses the HTTP POST parameters defined in OAuth 2.0 section 2.3.1.
- * "client_secret_basic": the client uses HTTP Basic defined in OAuth 2.0 section 2.3.1

Additional values can be defined via the IANA OAuth Token Endpoint Authentication Methods Registry established in Section 4.2.

Absolute URIs can also be used as values for this parameter without being registered. If unspecified or omitted, the default is "client_secret_basic", denoting HTTP Basic Authentication Scheme as specified in Section 2.3.1 of OAuth 2.0.

grant_types

Array of OAuth 2.0 grant type strings that the client can use at the token endpoint. These grant types are defined as follows:

- * "authorization_code": The Authorization Code Grant described in OAuth 2.0 Section 4.1
- * "implicit": The Implicit Grant described in OAuth 2.0 Section 4.2
- * "password": The Resource Owner Password Credentials Grant described in OAuth 2.0 Section 4.3
- * "client_credentials": The Client Credentials Grant described in OAuth 2.0 Section 4.4
- * "refresh_token": The Refresh Token Grant described in OAuth 2.0 Section 6.
- * "urn:ietf:params:oauth:grant-type:jwt-bearer": The JWT Bearer Grant defined in OAuth JWT Bearer Token Profiles [RFC7523].
- * "urn:ietf:params:oauth:grant-type:saml2-bearer": The SAML 2 Bearer Grant defined in OAuth SAML 2 Bearer Token Profiles [RFC7522].

If the token endpoint is used in the grant type, the value of this parameter MUST be the same as the value of the "grant_type" parameter passed to the token endpoint defined in the grant type definition. Authorization servers MAY allow for other values as defined in the grant type extension process described in OAuth 2.0 Section 2.5. If omitted, the default behavior is that the client will use only the "authorization_code" Grant Type.

response_types

Array of the OAuth 2.0 response type strings that the client can use at the authorization endpoint. These response types are defined as follows:

- * "code": The authorization code response described in OAuth 2.0 Section 4.1.

- * "token": The implicit response described in OAuth 2.0 Section 4.2.

If the authorization endpoint is used by the grant type, the value of this parameter MUST be the same as the value of the "response_type" parameter passed to the authorization endpoint defined in the grant type definition. Authorization servers MAY allow for other values as defined in the grant type extension process is described in OAuth 2.0 Section 2.5. If omitted, the default is that the client will use only the "code" response type.

client_name

Human-readable string name of the client to be presented to the end-user during authorization. If omitted, the authorization server MAY display the raw "client_id" value to the end-user instead. It is RECOMMENDED that clients always send this field. The value of this field MAY be internationalized, as described in Section 2.2.

client_uri

URL string of a web page providing information about the client. If present, the server SHOULD display this URL to the end-user in a clickable fashion. It is RECOMMENDED that clients always send this field. The value of this field MUST point to a valid web page. The value of this field MAY be internationalized, as described in Section 2.2.

logo_uri

URL string that references a logo for the client. If present, the server SHOULD display this image to the end-user during approval. The value of this field MUST point to a valid image file. The value of this field MAY be internationalized, as described in Section 2.2.

scope

String containing a space separated list of scope values (as described in Section 3.3 of OAuth 2.0 [RFC6749]) that the client can use when requesting access tokens. The semantics of values in this list is service specific. If omitted, an authorization server MAY register a client with a default set of scopes.

contacts

Array of strings representing ways to contact people responsible for this client, typically email addresses. The authorization server MAY make these contact addresses available to end-users for support requests for the client. See Section 6 for information on Privacy Considerations.

tos_uri

URL string that points to a human-readable terms of service document for the client that describes a contractual relationship between the end-user and the client that the end-user accepts when authorizing the client. The authorization server SHOULD display this URL to the end-user if it is provided. The value of this field MUST point to a valid web page. The value of this field MAY be internationalized, as described in Section 2.2.

policy_uri

URL string that points to a human-readable privacy policy document that describes how the deployment organization collects, uses, retains, and discloses personal data. The authorization server SHOULD display this URL to the end-user if it is provided. The value of this field MUST point to a valid web page. The value of this field MAY be internationalized, as described in Section 2.2.

jwtks_uri

URL string referencing the client's JSON Web Key Set [RFC7517] document, which contains the client's public keys. The value of this field MUST point to a valid JWK Set document. These keys can be used by higher level protocols that use signing or encryption. For instance, these keys might be used by some applications for validating signed requests made to the token endpoint when using JWTs for client authentication [RFC7523]. Use of this parameter is preferred over the "jwtks" parameter, as it allows for easier key rotation. The "jwtks_uri" and "jwtks" parameters MUST NOT both be present in the same request or response.

jwtks

Client's JSON Web Key Set [RFC7517] document value, which contains the client's public keys. The value of this field MUST be a JSON object containing a valid JWK Set. These keys can be used by higher level protocols that use signing or encryption. This parameter is intended to be used by clients that cannot use the "jwtks_uri" parameter, such as native clients that cannot host public URLs. The "jwtks_uri" and "jwtks" parameters MUST NOT both be present in the same request or response.

software_id

A unique identifier string (e.g. a UUID) assigned by the client developer or software publisher used by registration endpoints to identify the client software to be dynamically registered. Unlike "client_id", which is issued by the authorization server and SHOULD vary between instances, the "software_id" SHOULD remain the same for all instances of the client software. The "software_id" SHOULD remain the same across multiple updates or versions of the same piece of software. The value of this field is not intended

to be human-readable and is usually opaque to the client and authorization server.

software_version

A version identifier string for the client software identified by "software_id". The value of the "software_version" SHOULD change on any update to the client software identified by the same "software_id". The value of this field is intended to be compared using string equality matching and no other comparison semantics are defined by this specification. The value of this field is outside the scope of this specification, but it is not intended to be human readable and is usually opaque to the client and authorization server. The definition of what constitutes an update to client software that would trigger a change to this value is specific to the software itself and is outside the scope of this specification.

Extensions and profiles of this specification can expand this list with metadata names and descriptions registered in accordance with the IANA Considerations in Section 4 of this document. The authorization server MUST ignore any client metadata sent by the client that it does not understand (for instance, by silently removing unknown metadata from the client's registration record during processing). The authorization server MAY reject any requested client metadata values by replacing requested values with suitable defaults as described in Section 3.2.1 or by returning an error response as described in Section 3.2.2.

Client metadata values can either be communicated directly in the body of a registration request, as described in Section 3.1, or included as claims in a software statement, as described in Section 2.3, or a mixture of both. If the same client metadata name is present in both locations and the software statement is trusted by the authorization server, the value of a claim in the software statement MUST take precedence.

2.1. Relationship between Grant Types and Response Types

The "grant_types" and "response_types" values described above are partially orthogonal, as they refer to arguments passed to different endpoints in the OAuth protocol. However, they are related in that the "grant_types" available to a client influence the "response_types" that the client is allowed to use, and vice versa. For instance, a "grant_types" value that includes "authorization_code" implies a "response_types" value that includes "code", as both values are defined as part of the OAuth 2.0 authorization code grant. As such, a server supporting these fields SHOULD take steps to ensure that a client cannot register itself into

an inconsistent state, for example by returning an "invalid_client_metadata" error response to an inconsistent registration request.

The correlation between the two fields is listed in the table below.

grant_types value includes:	response_types value includes:
authorization_code	code
implicit	token
password	(none)
client_credentials	(none)
refresh_token	(none)
urn:ietf:params:oauth:grant-type:jwt-bearer	(none)
urn:ietf:params:oauth:grant-type:saml2-bearer	(none)

Extensions and profiles of this document that introduce new values to either the "grant_types" or "response_types" parameter MUST document all correspondences between these two parameter types.

2.2. Human-Readable Client Metadata

Human-readable client metadata values and client metadata values that reference human-readable values MAY be represented in multiple languages and scripts. For example, the values of fields such as "client_name", "tos_uri", "policy_uri", "logo_uri", and "client_uri" might have multiple locale-specific values in some client registrations to facilitate use in different locations.

To specify the languages and scripts, BCP47 [RFC5646] language tags are added to client metadata member names, delimited by a # character. Since JSON [RFC7159] member names are case sensitive, it is RECOMMENDED that language tag values used in Claim Names be spelled using the character case with which they are registered in the IANA Language Subtag Registry [IANA.Language]. In particular, normally language names are spelled with lowercase characters, region names are spelled with uppercase characters, and languages are spelled with mixed case characters. However, since BCP47 language tag values are case insensitive, implementations SHOULD interpret the language tag values supplied in a case insensitive manner. Per the recommendations in BCP47, language tag values used in metadata member names should only be as specific as necessary. For instance, using "fr" might be sufficient in many contexts, rather than "fr-CA" or "fr-FR".

For example, a client could represent its name in English as `"client_name#en": "My Client"` and its name in Japanese as `"client_name#ja-Jpan-JP": "\u30AF\u30E9\u30A4\u30A2\u30F3\u30C8\u540D"` within the same registration request. The authorization server MAY display any or all of these names to the resource owner during the authorization step, choosing which name to display based on system configuration, user preferences or other factors.

If any human-readable field is sent without a language tag, parties using it MUST NOT make any assumptions about the language, character set, or script of the string value, and the string value MUST be used as-is wherever it is presented in a user interface. To facilitate interoperability, it is RECOMMENDED that clients and servers use a human-readable field without any language tags in addition to any language-specific fields, and it is RECOMMENDED that any human-readable fields sent without language tags contain values suitable for display on a wide variety of systems.

Implementer's Note: Many JSON libraries make it possible to reference members of a JSON object as members of an object construct in the native programming environment of the library. However, while the `"#"` character is a valid character inside of a JSON object's member names, it is not a valid character for use in an object member name in many programming environments. Therefore, implementations will need to use alternative access forms for these claims. For instance, in JavaScript, if one parses the JSON as follows, `"var j = JSON.parse(json);"`, then as a workaround the member `"client_name#en-us"` can be accessed using the JavaScript syntax `j["client_name#en-us"]`.

2.3. Software Statement

A software statement is a JSON Web Token (JWT) [RFC7519] that asserts metadata values about the client software as a bundle. A set of claims that can be used in a software statement are defined in Section 2. When presented to the authorization server as part of a client registration request, the software statement MUST be digitally signed or MACed using JWS [RFC7515] and MUST contain an `"iss"` (issuer) claim denoting the party attesting to the claims in the software statement. It is RECOMMENDED that software statements be digitally signed using the `"RS256"` signature algorithm, although particular applications MAY specify the use of different algorithms. It is RECOMMENDED that software statements contain the `"software_id"` claim to allow authorization servers to correlate different instances of software using the same software statement.

For example, a software statement could contain the following claims:

```
{
  "software_id": "4NRB1-0XZABZI9E6-5SM3R",
  "client_name": "Example Statement-based Client",
  "client_uri": "https://client.example.net/"
}
```

The following non-normative example JWT includes these claims and has been asymmetrically signed using RS256:

Line breaks are for display purposes only

```
eyJhbGciOiJSUzI1NiJ9.  
eyJzb2Z0d2FyZV9pZCI6IjR0OUxLTBYWkFCWkk5RTYtNVNNM1IiLCJjbGll  
bnRfbmFtZSI6IktV4YW1wbGUU3RhdGVtZW50LWJhc2VkIENsaWVudCisImNs  
aWVudF91cmkiOiJodHRwczovL2NsaWVudC5leGFtcGxlLm5ldC8ifQ.  
GHfL4QNirQwL18BSRde595T9jbzqa06R9BT8w409x9oIcKaZo_mt15riEXHa  
zdISUvDIzhtiyNrSHQ8K4TvgWxH6uJgcmoodZdPwmWRIEYbQDLqPNxRetYn0  
5X3AR7ia4FRjQ2ojuZjk5fJqJdQ-JcfxyhK-P8BAWBd6I2LLA77IG32xtbhxy  
fHX7VhuU5ProJ02v3Ayy4XrhLZJY4yKfmyjiiKiPNe-Ia4SMY_d_QSWsx  
U5XIQL5Sa2YRPMbDRxttm2TfnZM1xx70DoYi8g6czZ-CPGRi4SW_S2RKHIJf  
IjoI3zTJ0Y2oe0_EJAiXbL6OyF9S5tKxDXV8JIndSA
```

The means by which a client or developer obtains a software statement are outside the scope of this specification. Some common methods could include a client developer generating a client-specific JWT by registering with a software API publisher to obtain a software statement for a class of clients. The software statement is typically distributed with all instances of a client application.

The criteria by which authorization servers determine whether to trust and utilize the information in a software statement are beyond the scope of this specification.

In some cases, authorization servers MAY choose to accept a software statement value directly as a client identifier in an authorization request, without a prior dynamic client registration having been performed. The circumstances under which an authorization server would do so, and the specific software statement characteristics required in this case, are beyond the scope of this specification.

3. Client Registration Endpoint

The client registration endpoint is an OAuth 2.0 endpoint defined in this document that is designed to allow a client to be registered with the authorization server. The client registration endpoint **MUST** accept HTTP POST messages with request parameters encoded in the entity body using the "application/json" format. The client

registration endpoint MUST be protected by a transport-layer security mechanism, as described in Section 5.

The client registration endpoint MAY be an OAuth 2.0 protected resource and accept an initial access token in the form of an OAuth 2.0 [RFC6749] access token to limit registration to only previously authorized parties. The method by which the initial access token is obtained by the client or developer is generally out-of-band and is out of scope for this specification. The method by which the initial access token is verified and validated by the client registration endpoint is out of scope for this specification.

To support open registration and facilitate wider interoperability, the client registration endpoint SHOULD allow registration requests with no authorization (which is to say, with no initial access token in the request). These requests MAY be rate-limited or otherwise limited to prevent a denial-of-service attack on the client registration endpoint.

3.1. Client Registration Request

This operation registers a client with the authorization server. The authorization server assigns this client a unique client identifier, optionally assigns a client secret, and associates the metadata provided in the request with the issued client identifier. The request includes any client metadata parameters being specified for the client during the registration. The authorization server MAY provision default values for any items omitted in the client metadata.

To register, the client or developer sends an HTTP POST to the client registration endpoint with a content type of "application/json". The HTTP Entity Payload is a JSON [RFC7159] document consisting of a JSON object and all requested client metadata values as top-level members of that JSON object.

For example, if the server supports open registration (with no initial access token), the client could send the following registration request to the client registration endpoint:

The following is a non-normative example request not using an initial access token (with line wraps within values for display purposes only):

```
POST /register HTTP/1.1
Content-Type: application/json
Accept: application/json
Host: server.example.com

{
  "redirect_uris":[
    "https://client.example.org/callback",
    "https://client.example.org/callback2"],
  "client_name":"My Example Client",
  "client_name#ja-Jpan-JP":
    "\u30AF\u30E9\u30A4\u30A2\u30F3\u30C8\u540D",
  "token_endpoint_auth_method":"client_secret_basic",
  "logo_uri":"https://client.example.org/logo.png",
  "jwks_uri":"https://client.example.org/my_public_keys.jwks",
  "example_extension_parameter": "example_value"
}
```

Alternatively, if the server supports authorized registration, the developer or the client will be provisioned with an initial access token. (The method by which the initial access token is obtained is out of scope for this specification.) The developer or client sends the following authorized registration request to the client registration endpoint. Note that the initial access token sent in this example as an OAuth 2.0 Bearer Token [RFC6750], but any OAuth 2.0 token type could be used by an authorization server.

The following is a non-normative example request using an initial access token and registering a JWK set by value (with line wraps within values for display purposes only):

```
POST /register HTTP/1.1
Content-Type: application/json
Accept: application/json
Authorization: Bearer ey23f2.adfj230.af32-developer321
Host: server.example.com

{
  "redirect_uris":["https://client.example.org/callback",
    "https://client.example.org/callback2"],
  "client_name":"My Example Client",
  "client_name#ja-Jpan-JP":
    "\u30AF\u30E9\u30A4\u30A2\u30F3\u30C8\u540D",
  "token_endpoint_auth_method":"client_secret_basic",
  "policy_uri":"https://client.example.org/policy.html",
  "jwks":{"keys":[{"e": "AQAB",
    "n": "nj3YJwsLUF19BmpAbkOswCNVx17Eh9wMO-_AReZwBqfaWFcfG
HrZXsIV2VMCNVNU8Tpb4obUaSXcRcQ-VMsfQPJm9IzgtRdAY8NN8Xb7PEcYyk
lBjvTtuPbpzIaqyiUepzUXNDFuA0Okriol3WmflPUUgMKULBN0EUd1fpOD70p
RM0rlp_gg_WNUKOW1V-3keYUJoXH9NztEDm_D2MQXj9eGOJJ8yPgGL8PAZMLe
2R7jb9TxOCPDED7tY_TU4nFPlxptw59A42mldEmViXsKQt60s1SLboazxFKve
qXC_jpLUt22OC6GUG63p-REw-ZOr3r845z50wMuzifQrMI9bQ",
    "kty": "RSA"
  }]}},
  "example_extension_parameter": "example_value"
}
```

3.1.1.1. Client Registration Request Using a Software Statement

In addition to JSON elements, client metadata values MAY also be provided in a software statement, as described in Section 2.3. The authorization server MAY ignore the software statement if it does not support this feature. If the server supports software statements, client metadata values conveyed in the software statement MUST take precedence over those conveyed using plain JSON elements.

Software statements are included in the requesting JSON object using this OPTIONAL member:

`software_statement`

A software statement containing client metadata values about the client software as claims. This is a string value containing the entire signed JWT.

In the following example, some registration parameters are conveyed as claims in a software statement from the example in Section 2.3, while some values specific to the client instance are conveyed as regular parameters (with line wraps within values for display purposes only):

```
POST /register HTTP/1.1
Content-Type: application/json
Accept: application/json
Host: server.example.com

{
  "redirect_uris":[
    "https://client.example.org/callback",
    "https://client.example.org/callback2"
  ],
  "software_statement":"eyJhbGciOiJSUzI1NiJ9.
eyJzb2Z0d2FyZV9pZCI6IjROUkIxLTBYWkFCWkk5RTYtNVNNM1IiLCJjbGll
bnRfbmFtZSI6IktV4YWlwbGUuG3RhZGVtZW50LWJhc2VkdjE5NSaWVudCIsImNs
aWVudF91cmkiOiJodHRwczovL2NsaWVudC5leGFtcGxlLm5ldC8ifQ.
GHfL4QNirQwL18BSRde595T9jbzqa06R9BT8w409x9oIcKaZo_mt15riEXHa
zdISUvDIZhtiyNrSHQ8K4TvqWxH6uJgcmoodZdPwmWRIEYbQDLqPNxREtYn0
5X3AR7ia4FRjQ2ojZjk5fJqJdQ-JcfxyhK-P8BAWBd6I2LLA77IG32xtbhxY
fHX7VhuU5ProJO8uvu3Ayv4XRhLZJY4yKfmyjiiKiPNe-Ia4SMY_d_QSWxsk
U5XIQL5Sa2YRPMbDRXttm2TfnZMlxx70DoYi8g6czz-CPGRI4SW_S2RKHIJf
IjoI3zTJ0Y2oe0_EJAiXbL6OyF9S5tKxDXV8JIndSA",
  "scope":"read write",
  "example_extension_parameter":"example_value"
}
```

3.2. Responses

Upon a successful registration request, the authorization server returns a client identifier for the client. The server responds with an HTTP 201 Created code and a body of type "application/json" with content as described in Section 3.2.1.

Upon an unsuccessful registration request, the authorization server responds with an error, as described in Section 3.2.2.

3.2.1. Client Information Response

The response contains the client identifier as well as the client secret, if the client is a confidential client. The response MAY contain additional fields as specified by extensions to this specification.

client_id

REQUIRED. OAuth 2.0 client identifier string. It SHOULD NOT be currently valid for any other registered client, though an authorization server MAY issue the same client identifier to multiple instances of a registered client at its discretion.

`client_secret`

OPTIONAL. OAuth 2.0 client secret string. If issued, this MUST be unique for each "client_id" and SHOULD be unique for multiple instances of a client using the same "client_id". This value is used by confidential clients to authenticate to the token endpoint as described in OAuth 2.0 [RFC6749] Section 2.3.1.

`client_id_issued_at`

OPTIONAL. Time at which the client identifier was issued. The time is represented as the number of seconds from 1970-01-01T0:0:0Z as measured in UTC until the date/time of issuance.

`client_secret_expires_at`

REQUIRED if "client_secret" is issued. Time at which the client secret will expire or 0 if it will not expire. The time is represented as the number of seconds from 1970-01-01T0:0:0Z as measured in UTC until the date/time of expiration.

Additionally, the authorization server MUST return all registered metadata about this client, including any fields provisioned by the authorization server itself. The authorization server MAY reject or replace any of the client's requested metadata values submitted during the registration and substitute them with suitable values. The client or developer can check the values in the response to determine if the registration is sufficient for use (e.g., the registered "token_endpoint_auth_method" is supported by the client software) and determine a course of action appropriate for the client software. The response to such a situation is out of scope for this specification but could include filing a report with the application developer or authorization server provider, attempted re-registration with different metadata values, or various other methods. For instance, if the server also supports a registration management mechanism such as that defined in [OAuth.Registration.Management], the client or developer could attempt to update the registration with different metadata values. This process could also be aided by a service discovery protocol such as [OpenID.Discovery] which can list a server's capabilities, allowing a client to make a more informed registration request. The use of any such management or discovery system is optional and outside the scope of this specification.

The successful registration response uses an HTTP 201 Created status code with a body of type "application/json" consisting of a single

JSON object [RFC7159] with all parameters as top-level members of the object.

If a software statement was used as part of the registration, its value MUST be returned unmodified in the response along with other metadata using the "software_statement" member name. Client metadata elements used from the software statement MUST also be returned directly as top-level client metadata values in the registration response (possibly with different values, since the values requested and the values used may differ).

Following is a non-normative example response of a successful registration:

```
HTTP/1.1 201 Created
Content-Type: application/json
Cache-Control: no-store
Pragma: no-cache

{
  "client_id": "s6BhdRkqt3",
  "client_secret": "cf136dc3c1fc93f31185e5885805d",
  "client_id_issued_at": 2893256800,
  "client_secret_expires_at": 2893276800,
  "redirect_uris": [
    "https://client.example.org/callback",
    "https://client.example.org/callback2"
  ],
  "grant_types": ["authorization_code", "refresh_token"],
  "client_name": "My Example Client",
  "client_name#ja-Jpan-JP":
    "\u30AF\u30E9\u30A4\u30A2\u30F3\u30C8\u540D",
  "token_endpoint_auth_method": "client_secret_basic",
  "logo_uri": "https://client.example.org/logo.png",
  "jwks_uri": "https://client.example.org/my_public_keys.jwks",
  "example_extension_parameter": "example_value"
}
```

3.2.2. Client Registration Error Response

When an OAuth 2.0 error condition occurs, such as the client presenting an invalid initial access token, the authorization server returns an error response appropriate to the OAuth 2.0 token type.

When a registration error condition occurs, the authorization server returns an HTTP 400 status code (unless otherwise specified) with content type "application/json" consisting of a JSON object [RFC7159] describing the error in the response body.

Two members are defined for inclusion in the JSON object:

`error`

REQUIRED. Single ASCII error code string.

`error_description`

OPTIONAL. Human-readable ASCII text description of the error used for debugging.

Other members MAY also be included, and if not understood, MUST be ignored.

This specification defines the following error codes:

`invalid_redirect_uri`

The value of one or more redirection URIs is invalid.

`invalid_client_metadata`

The value of one of the client metadata fields is invalid and the server has rejected this request. Note that an authorization server MAY choose to substitute a valid value for any requested parameter of a client's metadata.

`invalid_software_statement`

The software statement presented is invalid.

`unapproved_software_statement`

The software statement presented is not approved for use by this authorization server.

Following is a non-normative example of an error response resulting from a redirection URI that has been blacklisted by the authorization server (with line wraps within values for display purposes only):

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Cache-Control: no-store
Pragma: no-cache
```

```
{
  "error": "invalid_redirect_uri",
  "error_description": "The redirection URI
    http://sketchy.example.com is not allowed by this server."
}
```

Following is a non-normative example of an error response resulting from an inconsistent combination of "response_types" and "grant_types" values (with line wraps within values for display purposes only):

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Cache-Control: no-store
Pragma: no-cache
```

```
{
  "error": "invalid_client_metadata",
  "error_description": "The grant type 'authorization_code' must be
    registered along with the response type 'code' but found only
    'implicit' instead."
}
```

4. IANA Considerations

4.1. OAuth Dynamic Client Registration Metadata Registry

This specification establishes the OAuth Dynamic Client Registration Metadata registry.

OAuth registration client metadata names and descriptions are registered with a Specification Required ([RFC5226]) after a two-week review period on the `oauth-ext-review@ietf.org` mailing list, on the advice of one or more Designated Experts. However, to allow for the allocation of names prior to publication, the Designated Expert(s) may approve registration once they are satisfied that such a specification will be published, per [RFC7120].

Registration requests sent to the mailing list for review should use an appropriate subject (e.g., "Request to register OAuth Dynamic Client Registration Metadata name: example").

Within the review period, the Designated Expert(s) will either approve or deny the registration request, communicating this decision to the review list and IANA. Denials should include an explanation and, if applicable, suggestions as to how to make the request successful.

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

4.1.1.1. Registration Template

Client Metadata Name:

The name requested (e.g., "example"). This name is case sensitive. Names that match other registered names in a case insensitive manner SHOULD NOT be accepted.

Client Metadata Description:

Brief description of the metadata value (e.g., "Example description").

Change controller:

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

Specification document(s):

Reference to the document(s) that specify the token endpoint authorization method, preferably including a URI that can be used to retrieve a copy of the document(s). An indication of the relevant sections may also be included but is not required.

4.1.1.2. Initial Registry Contents

The initial contents of the OAuth Dynamic Client Registration Metadata registry are:

- o Client Metadata Name: "redirect_uris"
- o Client Metadata Description: Array of redirection URIs for use in redirect-based flows
- o Change controller: IESG
- o Specification document(s): [[this document]]

- o Client Metadata Name: "token_endpoint_auth_method"
- o Client Metadata Description: Requested authentication method for the token endpoint
- o Change controller: IESG
- o Specification document(s): [[this document]]

- o Client Metadata Name: "grant_types"
- o Client Metadata Description: Array of OAuth 2.0 grant types that the client may use
- o Change controller: IESG
- o Specification document(s): [[this document]]

- o Client Metadata Name: "response_types"
- o Client Metadata Description: Array of the OAuth 2.0 response types that the client may use

- o Change controller: IESG
- o Specification document(s): [[this document]]

- o Client Metadata Name: "client_name"
- o Client Metadata Description: Human-readable name of the client to be presented to the user
- o Change Controller: IESG
- o Specification Document(s): [[this document]]

- o Client Metadata Name: "client_uri"
- o Client Metadata Description: URL of a Web page providing information about the client
- o Change Controller: IESG
- o Specification Document(s): [[this document]]

- o Client Metadata Name: "logo_uri"
- o Client Metadata Description: URL that references a logo for the client
- o Change Controller: IESG
- o Specification Document(s): [[this document]]

- o Client Metadata Name: "scope"
- o Client Metadata Description: Space separated list of OAuth 2.0 scope values
- o Change Controller: IESG
- o Specification Document(s): [[this document]]

- o Client Metadata Name: "contacts"
- o Client Metadata Description: Array of strings representing ways to contact people responsible for this client, typically email addresses
- o Change Controller: IESG
- o Specification document(s): [[this document]]

- o Client Metadata Name: "tos_uri"
- o Client Metadata Description: URL that points to a human-readable Terms of Service document for the client
- o Change Controller: IESG
- o Specification Document(s): [[this document]]

- o Client Metadata Name: "policy_uri"
- o Client Metadata Description: URL that points to a human-readable Policy document for the client
- o Change Controller: IESG
- o Specification Document(s): [[this document]]

- o Client Metadata Name: "jwks_uri"

- o Client Metadata Description: URL referencing the client's JSON Web Key Set [RFC7517] document representing the client's public keys
- o Change Controller: IESG
- o Specification Document(s): [[this document]]

- o Client Metadata Name: "jwks"
- o Client Metadata Description: Client's JSON Web Key Set [RFC7517] document representing the client's public keys
- o Change Controller: IESG
- o Specification Document(s): [[this document]]

- o Client Metadata Name: "software_id"
- o Client Metadata Description: Identifier for the software that comprises a client
- o Change Controller: IESG
- o Specification Document(s): [[this document]]

- o Client Metadata Name: "software_version"
- o Client Metadata Description: Version identifier for the software that comprises a client
- o Change Controller: IESG
- o Specification Document(s): [[this document]]

- o Client Metadata Name: "client_id"
- o Client Metadata Description: Client identifier
- o Change Controller: IESG
- o Specification Document(s): [[this document]]

- o Client Metadata Name: "client_secret"
- o Client Metadata Description: Client secret
- o Change Controller: IESG
- o Specification Document(s): [[this document]]

- o Client Metadata Name: "client_id_issued_at"
- o Client Metadata Description: Time at which the client identifier was issued
- o Change Controller: IESG
- o Specification Document(s): [[this document]]

- o Client Metadata Name: "client_secret_expires_at"
- o Client Metadata Description: Time at which the client secret will expire
- o Change Controller: IESG
- o Specification Document(s): [[this document]]

4.2. OAuth Token Endpoint Authentication Methods Registry

This specification establishes the OAuth Token Endpoint Authentication Methods registry.

Additional values for use as "token_endpoint_auth_method" values are registered with a Specification Required ([RFC5226]) after a two-week review period on the `oauth-ext-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 Expert(s) may approve registration once they are satisfied that such a specification will be published, per [RFC7120].

Registration requests must be sent to the `oauth-ext-review@ietf.org` mailing list for review and comment, with an appropriate subject (e.g., "Request to register token_endpoint_auth_method value: example").

Within the review period, the Designated Expert(s) will either approve or deny the registration request, communicating this decision to the review list and IANA. Denials should include an explanation and, if applicable, suggestions as to how to make the request successful.

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

4.2.1. Registration Template

Token Endpoint Authorization Method Name:

The name requested (e.g., "example"). This name is case sensitive. Names that match other registered names in a case insensitive manner SHOULD NOT be accepted.

Change controller:

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

Specification document(s):

Reference to the document(s) that specify the token endpoint authorization method, preferably including a URI that can be used to retrieve a copy of the document(s). An indication of the relevant sections may also be included but is not required.

4.2.2. Initial Registry Contents

The initial contents of the OAuth Token Endpoint Authentication Methods registry are:

- o Token Endpoint Authorization Method Name: "none"
- o Change controller: IESG
- o Specification document(s): [[this document]]

- o Token Endpoint Authorization Method Name: "client_secret_post"
- o Change controller: IESG
- o Specification document(s): [[this document]]

- o Token Endpoint Authorization Method Name: "client_secret_basic"
- o Change controller: IESG
- o Specification document(s): [[this document]]

5. Security Considerations

Since requests to the client registration endpoint result in the transmission of clear-text credentials (in the HTTP request and response), the authorization server MUST require the use of a transport-layer security mechanism when sending requests to the registration endpoint. The server MUST support TLS 1.2 RFC 5246 [RFC5246] and MAY support additional transport-layer mechanisms meeting its security requirements. When using TLS, the client MUST perform a TLS/SSL server certificate check, per RFC 6125 [RFC6125]. Implementation security considerations can be found in Recommendations for Secure Use of TLS and DTLS [RFC7525].

For clients that use redirect-based grant types such as "authorization_code" and "implicit", authorization servers MUST require clients to register their redirection URI values. This can help mitigate attacks where rogue actors inject and impersonate a validly registered client and intercept its authorization code or tokens through an invalid redirection URI or open redirector. Additionally, in order to prevent hijacking of the return values of the redirection, registered redirection URI values MUST be one of:

- o A remote web site protected by TLS (e.g., `https://client.example.com/oauth_redirect`)
- o A web site hosted on the local machine using an HTTP URI (e.g., `http://localhost:8080/oauth_redirect`)
- o A non-HTTP application-specific URL that is available only to the client application (e.g., `exampleapp://oauth_redirect`)

Public clients MAY register with an authorization server using this protocol, if the authorization server's policy allows them. Public

clients use a "none" value for the "token_endpoint_auth_method" metadata field and are generally used with the "implicit" grant type. Often these clients will be short-lived in-browser applications requesting access to a user's resources and access is tied to a user's active session at the authorization server. Since such clients often do not have long-term storage, it is possible that such clients would need to re-register every time the browser application is loaded. To avoid the resulting proliferation of dead client identifiers, an authorization server MAY decide to expire registrations for existing clients meeting certain criteria after a period of time has elapsed. Alternatively, such clients could be registered on the server where the in-browser application's code is served from, and the client's configuration pushed to the browser along side the code.

Since different OAuth 2.0 grant types have different security and usage parameters, an authorization server MAY require separate registrations for a piece of software to support multiple grant types. For instance, an authorization server might require that all clients using the "authorization_code" grant type make use of a client secret for the "token_endpoint_auth_method", but any clients using the "implicit" grant type do not use any authentication at the token endpoint. In such a situation, a server MAY disallow clients from registering for both the "authorization_code" and "implicit" grant types simultaneously. Similarly, the "authorization_code" grant type is used to represent access on behalf of an end-user, but the "client_credentials" grant type represents access on behalf of the client itself. For security reasons, an authorization server could require that different scopes be used for these different use cases, and as a consequence it MAY disallow these two grant types from being registered together by the same client. In all of these cases, the authorization server would respond with an "invalid_client_metadata" error response.

Unless used as a claim in a software statement, the authorization server MUST treat all client metadata as self-asserted. For instance, a rogue client might use the name and logo of a legitimate client that it is trying to impersonate. Additionally, a rogue client might try to use the software identifier or software version of a legitimate client to attempt to associate itself on the authorization server with instances of the legitimate client. To counteract this, an authorization server MUST take appropriate steps to mitigate this risk by looking at the entire registration request and client configuration. For instance, an authorization server could issue a warning if the domain/site of the logo doesn't match the domain/site of redirection URIs. An authorization server could also refuse registration requests from a known software identifier that is requesting different redirection URIs or a different client

URI. An authorization server can also present warning messages to end-users about dynamically registered clients in all cases, especially if such clients have been recently registered or have not been trusted by any users at the authorization server before.

In a situation where the authorization server is supporting open client registration, it must be extremely careful with any URL provided by the client that will be displayed to the user (e.g. "logo_uri", "tos_uri", "client_uri", and "policy_uri"). For instance, a rogue client could specify a registration request with a reference to a drive-by download in the "policy_uri", enticing the user to click on it during the authorization. The authorization server SHOULD check to see if the "logo_uri", "tos_uri", "client_uri", and "policy_uri" have the same host and scheme as the those defined in the array of "redirect_uris" and that all of these URIs resolve to valid web pages. Since these URI values that are intended to be displayed to the user at the authorization page, the authorization server SHOULD protect the user from malicious content hosted at the URLs where possible. For instance, before presenting the URLs to the user at the authorization page, the authorization server could download the content hosted at the URLs, check the content against a malware scanner and blacklist filter, determine whether or not there is mixed secure and non-secure content at the URL, and other possible server-side mitigations. Note that the content in these URLs can change at any time and the authorization server cannot provide complete confidence in the safety of the URLs, but these practices could help. To further mitigate this kind of threat, the authorization server can also warn the user that the URL links have been provided by a third party, should be treated with caution, and are not hosted by the authorization server itself. For instance, instead of providing the links directly in an HTML anchor, the authorization server can direct the user to an interstitial warning page before allowing the user to continue to the target URL.

Clients MAY use both the direct JSON object and the JWT-encoded software statement to present client metadata to the authorization server as part of the registration request. A software statement is cryptographically protected and represents claims made by the issuer of the statement, while the JSON object represents the self-asserted claims made by the client or developer directly. If the software statement is valid and signed by an acceptable authority (such as the software API publisher), the values of client metadata within the software statement MUST take precedence over those metadata values presented in the plain JSON object, which could have been intercepted and modified.

Like all metadata values, the software statement is an item that is self-asserted by the client, even though its contents have been

digitally signed or MACed by the issuer of the software statement. As such, presentation of the software statement is not sufficient in most cases to fully identify a piece of client software. An initial access token, in contrast, does not necessarily contain information about a particular piece of client software but instead represents authorization to use the registration endpoint. An authorization server MUST consider the full registration request, including the software statement, initial access token, and JSON client metadata values, when deciding whether to honor a given registration request.

If an authorization server receives a registration request for a client that is not intended to have multiple instances registered simultaneously and the authorization server can infer a duplication of registration (e.g., it uses the same "software_id" and "software_version" values as another existing client), the server SHOULD treat the new registration as being suspect and reject the registration. It is possible that the new client is trying to impersonate the existing client in order to trick users into authorizing it, or that the original registration is no longer valid. The details of managing this situation are specific to the authorization server deployment and outside the scope of this specification.

Since a client identifier is a public value that can be used to impersonate a client at the authorization endpoint, an authorization server that decides to issue the same client identifier to multiple instances of a registered client needs to be very particular about the circumstances under which this occurs. For instance, the authorization server can limit a given client identifier to clients using the same redirect-based flow and the same redirection URIs. An authorization server SHOULD NOT issue the same client secret to multiple instances of a registered client, even if they are issued the same client identifier, or else the client secret could be leaked, allowing malicious impostors to impersonate a confidential client.

6. Privacy Considerations

As the protocol described in this specification deals almost exclusively with information about software and not about people, there are very few privacy concerns for its use. The notable exception is the "contacts" field as defined in Client Metadata (Section 2), which contains contact information for the developers or other parties responsible for the client software. These values are intended to be displayed to end-users and will be available to the administrators of the authorization server. As such, the developer may wish to provide an email address or other contact information expressly dedicated to the purpose of supporting the client instead

of using their personal or professional addresses. Alternatively, the developer may wish to provide a collective email address for the client to allow for continuing contact and support of the client software after the developer moves on and someone else takes over that responsibility.

In general, the metadata for a client, such as the client name and software identifier, are common across all instances of a piece of client software and therefore pose no privacy issues for end-users. Client identifiers, on the other hand, are often unique to a specific instance of a client. For clients such as web sites that are used by many users, there may not be significant privacy concerns regarding the client identifier, but for clients such as native applications that are installed on a single end-user's device, the client identifier could be uniquely tracked during OAuth 2.0 transactions and its use tied to that single end-user. However, as the client software still needs to be authorized by a resource owner through an OAuth 2.0 authorization grant, this type of tracking can occur whether or not the client identifier is unique by correlating the authenticated resource owner with the requesting client identifier.

Note that clients are forbidden by this specification from creating their own client identifier. If the client were able to do so, an individual client instance could be tracked across multiple colluding authorization servers, leading to privacy and security issues. Additionally, client identifiers are generally issued uniquely per registration request, even for the same instance of software. In this way, an application could marginally improve privacy by registering multiple times and appearing to be completely separate applications. However, this technique does incur significant usability cost in the form of requiring multiple authorizations per resource owner and is therefore unlikely to be used in practice.

7. References

7.1. Normative References

- [IANA.Language]
Internet Assigned Numbers Authority (IANA), "Language Subtag Registry", <<http://www.iana.org/assignments/language-subtag-registry>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.

- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.
- [RFC5646] Phillips, A. and M. Davis, "Tags for Identifying Languages", BCP 47, RFC 5646, September 2009.
- [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, March 2011.
- [RFC6749] Hardt, D., "The OAuth 2.0 Authorization Framework", RFC 6749, October 2012.
- [RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, October 2012.
- [RFC7120] Cotton, M., "Early IANA Allocation of Standards Track Code Points", BCP 100, RFC 7120, January 2014.
- [RFC7159] Bray, T., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, March 2014.
- [RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, May 2015.
- [RFC7517] Jones, M., "JSON Web Key (JWK)", RFC 7517, May 2015.
- [RFC7519] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, May 2015.
- [RFC7522] Campbell, B., Mortimore, C., and M. Jones, "Security Assertion Markup Language (SAML) 2.0 Profile for OAuth 2.0 Client Authentication and Authorization Grants", RFC 7522, May 2015.
- [RFC7523] Jones, M., Campbell, B., and C. Mortimore, "JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants", RFC 7523, May 2015.
- [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, May 2015.

7.2. Informative References

[I-D.hardjono-oauth-umacore]

Hardjono, T., Maler, E., Machulak, M., and D. Catalano,
"User-Managed Access (UMA) Profile of OAuth 2.0", draft-
hardjono-oauth-umacore-13 (work in progress), April 2015.

[OAuth.Registration.Management]

Richer, J., Jones, M., Bradley, J., and M. Machulak,
"OAuth 2.0 Dynamic Client Registration Management
Protocol", draft-ietf-oauth-dyn-reg-management (work in
progress), May 2015.

[OpenID.Discovery]

Sakimura, N., Bradley, J., Jones, M., and E. Jay, "OpenID
Connect Discovery 1.0", November 2014.

[OpenID.Registration]

Sakimura, N., Bradley, J., and M. Jones, "OpenID Connect
Dynamic Client Registration 1.0", November 2014.

Appendix A. Use Cases

This appendix describes different ways that this specification can be utilized, including describing some of the choices that may need to be made. Some of the choices are independent and can be used in combination, whereas some of the choices are interrelated.

A.1. Open versus Protected Dynamic Client Registration

A.1.1. Open Dynamic Client Registration

Authorization servers that support open registration allow registrations to be made with no initial access token. This allows all client software to register with the authorization server.

A.1.2. Protected Dynamic Client Registration

Authorization servers that support protected registration require that an initial access token be used when making registration requests. While the method by which a client or developer receives this initial access token and the method by which the authorization server validates this initial access token are out of scope for this specification, a common approach is for the developer to use a manual pre-registration portal at the authorization server that issues an initial access token to the developer.

A.2. Registration Without or With Software Statements

A.2.1. Registration Without a Software Statement

When a software statement is not used in the registration request, the authorization server must be willing to use client metadata values without them being digitally signed or MACed (and thereby attested to) by any authority. (Note that this choice is independent of the Open versus Protected choice, and that an initial access token is another possible form of attestation.)

A.2.2. Registration With a Software Statement

A software statement can be used in a registration request to provide attestation by an authority for a set of client metadata values. This can be useful when the authorization server wants to restrict registration to client software attested to by a set of authorities or when it wants to know that multiple registration requests refer to the same piece of client software.

A.3. Registration by the Client or Developer

A.3.1. Registration by the Client

In some use cases, client software will dynamically register itself with an authorization server to obtain a client identifier and other information needed to interact with the authorization server. In this case, no client identifier for the authorization server is packaged with the client software.

A.3.2. Registration by the Developer

In some cases, the developer (or development software being used by the developer) will pre-register the client software with the authorization server or a set of authorization servers. In this case, the client identifier value(s) for the authorization server(s) can be packaged with the client software.

A.4. Client ID per Client Instance or per Client Software

A.4.1. Client ID per Client Software Instance

In some cases, each deployed instance of a piece of client software will dynamically register and obtain distinct client identifier values. This can be advantageous, for instance, if the code flow is being used, as it also enables each client instance to have its own client secret. This can be useful for native clients, which cannot maintain the secrecy of a client secret value packaged with the

software, but which may be able to maintain the secrecy of a per-instance client secret.

A.4.2. Client ID Shared Among All Instances of Client Software

In some cases, each deployed instance of a piece of client software will share a common client identifier value. For instance, this is often the case for in-browser clients using the implicit flow, when no client secret is involved. Particular authorization servers might choose, for instance, to maintain a mapping between software statement values and client identifier values, and return the same client identifier value for all registration requests for a particular piece of software. The circumstances under which an authorization server would do so, and the specific software statement characteristics required in this case, are beyond the scope of this specification.

A.5. Stateful or Stateless Registration

A.5.1. Stateful Client Registration

In some cases, authorization servers will maintain state about registered clients, typically indexing this state using the client identifier value. This state would typically include the client metadata values associated with the client registration, and possibly other state specific to the authorization server's implementation. When stateful registration is used, operations to support retrieving and/or updating this state may be supported. One possible set of operations upon stateful registrations is described in the [OAuth.Registration.Management] specification.

A.5.2. Stateless Client Registration

In some cases, authorization servers will be implemented in a manner that enables them to not maintain any local state about registered clients. One means of doing this is to encode all the registration state in the returned client identifier value, and possibly encrypting the state to the authorization server to maintain the confidentiality and integrity of the state.

Appendix B. Acknowledgments

The authors thank the OAuth Working Group, the User-Managed Access Working Group, and the OpenID Connect Working Group participants for their input to this document. In particular, the following individuals have been instrumental in their review and contribution to various versions of this document: Amanda Anganes, Derek Atkins, Tim Bray, Domenico Catalano, Donald Coffin, Vladimir Dzhuvinov,

George Fletcher, Thomas Hardjono, Phil Hunt, William Kim, Torsten Lodderstedt, Eve Maler, Josh Mandel, Nov Mataka, Tony Nadalin, Nat Sakimura, Christian Scholz, and Hannes Tschofenig.

Appendix C. Document History

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

-30

- o Updated JOSE, JWT, and OAuth Assertion draft references to final RFC numbers.

-29

- o Described more possible client responses to the metadata fields returned by the server being different than those requested.
- o Added RFC 7120/BCP 100 references.
- o Added RFC 7525/BCP 195 reference to replace draft reference.

-28

- o Clarified all client metadata as JSON arrays, strings, or numbers.
- o Expanded security considerations advice around external URLs.
- o Added text to say what happens if the client doesn't get back the registration it expected in the response.
- o Added more explicit references to HTTP 201 response from registration.
- o Clarified client version definition.
- o Removed spurious reference to "delete action".
- o Fixed intended normative and non-normative language in several sections.
- o Stated what a server should do if a suspected duplicate client tries to register.

-27

- o Changed a registry name missed in -26.

-26

- o Used consistent registry name.

-25

- o Updated author information.
- o Clarified registry contents.
- o Added forward pointer to IANA from metadata section.

- o Clarified how to silently ignore errors.
- o Reformatted diagram text.

-24

- o Clarified some party definitions.
- o Clarified the opaqueness of `software_id` and `software_statement`.
- o Created a forward pointer to the Security Considerations section for TLS requirements on the registration endpoint.
- o Added a forward pointer to the Privacy Considerations section for the contacts field.
- o Wrote privacy considerations about `client_id` tracking.

-23

- o Updated author information.

-22

- o Reorganized registration response sections.
- o Addressed shepherd comments.
- o Added concrete JWK set to example.

-21

- o Applied minor editorial fixes.
- o Added software statement examples.
- o Moved software statement request details to sub-section.
- o Clarified that a server MAY ignore the software statement (just as it MAY ignore other metadata values).
- o Removed TLS 1.0.
- o Added privacy considerations around "contacts" field.
- o Marked `software_id` as RECOMMENDED inside of a software statement.

-20

- o Applied minor editorial fixes from working group comments.

-19

- o Added informative references to the OpenID Connect Dynamic Client Registration and UMA specifications in the introduction.
- o Clarified the "jwks" and "jwks_uri" descriptions and included an example situation in which they might be used.
- o Removed "application_type".
- o Added redirection URI usage restrictions to the Security Considerations section, based on the client type.
- o Expanded the "tos_uri" and "policy_uri" descriptions.

-18

- o Corrected an example HTTP response status code to be 201 Created.
- o Said more about who issues and uses initial access tokens and software statements.
- o Stated that the use of an initial access token is required when the authorization server limits the parties that can register a client.
- o Stated that the implementation and use of all client metadata fields is OPTIONAL, other than "redirect_uris", which MUST be used for redirect-based flows and implemented to fulfill the requirement in Section 2 of OAuth 2.0.
- o Added the "application_type" metadata value, which had somehow been omitted.
- o Added missing default metadata values, which had somehow been omitted.
- o Clarified that the "software_id" is ultimately asserted by the client developer.
- o Clarified that the "error" member is required in error responses, "error_description" member is optional, and other members may be present.
- o Added security consideration about registrations with duplicate "software_id" and "software_version" values.

-17

- o Merged draft-ietf-oauth-dyn-reg-metadata back into this document.
- o Removed "Core" from the document title.
- o Explicitly state that all metadata members are optional.
- o Clarified language around software statements for use in registration context.
- o Clarified that software statements need to be digitally signed or MACed.
- o Added a "jwks" metadata parameter to parallel the "jwks_uri" parameter.
- o Removed normative language from terminology.
- o Expanded abstract and introduction.
- o Addressed review comments from several working group members.

-16

- o Replaced references to draft-jones-oauth-dyn-reg-metadata and draft-jones-oauth-dyn-reg-management with draft-ietf-oauth-dyn-reg-metadata and draft-ietf-oauth-dyn-reg-management.
- o Addressed review comments by Phil Hunt and Tony Nadalin.

-15

- o Partitioned the Dynamic Client Registration specification into core, metadata, and management specifications. This built on work first published as draft-richer-oauth-dyn-reg-core-00 and draft-richer-oauth-dyn-reg-management-00.
- o Added the ability to use Software Statements. This built on work first published as draft-hunt-oauth-software-statement-00 and draft-hunt-oauth-client-association-00.
- o Created the IANA OAuth Registration Client Metadata registry for registering Client Metadata values.
- o Defined Client Instance term and stated that multiple instances can use the same client identifier value under certain circumstances.
- o Rewrote the introduction.
- o Rewrote the Use Cases appendix.

-14

- o Added software_id and software_version metadata fields
- o Added direct references to RFC6750 errors in read/update/delete methods

-13

- o Fixed broken example text in registration request and in delete request
- o Added security discussion of separating clients of different grant types
- o Fixed error reference to point to RFC6750 instead of RFC6749
- o Clarified that servers must respond to all requests to configuration endpoint, even if it's just an error code
- o Lowercased all Terms to conform to style used in RFC6750

-12

- o Improved definition of Initial Access Token
- o Changed developer registration scenario to have the Initial Access Token gotten through a normal OAuth 2.0 flow
- o Moved non-normative client lifecycle examples to appendix
- o Marked differentiating between auth servers as out of scope
- o Added protocol flow diagram
- o Added credential rotation discussion
- o Called out Client Registration Endpoint as an OAuth 2.0 Protected Resource
- o Cleaned up several pieces of text

-11

- o Added localized text to registration request and response examples.
- o Removed "client_secret_jwt" and "private_key_jwt".
- o Clarified "tos_uri" and "policy_uri" definitions.
- o Added the OAuth Token Endpoint Authentication Methods registry for registering "token_endpoint_auth_method" metadata values.
- o Removed uses of non-ASCII characters, per RFC formatting rules.
- o Changed "expires_at" to "client_secret_expires_at" and "issued_at" to "client_id_issued_at" for greater clarity.
- o Added explanatory text for different credentials (Initial Access Token, Registration Access Token, Client Credentials) and what they're used for.
- o Added Client Lifecycle discussion and examples.
- o Defined Initial Access Token in Terminology section.

-10

- o Added language to point out that scope values are service-specific
- o Clarified normative language around client metadata
- o Added extensibility to token_endpoint_auth_method using absolute URIs
- o Added security consideration about registering redirect URIs
- o Changed erroneous 403 responses to 401's with notes about token handling
- o Added example for initial registration credential

-09

- o Added method of internationalization for Client Metadata values
- o Fixed SAML reference

-08

- o Collapsed jwk_uri, jwk_encryption_uri, x509_uri, and x509_encryption_uri into a single jwks_uri parameter
- o Renamed grant_type to grant_types since it's a plural value
- o Formalized name of "OAuth 2.0" throughout document
- o Added JWT Bearer Assertion and SAML 2 Bearer Assertion to example grant types
- o Added response_types parameter and explanatory text on its use with and relationship to grant_types

-07

- o Changed registration_access_url to registration_client_uri
- o Fixed missing text in 5.1
- o Added Pragma: no-cache to examples
- o Changed "no such client" error to 403

- o Renamed Client Registration Access Endpoint to Client Configuration Endpoint
- o Changed all the parameter names containing "_url" to instead use "_uri"
- o Updated example text for forming Client Configuration Endpoint URL

-06

- o Removed secret_rotation as a client-initiated action, including removing client secret rotation endpoint and parameters.
- o Changed _links structure to single value registration_access_url.
- o Collapsed create/update/read responses into client info response.
- o Changed return code of create action to 201.
- o Added section to describe suggested generation and composition of Client Registration Access URL.
- o Added clarifying text to PUT and POST requests to specify JSON in the body.
- o Added Editor's Note to DELETE operation about its inclusion.
- o Added Editor's Note to registration_access_url about alternate syntax proposals.

-05

- o changed redirect_uri and contact to lists instead of space delimited strings
- o removed operation parameter
- o added _links structure
- o made client update management more RESTful
- o split endpoint into three parts
- o changed input to JSON from form-encoded
- o added READ and DELETE operations
- o removed Requirements section
- o changed token_endpoint_auth_type back to token_endpoint_auth_method to match OIDC who changed to match us

-04

- o removed default_acr, too undefined in the general OAuth2 case
- o removed default_max_auth_age, since there's no mechanism for supplying a non-default max_auth_age in OAuth2
- o clarified signing and encryption URLs
- o changed token_endpoint_auth_method to token_endpoint_auth_type to match OIDC

-03

- o added scope and grant_type claims
- o fixed various typos and changed wording for better clarity

- o endpoint now returns the full set of client information
- o operations on client_update allow for three actions on metadata:
leave existing value, clear existing value, replace existing value
with new value

-02

- o Reorganized contributors and references
- o Moved OAuth references to RFC
- o Reorganized model/protocol sections for clarity
- o Changed terminology to "client register" instead of "client
associate"
- o Specified that client_id must match across all subsequent requests
- o Fixed RFC2XML formatting, especially on lists

-01

- o Merged UMA and OpenID Connect registrations into a single document
- o Changed to form-parameter inputs to endpoint
- o Removed pull-based registration

-00

- o Imported original UMA draft specification

Authors' Addresses

Justin Richer (editor)

Email: ietf@justin.richer.org

Michael B. Jones
Microsoft

Email: mbj@microsoft.com

URI: <http://self-issued.info/>

John Bradley
Ping Identity

Email: ve7jtb@ve7jtb.com

Maciej Machulak
Newcastle University

Email: maciej.machulak@gmail.com

Phil Hunt
Oracle Corporation

Email: phil.hunt@yahoo.com

OAuth Working Group
Internet-Draft
Intended status: Experimental
Expires: November 6, 2015

J. Richer, Ed.

M. Jones
Microsoft
J. Bradley
Ping Identity
M. Machulak
Newcastle University
May 5, 2015

OAuth 2.0 Dynamic Client Registration Management Protocol
draft-ietf-oauth-dyn-reg-management-15

Abstract

This specification defines methods for management of dynamic OAuth 2.0 client registrations for use cases in which the properties of a registered client may need to be changed during the lifetime of the client. Not all authorization servers supporting dynamic client registration will support these management methods.

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 November 6, 2015.

Copyright Notice

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

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents

carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
1.1. Notational Conventions	3
1.2. Terminology	3
1.3. Protocol Flow	3
2. Client Configuration Endpoint	5
2.1. Client Read Request	6
2.2. Client Update Request	6
2.3. Client Delete Request	9
3. Client Information Response	10
4. IANA Considerations	11
5. Security Considerations	12
6. Privacy Considerations	13
7. Normative References	13
Appendix A. Acknowledgments	14
Appendix B. Registration Tokens and Client Credentials	14
B.1. Credential Rotation	15
Appendix C. Forming the Client Configuration Endpoint URL	15
Appendix D. Document History	16
Authors' Addresses	18

1. Introduction

In order for an OAuth 2.0 client to utilize an OAuth 2.0 authorization server, the client needs specific information to interact with the server, including an OAuth 2.0 client identifier to use with that server. The OAuth 2.0 Dynamic Client Registration Protocol [OAuth.Registration] specification describes how an OAuth 2.0 client can be dynamically registered with an authorization server to obtain this information and how metadata about the client can be registered with the server.

This specification extends the core registration specification by defining a set of methods for management of dynamic OAuth 2.0 client registrations beyond those defined in the core registration specification. In some situations, the registered metadata of a client can change over time, either by modification at the authorization server or by a change in the client software itself. This specification provides methods for the current registration state of a client to be queried at the authorization server, methods for the registration of a client to be updated at the authorization

server, and methods for the client to be unregistered from the authorization server.

This experimental draft is intended to encourage development and deployment of interoperable solutions with the intent that feedback from this experience will inform a future standard.

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 [RFC2119].

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

1.2. Terminology

This specification uses the terms "access token", "authorization code", "authorization endpoint", "authorization grant", "authorization server", "client", "client identifier", "client secret", "grant type", "protected resource", "redirection URI", "refresh token", "resource owner", "resource server", "response type", and "token endpoint" defined by OAuth 2.0 [RFC6749] and the terms defined by the OAuth 2.0 Client Dynamic Registration Protocol [OAuth.Registration].

This specification defines the following terms:

Client Configuration Endpoint

OAuth 2.0 endpoint through which registration information for a registered client can be managed. This URL for this endpoint is returned by the authorization server in the client information response.

Registration Access Token

OAuth 2.0 bearer token issued by the authorization server through the client registration endpoint that is used to authenticate the caller when accessing the client's registration information at the client configuration endpoint. This access token is associated with a particular registered client.

1.3. Protocol Flow

This extends the flow in the OAuth 2.0 Dynamic Client Registration Protocol [OAuth.Registration] specification as follows:

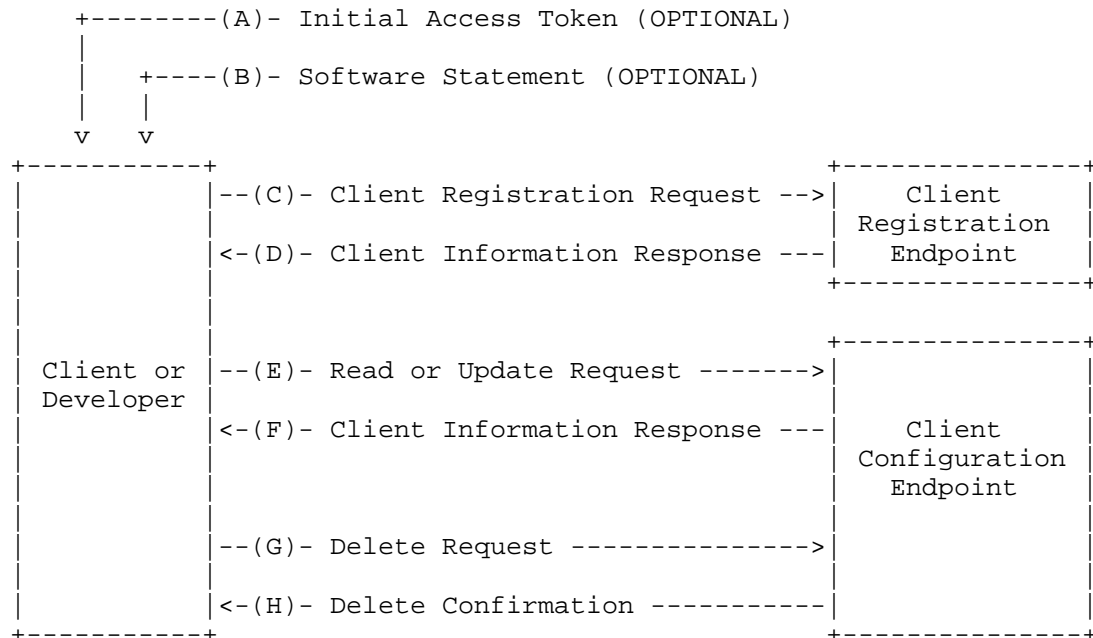


Figure 1: Abstract Extended Dynamic Client Registration Flow

The abstract OAuth 2.0 client dynamic registration flow illustrated in Figure 1 describes the interaction between the client or developer and the endpoints defined in this specification and its parent. This figure does not demonstrate error conditions. This flow includes the following steps:

- (A) Optionally, the client or developer is issued an initial access token for use with the client registration endpoint. The method by which the initial access token is issued to the client or developer is out of scope for this specification.
- (B) Optionally, the client or developer is issued a software statement for use with the client registration endpoint. The method by which the software statement is issued to the client or developer is out of scope for this specification.
- (C) The client or developer calls the client registration endpoint with its desired registration metadata, optionally including the initial access token from (A) if one is required by the authorization server.

- (D) The authorization server registers the client and returns:
- * the client's registered metadata,
 - * a client identifier that is unique to the server,
 - * a set of client credentials such as a client secret, if applicable for this client,
 - * a URI pointing to the client configuration endpoint, and
 - * a registration access token to be used when calling the client configuration endpoint.
- (E) The client or developer optionally calls the client configuration endpoint with a read or update request using the registration access token issued in (D). An update request contains all of the client's registered metadata.
- (F) The authorization server responds with the client's current configuration, potentially including a new registration access token and a new set of client credentials such as a client secret if applicable for this client. If a new registration access token is issued, it replaces the token issued in (D) for all subsequent calls to the client configuration endpoint.
- (G) The client or developer optionally calls the client configuration endpoint with a delete request using the registration access token issued in (D) or (F).
- (H) The authorization server deprovisions the client and responds with a confirmation that the deletion has taken place.

2. Client Configuration Endpoint

The client configuration endpoint is an OAuth 2.0 protected resource that is provisioned by the server to facilitate viewing, updating, and deleting a client's registered information. The location of this endpoint is communicated to the client through the "registration_client_uri" member of the client information response, as specified in Section 3. The client MUST use its registration access token in all calls to this endpoint as an OAuth 2.0 Bearer Token [RFC6750].

The client configuration endpoint MUST be protected by a transport-layer security mechanism, as described in Section 5.

Operations on this endpoint are switched through the use of different HTTP methods [RFC7231]. If an authorization server does not support a particular method on the client configuration endpoint, it MUST respond with the appropriate error code.

2.1. Client Read Request

To read the current configuration of the client on the authorization server, the client makes an HTTP GET request to the client configuration endpoint, authenticating with its registration access token.

Following is a non-normative example request (with line wraps for display purposes only):

```
GET /register/s6BhdRkqt3 HTTP/1.1
Accept: application/json
Host: server.example.com
Authorization: Bearer reg-23410913-abewfq.123483
```

Upon successful read of the information for a currently active client, the authorization server responds with an HTTP 200 OK with content type of "application/json" and a payload, as described in Section 3. Some values in the response, including the "client_secret" and "registration_access_token", MAY be different from those in the initial registration response. If the authorization server includes a new client secret and/or registration access token in its response, the client MUST immediately discard its previous client secret and/or registration access token. The value of the "client_id" MUST NOT change from the initial registration response.

If the registration access token used to make this request is not valid, the server MUST respond with an error as described in OAuth Bearer Token Usage [RFC6750].

If the client does not exist on this server, the server MUST respond with HTTP 401 Unauthorized and the registration access token used to make this request SHOULD be immediately revoked.

If the client does not have permission to read its record, the server MUST return an HTTP 403 Forbidden.

2.2. Client Update Request

To update previously-registered client's registration with an authorization server, the client makes an HTTP PUT request to the client configuration endpoint with a content type of "application/

json". The HTTP entity payload is a JSON [RFC7159] document consisting of a JSON object and all parameters as top-level members of that JSON object. This request is authenticated by the registration access token issued to the client.

This request MUST include all client metadata fields as returned to the client from a previous registration, read, or update operation. The updated client metadata fields request MUST NOT include the "registration_access_token", "registration_client_uri", "client_secret_expires_at", or "client_id_issued_at" fields described in Section 3.

Valid values of client metadata fields in this request MUST replace, not augment, the values previously associated with this client. Omitted fields MUST be treated as null or empty values by the server, indicating the client's request to delete them from the client's registration. The authorization server MAY ignore any null or empty value in the request just as any other value.

The client MUST include its "client_id" field in the request, and it MUST be the same as its currently-issued client identifier. If the client includes the "client_secret" field in the request, the value of this field MUST match the currently-issued client secret for that client. The client MUST NOT be allowed to overwrite its existing client secret with its own chosen value.

For all metadata fields, the authorization server MAY replace any invalid values with suitable default values, and it MUST return any such fields to the client in the response.

For example, a client could send the following request to the client registration endpoint to update the client registration in the above example with new information:

Following is a non-normative example request (with line wraps for display purposes only):

```
PUT /register/s6BhdRkqt3 HTTP/1.1
Accept: application/json
Host: server.example.com
Authorization: Bearer reg-23410913-abewfq.123483

{
  "client_id": "s6BhdRkqt3",
  "client_secret": "cf136dc3c1fc93f31185e5885805d",
  "redirect_uris": [
    "https://client.example.org/callback",
    "https://client.example.org/alt" ],
  "grant_types": [ "authorization_code", "refresh_token" ],
  "token_endpoint_auth_method": "client_secret_basic",
  "jwks_uri": "https://client.example.org/my_public_keys.jwks",
  "client_name": "My New Example",
  "client_name#fr": "Mon Nouvel Exemple",
  "logo_uri": "https://client.example.org/newlogo.png",
  "logo_uri#fr": "https://client.example.org/fr/newlogo.png"
}
```

This example uses client metadata values defined in [OAuth.Registration].

Upon successful update, the authorization server responds with an HTTP 200 OK Message with content type "application/json" and a payload, as described in Section 3. Some values in the response, including the "client_secret" and "registration_access_token", MAY be different from those in the initial registration response. If the authorization server includes a new client secret and/or registration access token in its response, the client MUST immediately discard its previous client secret and/or registration access token. The value of the "client_id" MUST NOT change from the initial registration response.

If the registration access token used to make this request is not valid, the server MUST respond with an error as described in OAuth Bearer Token Usage [RFC6750].

If the client does not exist on this server, the server MUST respond with HTTP 401 Unauthorized, and the registration access token used to make this request SHOULD be immediately revoked.

If the client is not allowed to update its records, the server MUST respond with HTTP 403 Forbidden.

If the client attempts to set an invalid metadata field and the authorization server does not set a default value, the authorization server responds with an error as described in [OAuth.Registration].

2.3. Client Delete Request

To deprovision itself on the authorization server, the client makes an HTTP DELETE request to the client configuration endpoint. This request is authenticated by the registration access token issued to the client as described in [RFC6749].

Following is a non-normative example request (with line wraps for display purposes only):

```
DELETE /register/s6BhdRkqt3 HTTP/1.1
Host: server.example.com
Authorization: Bearer reg-23410913-abewfq.123483
```

A successful delete action will invalidate the "client_id", "client_secret", and "registration_access_token" for this client, thereby preventing the "client_id" from being used at either the authorization endpoint or token endpoint of the authorization server. If possible, the authorization server SHOULD immediately invalidate all existing authorization grants and currently-active access tokens, refresh tokens, and other tokens associated with this client.

If a client has been successfully deprovisioned, the authorization server MUST respond with an HTTP 204 No Content message.

If the server does not support the delete method, the server MUST respond with an HTTP 405 Not Supported.

If the registration access token used to make this request is not valid, the server MUST respond with an error as described in OAuth Bearer Token Usage [RFC6750].

If the client does not exist on this server, the server MUST respond with HTTP 401 Unauthorized and the registration access token used to make this request SHOULD be immediately revoked, if possible.

If the client is not allowed to delete itself, the server MUST respond with HTTP 403 Forbidden.

Following is a non-normative example response:

```
HTTP/1.1 204 No Content
Cache-Control: no-store
Pragma: no-cache
```

3. Client Information Response

This specification extends the client information response defined in OAuth 2.0 Client Dynamic Registration [OAuth.Registration], which states that the response contains the client identifier (as well as the client secret if the client is a confidential client). When used with this specification, the client information response also contains the fully qualified URL of the client configuration endpoint (Section 2) for this specific client that the client or developer may use to manage the client's registration configuration, as well as a registration access token that is to be used by the client or developer to perform subsequent operations at the client configuration endpoint.

`registration_access_token`

REQUIRED. Access token string used at the client configuration endpoint to perform subsequent operations upon the client registration.

`registration_client_uri`

REQUIRED. Fully qualified URL string of the client configuration endpoint for this client.

Additionally, the authorization server MUST return all registered metadata about this client, including any fields provisioned by the authorization server itself. The authorization server MAY reject or replace any of the client's requested metadata values submitted during the registration or update requests and substitute them with suitable values.

The response is an "application/json" document with all parameters as top-level members of a JSON object [RFC7159].

Following is a non-normative example response:

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

{
  "registration_access_token": "reg-23410913-abewfq.123483",
  "registration_client_uri":
    "https://server.example.com/register/s6BhdRkqt3",
  "client_id": "s6BhdRkqt3",
  "client_secret": "cf136dc3c1fc93f31185e5885805d",
  "client_id_issued_at": 2893256800,
  "client_secret_expires_at": 2893276800,
  "client_name": "My Example Client",
  "client_name#ja-Jpan-JP":
    "\u30AF\u30E9\u30A4\u30A2\u30F3\u30C8\u540D",
  "redirect_uris": [
    "https://client.example.org/callback",
    "https://client.example.org/callback2" ],
  "grant_types": [ "authorization_code", "refresh_token" ],
  "token_endpoint_auth_method": "client_secret_basic",
  "logo_uri": "https://client.example.org/logo.png",
  "jwks_uri": "https://client.example.org/my_public_keys.jwks"
}
```

4. IANA Considerations

This specification registers the following client metadata names and descriptions in the OAuth Dynamic Client Registration Metadata registry established by [OAuth.Registration]:

- o Client Metadata Name: "registration_access_token"
- o Client Metadata Description: OAuth 2.0 bearer token used to access the client configuration endpoint
- o Change controller: IESG
- o Specification document(s): [[this document]]
- o Client Metadata Name: "registration_client_uri"
- o Client Metadata Description: Fully qualified URI of the client registration endpoint
- o Change controller: IESG

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

5. Security Considerations

While the client secret can expire, the registration access token SHOULD NOT expire while a client is still actively registered. If this token were to expire, a developer or client could be left in a situation where they have no means of retrieving, updating, or deleting the client's registration information. Were that the case, a new registration would be required, thereby generating a new client identifier. However, to limit the exposure surface of the registration access token, the registration access token MAY be rotated when the developer or client does a read or update operation on the client's client configuration endpoint. As the registration access tokens are relatively long-term credentials, and since the registration access token is a Bearer token and acts as the sole authentication for use at the client configuration endpoint, it MUST be protected by the developer or client as described in OAuth 2.0 Bearer Token Usage [RFC6750].

Since requests to the client configuration endpoint result in the transmission of clear-text credentials (in the HTTP request and response), the authorization server MUST require the use of a transport-layer security mechanism when sending requests to the endpoint. The server MUST support TLS 1.2 RFC 5246 [RFC5246] and MAY support additional transport-layer mechanisms meeting its security requirements. When using TLS, the client MUST perform a TLS/SSL server certificate check, per RFC 6125 [RFC6125]. Implementation security considerations can be found in Recommendations for Secure Use of TLS and DTLS [RFC7525].

Since possession of the registration access token authorizes the holder to potentially read, modify, or delete a client's registration (including its credentials such as a client_secret), the registration access token MUST contain sufficient entropy to prevent a random guessing attack of this token, such as described in [RFC6750] Section 5.2 and [RFC6819] Section 5.1.4.2.2.

If a client is deprovisioned from a server, any outstanding registration access token for that client MUST be invalidated at the same time. Otherwise, this can lead to an inconsistent state wherein a client could make requests to the client configuration endpoint where the authentication would succeed but the action would fail because the client is no longer valid. The authorization server MUST treat all such requests as if the registration access token was invalid by returning an HTTP 401 Unauthorized error, as described.

6. Privacy Considerations

This specification poses no additional privacy considerations beyond those described in the core OAuth 2.0 Dynamic Client Registration [OAuth.Registration] specification.

7. Normative References

[OAuth.Registration]

Richer, J., Ed., Jones, M., Bradley, J., Machulak, M., and P. Hunt, "OAuth 2.0 Dynamic Client Registration Protocol", draft-ietf-oauth-dyn-reg (work in progress), May 2015.

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

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

[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, March 2011.

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

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

[RFC6819] Lodderstedt, T., McGloin, M., and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations", RFC 6819, January 2013.

[RFC7159] Bray, T., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, March 2014.

[RFC7231] Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, June 2014.

[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, May 2015.

Appendix A. Acknowledgments

The authors thank the OAuth Working Group, the User-Managed Access Working Group, and the OpenID Connect Working Group participants for their input to this document. In particular, the following individuals have been instrumental in their review and contribution to various versions of this document: Amanda Anganes, Derek Atkins, Tim Bray, Domenico Catalano, Donald Coffin, Vladimir Dzhuvinov, George Fletcher, Thomas Hardjono, Phil Hunt, William Kim, Torsten Lodderstedt, Eve Maler, Josh Mandel, Nov Mataka, Tony Nadalin, Nat Sakimura, Christian Scholz, and Hannes Tschofenig.

Appendix B. Registration Tokens and Client Credentials

Throughout the course of the dynamic registration protocol, there are three different classes of credentials in play, each with different properties and targets.

- o The initial access token is optionally used by the client or developer at the registration endpoint. This is an OAuth 2.0 token that is used to authorize the initial client registration request. The content, structure, generation, and validation of this token are out of scope for this specification. The authorization server can use this token to verify that the presenter is allowed to dynamically register new clients. This token may be shared among multiple instances of a client to allow them to each register separately, thereby letting the authorization server use this token to tie multiple instances of registered clients (each with their own distinct client identifier) back to the party to whom the initial access token was issued, usually an application developer. This token is usually intended to be used only at the client registration endpoint.
- o The registration access token is used by the client or developer at the client configuration endpoint and represents the holder's authorization to manage the registration of a client. This is an OAuth 2.0 bearer token that is issued from the client registration endpoint in response to a client registration request and is returned in a client information response. The registration access token is uniquely bound to the client identifier and is required to be presented with all calls to the client configuration endpoint. The registration access token should be protected as described in [RFC6750] and should not be shared between instances of a client. If a registration access token is shared between client instances, one instance could change or delete registration values for all other instances of the client. The registration access token can be rotated through the use of the client read or update method on the client configuration

endpoint. The registration access token is intended to be used only at the client configuration endpoint.

- o The client credentials (such as "client_secret") are optional depending on the type of client and are used to retrieve OAuth tokens. Client credentials are most often bound to particular instances of a client and should not be shared between instances. Note that since not all types of clients have client credentials, they cannot be used to manage client registrations at the client configuration endpoint. The client credentials can be rotated through the use of the client read or update method on the client configuration endpoint. The client credentials are intended to be used only at the token endpoint.

B.1. Credential Rotation

The authorization server may be configured to issue new registration access tokens and/or client credentials (such as a "client_secret") throughout the lifetime of the client. This may help minimize the impact of exposed credentials. The authorization server conveys new registration access tokens and client credentials (if applicable) to the client in the client information response of either a read or update request to the client configuration endpoint. The client's current registration access token and client credentials (if applicable) MUST be included in the client information response.

The registration access token SHOULD be rotated only in response to a read or update request to the client configuration endpoint, at which point the new registration access token is returned to the client and the old registration access token MUST be discarded by the client and SHOULD be discarded by the server, if possible. If instead the registration access token were to expire or be invalidated outside of such requests, the client or developer might be locked out of managing the client's configuration.

Note that the authorization server decides the frequency of the credential rotation and not the client. Methods by which the client can request credential rotation are outside the scope of this document.

Appendix C. Forming the Client Configuration Endpoint URL

The authorization server MUST provide the client with the fully qualified URL in the "registration_client_uri" element of the Client Information Response, as specified in Section 3. The authorization server MUST NOT expect the client to construct or discover this URL on its own. The client MUST use the URL as given by the server and MUST NOT construct this URL from component pieces.

Depending on deployment characteristics, the client configuration endpoint URL may take any number of forms. It is RECOMMENDED that this endpoint URL be formed through the use of a server-constructed URL string which combines the client registration endpoint's URL and the issued "client_id" for this client, with the latter as either a path parameter or a query parameter. For example, a client with the client identifier "s6BhdRkqt3" could be given a client configuration endpoint URL of "https://server.example.com/register/s6BhdRkqt3" (path parameter) or of "https://server.example.com/register?client_id=s6BhdRkqt3" (query parameter). In both of these cases, the client simply uses the URL as given by the authorization server.

These common patterns can help the server to more easily determine the client to which the request pertains, which MUST be matched against the client to which the registration access token was issued. If desired, the server MAY simply return the client registration endpoint URL as the client configuration endpoint URL and change behavior based on the authentication context provided by the registration access token.

Appendix D. Document History

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

-15

- o Added RFC 7525/BCP 195 reference to replace draft reference.

-14

- o Clarified all client metadata as JSON arrays, strings, or numbers.
- o Clarified experimental nature of the draft.

-13

- o Changed rate-limiting suggestion to a complexity requirement.

-12

- o Used consistent registry name.

-11

- o Fixed a series of nits from Peter Yee's Gen-ART review.

-10

- o Updated author information.
- o Updated TLS information, imported from Dynamic Registration core.
- o Expanded introduction.
- o Reformatted diagram text.
- o Added privacy considerations section.

-09

- o Updated author information.

-08

- o Updated HTTP RFC reference.

-07

- o Editorial clarifications due to document shepherd feedback.

-06

- o Removed TLS 1.0.
- o Moved several explanatory sections to the appendix.
- o Clarified read operations.
- o Added IANA request.

-05

- o Removed Phil Hunt from authors list, per request.
- o Applied various minor editorial changes from working group comments.

-04

- o Incorrect XML uploaded for -03

-03

- o Changed draft to be Experimental instead of Standards Track.

-02

- o Added more context information to the abstract.

-01

- o Addressed issues that arose from last call comments on draft-ietf-oauth-dyn-reg and draft-ietf-oauth-dyn-reg-metadata.

-00

- o Created from draft-jones-oauth-dyn-reg-management-00.

Authors' Addresses

Justin Richer (editor)

Email: ietf@justin.richer.org

Michael B. Jones
Microsoft

Email: mbj@microsoft.com

URI: <http://self-issued.info/>

John Bradley
Ping Identity

Email: ve7jtb@ve7jtb.com

Maciej Machulak
Newcastle University

Email: maciej.machulak@gmail.com

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: January 4, 2016

J. Richer, Ed.
July 3, 2015

OAuth 2.0 Token Introspection
draft-ietf-oauth-introspection-11

Abstract

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

Requirements Language

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

Status of This Memo

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

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

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

This Internet-Draft will expire on January 4, 2016.

Copyright Notice

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

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents

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

Table of Contents

1. Introduction	2
1.1. Notational Conventions	3
1.2. Terminology	3
2. Introspection Endpoint	4
2.1. Introspection Request	4
2.2. Introspection Response	6
2.3. Error Response	8
3. IANA Considerations	9
3.1. OAuth Token Introspection Response Registry	9
3.1.1. Registration Template	9
3.1.2. Initial Registry Contents	10
4. Security Considerations	11
5. Privacy Considerations	14
6. Acknowledgements	14
7. References	14
7.1. Normative References	14
7.2. Informative References	15
Appendix A. Use with Proof of Possession Tokens	15
Appendix B. Document History	16
Author's Address	17

1. Introduction

In OAuth 2.0, the contents of tokens are opaque to clients. This means that the client does not need to know anything about the content or structure of the token itself, if there is any. However, there is still a large amount of metadata that may be attached to a token, such as its current validity, approved scopes, and information about the context in which the token was issued. These pieces of information are often vital to protected resources making authorization decisions based on the tokens being presented. Since OAuth 2.0 [RFC6749] does not define a protocol for the resource server to learn meta-information about a token that it has received from an authorization server, several different approaches have been developed to bridge this gap. These include using structured token formats such as JWT [RFC7519] or proprietary inter-service communication mechanisms (such as shared databases and protected enterprise service buses) that convey token information.

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

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 [RFC2119].

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

1.2. Terminology

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

This specification uses the terms "access token", "authorization endpoint", "authorization grant", "authorization server", "client", "client identifier", "protected resource", "refresh token", "resource owner", "resource server", and "token endpoint" defined by OAuth 2.0 [RFC6749], and the terms "claim names" and "claim values" defined by JSON Web Token (JWT) [RFC7519].

This specification defines the following terms:

Token Introspection

The act of inquiring about the current state of an OAuth 2.0 token through use of the network protocol defined in this document.

Introspection Endpoint

The OAuth 2.0 endpoint through which the token introspection operation is accomplished..

2. Introspection Endpoint

The introspection endpoint is an OAuth 2.0 endpoint that takes a parameter representing an OAuth 2.0 token and returns a JSON [RFC7159] document representing the meta information surrounding the token, including whether this token is currently active. The definition of an active token is dependent upon the authorization server, but this is commonly a token that has been issued by this authorization server, is not expired, has not been revoked, and valid for use at the protected resource making the introspection call.

The introspection endpoint MUST be protected by a transport-layer security mechanism as described in Section 4. The means by which the protected resource discovers the location of the introspection endpoint are outside the scope of this specification.

2.1. Introspection Request

The protected resource calls the introspection endpoint using an HTTP POST [RFC7231] request with parameters sent as "application/x-www-form-urlencoded" data as defined in [W3C.REC-html5-20141028]. The protected resource sends a parameter representing the token along with optional parameters representing additional context that is known by the protected resource to aid the authorization server in its response.

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

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

The introspection endpoint MAY accept other OPTIONAL parameters to provide further context to the query. For instance, an authorization server may desire to know the IP address of the client accessing the protected resource to determine if the correct client is likely to be presenting the token. The definition of this or any other parameters

are outside the scope of this specification, to be defined by service documentation or extensions to this specification. If the authorization server is unable to determine the state of the token without additional information, it SHOULD return an introspection response indicating the token is not active as described in Section 2.2.

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

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

Following is a non-normative example request:

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

token=2YotnFZFEjrlzCsicMWpAA
```

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

Following is a non-normative example request:

```
POST /introspect HTTP/1.1
Host: server.example.com
Accept: application/json
Content-Type: application/x-www-form-urlencoded
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW

token=mF_9.B5f-4.1JqM&token_type_hint=access_token
```

2.2. Introspection Response

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

active

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

scope

OPTIONAL. A JSON string containing a space-separated list of scopes associated with this token, in the format described in section 3.3 of OAuth 2.0 [RFC6749].

client_id

OPTIONAL. Client identifier for the OAuth 2.0 client that requested this token.

username

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

token_type

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

exp

OPTIONAL. Integer timestamp, measured in the number of seconds since January 1 1970 UTC, indicating when this token will expire, as defined in JWT [RFC7519].

iat

OPTIONAL. Integer timestamp, measured in the number of seconds since January 1 1970 UTC, indicating when this token was originally issued, as defined in JWT [RFC7519].

nbf

OPTIONAL. Integer timestamp, measured in the number of seconds since January 1 1970 UTC, indicating when this token is not to be used before, as defined in JWT [RFC7519].

sub

OPTIONAL. Subject of the token, as defined in JWT [RFC7519]. Usually a machine-readable identifier of the resource owner who authorized this token.

aud

OPTIONAL. Service-specific string identifier or list of string identifiers representing the intended audience for this token, as defined in JWT [RFC7519].

iss

OPTIONAL. String representing the issuer of this token, as defined in JWT [RFC7519].

jti

OPTIONAL. String identifier for the token, as defined in JWT [RFC7519].

Specific implementations MAY extend this structure with their own service-specific response names as top-level members of this JSON object. Response names intended to be used across domains MUST be registered in the OAuth Token Introspection Response registry defined in Section 3.1.

The authorization server MAY respond differently to different protected resources making the same request. For instance, an authorization server MAY limit which scopes from a given token are returned for each protected resource to prevent protected resources from learning more about the larger network than is necessary for its operation.

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

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

Following is a non-normative example response:

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

{
  "active": true,
  "client_id": "1238j323ds-23ij4",
  "username": "jdoe",
  "scope": "read write dolphin",
  "sub": "Z503upPC88QrAjx00dis",
  "aud": "https://protected.example.net/resource",
  "iss": "https://server.example.com/",
  "exp": 1419356238,
  "iat": 1419350238,
  "extension_field": "twenty-seven"
}
```

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

Following is a non-normative example response:

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

{
  "active": false
}
```

2.3. Error Response

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

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

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

3. IANA Considerations

3.1. OAuth Token Introspection Response Registry

This specification establishes the OAuth Token Introspection Response registry.

OAuth registration client metadata names and descriptions are registered with a Specification Required ([RFC5226]) after a two-week review period on the `oauth-ext-review@ietf.org` mailing list, on the advice of one or more Designated Experts. However, to allow for the allocation of names prior to publication, the Designated Expert(s) may approve registration once they are satisfied that such a specification will be published.

Registration requests sent to the mailing list for review should use an appropriate subject (e.g., "Request to register OAuth Token Introspection Response name: example").

Within the review period, the Designated Expert(s) will either approve or deny the registration request, communicating this decision to the review list and IANA. Denials should include an explanation and, if applicable, suggestions as to how to make the request successful.

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

3.1.1. Registration Template

Name:

The name requested (e.g., "example"). This name is case sensitive. Names that match other registered names in a case

insensitive manner SHOULD NOT be accepted. Names that match claims registered in the JSON Web Token Claims registry established by [RFC7519] SHOULD have comparable definitions and semantics.

Description:

Brief description of the metadata value (e.g., "Example description").

Change controller:

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

Specification document(s):

Reference to the document(s) that specify the token endpoint authorization method, preferably including a URI that can be used to retrieve a copy of the document(s). An indication of the relevant sections may also be included but is not required.

3.1.2. Initial Registry Contents

The initial contents of the OAuth Token Introspection Response registry are:

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

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

- o Name: "client_id"
- o Description: Client identifier of the client
- o Change Controller: IESG
- o Specification Document(s): Section 2.2 of [[this document]].

- o Name: "scope"
- o Description: Authorized scopes of the token
- o Change Controller: IESG
- o Specification Document(s): Section 2.2 of [[this document]].

- o Name: "token_type"
- o Description: Type of the token
- o Change Controller: IESG

- o Specification Document(s): Section 2.2 of [[this document]].
- o Name: "exp"
- o Description: Expiration timestamp of the token
- o Change Controller: IESG
- o Specification Document(s): Section 2.2 of [[this document]].
- o Name: "iat"
- o Description: Issuance timestamp of the token
- o Change Controller: IESG
- o Specification Document(s): Section 2.2 of [[this document]].
- o Name: "nbf"
- o Description: Timestamp which the token is not valid before
- o Change Controller: IESG
- o Specification Document(s): Section 2.2 of [[this document]].
- o Name: "sub"
- o Description: Subject of the token
- o Change Controller: IESG
- o Specification Document(s): Section 2.2 of [[this document]].
- o Name: "aud"
- o Description: Audience of the token
- o Change Controller: IESG
- o Specification Document(s): Section 2.2 of [[this document]].
- o Name: "iss"
- o Description: Issuer of the token
- o Change Controller: IESG
- o Specification Document(s): Section 2.2 of [[this document]].
- o Name: "jti"
- o Description: Unique identifier of the token
- o Change Controller: IESG
- o Specification Document(s): Section 2.2 of [[this document]].

4. Security Considerations

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

- o If the token can expire, the authorization server MUST determine whether or not the token has expired.
- o If the token can be issued before it is able to be used, the authorization server MUST determine whether or not a token's valid period has started yet.
- o If the token can be revoked after it was issued, the authorization server MUST determine whether or not such a revocation has taken place.
- o If the token has been signed, the authorization server MUST validate the signature.
- o If the token can be used only at certain resource servers, the authorization server MUST determine whether or not the token can be used at the resource server making the introspection call.

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

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

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

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

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

Since a protected resource MAY cache the response of the introspection endpoint, designers of an OAuth 2.0 system using this protocol MUST consider the performance and security trade-offs inherent in caching security information such as this. A less aggressive cache with a short timeout will provide the protected resource with more up to date information (due to it needing to query the introspection endpoint more often) at the cost of increased network traffic and load on the introspection endpoint. A more aggressive cache with a longer duration will minimize network traffic and load on the introspection endpoint, but at the risk of stale information about the token. For example, the token may be revoked while the protected resource is relying on the value of the cached response to make authorization decisions. This creates a window during which a revoked token could be used at the protected resource. Consequently, an acceptable cache validity duration needs to be carefully considered given the concerns and sensitivities of the protected resource being accessed and the likelihood of a token being revoked or invalidated in the interim period. Highly sensitive environments can opt to disable caching entirely on the protected resource to eliminate the risk of stale cached information entirely, again at the cost of increased network traffic and server load. If the response contains the "exp" parameter (expiration), the response MUST NOT be cached beyond the time indicated therein.

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

that in cases where the authorization server stores no information about the token and has no means of accessing information about the token by parsing the token itself, it can not likely offer an introspection service.

5. Privacy Considerations

The introspection response may contain privacy-sensitive information such as user identifiers for resource owners. When this is the case, measures MUST be taken to prevent disclosure of this information to unintended parties. One method is to transmit user identifiers as opaque service-specific strings, potentially returning different identifiers to each protected resource.

If the protected resource sends additional information about the client's request to the authorization server (such as the client's IP address) using an extension of this specification, such information could have additional privacy considerations that the extension should detail. However, the nature and implications of such extensions are outside the scope of this specification.

Omitting privacy-sensitive information from an introspection response is the simplest way of minimizing privacy issues.

6. Acknowledgements

Thanks to the OAuth Working Group and the User Managed Access Working Group for feedback and review of this document, and to the various implementors of both the client and server components of this specification. In particular, the author would like to thank Amanda Anganes, John Bradley, Thomas Broyer, Brian Campbell, George Fletcher, Paul Freemantle, Thomas Hardjono, Eve Maler, Josh Mandel, Steve Moore, Mike Schwartz, Prabath Siriwardena, Sarah Squire, and Hannes Tschofennig.

7. References

7.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

- [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, March 2011.
- [RFC6749] Hardt, D., "The OAuth 2.0 Authorization Framework", RFC 6749, October 2012.
- [RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, October 2012.
- [RFC7009] Lodderstedt, T., Dronia, S., and M. Scurtescu, "OAuth 2.0 Token Revocation", RFC 7009, August 2013.
- [RFC7159] Bray, T., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, March 2014.
- [RFC7231] Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, June 2014.
- [RFC7519] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, May 2015.
- [W3C.REC-html5-20141028]
Hickson, I., Berjon, R., Faulkner, S., Leithead, T., Navara, E., O'Connor, E., and S. Pfeiffer, "HTML5", World Wide Web Consortium Recommendation REC-html5-20141028, October 2014, <<http://www.w3.org/TR/2014/REC-html5-20141028>>.

7.2. Informative References

- [TLS.BCP] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of TLS and DTLS", November 2014.

Appendix A. Use with Proof of Possession Tokens

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

request. The details of this usage are outside the scope of this specification and will be defined in an extension to this specification.

Appendix B. Document History

[[To be removed by the RFC Editor.]]

-11

- o Minor wording tweaks from IESG review.

-10

- o Added missing 2119 section to terminology.
- o Removed optional HTTP GET at introspection endpoint.
- o Added terminology.
- o Renamed this "a protocol" instead of "web API".
- o Moved JWT to normative reference.
- o Reworded definition of "scope" value.
- o Clarified extensibility of input parameters.
- o Noted that discover is out of scope.
- o Fixed several typos and imprecise references.

-09

- o Updated JOSE, JWT, and OAuth Assertion draft references to final RFC numbers.

-08

- o Added privacy considerations note about extensions.
- o Added acknowledgements (finally).

-07

- o Created a separate IANA registry for introspection responses, importing the values from JWT.

-06

- o Clarified relationship between AS and RS in introduction.
- o Used updated TLS text imported from Dyn-Reg drafts.
- o Clarified definition of active state.
- o Added some advice on caching responses.
- o Added security considerations on active state implementation.
- o Changed user_id to username based on WG feedback.

-05

- o Typo fix.
- o Updated author information.
- o Removed extraneous "linewrap" note from examples.

- 04

- o Removed "resource_id" from request.
- o Added examples.

- 03

- o Updated HTML and HTTP references.
- o Call for registration of parameters in the JWT registry.

- 02

- o Removed SAML pointer.
- o Clarified what an "active" token could be.
- o Explicitly declare introspection request as x-www-form-urlencoded format.
- o Added extended example.
- o Made protected resource authentication a MUST.

- 01

- o Fixed casing and consistent term usage.
- o Incorporated working group comments.
- o Clarified that authorization servers need to be able to understand the token if they're to introspect it.
- o Various editorial cleanups.

- 00

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

Author's Address

Justin Richer (editor)

Email: ietf@justin.richer.org

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: June 12, 2015

M. Jones
Microsoft
J. Bradley
Ping Identity
N. Sakimura
NRI
December 9, 2014

JSON Web Token (JWT)
draft-ietf-oauth-json-web-token-32

Abstract

JSON Web Token (JWT) is a compact, URL-safe means of representing claims to be transferred between two parties. The claims in a JWT are encoded as a JavaScript Object Notation (JSON) object that is used as the payload of a JSON Web Signature (JWS) structure or as the plaintext of a JSON Web Encryption (JWE) structure, enabling the claims to be digitally signed or MACed and/or encrypted.

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 12, 2015.

Copyright Notice

Copyright (c) 2014 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect

to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	4
1.1. Notational Conventions	4
2. Terminology	4
3. JSON Web Token (JWT) Overview	6
3.1. Example JWT	6
4. JWT Claims	8
4.1. Registered Claim Names	8
4.1.1. "iss" (Issuer) Claim	9
4.1.2. "sub" (Subject) Claim	9
4.1.3. "aud" (Audience) Claim	9
4.1.4. "exp" (Expiration Time) Claim	9
4.1.5. "nbf" (Not Before) Claim	9
4.1.6. "iat" (Issued At) Claim	10
4.1.7. "jti" (JWT ID) Claim	10
4.2. Public Claim Names	10
4.3. Private Claim Names	10
5. JOSE Header	10
5.1. "typ" (Type) Header Parameter	11
5.2. "cty" (Content Type) Header Parameter	11
5.3. Replicating Claims as Header Parameters	11
6. Unsecured JWTs	12
6.1. Example Unsecured JWT	12
7. Creating and Validating JWTs	13
7.1. Creating a JWT	13
7.2. Validating a JWT	14
7.3. String Comparison Rules	15
8. Implementation Requirements	16
9. URI for Declaring that Content is a JWT	16
10. IANA Considerations	16
10.1. JSON Web Token Claims Registry	16
10.1.1. Registration Template	18
10.1.2. Initial Registry Contents	18
10.2. Sub-Namespace Registration of urn:ietf:params:oauth:token-type:jwt	19
10.2.1. Registry Contents	19
10.3. Media Type Registration	19
10.3.1. Registry Contents	19
10.4. Header Parameter Names Registration	20
10.4.1. Registry Contents	20
11. Security Considerations	21

11.1. Trust Decisions	21
11.2. Signing and Encryption Order	21
12. Privacy Considerations	22
13. References	22
13.1. Normative References	22
13.2. Informative References	23
Appendix A. JWT Examples	24
A.1. Example Encrypted JWT	24
A.2. Example Nested JWT	25
Appendix B. Relationship of JWTs to SAML Assertions	26
Appendix C. Relationship of JWTs to Simple Web Tokens (SWTs)	27
Appendix D. Acknowledgements	27
Appendix E. Document History	28
Authors' Addresses	34

1. Introduction

JSON Web Token (JWT) is a compact claims representation format intended for space constrained environments such as HTTP Authorization headers and URI query parameters. JWTs encode claims to be transmitted as a JavaScript Object Notation (JSON) [RFC7159] object that is used as the payload of a JSON Web Signature (JWS) [JWS] structure or as the plaintext of a JSON Web Encryption (JWE) [JWE] structure, enabling the claims to be digitally signed or MACed and/or encrypted. JWTs are always represented using the JWS Compact Serialization or the JWE Compact Serialization.

The suggested pronunciation of JWT is the same as the English word "jot".

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 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.

2. Terminology

These terms defined by the JSON Web Signature (JWS) [JWS] specification are incorporated into this specification: "JSON Web Signature (JWS)", "Base64url Encoding", "Header Parameter", "JOSE Header", "JWS Compact Serialization", "JWS Payload", "JWS Signature", and "Unsecured JWS".

These terms defined by the JSON Web Encryption (JWE) [JWE] specification are incorporated into this specification: "JSON Web Encryption (JWE)", "Content Encryption Key (CEK)", "JWE Compact Serialization", "JWE Encrypted Key", "JWE Initialization Vector", and "JWE Plaintext".

These terms defined by the Internet Security Glossary, Version 2 [RFC4949] are incorporated into this specification: "Ciphertext", "Digital Signature", "Message Authentication Code (MAC)", and "Plaintext".

These terms are defined by this specification:

JSON Web Token (JWT)

A string representing a set of claims as a JSON object that is encoded in a JWS or JWE, enabling the claims to be digitally signed or MACed and/or encrypted.

JWT Claims Set

A JSON object that contains the Claims conveyed by the JWT.

Claim

A piece of information asserted about a subject. A Claim is represented as a name/value pair consisting of a Claim Name and a Claim Value.

Claim Name

The name portion of a Claim representation. A Claim Name is always a string.

Claim Value

The value portion of a Claim representation. A Claim Value can be any JSON value.

Encoded JOSE Header

Base64url encoding of the JOSE Header.

Nested JWT

A JWT in which nested signing and/or encryption are employed. In nested JWTs, a JWT is used as the payload or plaintext value of an enclosing JWS or JWE structure, respectively.

Unsecured JWT

A JWT whose Claims are not integrity protected or encrypted.

Collision-Resistant Name

A name in a namespace that enables names to be allocated in a manner such that they are highly unlikely to collide with other names. Examples of collision-resistant namespaces include: Domain Names, Object Identifiers (OIDs) as defined in the ITU-T X.660 and X.670 Recommendation series, and Universally Unique Identifiers (UUIDs) [RFC4122]. When using an administratively delegated namespace, the definer of a name needs to take reasonable precautions to ensure they are in control of the portion of the namespace they use to define the name.

StringOrURI

A JSON string value, with the additional requirement that while arbitrary string values MAY be used, any value containing a ":" character MUST be a URI [RFC3986]. StringOrURI values are compared as case-sensitive strings with no transformations or

canonicalizations applied.

NumericDate

A JSON numeric value representing the number of seconds from 1970-01-01T00:00:00Z UTC until the specified UTC date/time, ignoring leap seconds. This is equivalent to the IEEE Std 1003.1, 2013 Edition [POSIX.1] definition "Seconds Since the Epoch", in which each day is accounted for by exactly 86400 seconds, other than that non-integer values can be represented. See RFC 3339 [RFC3339] for details regarding date/times in general and UTC in particular.

3. JSON Web Token (JWT) Overview

JWTs represent a set of claims as a JSON object that is encoded in a JWS and/or JWE structure. This JSON object is the JWT Claims Set. As per Section 4 of RFC 7159 [RFC7159], the JSON object consists of zero or more name/value pairs (or members), where the names are strings and the values are arbitrary JSON values. These members are the claims represented by the JWT. This JSON object MAY contain white space and/or line breaks before or after any JSON values or structural characters, in accordance with Section 2 of RFC 7159 [RFC7159].

The member names within the JWT Claims Set are referred to as Claim Names. The corresponding values are referred to as Claim Values.

The contents of the JOSE Header describe the cryptographic operations applied to the JWT Claims Set. If the JOSE Header is for a JWS, the JWT is represented as a JWS and the claims are digitally signed or MACed, with the JWT Claims Set being the JWS Payload. If the JOSE Header is for a JWE, the JWT is represented as a JWE and the claims are encrypted, with the JWT Claims Set being the JWE Plaintext. A JWT may be enclosed in another JWE or JWS structure to create a Nested JWT, enabling nested signing and encryption to be performed.

A JWT is represented as a sequence of URL-safe parts separated by period ('.') characters. Each part contains a base64url encoded value. The number of parts in the JWT is dependent upon the representation of the resulting JWS using the JWS Compact Serialization or JWE using the JWE Compact Serialization.

3.1. Example JWT

The following example JOSE Header declares that the encoded object is a JSON Web Token (JWT) and the JWT is a JWS that is MACed using the HMAC SHA-256 algorithm:

```
{ "typ": "JWT",  
  "alg": "HS256" }
```

To remove potential ambiguities in the representation of the JSON object above, the octet sequence for the actual UTF-8 representation used in this example for the JOSE Header above is also included below. (Note that ambiguities can arise due to differing platform representations of line breaks (CRLF versus LF), differing spacing at the beginning and ends of lines, whether the last line has a terminating line break or not, and other causes. In the representation used in this example, the first line has no leading or trailing spaces, a CRLF line break (13, 10) occurs between the first and second lines, the second line has one leading space (32) and no trailing spaces, and the last line does not have a terminating line break.) The octets representing the UTF-8 representation of the JOSE Header in this example (using JSON array notation) are:

```
[123, 34, 116, 121, 112, 34, 58, 34, 74, 87, 84, 34, 44, 13, 10, 32,  
34, 97, 108, 103, 34, 58, 34, 72, 83, 50, 53, 54, 34, 125]
```

Base64url encoding the octets of the UTF-8 representation of the JOSE Header yields this Encoded JOSE Header value:

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
```

The following is an example of a JWT Claims Set:

```
{ "iss": "joe",  
  "exp": 1300819380,  
  "http://example.com/is_root": true }
```

The following octet sequence, which is the UTF-8 representation used in this example for the JWT Claims Set above, is the JWS Payload:

```
[123, 34, 105, 115, 115, 34, 58, 34, 106, 111, 101, 34, 44, 13, 10,  
32, 34, 101, 120, 112, 34, 58, 49, 51, 48, 48, 56, 49, 57, 51, 56,  
48, 44, 13, 10, 32, 34, 104, 116, 116, 112, 58, 47, 47, 101, 120, 97,  
109, 112, 108, 101, 46, 99, 111, 109, 47, 105, 115, 95, 114, 111,  
111, 116, 34, 58, 116, 114, 117, 101, 125]
```

Base64url encoding the JWS Payload yields this encoded JWS Payload (with line breaks for display purposes only):

```
eyJpc3MiOiJqb2UiLA0KICJleHAiOiJleMDA4MTkzODAsDQogImh0dHA6Ly  
9leGFtcGxlLnNvbn9pc19yb290Ijp0cnVlfQ
```

Computing the MAC of the encoded JOSE Header and encoded JWS Payload with the HMAC SHA-256 algorithm and base64url encoding the HMAC value

in the manner specified in [JWS], yields this encoded JWS Signature:

```
dBjftJeZ4CVP-mB92K27uhbUJU1plr_wWlgFWFOEjXk
```

Concatenating these encoded parts in this order with period ('.') characters between the parts yields this complete JWT (with line breaks for display purposes only):

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGft
cGxlImNvbS9pc19yb290Ijp0cnVlfQ
.
dBjftJeZ4CVP-mB92K27uhbUJU1plr_wWlgFWFOEjXk
```

This computation is illustrated in more detail in Appendix A.1 of [JWS]. See Appendix A.1 for an example of an encrypted JWT.

4. JWT Claims

The JWT Claims Set represents a JSON object whose members are the claims conveyed by the JWT. The Claim Names within a JWT Claims Set MUST be unique; JWT parsers MUST either reject JWTs with duplicate Claim Names or use a JSON parser that returns only the lexically last duplicate member name, as specified in Section 15.12 (The JSON Object) of ECMAScript 5.1 [ECMAScript].

The set of claims 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 claims in particular ways. However, in the absence of such requirements, all claims that are not understood by implementations MUST be ignored.

There are three classes of JWT Claim Names: Registered Claim Names, Public Claim Names, and Private Claim Names.

4.1. Registered Claim Names

The following Claim Names are registered in the IANA JSON Web Token Claims registry defined in Section 10.1. None of the claims defined below are intended to be mandatory to use or implement in all cases, but rather, provide a starting point for a set of useful, interoperable claims. Applications using JWTs should define which specific claims they use and when they are required or optional. All the names are short because a core goal of JWTs is for the representation to be compact.

4.1.1. "iss" (Issuer) Claim

The "iss" (issuer) claim identifies the principal that issued the JWT. The processing of this claim is generally application specific. The "iss" value is a case-sensitive string containing a StringOrURI value. Use of this claim is OPTIONAL.

4.1.2. "sub" (Subject) Claim

The "sub" (subject) claim identifies the principal that is the subject of the JWT. The Claims in a JWT are normally statements about the subject. The subject value MUST either be scoped to be locally unique in the context of the issuer or be globally unique. The processing of this claim is generally application specific. The "sub" value is a case-sensitive string containing a StringOrURI value. Use of this claim is OPTIONAL.

4.1.3. "aud" (Audience) Claim

The "aud" (audience) claim identifies the recipients that the JWT is intended for. Each principal intended to process the JWT MUST identify itself with a value in the audience claim. If the principal processing the claim does not identify itself with a value in the "aud" claim when this claim is present, then the JWT MUST be rejected. In the general case, the "aud" value is an array of case-sensitive strings, each containing a StringOrURI value. In the special case when the JWT has one audience, the "aud" value MAY be a single case-sensitive string containing a StringOrURI value. The interpretation of audience values is generally application specific. Use of this claim is OPTIONAL.

4.1.4. "exp" (Expiration Time) Claim

The "exp" (expiration time) claim identifies the expiration time on or after which the JWT MUST NOT be accepted for processing. The processing of the "exp" claim requires that the current date/time MUST be before the expiration date/time listed in the "exp" claim. Implementers MAY provide for some small leeway, usually no more than a few minutes, to account for clock skew. Its value MUST be a number containing a NumericDate value. Use of this claim is OPTIONAL.

4.1.5. "nbf" (Not Before) Claim

The "nbf" (not before) claim identifies the time before which the JWT MUST NOT be accepted for processing. The processing of the "nbf" claim requires that the current date/time MUST be after or equal to the not-before date/time listed in the "nbf" claim. Implementers MAY provide for some small leeway, usually no more than a few minutes, to

account for clock skew. Its value MUST be a number containing a NumericDate value. Use of this claim is OPTIONAL.

4.1.6. "iat" (Issued At) Claim

The "iat" (issued at) claim identifies the time at which the JWT was issued. This claim can be used to determine the age of the JWT. Its value MUST be a number containing a NumericDate value. Use of this claim is OPTIONAL.

4.1.7. "jti" (JWT ID) Claim

The "jti" (JWT ID) claim provides a unique identifier for the JWT. The identifier value MUST be assigned in a manner that ensures that there is a negligible probability that the same value will be accidentally assigned to a different data object; if the application uses multiple issuers, collisions MUST be prevented among values produced by different issuers as well. The "jti" claim can be used to prevent the JWT from being replayed. The "jti" value is a case-sensitive string. Use of this claim is OPTIONAL.

4.2. Public Claim Names

Claim Names can be defined at will by those using JWTs. However, in order to prevent collisions, any new Claim Name should either be registered in the IANA JSON Web Token Claims registry defined in Section 10.1 or be a Public Name: a value that contains a Collision-Resistant Name. In each case, the definer of the name or value needs to take reasonable precautions to make sure they are in control of the part of the namespace they use to define the Claim Name.

4.3. Private Claim Names

A producer and consumer of a JWT MAY agree to use Claim Names that are Private Names: names that are not Registered Claim Names Section 4.1 or Public Claim Names Section 4.2. Unlike Public Claim Names, Private Claim Names are subject to collision and should be used with caution.

5. JOSE Header

For a JWT object, the members of the JSON object represented by the JOSE Header describe the cryptographic operations applied to the JWT and optionally, additional properties of the JWT. Depending upon whether the JWT is a JWS or JWE, the corresponding rules for the JOSE Header values apply.

This specification further specifies the use of the following Header Parameters in both the cases where the JWT is a JWS and where it is a JWE.

5.1. "typ" (Type) Header Parameter

The "typ" (type) Header Parameter defined by [JWS] and [JWE] is used by JWT applications to declare the MIME Media Type [IANA.MediaTypes] of this complete JWT. This is intended for use by the JWT application when values that are not JWTs could also be present in an application data structure that can contain a JWT object; the application can use this value to disambiguate among the different kinds of objects that might be present. It will typically not be used by applications when it is already known that the object is a JWT. This parameter is ignored by JWT implementations; any processing of this parameter is performed by the JWT application. If present, it is RECOMMENDED that its value be "JWT" to indicate that this object is a JWT. While media type names are not case-sensitive, it is RECOMMENDED that "JWT" always be spelled using uppercase characters for compatibility with legacy implementations. Use of this Header Parameter is OPTIONAL.

5.2. "cty" (Content Type) Header Parameter

The "cty" (content type) Header Parameter defined by [JWS] and [JWE] is used by this specification to convey structural information about the JWT.

In the normal case in which nested signing or encryption operations are not employed, the use of this Header Parameter is NOT RECOMMENDED. In the case that nested signing or encryption is employed, this Header Parameter MUST be present; in this case, the value MUST be "JWT", to indicate that a Nested JWT is carried in this JWT. While media type names are not case-sensitive, it is RECOMMENDED that "JWT" always be spelled using uppercase characters for compatibility with legacy implementations. See Appendix A.2 for an example of a Nested JWT.

5.3. Replicating Claims as Header Parameters

In some applications using encrypted JWTs, it is useful to have an unencrypted representation of some Claims. This might be used, for instance, in application processing rules to determine whether and how to process the JWT before it is decrypted.

This specification allows Claims present in the JWT Claims Set to be replicated as Header Parameters in a JWT that is a JWE, as needed by the application. If such replicated Claims are present, the

application receiving them SHOULD verify that their values are identical, unless the application defines other specific processing rules for these Claims. It is the responsibility of the application to ensure that only claims that are safe to be transmitted in an unencrypted manner are replicated as Header Parameter values in the JWT.

Section 10.4.1 of this specification registers the "iss" (issuer), "sub" (subject), and "aud" (audience) Header Parameter names for the purpose of providing unencrypted replicas of these Claims in encrypted JWTs for applications that need them. Other specifications MAY similarly register other names that are registered Claim Names as Header Parameter names, as needed.

6. Unsecured JWTs

To support use cases in which the JWT content is secured by a means other than a signature and/or encryption contained within the JWT (such as a signature on a data structure containing the JWT), JWTs MAY also be created without a signature or encryption. An Unsecured JWT is a JWS using the "alg" Header Parameter value "none" and with the empty string for its JWS Signature value, as defined in JSON Web Algorithms (JWA) [JWA]; it is an Unsecured JWS with the JWT Claims Set as its JWS Payload.

6.1. Example Unsecured JWT

The following example JOSE Header declares that the encoded object is an Unsecured JWT:

```
{"alg":"none"}
```

Base64url encoding the octets of the UTF-8 representation of the JOSE Header yields this Encoded JOSE Header:

```
eyJhbGciOiJub25lIn0
```

The following is an example of a JWT Claims Set:

```
{"iss":"joe",  
 "exp":1300819380,  
 "http://example.com/is_root":true}
```

Base64url encoding the octets of the UTF-8 representation of the JWT Claims Set yields this encoded JWS Payload (with line breaks for display purposes only):

```
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFT  
cGx1LmNvbS9pc19yb290Ijp0cnVlfQ
```

The encoded JWS Signature is the empty string.

Concatenating these encoded parts in this order with period ('.') characters between the parts yields this complete JWT (with line breaks for display purposes only):

```
eyJhbGciOiJub251In0  
.  
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFT  
cGx1LmNvbS9pc19yb290Ijp0cnVlfQ  
.
```

7. Creating and Validating JWTs

7.1. Creating a JWT

To create a JWT, the following steps are performed. The order of the steps is not significant in cases where there are no dependencies between the inputs and outputs of the steps.

1. Create a JWT Claims Set containing the desired claims. Note that white space is explicitly allowed in the representation and no canonicalization need be performed before encoding.
2. Let the Message be the octets of the UTF-8 representation of the JWT Claims Set.
3. Create a JOSE Header containing the desired set of Header Parameters. The JWT MUST conform to either the [JWS] or [JWE] specification. Note that white space is explicitly allowed in the representation and no canonicalization need be performed before encoding.
4. Depending upon whether the JWT is a JWS or JWE, there are two cases:
 - * If the JWT is a JWS, create a JWS using the Message as the JWS Payload; all steps specified in [JWS] for creating a JWS MUST be followed.
 - * Else, if the JWT is a JWE, create a JWE using the Message as the JWE Plaintext; all steps specified in [JWE] for creating a JWE MUST be followed.

5. If a nested signing or encryption operation will be performed, let the Message be the JWS or JWE, and return to Step 3, using a "cty" (content type) value of "JWT" in the new JOSE Header created in that step.
6. Otherwise, let the resulting JWT be the JWS or JWE.

7.2. Validating a JWT

When validating a JWT, the following steps are performed. The order of the steps is not significant in cases where there are no dependencies between the inputs and outputs of the steps. If any of the listed steps fails then the JWT MUST be rejected -- treated by the application as an invalid input.

1. Verify that the JWT contains at least one period ('.') character.
2. Let the Encoded JOSE Header be the portion of the JWT before the first period ('.') character.
3. Base64url decode the Encoded JOSE Header following the restriction that no line breaks, white space, or other additional characters have been used.
4. Verify that the resulting octet sequence is a UTF-8 encoded representation of a completely valid JSON object conforming to RFC 7159 [RFC7159]; let the JOSE Header be this JSON object.
5. Verify that the resulting JOSE Header includes only parameters and values whose syntax and semantics are both understood and supported or that are specified as being ignored when not understood.
6. Determine whether the JWT is a JWS or a JWE using any of the methods described in Section 9 of [JWE].
7. Depending upon whether the JWT is a JWS or JWE, there are two cases:
 - * If the JWT is a JWS, follow the steps specified in [JWS] for validating a JWS. Let the Message be the result of base64url decoding the JWS Payload.
 - * Else, if the JWT is a JWE, follow the steps specified in [JWE] for validating a JWE. Let the Message be the JWE Plaintext.

8. If the JOSE Header contains a "cty" (content type) value of "JWT", then the Message is a JWT that was the subject of nested signing or encryption operations. In this case, return to Step 1, using the Message as the JWT.
9. Otherwise, base64url decode the Message following the restriction that no line breaks, white space, or other additional characters have been used.
10. Verify that the resulting octet sequence is a UTF-8 encoded representation of a completely valid JSON object conforming to RFC 7159 [RFC7159]; let the JWT Claims Set be this JSON object.

Finally, note that it is an application decision which algorithms may be used in a given context. Even if a JWT can be successfully validated, unless the algorithm(s) used in the JWT are acceptable to the application, it SHOULD reject the JWT.

7.3. String Comparison Rules

Processing a JWT inevitably requires comparing known strings to members and values in JSON objects. For example, in checking what the algorithm is, the Unicode string encoding "alg" will be checked against the member names in the JOSE Header to see if there is a matching Header Parameter name.

The JSON rules for doing member name comparison are described in Section 8.3 of RFC 7159 [RFC7159]. Since the only string comparison operations that are performed are equality and inequality, the same rules can be used for comparing both member names and member values against known strings.

These comparison rules MUST be used for all JSON string comparisons except in cases where the definition of the member explicitly calls out that a different comparison rule is to be used for that member value. In this specification, only the "typ" and "cty" member values do not use these comparison rules.

Some applications may include case-insensitive information in a case-sensitive value, such as including a DNS name as part of the "iss" (issuer) claim value. In those cases, the application may need to define a convention for the canonical case to use for representing the case-insensitive portions, such as lowercasing them, if more than one party might need to produce the same value so that they can be compared. (However if all other parties consume whatever value the producing party emitted verbatim without attempting to compare it to an independently produced value, then the case used by the producer will not matter.)

8. Implementation Requirements

This section defines which algorithms and features of this specification are mandatory to implement. Applications using this specification can impose additional requirements upon implementations that they use. For instance, one application might require support for encrypted JWTs and Nested JWTs, while another might require support for signing JWTs with ECDSA using the P-256 curve and the SHA-256 hash algorithm ("ES256").

Of the signature and MAC algorithms specified in JSON Web Algorithms (JWA) [JWA], only HMAC SHA-256 ("HS256") and "none" MUST be implemented by conforming JWT implementations. It is RECOMMENDED that implementations also support RSASSA-PKCS1-V1_5 with the SHA-256 hash algorithm ("RS256") and ECDSA using the P-256 curve and the SHA-256 hash algorithm ("ES256"). Support for other algorithms and key sizes is OPTIONAL.

Support for encrypted JWTs is OPTIONAL. If an implementation provides encryption capabilities, of the encryption algorithms specified in [JWA], only RSAES-PKCS1-V1_5 with 2048 bit keys ("RSA1_5"), AES Key Wrap with 128 and 256 bit keys ("A128KW" and "A256KW"), and the composite authenticated encryption algorithm using AES CBC and HMAC SHA-2 ("A128CBC-HS256" and "A256CBC-HS512") MUST be implemented by conforming implementations. It is RECOMMENDED that implementations also support using ECDH-ES to agree upon a key used to wrap the Content Encryption Key ("ECDH-ES+A128KW" and "ECDH-ES+A256KW") and AES in Galois/Counter Mode (GCM) with 128 bit and 256 bit keys ("A128GCM" and "A256GCM"). Support for other algorithms and key sizes is OPTIONAL.

Support for Nested JWTs is OPTIONAL.

9. URI for Declaring that Content is a JWT

This specification registers the URN "urn:ietf:params:oauth:token-type:jwt" for use by applications that declare content types using URIs (rather than, for instance, MIME Media Types) to indicate that the content referred to is a JWT.

10. IANA Considerations

10.1. JSON Web Token Claims Registry

This specification establishes the IANA JSON Web Token Claims registry for JWT Claim Names. The registry records the Claim Name

and a reference to the specification that defines it. This specification registers the Claim Names defined in Section 4.1.

Values are registered on a Specification Required [RFC5226] basis after a three-week review period on the `jwt-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 Expert(s) may approve registration once they are satisfied that such a specification will be published.

Registration requests must be sent to the `jwt-reg-review@ietf.org` mailing list for review and comment, with an appropriate subject (e.g., "Request to register claim: example").

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

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

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

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

[[Note to the RFC Editor and IANA: Pearl Liang of ICANN had requested that the draft supply the following proposed registry description information.

- o Protocol Category: JSON Web Token (JWT)
- o Registry Location: <http://www.iana.org/assignments/jwt>

- o Webpage Title: (same as the protocol category)
- o Registry Name: JSON Web Token Claims

]]

10.1.1.1. Registration Template

Claim Name:

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

Claim Description:

Brief description of the Claim (e.g., "Issuer").

Change Controller:

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

Specification Document(s):

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

10.1.1.2. Initial Registry Contents

- o Claim Name: "iss"
- o Claim Description: Issuer
- o Change Controller: IESG
- o Specification Document(s): Section 4.1.1 of [[this document]]

- o Claim Name: "sub"
- o Claim Description: Subject
- o Change Controller: IESG
- o Specification Document(s): Section 4.1.2 of [[this document]]

- o Claim Name: "aud"
- o Claim Description: Audience

- o Change Controller: IESG
- o Specification Document(s): Section 4.1.3 of [[this document]]
- o Claim Name: "exp"
- o Claim Description: Expiration Time
- o Change Controller: IESG
- o Specification Document(s): Section 4.1.4 of [[this document]]
- o Claim Name: "nbf"
- o Claim Description: Not Before
- o Change Controller: IESG
- o Specification Document(s): Section 4.1.5 of [[this document]]
- o Claim Name: "iat"
- o Claim Description: Issued At
- o Change Controller: IESG
- o Specification Document(s): Section 4.1.6 of [[this document]]
- o Claim Name: "jti"
- o Claim Description: JWT ID
- o Change Controller: IESG
- o Specification Document(s): Section 4.1.7 of [[this document]]

10.2. Sub-Namespace Registration of urn:ietf:params:oauth:token-type:jwt

10.2.1. Registry Contents

This specification registers the value "token-type:jwt" in the IANA urn:ietf:params:oauth registry established in An IETF URN Sub-Namespace for OAuth [RFC6755], which can be used to indicate that the content is a JWT.

- o URN: urn:ietf:params:oauth:token-type:jwt
- o Common Name: JSON Web Token (JWT) Token Type
- o Change Controller: IESG
- o Specification Document(s): [[this document]]

10.3. Media Type Registration

10.3.1. Registry Contents

This specification registers the "application/jwt" Media Type [RFC2046] in the MIME Media Types registry [IANA.MediaTypes] in the manner described in RFC 6838 [RFC6838], which can be used to indicate that the content is a JWT.

- o Type Name: application
- o Subtype Name: jwt
- o Required Parameters: n/a
- o Optional Parameters: n/a
- o Encoding considerations: 8bit; JWT values are encoded as a series of base64url encoded values (some of which may be the empty string) separated by period ('.') characters.
- o Security Considerations: See the Security Considerations section of [[this document]]
- o Interoperability Considerations: n/a
- o Published Specification: [[this document]]
- o Applications that use this media type: OpenID Connect, Mozilla Persona, Salesforce, Google, Android, Windows Azure, Amazon Web Services, and numerous others
- o Fragment identifier considerations: n/a
- o Additional Information: Magic number(s): n/a, File extension(s): n/a, Macintosh file type code(s): n/a
- o Person & email address to contact for further information: Michael B. Jones, mbj@microsoft.com
- o Intended Usage: COMMON
- o Restrictions on Usage: none
- o Author: Michael B. Jones, mbj@microsoft.com
- o Change Controller: IESG
- o Provisional registration? No

10.4. Header Parameter Names Registration

This specification registers specific Claim Names defined in Section 4.1 in the IANA JSON Web Signature and Encryption Header Parameters registry defined in [JWS] for use by Claims replicated as Header Parameters in JWEs, per Section 5.3.

10.4.1. Registry Contents

- o Header Parameter Name: "iss"
- o Header Parameter Description: Issuer
- o Header Parameter Usage Location(s): JWE
- o Change Controller: IESG
- o Specification Document(s): Section 4.1.1 of [[this document]]
- o Header Parameter Name: "sub"
- o Header Parameter Description: Subject
- o Header Parameter Usage Location(s): JWE
- o Change Controller: IESG
- o Specification Document(s): Section 4.1.2 of [[this document]]

- o Header Parameter Name: "aud"
- o Header Parameter Description: Audience
- o Header Parameter Usage Location(s): JWE
- o Change Controller: IESG
- o Specification Document(s): Section 4.1.3 of [[this document]]

11. Security Considerations

All of the security issues that are pertinent to any cryptographic application must be addressed by JWT/JWS/JWE/JWK agents. Among these issues are protecting the user's asymmetric private and symmetric secret keys and employing countermeasures to various attacks.

All the security considerations in the JWS specification also apply to JWT, as do the JWE security considerations when encryption is employed. In particular, the JWS JSON Security Considerations and Unicode Comparison Security Considerations apply equally to the JWT Claims Set in the same manner that they do to the JOSE Header.

11.1. Trust Decisions

The contents of a JWT cannot be relied upon in a trust decision unless its contents have been cryptographically secured and bound to the context necessary for the trust decision. In particular, the key(s) used to sign and/or encrypt the JWT will typically need to verifiably be under the control of the party identified as the issuer of the JWT.

11.2. Signing and Encryption Order

While syntactically the signing and encryption operations for Nested JWTs may be applied in any order, if both signing and encryption are necessary, normally producers should sign the message and then encrypt the result (thus encrypting the signature). This prevents attacks in which the signature is stripped, leaving just an encrypted message, as well as providing privacy for the signer. Furthermore, signatures over encrypted text are not considered valid in many jurisdictions.

Note that potential concerns about security issues related to the order of signing and encryption operations are already addressed by the underlying JWS and JWE specifications; in particular, because JWE only supports the use of authenticated encryption algorithms, cryptographic concerns about the potential need to sign after encryption that apply in many contexts do not apply to this specification.

12. Privacy Considerations

A JWT may contain privacy-sensitive information. When this is the case, measures MUST be taken to prevent disclosure of this information to unintended parties. One way to achieve this is to use an encrypted JWT and authenticate the recipient. Another way is to ensure that JWTs containing unencrypted privacy-sensitive information are only transmitted using protocols utilizing encryption that support endpoint authentication, such as TLS. Omitting privacy-sensitive information from a JWT is the simplest way of minimizing privacy issues.

13. References

13.1. Normative References

- [ECMAScript]
Ecma International, "ECMAScript Language Specification, 5.1 Edition", ECMA 262, June 2011.
- [IANA.MediaType]
Internet Assigned Numbers Authority (IANA), "MIME Media Types", 2005.
- [JWA]
Jones, M., "JSON Web Algorithms (JWA)", draft-ietf-jose-json-web-algorithms (work in progress), December 2014.
- [JWE]
Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)", draft-ietf-jose-json-web-encryption (work in progress), December 2014.
- [JWS]
Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", draft-ietf-jose-json-web-signature (work in progress), December 2014.
- [RFC20]
Cerf, V., "ASCII format for Network Interchange", RFC 20, October 1969.
- [RFC2046]
Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types", RFC 2046, November 1996.
- [RFC2119]
Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC3986]
Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform

Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.

[RFC4949] Shirey, R., "Internet Security Glossary, Version 2", RFC 4949, August 2007.

[RFC7159] Bray, T., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, March 2014.

13.2. Informative References

[CanvasApp]

Facebook, "Canvas Applications", 2010.

[JSS] Bradley, J. and N. Sakimura (editor), "JSON Simple Sign", September 2010.

[MagicSignatures]

Panzer (editor), J., Laurie, B., and D. Balfanz, "Magic Signatures", January 2011.

[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.

[POSIX.1] Institute of Electrical and Electronics Engineers, "The Open Group Base Specifications Issue 7", IEEE Std 1003.1, 2013 Edition, 2013.

[RFC3275] Eastlake, D., Reagle, J., and D. Solo, "(Extensible Markup Language) XML-Signature Syntax and Processing", RFC 3275, March 2002.

[RFC3339] Klyne, G., Ed. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, July 2002.

[RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique Identifier (UUID) URN Namespace", RFC 4122, July 2005.

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

[RFC6755] Campbell, B. and H. Tschofenig, "An IETF URN Sub-Namespace for OAuth", RFC 6755, October 2012.

- [RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, January 2013.
- [SWT] Hardt, D. and Y. Goland, "Simple Web Token (SWT)", Version 0.9.5.1, November 2009.
- [W3C.CR-xml11-20021015] Cowan, J., "Extensible Markup Language (XML) 1.1", W3C CR CR-xml11-20021015, October 2002.
- [W3C.REC-xml-c14n-20010315] Boyer, J., "Canonical XML Version 1.0", World Wide Web Consortium Recommendation REC-xml-c14n-20010315, March 2001, <<http://www.w3.org/TR/2001/REC-xml-c14n-20010315>>.

Appendix A. JWT Examples

This section contains examples of JWTs. For other example JWTs, see Section 6.1 and Appendices A.1, A.2, and A.3 of [JWS].

A.1. Example Encrypted JWT

This example encrypts the same claims as used in Section 3.1 to the recipient using RSAES-PKCS1-V1_5 and AES_128_CBC_HMAC_SHA_256.

The following example JOSE Header declares that:

- o The Content Encryption Key is encrypted to the recipient using the RSAES-PKCS1-V1_5 algorithm to produce the JWE Encrypted Key.
- o Authenticated encryption is performed on the Plaintext using the AES_128_CBC_HMAC_SHA_256 algorithm to produce the JWE Ciphertext and the JWE Authentication Tag.

```
{"alg":"RSA1_5","enc":"A128CBC-HS256"}
```

Other than using the octets of the UTF-8 representation of the JWT Claims Set from Section 3.1 as the plaintext value, the computation of this JWT is identical to the computation of the JWE in Appendix A.2 of [JWE], including the keys used.

The final result in this example (with line breaks for display purposes only) is:

```

eyJhbGciOiJSU0ExXzUiLCJlbmMiOiJBMTI4Q0JDLUhTMjU2In0.
QR1Owv2ug2WyPBnbQrRARTEk9kDO2w8qDcjiHnSJflSdvliNqhWXaKH4MqAkQtM
oNfABIPJaZm0HaA415sv3aeuBWnD8J-Ui7Ah6cWafs3ZwwFKDFUUsWHSK-IPKxLG
TkND09XyJORj_CHAgOPJ-Sd8ONQRnJvWn_hXV1BNMHZUjPyYwEsRhDhzjAD26ima
sOTsgruobpYGoQcXUwFDn7moXPRfDE8-NoQX7N7ZYMmpUDkR-Cx9obNGwJQ3nM52
YCitxoQVPzjbl7WBuB7AohdBoZOdZ24WlN1lVieh8v1K4krB8xgKvRU8kgFrEn_a
lrZgN5TiySnmzTROF869lQ.
AXY8DctDaGlsbGljb3RoZQ.
MKOle7UQrG6nSxTLX6Mqwt0orbHvAKeWnDYvpIAeZ72deHxz3roJDXQyhxx0wKaM
HDjUEOKIwrthKthpQEanSBNYHZgmNOV7slN1Eu9g3J8.
fiK5lVwhsxJ-siBMR-YFiA

```

A.2. Example Nested JWT

This example shows how a JWT can be used as the payload of a JWE or JWS to create a Nested JWT. In this case, the JWT Claims Set is first signed, and then encrypted.

The inner signed JWT is identical to the example in Appendix A.2 of [JWS]. Therefore, its computation is not repeated here. This example then encrypts this inner JWT to the recipient using RSAES-PKCS1-V1_5 and AES_128_CBC_HMAC_SHA_256.

The following example JOSE Header declares that:

- o The Content Encryption Key is encrypted to the recipient using the RSAES-PKCS1-V1_5 algorithm to produce the JWE Encrypted Key.
- o Authenticated encryption is performed on the Plaintext using the AES_128_CBC_HMAC_SHA_256 algorithm to produce the JWE Ciphertext and the JWE Authentication Tag.
- o The Plaintext is itself a JWT.

```

{"alg":"RSA1_5","enc":"A128CBC-HS256","cty":"JWT"}

```

Base64url encoding the octets of the UTF-8 representation of the JOSE Header yields this encoded JOSE Header value:

```

eyJhbGciOiJSU0ExXzUiLCJlbmMiOiJBMTI4Q0JDLUhTMjU2Iiwia3R5IjoiaSldUIn0

```

The computation of this JWT is identical to the computation of the JWE in Appendix A.2 of [JWE], other than that different JOSE Header, Plaintext, JWE Initialization Vector, and Content Encryption Key values are used. (The RSA key used is the same.)

The Payload used is the octets of the ASCII [RFC20] representation of

the JWT at the end of Appendix A.2.1 of [JWS] (with all whitespace and line breaks removed), which is a sequence of 458 octets.

The JWE Initialization Vector value used (using JSON array notation) is:

```
[82, 101, 100, 109, 111, 110, 100, 32, 87, 65, 32, 57, 56, 48, 53, 50]
```

This example uses the Content Encryption Key represented by the base64url encoded value below:

```
GawggguFyGrWKav7AX4VKUg
```

The final result for this Nested JWT (with line breaks for display purposes only) is:

```
eyJhbGciOiJSU0ExXzUiLCJlbmMiOiJBMTI4Q0JDLUhTMjU2Iiwia3R5IjoiaSldU  
In0.  
g_hEwksO1Ax8Qn7HoN-BVeBoa8FXe0kpyk_XdcSmxvcM5_P296JXXtoHISr_DD_M  
qewaQSH4dZOQH0UgKLeFly-9RI11TG-_GelbZFazBPwKC5lJ6OLANLMD0QSL4fYE  
b9ERe-epKYE3xb2jfY1AltHqBO-PM6j23Guj2yDKnFv6WO72tteVzm_2n17SBFvh  
DuR9a2nHTE67pe0XGBUS_TK7eca-iVq5COeVdJR4U4VZGGlxRGPLRHvolVLEHx6D  
YyLpw30Ay9R6d68YCLi9FYTq3hIXPK_-dmPlOUlKvPr1GgJzRoeC9G5qCvdchWsq  
JGTQ_z3Wfo5zsqwkxruxwA.  
UmVkbW9uZCBXQSA5ODAlMg.  
VwHERHPvCNcHHpTjkoigx3_ExK0Qc71RMEParpatm0X_qpg-w8kozSjfnIPPXiTb  
BLXR65CIPkFqz4l1Ae9w_uowKiwyi9acgVztAi-pSL8GQsXnaamh9kXlmdh3M_TT  
-FZGQFQsFhu0Z72gJKGdfGE-OE7hS1zuBD5oEUfk0Dmb0VzWEzpxxiSSBbBAzP10  
l56pPfAtrjEYw-7ygeMkwBl6Z_mLS6w6xUgKlvW6ULmkV-uLC4FUiYKECK4e3WZY  
KwlbpgIqGYsw2v_grHjszJZ-_I5uM-9RA8ycX9KqPRp9gc6pXmoU_-27ATs9XCvr  
ZXUtK2902AUzqpeEUJYjWWxSNsS-rlTJlI-FMJ4XyAiGrfmo9hQPcNBYxPz3GQb2  
8Y5CLSQfNgKSGt0A4isplhBUXBHAndgtcslt7ZoQJaKe_nNJgNliWtWpJ_ebuOpE  
l8jdhehdccnRMIwAmUln7SPkmhI1lHlSOpvcvDfhUN5wuqU955vOBvfkBOh5A1lU  
zBuo2WlgZ6hYi9-e3w29bR0C2-pp3jbxqEDw3iWaf2dc5b-LnR0FEYXvI_tYk5rd  
_J9N0mg0tQ6RbpxNEMNoA9QWk5lgdPvbh9Ba0195abQ.  
AVO9iT5AV4CzvDJCdhSFlQ
```

Appendix B. Relationship of JWTs to SAML Assertions

SAML 2.0 [OASIS.saml-core-2.0-os] provides a standard for creating security tokens with greater expressivity and more security options than supported by JWTs. However, the cost of this flexibility and expressiveness is both size and complexity. SAML's use of XML [W3C.CR-xml11-20021015] and XML DSIG [RFC3275] contributes to the size of SAML assertions; its use of XML and especially XML Canonicalization [W3C.REC-xml-c14n-20010315] contributes to their

complexity.

JWTs are intended to provide a simple security token format that is small enough to fit into HTTP headers and query arguments in URIs. It does this by supporting a much simpler token model than SAML and using the JSON [RFC7159] object encoding syntax. It also supports securing tokens using Message Authentication Codes (MACs) and digital signatures using a smaller (and less flexible) format than XML DSIG.

Therefore, while JWTs can do some of the things SAML assertions do, JWTs are not intended as a full replacement for SAML assertions, but rather as a token format to be used when ease of implementation or compactness are considerations.

SAML Assertions are always statements made by an entity about a subject. JWTs are often used in the same manner, with the entity making the statements being represented by the "iss" (issuer) claim, and the subject being represented by the "sub" (subject) claim. However, with these claims being optional, other uses of the JWT format are also permitted.

Appendix C. Relationship of JWTs to Simple Web Tokens (SWTs)

Both JWTs and Simple Web Tokens SWT [SWT], at their core, enable sets of claims to be communicated between applications. For SWTs, both the claim names and claim values are strings. For JWTs, while claim names are strings, claim values can be any JSON type. Both token types offer cryptographic protection of their content: SWTs with HMAC SHA-256 and JWTs with a choice of algorithms, including signature, MAC, and encryption algorithms.

Appendix D. Acknowledgements

The authors acknowledge that the design of JWTs was intentionally influenced by the design and simplicity of Simple Web Tokens [SWT] and ideas for JSON tokens that Dick Hardt discussed within the OpenID community.

Solutions for signing JSON content were previously explored by Magic Signatures [MagicSignatures], JSON Simple Sign [JSS], and Canvas Applications [CanvasApp], all of which influenced this draft.

This specification is the work of the OAuth Working Group, which includes dozens of active and dedicated participants. In particular, the following individuals contributed ideas, feedback, and wording that influenced this specification:

Dirk Balfanz, Richard Barnes, Brian Campbell, Alissa Cooper, Breno de Medeiros, Stephen Farrell, Dick Hardt, Joe Hildebrand, Jeff Hodges, Edmund Jay, Yaron Y. Goland, Warren Kumari, Ben Laurie, Barry Leiba, Ted Lemon, James Manger, Prateek Mishra, Kathleen Moriarty, Tony Nadalin, Axel Nennker, John Panzer, Emmanuel Raviart, David Recordon, Eric Rescorla, Jim Schaad, Paul Tarjan, Hannes Tschofenig, Sean Turner, and Tom Yu.

Hannes Tschofenig and Derek Atkins chaired the OAuth working group and Sean Turner, Stephen Farrell, and Kathleen Moriarty served as Security area directors during the creation of this specification.

Appendix E. Document History

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

-32

- o Replaced uses of the phrases "JWS object" and "JWE object" with "JWS" and "JWE".
- o Applied other minor editorial improvements.

-31

- o Updated the example IANA registration request subject line.

-30

- o Applied privacy wording supplied by Stephen Farrell.
- o Clarified where white space and line breaks may occur in JSON objects by referencing Section 2 of RFC 7159.
- o Specified that registration reviews occur on the `jwt-reg-review@ietf.org` mailing list.

-29

- o Used real values for examples in the IANA Registration Template.

-28

- o Addressed IESG review comments by Alissa Cooper, Barry Leiba, Stephen Farrell, Ted Lemon, and Richard Barnes.

- o Changed the RFC 6755 reference to be informative, based upon related IESG review feedback on draft-ietf-oauth-saml2-bearer.

-27

- o Removed unused reference to RFC 4648.
- o Changed to use the term "authenticated encryption" instead of "encryption", where appropriate.
- o Changed the registration review period to three weeks.
- o Acknowledged additional contributors.

-26

- o Removed an ambiguity in numeric date representations by specifying that leap seconds are handled in the manner specified by POSIX.1.
- o Addressed Gen-ART review comments by Russ Housley.
- o Addressed secdir review comments by Warren Kumari and Stephen Kent.
- o Replaced the terms Plaintext JWS and Plaintext JWT with Unsecured JWS and Unsecured JWT.

-25

- o Reworded the language about JWT implementations ignoring the "typ" parameter, explicitly saying that its processing is performed by JWT applications.
- o Added a Privacy Considerations section.

-24

- o Cleaned up the reference syntax in a few places.
- o Applied minor wording changes to the Security Considerations section.

-23

- o Replaced the terms JWS Header, JWE Header, and JWT Header with a single JOSE Header term defined in the JWS specification. This also enabled a single Header Parameter definition to be used and reduced other areas of duplication between specifications.

-22

- o Revised the introduction to the Security Considerations section. Also introduced subsection headings for security considerations items.
- o Added text about when applications typically would and would not use the "typ" header parameter.

-21

- o Removed unnecessary informative JWK spec reference.

-20

- o Changed the RFC 6755 reference to be normative.
- o Changed the JWK reference to be informative.
- o Described potential sources of ambiguity in representing the JSON objects used in the examples. The octets of the actual UTF-8 representations of the JSON objects used in the examples are included to remove these ambiguities.
- o Noted that octet sequences are depicted using JSON array notation.

-19

- o Specified that support for Nested JWTs is optional and that applications using this specification can impose additional requirements upon implementations that they use.
- o Updated the JSON reference to RFC 7159.

-18

- o Clarified that the base64url encoding includes no line breaks, white space, or other additional characters.
- o Removed circularity in the audience claim definition.
- o Clarified that it is entirely up to applications which claims to use.
- o Changed "SHOULD" to "MUST" in "in the absence of such requirements, all claims that are not understood by implementations MUST be ignored".

- o Clarified that applications can define their own processing rules for claims replicated in header parameters, rather than always requiring that they be identical in the JWT Header and JWT Claims Set.
- o Removed a JWT creation step that duplicated a step in the underlying JWS or JWE creation.
- o Added security considerations about using JWTs in trust decisions.

-17

- o Corrected RFC 2119 terminology usage.
- o Replaced references to draft-ietf-json-rfc4627bis with RFC 7158.

-16

- o Changed some references from being normative to informative, per JOSE issue #90.

-15

- o Replaced references to RFC 4627 with draft-ietf-json-rfc4627bis.

-14

- o Referenced the JWE section on Distinguishing between JWS and JWE Objects.

-13

- o Added Claim Description registry field.
- o Used Header Parameter Description registry field.
- o Removed the phrases "JWA signing algorithms" and "JWA encryption algorithms".
- o Removed the term JSON Text Object.

-12

- o Tracked the JOSE change refining the "typ" and "cty" definitions to always be MIME Media Types, with the omission of "application/" prefixes recommended for brevity. For compatibility with legacy implementations, it is RECOMMENDED that "JWT" always be spelled using uppercase characters when used as a "typ" or "cty" value.

As side effects, this change removed the "typ" Claim definition and narrowed the uses of the URI "urn:ietf:params:oauth:token-type:jwt".

- o Updated base64url definition to match JOSE definition.
- o Changed terminology from "Reserved Claim Name" to "Registered Claim Name" to match JOSE terminology change.
- o Applied other editorial changes to track parallel JOSE changes.
- o Clarified that the subject value may be scoped to be locally unique in the context of the issuer or may be globally unique.

-11

- o Added a Nested JWT example.
- o Added "sub" to the list of Claims registered for use as Header Parameter values when an unencrypted representation is required in an encrypted JWT.

-10

- o Allowed Claims to be replicated as Header Parameters in encrypted JWTs as needed by applications that require an unencrypted representation of specific Claims.

-09

- o Clarified that the "typ" header parameter is used in an application-specific manner and has no effect upon the JWT processing.
- o Stated that recipients MUST either reject JWTs with duplicate Header Parameter Names or with duplicate Claim Names or use a JSON parser that returns only the lexically last duplicate member name.

-08

- o Tracked a change to how JWEs are computed (which only affected the example encrypted JWT value).

-07

- o Defined that the default action for claims that are not understood is to ignore them unless otherwise specified by applications.

- o Changed from using the term "byte" to "octet" when referring to 8 bit values.
- o Tracked encryption computation changes in the JWE specification.

-06

- o Changed the name of the "prn" claim to "sub" (subject) both to more closely align with SAML name usage and to use a more intuitive name.
- o Allow JWTs to have multiple audiences.
- o Applied editorial improvements suggested by Jeff Hodges, Prateek Mishra, and Hannes Tschofenig. Many of these simplified the terminology used.
- o Explained why Nested JWTs should be signed and then encrypted.
- o Clarified statements of the form "This claim is OPTIONAL" to "Use of this claim is OPTIONAL".
- o Referenced String Comparison Rules in JWS.
- o Added seriesInfo information to Internet Draft references.

-05

- o Updated values for example AES CBC calculations.

-04

- o Promoted Initialization Vector from being a header parameter to being a top-level JWE element. This saves approximately 16 bytes in the compact serialization, which is a significant savings for some use cases. Promoting the Initialization Vector out of the header also avoids repeating this shared value in the JSON serialization.
- o Applied changes made by the RFC Editor to RFC 6749's registry language to this specification.
- o Reference RFC 6755 -- An IETF URN Sub-Namespace for OAuth.

-03

- o Added statement that "StringOrURI values are compared as case-sensitive strings with no transformations or canonicalizations

applied".

- o Indented artwork elements to better distinguish them from the body text.

-02

- o Added an example of an encrypted JWT.
- o Added this language to Registration Templates: "This name is case sensitive. Names that match other registered names in a case insensitive manner SHOULD NOT be accepted."
- o Applied editorial suggestions.

-01

- o Added the "cty" (content type) header parameter for declaring type information about the secured content, as opposed to the "typ" (type) header parameter, which declares type information about this object. This significantly simplified nested JWTs.
- o Moved description of how to determine whether a header is for a JWS or a JWE from the JWT spec to the JWE spec.
- o Changed registration requirements from RFC Required to Specification Required with Expert Review.
- o Added Registration Template sections for defined registries.
- o Added Registry Contents sections to populate registry values.
- o Added "Collision Resistant Namespace" to the terminology section.
- o Numerous editorial improvements.

-00

- o Created the initial IETF draft based upon draft-jones-json-web-token-10 with no normative changes.

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/>

Nat Sakimura
Nomura Research Institute

Email: n-sakimura@nri.co.jp
URI: <http://nat.sakimura.org/>

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: October 10, 2021

N. Sakimura
NAT.Consulting
J. Bradley
Yubico
M. Jones
Microsoft
April 8, 2021

The OAuth 2.0 Authorization Framework: JWT Secured Authorization Request
(JAR)
draft-ietf-oauth-jwsreq-34

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 is not integrity protected and thus the parameters can be tainted, (b) the source of the communication is not authenticated, and (c) the communication through the user agents can be monitored. 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 October 10, 2021.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction	3
1.1. Requirements Language	5
2. Terminology	6
2.1. Request Object	6
2.2. Request Object URI	6
3. Symbols and abbreviated terms	6
4. Request Object	7
5. Authorization Request	9
5.1. Passing a Request Object by Value	10
5.2. Passing a Request Object by Reference	10
5.2.1. URI Referencing the Request Object	11
5.2.2. Request using the "request_uri" Request Parameter . .	12
5.2.3. Authorization Server Fetches Request Object	12
6. Validating JWT-Based Requests	13
6.1. JWE Encrypted Request Object	13
6.2. JWS Signed Request Object	13
6.3. Request Parameter Assembly and Validation	14
7. Authorization Server Response	14
8. TLS Requirements	14
9. IANA Considerations	15
9.1. OAuth Parameters Registration	15
9.2. OAuth Authorization Server Metadata Registry	16
9.3. OAuth Dynamic Client Registration Metadata Registry . . .	17
9.4. Media Type Registration	17
9.4.1. Registry Contents	17
10. Security Considerations	18
10.1. Choice of Algorithms	18
10.2. Request Source Authentication	18
10.3. Explicit Endpoints	19

10.4.	Risks Associated with request_uri	19
10.4.1.	DDoS Attack on the Authorization Server	20
10.4.2.	Request URI Rewrite	20
10.5.	Downgrade Attack	20
10.6.	TLS Security Considerations	21
10.7.	Parameter Mismatches	21
10.8.	Cross-JWT Confusion	21
11.	Privacy Considerations	22
11.1.	Collection limitation	22
11.2.	Disclosure Limitation	23
11.2.1.	Request Disclosure	23
11.2.2.	Tracking using Request Object URI	23
12.	Acknowledgements	24
13.	Revision History	24
14.	References	32
14.1.	Normative References	32
14.2.	Informative References	34
	Authors' Addresses	35

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 to be used to provide confidentiality and integrity protection. 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 is 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, 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 trusted third party so that a client application cannot request more permissions than previously agreed.

Furthermore, passing 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 as a Proposed Standard, along with the associated signing and encryption methods [RFC7515] [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. Note that, in contrast to RFC 7519, the elements of the Claims Set are encoded OAuth Request Parameters [IANA.OAuth.Parameters], supplemented with only a few of the IANA-managed JSON Web Token Claims [IANA.JWT.Claims] - in particular "iss" and "aud". The JWT in the "request" parameter 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 (including at and before user-agents).
- (d) (collection minimization) The request can be signed by a trusted third party attesting that the authorization request is compliant with a certain policy. For example, a request can be pre-examined by a trusted third party that all the personal data requested is strictly necessary to perform the process that the end-user asked for, and signed by that trusted 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 application level cryptography. 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", "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.

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

JSON Web Token (JWT) [RFC7519] whose JWT Claims Set holds the JSON encoded OAuth 2.0 authorization request parameters.

2.2. Request Object URI

Absolute URI that references the set of parameters comprising an OAuth 2.0 authorization request. The contents of the resource referenced by the URI are a Request Object (Section 2.1), unless the URI was provided to the client by the same Authorization Server, in which case the content is an implementation detail at the discretion of the Authorization Server. The former is to ensure interoperability in cases where the provider of the request_uri is a separate entity from the consumer, such as when a client provides a URI referencing a Request Object stored on the client's backend service and made accessible via HTTPS. In the latter case where the Authorization Server is both provider and consumer of the URI, such as when it offers an endpoint that provides a URI in exchange for a Request Object, this interoperability concern does not apply.

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 contain all the parameters (including extension parameters) used to process the OAuth 2.0 [RFC6749] authorization request except the "request" and "request_uri" parameters that are defined in this document. The parameters are represented as the JWT claims of the object. 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 [RFC8259] object 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. The value of "aud" should be the value of the Authorization Server (AS) "issuer" as defined in RFC8414 [RFC8414].

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

The client determines the algorithms used to sign and encrypt Request Objects. The algorithms chosen need to be supported by both the client and the authorization server. The client can inform the authorization server of the algorithms that it supports in its dynamic client registration metadata [RFC7591], specifically, the metadata values "request_object_signing_alg", "request_object_encryption_alg", and "request_object_encryption_enc". Likewise, the authorization server can inform the client of the algorithms that it supports in its authorization server metadata [RFC8414], specifically, the metadata values "request_object_signing_alg_values_supported", "request_object_encryption_alg_values_supported", and "request_object_encryption_enc_values_supported".

The 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.

A Request Object (Section 2.1) has the media type [RFC2046] "application/oauth-authz-req+jwt". Note that some existing deployments may alternatively be using the type "application/jwt".

The following is an example of the Claims in a Request Object before base64url [RFC7515] encoding and signing. Note that it includes the extension parameters "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 [RFC7518] results in this Request Object value (with line wraps within values for display purposes only):

```
eyYhbGciOiJSUzI1NiIsImtpZCI6ImSyYmRjIn0.ewogICAgImIzcyI6ICJzNkJoZlZJrcXQzIiwKICAgICJhdWQiOiAiaHR0cHM6Ly9zZXJ2ZXIuZXhhbXBsZS5jb20iLAogICAgInJlc3BvbmlX3R5cGUOiAiY29kZSBpZF90b2t1biIsCiAgICAiY2xpZW50X2lkIjogInM2QmhhUmtxdDMLAogICAgInJlZGlyZWNOX3VyaSI6ICJodHRwczovL2NsYWVudC5leGFTcGx1Lm9yZy9jYiIsCiAgICAic2NvcGUOiAib3BlbmklIiwKICAgICJzdGF0ZSI6ICJhZjBpZmpzbGRraiiIsCiAgICAibm9uY2UiOiAibi0wUzZfV3pBMklqIiwKICAgICJtYXhfYWdlIjogODY0MDAKfQ.Nsxa_18VUElVaPjqW_ToI1yrEJ67BgKb5xsuZRVqzGkfKrOIX7BCx0biSxYgmjK9KJPctH1OC0iQJwXu5YVY-vnW0_PLJb1C2HG-ztVzcnKpZcgle4i0vgQcpkH00CpW3SEYXnyWnKzUkZqSblwAZALo5f89B_p6QA6j6JwBSRvdVsDPdulw8lKxGTbH2CpCaQ50rLAg3EYLYaCb41k4I1zGXE4fvim9FIMs80CmmzwIB5S-ujfFzwFjoyuPEV4hJnoVUMR_9Wt9ypPf846lGwA8h9G9oNTIuX8ft2jfpnZdFmLq3_wr3Wa5q3a-lfbqF3S9H_8nN3jli7tLR_5Nz-q
```

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": "x5RbkAZkmpRxia65qRQ1wwSMSxQUnS7gcpVTV_cdHmfmG2ltd2yabEO9XadD8
    pJNZubINPpmgHh3J1aD9WRwS05ucmFq3CfFsluLt13_7oX5yDRSKX7poXmT_5
    ko8k4NJZPMAO8fPToDTH7kHYbONSE2FYa5GZ60CUsFhSonI-dcMDJ0Ary9lxI
    w5k2z4TAdARVWcs7sD07VhlMMshrwsPHBQgTatlkxyIHxbYdtak8fqvNAwr7O
    lVEvM_Ipf5OfmdB8Sd-wjzaBsyP4VhJKoi_qdgSzpC694XZeYPq45Sw-q5liF
    Ulc0lTCI7z6jltUtnR6ySn6XDGFnzH5Fe5ypw",
  "e": "AQAB"
}
```

5. Authorization Request

The client constructs the authorization request URI by adding the following parameters to the query component of the authorization endpoint URI using the "application/x-www-form-urlencoded" format:

request

REQUIRED unless "request_uri" is specified. The Request Object (Section 2.1) that holds authorization request parameters stated in section 4 of OAuth 2.0 [RFC6749]. If this parameter is present in the authorization request, "request_uri" MUST NOT be present.

request_uri

REQUIRED unless "request" is specified. The absolute URI as defined by RFC3986 [RFC3986] that is the Request Object URI (Section 2.2) referencing the authorization request parameters stated in section 4 of OAuth 2.0 [RFC6749]. If this parameter is present in the authorization request, "request" MUST NOT be present.

client_id

REQUIRED. OAuth 2.0 [RFC6749] "client_id". The value MUST match the "request" or "request_uri" Request Object's (Section 2.1) "client_id".

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?client_id=s6BhdRkqt3&request=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXZW80In0%3D 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?client_id=s6BhdRkqt3&
request=eyJhbGciOiJSUzI1NiIsImtpZCI6Im9ybmRjIn0.ewogICAgImIzcyI6
ICJzNkJoZjZJrcXQzIiwKIICAgICJhdWQiOiAiAiaHR0cHM6Ly9zZXJ2ZXIuZXhhbXBs
ZS5jb20iLAogICAgInJlc3BvbmlX3R5cGUiOiAiY29kZSBpZF90b2t1biIsCiAg
ICAiY2xpZW50X2lkIjogInM2QmhhkUmtxdDMLAogICAgInJlZGlyZWNOX3VyaSI6
ICJodHRwczovL2NsaWVudC5leGFtcGxlLm9yZy9jYiIsCiAgICAIc2NvcGUiOiAi
b3BlbmklIiwKIICAgICJzdGF0ZSI6ICJhZjBpZmpzbGRraiIsCiAgICAIbm9uY2Ui
OiAibi0wUzZfV3pBMklqIiwKIICAgICJtYXhfYWdlIjogODY0MDAKfQ.Nsxa_18VU
ElVaPjqW_ToIlyrEJ67BgKb5xsuZRVqzGkfKrOIX7BCx0biSxYGmjK9KJPctH1OC
0iQJwXu5YVY-vnW0_PLJb1C2HG-ztVzcnKZC2gE4i0vgQcpkUOCpW3SEYXnyWnKz
uKzqSblwAZALo5f89B_p6QA6j6JwBSRvdVsDPdulW8lKxGTbH82czCaQ50rLAg3E
YLyaCb4ik4I1zGXE4fvim9FIMs8OCmzwIB5S-uJfFzwFjjoyuPEV4hJnoVUmXR_W
9t9pPf846lGwA8h9G9oNTIuX8Ft2jfpnZdFmLg3_wr3Wa5q3a-lfbgF3S9H_8nN3
jli7tLR_5Nz-q
```

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 SHOULD NOT exceed 512 ASCII characters. There are two 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. 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 "request_uri" MUST be a Request Object and MUST be reachable by the Authorization Server unless the URI was provided to the client by the Authorization Server. In the first case, the "request_uri" MUST be an "https" URI, as specified in Section 2.7.2 of RFC7230 [RFC7230]. In the second case, it MUST be a URN, as specified in RFC8141 [RFC8141].

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):

```
eyYhbGciOiJSUzI1NiIsImtpZCI6ImSyYmRjIn0.ewogICAgImIzcyI6ICJzNkJoZFJrcXQzIiwKICAgICJhdWQiOiAiAHR0cHM6Ly9zZXJ2ZXIuZXhhbXBsZS5jb20iLAogICAgInJlc3BvbmlkX3R5cGUOiAiY29kZSBpZF90b2t1biIsCiAgICAiY2xpZW50X2lkIjogInM2QmhhGmtxdMDiLAogICAgInJlZGlyZWNOX3VyaSI6ICJodHRwczowL2NsAWVudC5leGftcGx1Lm9yYz9yIiIsCiAgICAic2NvcGU0IiAib3B1bmlkIiwKICAgICJzZGF0ZSI6ICJhZGJpZmpzbGRraiIsCiAgICAibm9uY2UiOiAibi0wUzFvF3pBmk1qIiwKICAgICJtYXhfYWdlIjogODY0MDAKfQ.Nsxa_18VUElVaPjqW_ToIlyrEJ67BgKb5xsuZRVqzGkfKrOIX7BCx0biSxYGmjK9KJPctH1OC0iQJwXu5YVY-vnW0_PLJb1C2HG-ztVzcnKZC2gE4i0vgQcpkUOCpW3SEYXnyWnKzuKzqSb1wAZALo5f89B_p6QA6j6JwBSRvdVsDPdulW8lKxGTbH82czCaQ50rLAg3EYLYaC64ik4I1zGXE4fvim9FIMs8OCmmzwIB5S-ujFfzwFjoyuPEV4hjNoVUmXR_W9tPyPf846lwA8h9G9oNTIuX8ft2jfpnzZfMlG3-w37Wa5q3a-lfbqF3S9H_8nN3j1i7tLR_5Nz-q
```

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 trusted third party. For example, the authorization server may provide a URL to which the client POSTs the request object and obtains the Request 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 so that the URI is not guessable if publicly retrievable. For the guidance, refer to 5.1.4.2.2 of [RFC6819] and Good Practices for Capability URLs [CapURLs]. 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). In this example, a trusted third-party service hosts the Request Object.

```
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?  
client_id=s6BhdRkqt3  
&request_uri=https%3A%2F%2Ftfp.example.org%2Frequest.jwt  
%2FGkurKxf5T0Y-mnPFCHqWOMiZi4VS138cQO_V7PZHAdM
```

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. In this example, a trusted third-party service hosts the Request Object.

```
GET /request.jwt/GkurKxf5T0Y-mnPFCHqWOMiZi4VS138cQO_V7PZHAdM HTTP/1.1  
Host: tfp.example.org
```

The following is an example of the fetch response:

```
HTTP/1.1 200 OK
Date: Thu, 20 Aug 2020 23:52:39 GMT
Server: Apache/2.4.43 (tfp.example.org)
Content-type: application/oauth-authz-req+jwt
Content-Length: 797
Last-Modified: Wed, 19 Aug 2020 23:52:32 GMT

eyJhbGciOiJSUzI1NiIsImtpZCI6Im9yYmRjIn0.ewogICAgImIzcyI6ICJzNkJoZFJrcXQzIiwKICAgICJhdWQiOiAiaHR0cHM6Ly9zZXJ2ZXIuZXhhbXBsZS5jb20iLAogICAgInJlc3Bvb3R5cGUiOiAiY29kZSBpZF90b2t1biIsCiAgICAiY2xpZW50X2lkIjogInM2QmhhkUmtxdDMiLAogICAgInJlZGlyZWNOX3VyaSI6ICJodHRwczovL2NsYWVudC5leGFtcGxlLm9yZy9jYiIsCiAgICAic2NvcGUiOiAib3BlbmlkIiwKICAgICJzdGF0ZSI6ICJhZjBpZmpzbGRraiiIsCiAgICAibm9uY2UiOiAibi0wUzZfV3pBMk1qIiwKICAgICJtYXhfYWdlIjogODY0MDAKfQ.Nsxa_18VUELvaPjqW_ToIlyrEJ67BgKb5xsuZRVqzGkfKrOIX7BCx0biSxYGmJk9KJPctH1OC0iQJwXu5YVY-vnW0_PLJb1C2HG-ztVzcnKZC2gE4i0vgQcpkUOCpW3SEYXnyWnKzuKzqSblwAZALo5f89B_p6QA6j6JwBSRvdVsDPdulW8lKxGTbH82czCaQ50rLAg3EYLYaCb4ik4I1zGXE4fvim9FIMS8OCMmzwIB5S-ujFfzwFjoyuPEV4hJnoVUmXR_W9typPf846lGwA8h9G9oNTIuX8Ft2jfpnZdFmLg3_wr3Wa5q3a-lfbgF3S9H_8nN3j1i7tLR_5Nz-g
```

6. Validating JWT-Based Requests

6.1. JWE 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 to the client in response to the authorization request.

6.2. JWS Signed Request Object

The Authorization Server MUST validate the signature of the JSON Web Signature [RFC7515] signed Request Object. If a "kid" Header Parameter is present, the key identified MUST be the key used, and MUST be a key associated with the client. The signature MUST be validated using a key associated with the client and the algorithm specified in the "alg" Header Parameter. Algorithm verification MUST be performed, as specified in Sections 3.1 and 3.2 of [RFC8725].

If the key is not associated with the client or if signature validation fails, the Authorization Server MUST return an

"invalid_request_object" error to the client in response to the authorization request.

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 Client ID values in the "client_id" request parameter and in the Request Object "client_id" claim MUST be identical. The Authorization Server then validates the request as specified in OAuth 2.0 [RFC6749].

If the Client ID check or the request validation fails, then the Authorization Server MUST return an error to the client in response to the authorization request, as specified in Section 5.2 of 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 DNS-ID [RFC6125], to avoid man-in-the-middle attacks. The rules and guidelines defined in [RFC6125] apply here, with the following considerations:

- o 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.
- o DNS names in server certificates MAY contain the wildcard character `"*"`.
- o 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].
- o SRV-ID and URI-ID as described in Section 6.5 of [RFC6125] MUST NOT be used for comparison.

9. IANA Considerations

9.1. OAuth Parameters Registration

Since the request object is a JWT, the core JWT claims cannot be used for any purpose in the request object other than for what JWT dictates. Thus, they need to be registered as OAuth Authorization Request parameters to avoid future OAuth extensions using them with different meanings.

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

- o Name: `"iss"`
- o Parameter Usage Location: authorization request
- o Change Controller: IETF
- o Specification Document(s): Section 4.1.1 of [RFC7519] and this document.
- o Name: `"sub"`
- o Parameter Usage Location: authorization request
- o Change Controller: IETF

- o Specification Document(s): Section 4.1.2 of [RFC7519] and this document.
- o Name: "aud"
- o Parameter Usage Location: authorization request
- o Change Controller: IETF
- o Specification Document(s): Section 4.1.3 of [RFC7519] and this document.
- o Name: "exp"
- o Parameter Usage Location: authorization request
- o Change Controller: IETF
- o Specification Document(s): Section 4.1.4 of [RFC7519] and this document.
- o Name: "nbf"
- o Parameter Usage Location: authorization request
- o Change Controller: IETF
- o Specification Document(s): Section 4.1.5 of [RFC7519] and this document.
- o Name: "iat"
- o Parameter Usage Location: authorization request
- o Change Controller: IETF
- o Specification Document(s): Section 4.1.6 of [RFC7519] and this document.
- o Name: "jti"
- o Parameter Usage Location: authorization request
- o Change Controller: IETF
- o Specification Document(s): Section 4.1.7 of [RFC7519] and this document.

9.2. OAuth Authorization Server Metadata Registry

This specification adds the following value to the "OAuth Authorization Server Metadata" registry [IANA.OAuth.Parameters] established by [RFC8414].

- o Metadata Name: "require_signed_request_object"
- o Metadata Description: Indicates where authorization request needs to be protected as Request Object and provided through either "request" or "request_uri" parameter.
- o Change Controller: IETF
- o Specification Document(s): Section 10.5 of this document.

9.3. OAuth Dynamic Client Registration Metadata Registry

This specification adds the following value to the "OAuth Dynamic Client Registration Metadata" registry [IANA.OAuth.Parameters] established by [RFC7591].

- o Metadata Name: "require_signed_request_object"
- o Metadata Description: Indicates where authorization request needs to be protected as Request Object and provided through either "request" or "request_uri" parameter.
- o Change Controller: IETF
- o Specification Document(s): Section 10.5 of this document.

9.4. Media Type Registration

9.4.1. Registry Contents

This section registers the "application/oauth-authz-req+jwt" media type [RFC2046] in the "Media Types" registry [IANA.MediaTypes] in the manner described in [RFC6838], which can be used to indicate that the content is a JWT containing Request Object claims.

- o Type name: application
- o Subtype name: oauth-authz-req+jwt
- o Required parameters: n/a
- o Optional parameters: n/a
- o Encoding considerations: binary; A Request Object is a JWT; JWT values are encoded as a series of base64url-encoded values (some of which may be the empty string) separated by period ('.') characters.
- o Security considerations: See Section 10 of [[this specification]]
- o Interoperability considerations: n/a
- o Published specification: Section 4 of [[this specification]]
- o Applications that use this media type: Applications that use Request Objects to make an OAuth 2.0 Authorization Request
- o Fragment identifier considerations: n/a
- o Additional information:

Magic number(s): n/a

File extension(s): n/a

Macintosh file type code(s): n/a

- o Person & email address to contact for further information:
Nat Sakimura, nat@nat.consulting
- o Intended usage: COMMON
- o Restrictions on usage: none
- o Author: Nat Sakimura, nat@nat.consulting

- o Change controller: IETF
- o Provisional registration? No

10. Security Considerations

In addition to the all the security considerations discussed in OAuth 2.0 [RFC6819], the security considerations in [RFC7515], [RFC7516], [RFC7518], and [RFC8725] need 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 signed then encrypted using JWS [RFC7515] and JWE [RFC7516] respectively, with then considered appropriate algorithms.

10.2. Request Source Authentication

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

- (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. Note however, that if public key encryption is used, no source authentication is enabled by the encryption, as any party can encrypt content to the public key.
- (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) When an Authorization Server implements a service that returns a Request Object URI in exchange for a Request Object, 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. It MUST validate the signature, per (a). 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) When a trusted third-party service returns a Request Object URI in exchange for a Request Object, it MUST validate the signature, per (a). In addition, the Authorization Server MUST be trusted by the third-party service and MUST know out-of-band that the client is also trusted by it.

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 following endpoints defined in [RFC6749], [RFC6750], and [RFC8414] are RECOMMENDED by this specification to use this practice :

- (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 this practice also applies to the discovery related endpoints.

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

10.4. Risks Associated with request_uri

The introduction of "request_uri" introduces several attack possibilities. Consult the security considerations in Section 7 of RFC3986 [RFC3986] for more information regarding risks associated with URIs.

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 media type of the response is "application/oauth-authz-req+jwt", (c) implement a time-out for obtaining the content of "request_uri", and (d) 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 media type of the response is "application/oauth-authz-req+jwt", and (c) implement a time-out for obtaining the content of "request_uri".

10.5. Downgrade Attack

Unless the protocol used by client and the server is locked down to use OAuth JAR, it is possible for an attacker to use RFC6749 requests to bypass all the protection provided by this specification.

To prevent it, this specification defines a new client metadata and server metadata "require_signed_request_object" whose value is a boolean.

When the value of it as a client metadata is "true", then the server MUST reject the authorization request from the client that does not conform to this specification. It MUST also reject the request if the request object uses "alg":"none" when this client metadata value is "true". If omitted, the default value is "false".

When the value of it as a server metadata is "true", then the server MUST reject the authorization request from any client that does not conform to this specification. It MUST also reject the request if the request object uses "alg":"none" when this server metadata value is "true". If omitted, the default value is "false".

Note that even if "require_signed_request_object" metadata values are not present, the client MAY use signed request objects, provided that there are signing algorithms mutually supported by the client and the server. Use of signing algorithm metadata is described in Section 4.

10.6. TLS Security Considerations

Current 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].

10.7. Parameter Mismatches

Given that OAuth parameter values are being sent in two different places, as normal OAuth parameters and as Request Object claims, implementations must guard against attacks that could use mismatching parameter values to obtain unintended outcomes. That is the reason that the two Client ID values MUST match, the reason that only the parameter values from the Request Object are to be used, and the reason that neither "request" nor "request_uri" can appear in a Request Object.

10.8. Cross-JWT Confusion

As described in Section 2.8 of [RFC8725], attackers may attempt to use a JWT issued for one purpose in a context that it was not intended for. The mitigations described for these attacks can be applied to Request Objects.

One way that an attacker might attempt to repurpose a Request Object is to try to use it as a client authentication JWT, as described in Section 2.2 of [RFC7523]. A simple way to prevent this is to never use the Client ID as the "sub" value in a Request Object.

Another way to prevent cross-JWT confusion is to use explicit typing, as described in Section 3.11 of [RFC8725]. One would explicitly type a Request Object by including a "typ" Header Parameter with the value "oauth-authz-req+jwt" (which is registered in Section 9.4.1. Note however, that requiring explicitly typed Requests Objects at existing authorization servers will break most existing deployments, as existing clients are already commonly using untyped Request Objects, especially with OpenID Connect [OpenID.Core]. However, requiring

explicit typing would be a good idea for new OAuth deployment profiles where compatibility with existing deployments is not a consideration.

Finally, yet another way to prevent cross-JWT confusion is to use a key management regime in which keys used to sign Request Objects are identifiably distinct from those used for other purposes. Then, if an adversary attempts to repurpose the Request Object in another context, a key mismatch will occur, thwarting the attack.

11. 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.

11.1. 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 trusted third-party service 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 trusted third-party service to obtain the Request Object URI. This process is two steps:

- (1) (Certification Process) The trusted third-party service examines the business process of the client and determines what claims they need: This is the certification process. Once the client is certified, then they are issued a client credential to authenticate against to push request objects to the trusted third-party service to get the "request_uri".

- (2) (Translation Process) The client uses the client credential that it got to push the request object to the trusted third-party service to get the "request_uri". The trusted third-party service also verifies that the Request Object is consistent with the claims that the client is eligible for, per prior step.

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 trusted third-party service. 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 trusted third-party service for adherence to the Collection Limitation principle.

11.2. Disclosure Limitation

11.2.1. 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.

11.2.2. 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 static per-user

Request Object URIs are 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 persistent Request Object URIs should be avoided. Single-use Request Object URIs are one alternative.

12. Acknowledgements

The following people contributed to the creation of this document in the OAuth working group and other IETF roles. (Affiliations at the time of the contribution are used.)

Annabelle Backman (Amazon), Dirk Balfanz (Google), Sergey Beryozkin, Ben Campbell (as AD), Brian Campbell (Ping Identity), Roman Danyliw (as AD), Martin Duke (as AD), Vladimir Dzhuvinov (Connect2id), Lars Eggert (as AD), Joel Halpern (as GENART), Benjamin Kaduk (as AD), Stephen Kent (as SECDIR), Murray Kucherawy (as AD), Warren Kumari (as OPSDIR), Watson Ladd (as SECDIR), Torsten Lodderstedt (yes.com), Jim Manico, Axel Nennker (Deutsche Telecom), Hannes Tschofenig (ARM), James H. Manger (Telstra), Kathleen Moriarty (as AD), John Panzer (Google), Francesca Palombini (as AD), David Recordon (Facebook), Marius Scurtescu (Google), Luke Shepard (Facebook), Filip Skokan (Auth0), Eric Vyncke (as AD), and Robert Wilton (as AD).

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), Breno de Medeiros (Google), Hideki Nara (TACT), Justin Richer (MITRE).

13. Revision History

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

-34

- o Addressed additional IESG comments by Murray Kucherawy and Francesca Palombini.

-33

- o Addressed IESG comments prior to 8-Apr-21 telechat. Thanks to Martin Duke, Lars Eggert, Benjamin Kaduk, Francesca Palombini, and Eric Vyncke for their reviews.

-32

- o Removed outdated JSON reference.

-31

- o Addressed SecDir review comments by Watson Ladd.

-30

- o Changed the MIME Type from "oauth.authz.req+jwt" to "oauth-authz-req+jwt", per advice from the designated experts.

-29

- o Uniformly use the Change Controller "IETF".

-28

- o Removed unused references, as suggested by Roman Danyliw.

-27

- o Edits by Mike Jones to address IESG and working group review comments, including:
- o Added Security Considerations text saying not to use the Client ID as the "sub" value to prevent Cross-JWT Confusion.
- o Added Security Considerations text about using explicit typing to prevent Cross-JWT Confusion.
- o Addressed Eric Vyncke's review comments.
- o Addressed Robert Wilton's review comments.
- o Addressed Murray Kucherawy's review comments.
- o Addressed Benjamin Kaduk's review comments.
- o Applied spelling and grammar corrections.

-20

- o BK comments
- o Section 3 Removed WAP

- o Section 4. Clarified authorization request object parameters, removed extension parameters from examples
- o Section 4. Specifies application/oauth.authz.req+jwt as mime-type for request objects
- o Section 5.2.1 Added reference to Capability URLs
- o Section 5.2.3. Added entropy fragment to example request
- o Section 8. Replaced "subjectAltName dnsName" with "DNS-ID"
- o Section 9. Registers authorization request parameters in JWT Claims Registry.
- o Section 9. Registers application/oauth.authz.req in IANA mime-types registry
- o Section 10.1. Clarified encrypted request objects are "signed then encrypted" to maintain consistency
- o Section 10.2. Clarifies trust between AS and TFP
- o Section 10.3. Clarified endpoints subject to the practice
- o Section 10.4 Replaced "redirect_uri" to "request_uri"
- o Section 10.4. Added reference to RFC 3986 for risks
- o Section 10.4.1.d Deleted "do" to maintain grammar flow
- o Section 10.4.1, 10.4.2 Replaced "application/jose" to "application/jwt"
- o Section 12.1. Extended description for submitting authorization request to TFP to obtain request object
- o Section 12.2.2. Replaced per-user Request Object URI with static per-user Request URIs
- o Section 13. Combined OAuth WG contributors together
- o Section Whole doc Replaced application/jwt with application/oauth.authz.req+jwt

-19

- o AD comments

- o Section 5.2.1. s/Requiest URI/Request URI/
- o Section 8 s/[BCP195] ./[BCP195]./
- o Section 10.3. s/sited/cited/
- o Section 11. Typo. s/Curent/Current/

-17

- o #78 Typos in content-type

-16

- o Treated remaining Ben Campbell comments.

-15

- o Removed further duplication

-14

- o #71 Reiterate dynamic params are included.
- o #70 Made clear that AS must return error.
- o #69 Inconsistency of the need to sign.
- o Fixed Mime-type.
- o #67 Inconsistence in requiring HTTPS in request URI.
- o #66 Dropped ISO 29100 reference.
- o #25 Removed Encrypt only option.
- o #59 Same with #25.

-13

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

-12

- o fixes #62 - Alexey Melnikov Discuss
- o fixes #48 - OPSDIR Review : General - delete semicolons after list items
- o fixes #58 - DP Comments for the Last Call
- o fixes #57 - GENART - Remove "non-normative ... " from examples.
- o fixes #45 - OPSDIR Review : Introduction - are attacks discovered or already opened
- o fixes #49 - OPSDIR Review : Introduction - Inconsistent colons after initial sentence of list items.
- o fixes #53 - OPSDIR Review : 6.2 JWS Signed Request Object - Clarify JOSE Header
- o fixes #42 - OPSDIR Review : Introduction - readability of 'and' is confusing
- o fixes #50 - OPSDIR Review : Section 4 Request Object - Clarify 'signed, encrypted, or signed and encrypted'
- o fixes #39 - OPSDIR Review : Abstract - Explain/Clarify JWS and JWE
- o fixed #50 - OPSDIR Review : Section 4 Request Object - Clarify 'signed, encrypted, or signed and encrypted'
- o fixes #43 - OPSDIR Review : Introduction - 'properties' sounds awkward and are not exactly 'properties'
- o fixes #56 - OPSDIR Review : 12 Acknowledgements - 'contribution is' => 'contribution are'
- o fixes #55 - OPSDIR Review : 11.2.2 Privacy Considerations - ' It is in a way' => 'In a way, it is'
- o fixes #54 - OPSDIR Review : 11 Privacy Considerations - 'and not specific' => 'and are not specific'
- o fixes #51 - OPSDIR Review : Section 4 Request Object - 'It is fine' => 'It is recommended'
- o fixes #47 - OPSDIR Review : Introduction - 'over- the- wire' => 'over-the-wire'

- o fixes #46 - OPSDIR Review : Introduction - 'It allows' => 'The use of application security' for
- o fixes #44 - OPSDIR Review : Introduction - 'has' => 'have'
- o fixes #41 - OPSDIR Review : Introduction - missing 'is' before 'typically sent'
- o fixes #38 - OPSDIR Review : Section 11 - Delete 'freely accessible' regarding ISO 29100

-11

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

-10

- o #20: KM1 -- some wording that is awkward in the TLS section.
- o #21: KM2 - the additional attacks against OAuth 2.0 should also have a pointer
- o #22: KM3 -- Nit: in the first line of 10.4:
- o #23: KM4 -- Mention RFC6973 in Section 11 in addition to ISO 29100
- o #24: SECDIR review: Section 4 -- Confusing requirements for sign+encrypt
- o #25: SECDIR review: Section 6 -- authentication and integrity need not be provided if the requestor encrypts the token?
- o #26: SECDIR Review: Section 10 -- why no reference for JWS algorithms?
- o #27: SECDIR Review: Section 10.2 - how to do the agreement between client and server "a priori"?
- o #28: SECDIR Review: Section 10.3 - Indication on "large entropy" and "short lifetime" should be indicated
- o #29: SECDIR Review: Section 10.3 - Typo
- o #30: SECDIR Review: Section 10.4 - typos and missing articles

- o #31: SECDIR Review: Section 10.4 - Clearer statement on the lack of endpoint identifiers needed
- o #32: SECDIR Review: Section 11 - ISO29100 needs to be moved to normative reference
- o #33: SECDIR Review: Section 11 - Better English and Entropy language needed
- o #34: Section 4: Typo
- o #35: More Acknowledgment
- o #36: DP - More precise qualification on Encryption needed.

-09

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

-08

- o Applied changes proposed by Hannes on 2016-06-29 on IETF OAuth list recorded as <https://bitbucket.org/Nat/oauth-jwsreq/issues/12/>.
- o TLS requirements added.
- o Security Consideration reinforced.
- o Privacy Consideration added.
- o Introduction improved.

-07

- o Changed the abbrev to OAuth JAR from oauth-jar.
- o Clarified sig and enc methods.
- o Better English.

- o Removed claims from one of the example.
- o Re-worded the URI construction.
- o Changed the example to use request instead of request_uri.
- o Clarified that Request Object parameters take precedence regardless of request or request_uri parameters were used.
- o Generalized the language in 4.2.1 to convey the intent more clearly.
- o Changed "Server" to "Authorization Server" as a clarification.
- o Stopped talking about request_object_signing_alg.
- o IANA considerations now reflect the current status.
- o 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.
- o Added "older versions of " to the reference to IE URI length limitations.
- o Stopped talking about signed or unsigned JWS etc.
- o 1.Introduction improved.

-06

- o Added explanation on the 512 chars URL restriction.
- o Updated Acknowledgements.

-05

- o More alignment with OpenID Connect.

-04

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

-03

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

-02

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

-01

- o Copy Edits.

14. References

14.1. 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.
- [IANA.MediaTypes] IANA, "Media Types", <<http://www.iana.org/assignments/media-types>>.
- [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>>.
- [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>>.
- [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>>.

- [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>>.
- [RFC8414] Jones, M., Sakimura, N., and J. Bradley, "OAuth 2.0 Authorization Server Metadata", RFC 8414, DOI 10.17487/RFC8414, June 2018, <<https://www.rfc-editor.org/info/rfc8414>>.

14.2. 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>>.
- [CapURLs] Tennison, J., "Good Practices for Capability URLs", W3C Working Draft, February 2014, <<https://www.w3.org/TR/capability-urls/>>.
- [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>>.
- [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>.
- [RFC2046] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types", RFC 2046, DOI 10.17487/RFC2046, November 1996, <<https://www.rfc-editor.org/info/rfc2046>>.
- [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>>.

- [RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, DOI 10.17487/RFC6838, January 2013, <<https://www.rfc-editor.org/info/rfc6838>>.
- [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>>.
- [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>>.
- [RFC7591] 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>>.
- [RFC8725] Sheffer, Y., Hardt, D., and M. Jones, "JSON Web Token Best Current Practices", BCP 225, RFC 8725, DOI 10.17487/RFC8725, February 2020, <<https://www.rfc-editor.org/info/rfc8725>>.

Authors' Addresses

Nat Sakimura
NAT.Consulting
2-22-17 Naka
Kunitachi, Tokyo 186-0004
Japan

Phone: +81-42-580-7401
Email: nat@nat.consulting
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/>

Michael B. Jones
Microsoft
One Microsoft Way
Redmond, Washington 98052
United States of America

Email: mbj@microsoft.com
URI: <https://self-issued.info/>

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: May 16, 2015

M. Jones
Microsoft
B. Campbell
Ping Identity
C. Mortimore
Salesforce
November 12, 2014

JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and
Authorization Grants
draft-ietf-oauth-jwt-bearer-12

Abstract

This specification defines the use of a JSON Web Token (JWT) Bearer Token as a means for requesting an OAuth 2.0 access token as well as for use as a means of client authentication.

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 May 16, 2015.

Copyright Notice

Copyright (c) 2014 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of

the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
1.1. Notational Conventions	4
1.2. Terminology	4
2. HTTP Parameter Bindings for Transporting Assertions	4
2.1. Using JWTs as Authorization Grants	4
2.2. Using JWTs for Client Authentication	5
3. JWT Format and Processing Requirements	5
3.1. Authorization Grant Processing	7
3.2. Client Authentication Processing	8
4. Authorization Grant Example	8
5. Interoperability Considerations	9
6. Security Considerations	9
7. Privacy Considerations	10
8. IANA Considerations	10
8.1. Sub-Namespace Registration of urn:ietf:params:oauth: :grant-type:jwt-bearer	10
8.2. Sub-Namespace Registration of urn:ietf:params:oauth: :client-assertion-type:jwt-bearer	10
9. References	11
9.1. Normative References	11
9.2. Informative References	11
Appendix A. Acknowledgements	12
Appendix B. Document History	12
Authors' Addresses	14

1. Introduction

JSON Web Token (JWT) [JWT] is a JavaScript Object Notation (JSON) [RFC7159] based security token encoding that enables identity and security information to be shared across security domains. A security token is generally issued by an identity provider and consumed by a relying party that relies on its content to identify the token's subject for security related purposes.

The OAuth 2.0 Authorization Framework [RFC6749] provides a method for making authenticated HTTP requests to a resource using an access token. Access tokens are issued to third-party clients by an authorization server (AS) with the (sometimes implicit) approval of the resource owner. In OAuth, an authorization grant is an abstract term used to describe intermediate credentials that represent the resource owner authorization. An authorization grant is used by the client to obtain an access token. Several authorization grant types are defined to support a wide range of client types and user

experiences. OAuth also allows for the definition of new extension grant types to support additional clients or to provide a bridge between OAuth and other trust frameworks. Finally, OAuth allows the definition of additional authentication mechanisms to be used by clients when interacting with the authorization server.

The Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants [I-D.ietf-oauth-assertions] specification is an abstract extension to OAuth 2.0 that provides a general framework for the use of Assertions (a.k.a. Security Tokens) as client credentials and/or authorization grants with OAuth 2.0. This specification profiles the Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants [I-D.ietf-oauth-assertions] specification to define an extension grant type that uses a JSON Web Token (JWT) Bearer Token to request an OAuth 2.0 access token as well as for use as client credentials. The format and processing rules for the JWT defined in this specification are intentionally similar, though not identical, to those in the closely related SAML 2.0 Profile for OAuth 2.0 Client Authentication and Authorization Grants [I-D.ietf-oauth-saml2-bearer] specification. The differences arise where the structure and semantics of JWTs differ from SAML assertions. JWTs, for example, have no direct equivalent to the <SubjectConfirmation> or <AuthnStatement> elements of SAML assertions.

This document defines how a JSON Web Token (JWT) Bearer Token can be used to request an access token when a client wishes to utilize an existing trust relationship, expressed through the semantics of (and digital signature or Message Authentication Code calculated over) the JWT, without a direct user approval step at the authorization server. It also defines how a JWT can be used as a client authentication mechanism. The use of a security token for client authentication is orthogonal to and separable from using a security token as an authorization grant. They can be used either in combination or separately. Client authentication using a JWT is nothing more than an alternative way for a client to authenticate to the token endpoint and must be used in conjunction with some grant type to form a complete and meaningful protocol request. JWT authorization grants may be used with or without client authentication or identification. Whether or not client authentication is needed in conjunction with a JWT authorization grant, as well as the supported types of client authentication, are policy decisions at the discretion of the authorization server.

The process by which the client obtains the JWT, prior to exchanging it with the authorization server or using it for client authentication, is out of scope.

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

All terms are as defined in The OAuth 2.0 Authorization Framework [RFC6749], the Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants [I-D.ietf-oauth-assertions], and the JSON Web Token (JWT) [JWT] specifications.

2. HTTP Parameter Bindings for Transporting Assertions

The Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants [I-D.ietf-oauth-assertions] specification defines generic HTTP parameters for transporting Assertions (a.k.a. Security Tokens) during interactions with a token endpoint. This section defines specific parameters and treatments of those parameters for use with JWT bearer tokens.

2.1. Using JWTs as Authorization Grants

To use a Bearer JWT as an authorization grant, the client uses an access token request as defined in Section 4 of the Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants [I-D.ietf-oauth-assertions] specification with the following specific parameter values and encodings.

The value of the "grant_type" is "urn:ietf:params:oauth:grant-type:jwt-bearer".

The value of the "assertion" parameter MUST contain a single JWT.

The "scope" parameter may be used, as defined in the Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants [I-D.ietf-oauth-assertions] specification, to indicate the requested scope.

Authentication of the client is optional, as described in Section 3.2.1 of OAuth 2.0 [RFC6749] and consequently, the "client_id" is only needed when a form of client authentication that relies on the parameter is used.

The following example demonstrates an Access Token Request with a JWT as an authorization grant (with extra line breaks for display purposes only):

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

grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Ajwt-bearer
&assertion=eyJhbGciOiJFUzI1NiJ9.
eyJpc3Mi[...omitted for brevity...].
J9l-ZhwP[...omitted for brevity...]
```

2.2. Using JWTs for Client Authentication

To use a JWT Bearer Token for client authentication, the client uses the following parameter values and encodings.

The value of the "client_assertion_type" is
"urn:ietf:params:oauth:client-assertion-type:jwt-bearer".

The value of the "client_assertion" parameter contains a single JWT.
It MUST NOT contain more than one JWT.

The following example demonstrates client authentication using a JWT during the presentation of an authorization code grant in an Access Token Request (with extra line breaks for display purposes only):

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

grant_type=authorization_code&
code=vAZEIHjQTHuGgaSvyW9hO0RpusLzkvTOww3trZBxZpo&
client_assertion_type=urn%3Aietf%3Aparams%3Aoauth%3A
client-assertion-type%3Ajwt-bearer&
client_assertion=eyJhbGciOiJSUzI1NiJ9.
eyJpc3Mi[...omitted for brevity...].
cC4hiUPo[...omitted for brevity...]
```

3. JWT Format and Processing Requirements

In order to issue an access token response as described in OAuth 2.0 [RFC6749] or to rely on a JWT for client authentication, the authorization server MUST validate the JWT according to the criteria below. Application of additional restrictions and policy are at the discretion of the authorization server.

1. The JWT MUST contain an "iss" (issuer) claim that contains a unique identifier for the entity that issued the JWT. In the absence of an application profile specifying otherwise, compliant applications MUST compare Issuer values using the Simple String Comparison method defined in Section 6.2.1 of RFC 3986 [RFC3986].
2. The JWT MUST contain a "sub" (subject) claim identifying the principal that is the subject of the JWT. Two cases need to be differentiated:
 - A. For the authorization grant, the subject typically identifies an authorized accessor for which the access token is being requested (i.e., the resource owner or an authorized delegate), but in some cases, may be a pseudonymous identifier or other value denoting an anonymous user.
 - B. For client authentication, the subject MUST be the "client_id" of the OAuth client.
3. The JWT MUST contain an "aud" (audience) claim containing a value that identifies the authorization server as an intended audience. The token endpoint URL of the authorization server MAY be used as a value for an "aud" element to identify the authorization server as an intended audience of the JWT. The Authorization Server MUST reject any JWT that does not contain its own identity as the intended audience. In the absence of an application profile specifying otherwise, compliant applications MUST compare the audience values using the Simple String Comparison method defined in Section 6.2.1 of RFC 3986 [RFC3986]. As noted in Section 5, the precise strings to be used as the audience for a given Authorization Server must be configured out-of-band by the Authorization Server and the Issuer of the JWT.
4. The JWT MUST contain an "exp" (expiration) claim that limits the time window during which the JWT can be used. The authorization server MUST reject any JWT with an expiration time that has passed, subject to allowable clock skew between systems. Note that the authorization server may reject JWTs with an "exp" claim value that is unreasonably far in the future.
5. The JWT MAY contain an "nbf" (not before) claim that identifies the time before which the token MUST NOT be accepted for processing.

6. The JWT MAY contain an "iat" (issued at) claim that identifies the time at which the JWT was issued. Note that the authorization server may reject JWTs with an "iat" claim value that is unreasonably far in the past.
7. The JWT MAY contain a "jti" (JWT ID) claim that provides a unique identifier for the token. The authorization server MAY ensure that JWTs are not replayed by maintaining the set of used "jti" values for the length of time for which the JWT would be considered valid based on the applicable "exp" instant.
8. The JWT MAY contain other claims.
9. The JWT MUST be digitally signed or have a Message Authentication Code applied by the issuer. The authorization server MUST reject JWTs with an invalid signature or Message Authentication Code.
10. The authorization server MUST reject a JWT that is not valid in all other respects per JSON Web Token (JWT) [JWT].

3.1. Authorization Grant Processing

JWT authorization grants may be used with or without client authentication or identification. Whether or not client authentication is needed in conjunction with a JWT authorization grant, as well as the supported types of client authentication, are policy decisions at the discretion of the authorization server. However, if client credentials are present in the request, the authorization server MUST validate them.

If the JWT is not valid, or the current time is not within the token's valid time window for use, the authorization server constructs an error response as defined in OAuth 2.0 [RFC6749]. The value of the "error" parameter MUST be the "invalid_grant" error code. The authorization server MAY include additional information regarding the reasons the JWT was considered invalid using the "error_description" or "error_uri" parameters.

For example:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Cache-Control: no-store

{
  "error": "invalid_grant",
  "error_description": "Audience validation failed"
}
```

3.2. Client Authentication Processing

If the client JWT is not valid, the authorization server constructs an error response as defined in OAuth 2.0 [RFC6749]. The value of the "error" parameter MUST be the "invalid_client" error code. The authorization server MAY include additional information regarding the reasons the JWT was considered invalid using the "error_description" or "error_uri" parameters.

4. Authorization Grant Example

The following examples illustrate what a conforming JWT and an access token request would look like.

The example shows a JWT issued and signed by the system entity identified as "https://jwt-idp.example.com". The subject of the JWT is identified by email address as "mike@example.com". The intended audience of the JWT is "https://jwt-rp.example.net", which is an identifier with which the authorization server identifies itself. The JWT is sent as part of an access token request to the authorization server's token endpoint at "https://authz.example.net/token.oauth2".

Below is an example JSON object that could be encoded to produce the JWT Claims Object for a JWT:

```
{
  "iss": "https://jwt-idp.example.com",
  "sub": "mailto:mike@example.com",
  "aud": "https://jwt-rp.example.net",
  "nbf": 1300815780,
  "exp": 1300819380,
  "http://claims.example.com/member": true
}
```

The following example JSON object, used as the header of a JWT, declares that the JWT is signed with the ECDSA P-256 SHA-256 algorithm.

```
{"alg": "ES256"}
```

To present the JWT with the claims and header shown in the previous example as part of an access token request, for example, the client might make the following HTTPS request (with extra line breaks for display purposes only):

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

grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Ajwt-bearer
&assertion=eyJhbGciOiJIUzI1NiJ9.
eyJpc3MiOiI...omitted for brevity...
J9l-ZhwP[...omitted for brevity...]
```

5. Interoperability Considerations

Agreement between system entities regarding identifiers, keys, and endpoints is required in order to achieve interoperable deployments of this profile. Specific items that require agreement are as follows: values for the issuer and audience identifiers, the location of the token endpoint, the key used to apply and verify the digital signature or Message Authentication Code over the JWT, one-time use restrictions on the JWT, maximum JWT lifetime allowed, and the specific subject and claim requirements of the JWT. The exchange of such information is explicitly out of scope for this specification. In some cases, additional profiles may be created that constrain or prescribe these values or specify how they are to be exchanged. Examples of such profiles include the OAuth 2.0 Dynamic Client Registration Core Protocol [I-D.ietf-oauth-dyn-reg], OpenID Connect Dynamic Client Registration 1.0 [OpenID.Registration], and OpenID Connect Discovery 1.0 [OpenID.Discovery].

The "RS256" algorithm, from [I-D.ietf-jose-json-web-algorithms], is a mandatory to implement JSON Web Signature algorithm for this profile.

6. Security Considerations

The security considerations described within the Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants [I-D.ietf-oauth-assertions], The OAuth 2.0 Authorization Framework [RFC6749], and the JSON Web Token (JWT) [JWT] specifications are all applicable to this document.

The specification does not mandate replay protection for the JWT usage for either the authorization grant or for client authentication. It is an optional feature, which implementations may employ at their own discretion.

7. Privacy Considerations

A JWT may contain privacy-sensitive information and, to prevent disclosure of such information to unintended parties, should only be transmitted over encrypted channels, such as TLS. In cases where it is desirable to prevent disclosure of certain information to the client, the JWT should be encrypted to the authorization server.

Deployments should determine the minimum amount of information necessary to complete the exchange and include only such claims in the JWT. In some cases, the "sub" (subject) claim can be a value representing an anonymous or pseudonymous user, as described in Section 6.3.1 of the Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants [I-D.ietf-oauth-assertions].

8. IANA Considerations

8.1. Sub-Namespace Registration of urn:ietf:params:oauth:grant-type:jwt-bearer

This specification registers the value "grant-type:jwt-bearer" in the IANA urn:ietf:params:oauth registry established in An IETF URN Sub-Namespace for OAuth [RFC6755].

- o URN: urn:ietf:params:oauth:grant-type:jwt-bearer
- o Common Name: JWT Bearer Token Grant Type Profile for OAuth 2.0
- o Change controller: IESG
- o Specification Document: [[this document]]

8.2. Sub-Namespace Registration of urn:ietf:params:oauth:client-assertion-type:jwt-bearer

This specification registers the value "client-assertion-type:jwt-bearer" in the IANA urn:ietf:params:oauth registry established in An IETF URN Sub-Namespace for OAuth [RFC6755].

- o URN: urn:ietf:params:oauth:client-assertion-type:jwt-bearer
- o Common Name: JWT Bearer Token Profile for OAuth 2.0 Client Authentication

- o Change controller: IESG
- o Specification Document: [[this document]]

9. References

9.1. Normative References

- [I-D.ietf-jose-json-web-algorithms]
Jones, M., "JSON Web Algorithms (JWA)", draft-ietf-jose-json-web-algorithms-36 (work in progress), October 2014.
- [I-D.ietf-oauth-assertions]
Campbell, B., Mortimore, C., Jones, M., and Y. Goland, "Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants", draft-ietf-oauth-assertions (work in progress), October 2014.
- [JWT] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", draft-ietf-oauth-json-web-token (work in progress), October 2014.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.
- [RFC6749] Hardt, D., "The OAuth 2.0 Authorization Framework", RFC 6749, October 2012.
- [RFC7159] Bray, T., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, March 2014.

9.2. Informative References

- [I-D.ietf-oauth-dyn-reg]
Richer, J., Jones, M., Bradley, J., Machulak, M., and P. Hunt, "OAuth 2.0 Dynamic Client Registration Protocol", draft-ietf-oauth-dyn-reg-20 (work in progress), August 2014.
- [I-D.ietf-oauth-saml2-bearer]
Campbell, B., Mortimore, C., and M. Jones, "SAML 2.0 Profile for OAuth 2.0 Client Authentication and Authorization Grants", draft-ietf-oauth-saml2-bearer (work in progress), November 2014.

[OpenID.Discovery]

Sakimura, N., Bradley, J., Jones, M., and E. Jay, "OpenID Connect Discovery 1.0", February 2014.

[OpenID.Registration]

Sakimura, N., Bradley, J., and M. Jones, "OpenID Connect Dynamic Client Registration 1.0", February 2014.

[RFC6755] Campbell, B. and H. Tschofenig, "An IETF URN Sub-Namespace for OAuth", RFC 6755, October 2012.

Appendix A. Acknowledgements

This profile was derived from SAML 2.0 Profile for OAuth 2.0 Client Authentication and Authorization Grants [I-D.ietf-oauth-saml2-bearer] by Brian Campbell and Chuck Mortimore.

Appendix B. Document History

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

draft-ietf-oauth-jwt-bearer-12

- o Fix typo per <http://www.ietf.org/mail-archive/web/oauth/current/msg13790.html>

draft-ietf-oauth-jwt-bearer-11

- o Changes/suggestions from IESG reviews.

draft-ietf-oauth-jwt-bearer-10

- o Added Privacy Considerations section per AD review discussion <http://www.ietf.org/mail-archive/web/oauth/current/msg13148.html> and <http://www.ietf.org/mail-archive/web/oauth/current/msg13144.html>

draft-ietf-oauth-jwt-bearer-09

- o Clarified some text around the treatment of subject based on the rough rough consensus from the thread staring at <http://www.ietf.org/mail-archive/web/oauth/current/msg12630.html>

draft-ietf-oauth-jwt-bearer-08

- o Updated references, including replacing references to RFC 4627 with RFC 7159.

draft-ietf-oauth-jwt-bearer-07

- o Clean up language around subject per <http://www.ietf.org/mail-archive/web/oauth/current/msg12250.html>.
- o As suggested in <http://www.ietf.org/mail-archive/web/oauth/current/msg12251.html> stated that "In the absence of an application profile specifying otherwise, compliant applications MUST compare the audience values using the Simple String Comparison method defined in Section 6.2.1 of RFC 3986."
- o Added one-time use, maximum lifetime, and specific subject and attribute requirements to Interoperability Considerations based on <http://www.ietf.org/mail-archive/web/oauth/current/msg12252.html>.
- o Remove "or its subject confirmation requirements cannot be met" text.
- o Reword security considerations and mention that replay protection is not mandated based on <http://www.ietf.org/mail-archive/web/oauth/current/msg12259.html>.

-06

- o Stated that issuer and audience values SHOULD be compared using the Simple String Comparison method defined in Section 6.2.1 of RFC 3986 unless otherwise specified by the application.

-05

- o Changed title from "JSON Web Token (JWT) Bearer Token Profiles for OAuth 2.0" to "JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants" to be more explicit about the scope of the document per <http://www.ietf.org/mail-archive/web/oauth/current/msg11063.html>.
- o Numbered the list of processing rules.
- o Smallish editorial cleanups to try and improve readability and comprehensibility.
- o Cleaner split out of the processing rules in cases where they differ for client authentication and authorization grants.
- o Clarified the parameters that are used/available for authorization grants.
- o Added Interoperability Considerations section.

- o Added more explanatory context to the example in Section 4.

-04

- o Changed the name of the "prn" claim to "sub" (subject) both to more closely align with SAML name usage and to use a more intuitive name.
- o Added seriesInfo information to Internet Draft references.

-03

- o Reference RFC 6749 and RFC 6755.

-02

- o Add more text to intro explaining that an assertion/JWT grant type can be used with or without client authentication/identification and that client assertion/JWT authentication is nothing more than an alternative way for a client to authenticate to the token endpoint
- o Add examples to Sections 2.1 and 2.2
- o Update references

-01

- o Tracked specification name changes: "The OAuth 2.0 Authorization Protocol" to "The OAuth 2.0 Authorization Framework" and "OAuth 2.0 Assertion Profile" to "Assertion Framework for OAuth 2.0".
- o Merged in changes between draft-ietf-oauth-saml2-bearer-11 and draft-ietf-oauth-saml2-bearer-13. All changes were strictly editorial.

-00

- o Created the initial IETF draft based upon draft-jones-oauth-jwt-bearer-04 with no normative changes.

Authors' Addresses

Michael B. Jones
Microsoft

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

Brian Campbell
Ping Identity

Email: brian.d.campbell@gmail.com

Chuck Mortimore
Salesforce

Email: cmortimore@salesforce.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.

Copyright Notice

Copyright (c) 2016 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction	3
2. Terminology	3
3. Use Cases	3
3.1. Preventing Access Token Re-Use by the Resource Server . .	4
3.2. TLS and DTLS Channel Binding Support	4
3.3. Access to a Non-TLS Protected Resource	4
3.4. Offering Application Layer End-to-End Security	5
4. Security and Privacy Threats	5
5. Requirements	6
6. Threat Mitigation	10
6.1. Confidentiality Protection	11
6.2. Sender Constraint	11
6.3. Key Confirmation	12
6.4. Summary	13
7. Architecture	14
7.1. Client and Authorization Server Interaction	15
7.1.1. Symmetric Keys	15
7.1.2. Asymmetric Keys	16
7.2. Client and Resource Server Interaction	17
7.3. Resource and Authorization Server Interaction (Token Introspection)	18
8. Security Considerations	19
9. IANA Considerations	19
10. Acknowledgments	19
11. References	20
11.1. Normative References	20
11.2. Informative References	21
Authors' Addresses	22

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

- o security properties
- o available infrastructure
- o library support
- o credential cost (financial)
- o performance
- o integration into the existing IT infrastructure
- o 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 $K(\text{Authorization Server-Resource Server})$. 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 K_s and places it into the token encrypted with the long term key $K(\text{Authorization Server-Resource Server})$. Additionally, the authorization server attaches K_s 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 K_s 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 $K(\text{Authorization Server-Resource Server})$ to obtain K_s . This key K_s 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

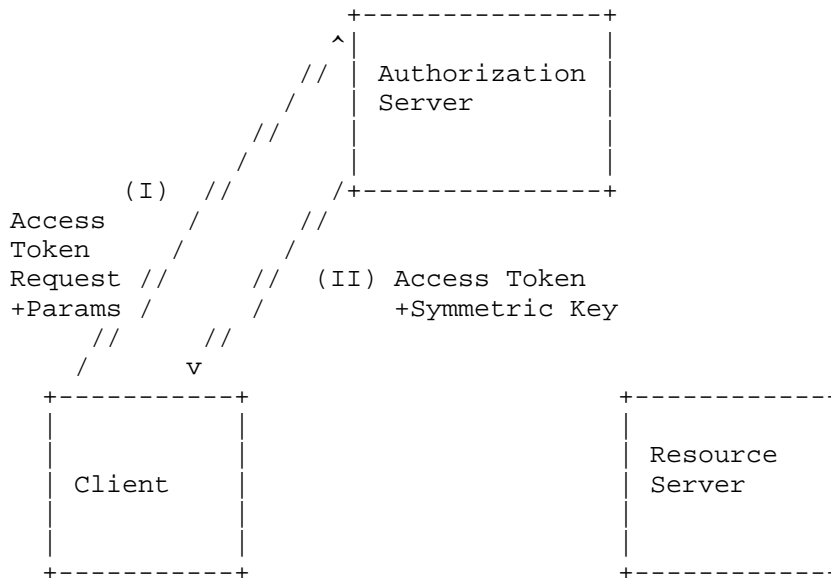


Figure 1: Interaction between the Client and the Authorization Server (Symmetric Keys).

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

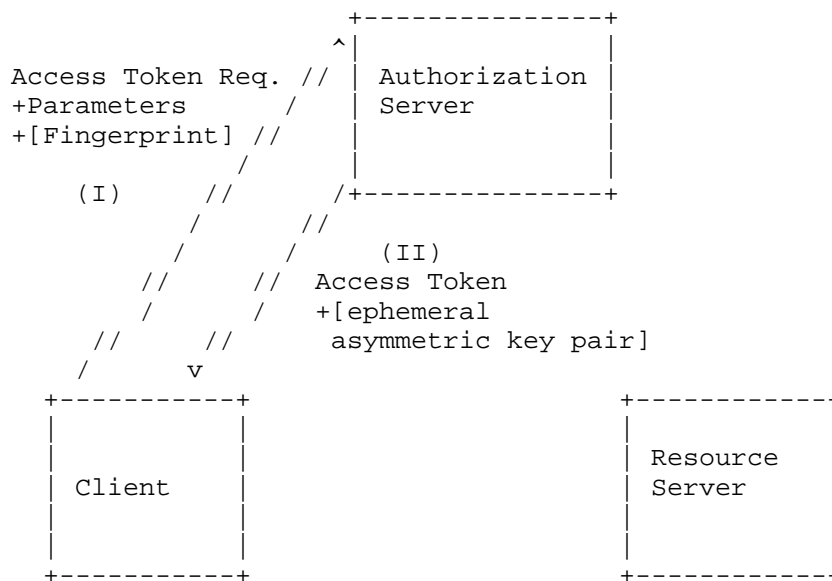


Figure 2: Interaction between the Client and the Authorization Server (Asymmetric Keys).

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.

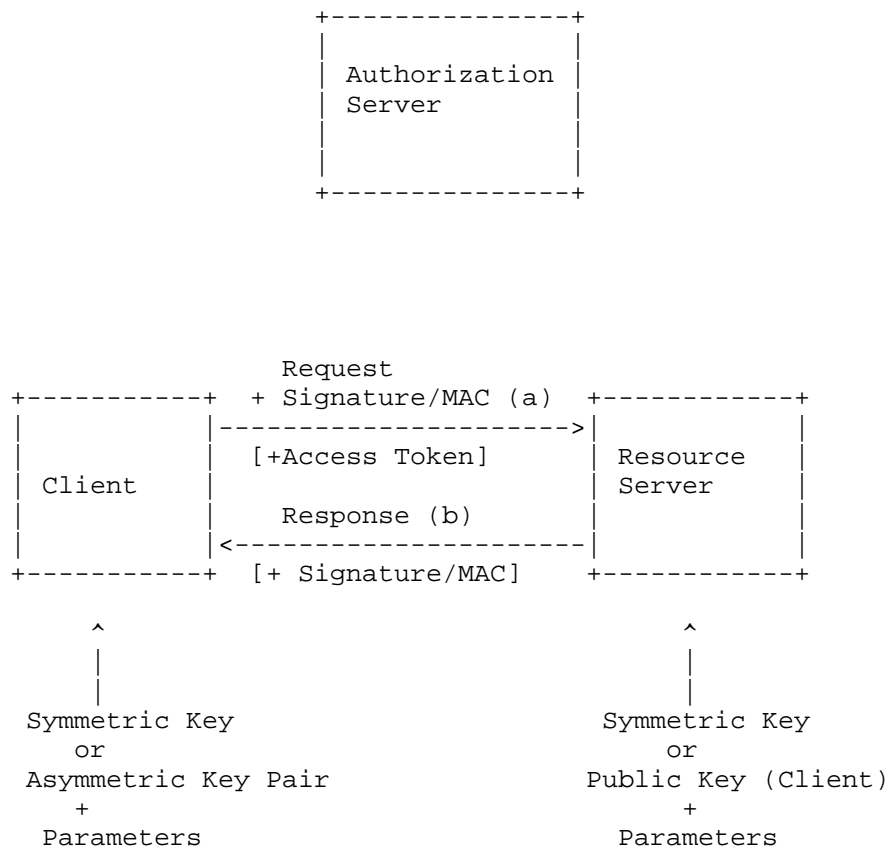


Figure 3: Client Demonstrates PoP.

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.

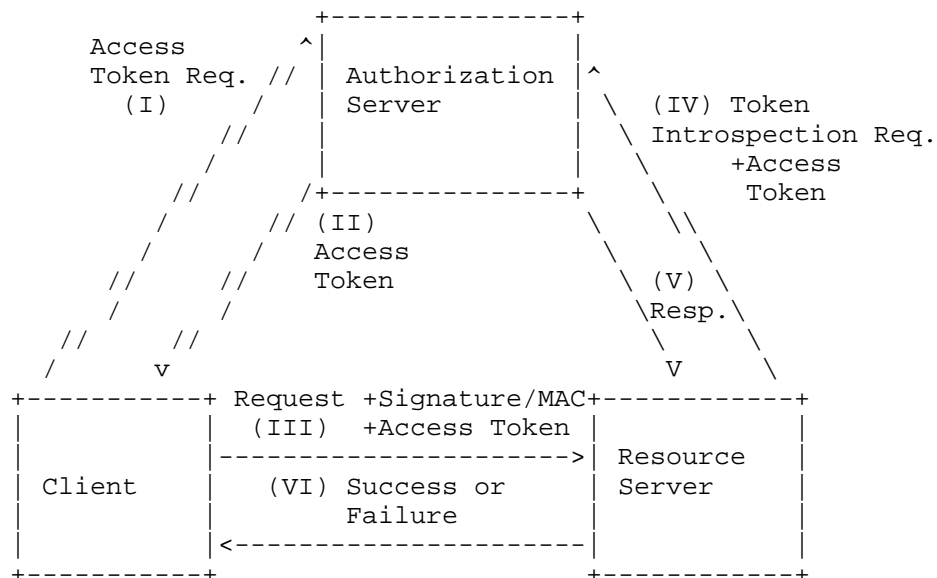


Figure 4: Token Introspection and Access Token Handles.

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: September 28, 2019

J. Bradley
Ping Identity
P. Hunt
Oracle Corporation
M. Jones
Microsoft
H. Tschofenig
Arm Ltd.
M. Meszaros
GITDA
March 27, 2019

OAuth 2.0 Proof-of-Possession: Authorization Server to Client Key
Distribution
draft-ietf-oauth-pop-key-distribution-07

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 September 28, 2019.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction	2
2. Terminology	4
3. Processing Instructions	4
4. Examples	5
4.1. Symmetric Key Transport	5
4.1.1. Client-to-AS Request	5
4.1.2. Client-to-AS Response	6
4.2. Asymmetric Key Transport	9
4.2.1. Client-to-AS Request	9
4.2.2. Client-to-AS Response	10
5. Security Considerations	11
6. IANA Considerations	13
6.1. OAuth Access Token Types	13
6.2. OAuth Parameters Registration	13
6.3. OAuth Extensions Error Registration	13
7. Acknowledgements	13
8. References	14
8.1. Normative References	14
8.2. Informative References	15
Authors' Addresses	16

1. Introduction

The work on proof-of-possession tokens, an extended token security mechanisms for OAuth 2.0, is motivated in [22]. 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.

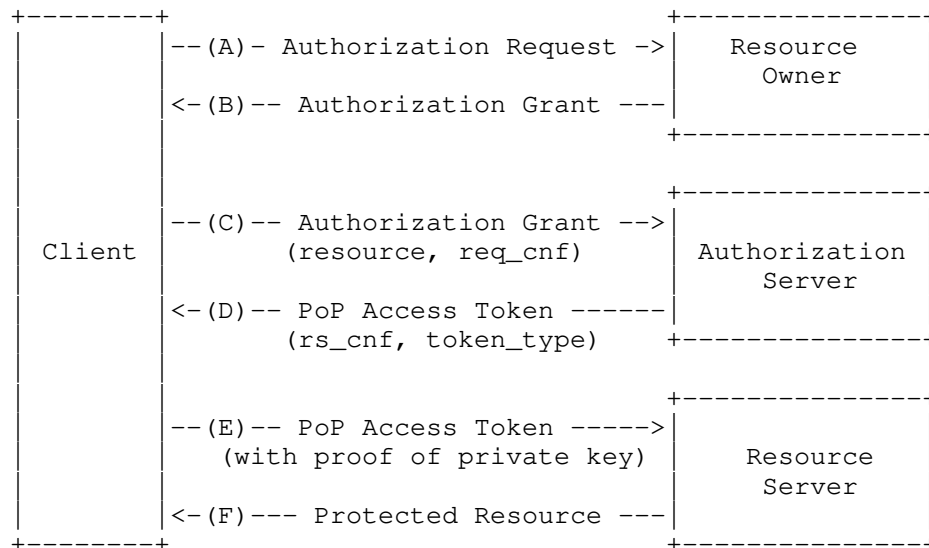


Figure 1: Augmented OAuth 2.0 Protocol Flow

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 [19] 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 [24].

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 identifier parameter, defined in [16], or the audience parameter, defined in [15], when symmetric PoP tokens are used. For asymmetric PoP tokens the use of resource indicators and audience is optional but

RECOMMENDED. The parameters 'audience' and 'resource' both allow the client to express the location of the target service and the difference between the two is described in [15]. As a summary, 'audience' allows expressing a logical name while 'resource' contains an absolute URI. More details about the 'resource' parameter can be found in [16].

Step (3): 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 [23].

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://resource.example.com`


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

grant_type=authorization_code
&code=Sp1xl0BeZQQYbYS6WxSbIA
&scope=calendar%20contacts
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
&resource=https%3A%2F%2Fresource.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":
      "eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkJExMjhdQkMtSFMyNTYifQ.
      (remainder of JWE omitted for brevity)"
    }
  }
}

```

Figure 3: Example: Encrypted Symmetric 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",
  "exp": 1311281970,
  "iat": 1311280970,
  "cnf":{
    "jwe":
      "eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkJExMjhdQkMtSFMyNTYifQ.
      (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 [22].

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=Sp1x10BeZQQYbYS6WxSbIA
&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....jrlzCsicMWpAA",
  "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": "https://authz.example.com",
  "aud": "https://resource.example.com",
  "exp": "1361398824",
  "nbf": "1360189224",
  "cnf": {
    "jwk" : {
      "kty" : "EC",
      "crv" : "P-256",
      "x" : "usWxHK2PmfHnHKwXPS54m0kTcGJ90UiglWiGahtagnv8",
      "y" : "IBOL+C3BttVivg+1SreASjpkttcsz+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

[22] 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 [17] 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" [25] established by [17].

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

6.2. OAuth Parameters Registration

This specification registers the following value in the IANA "OAuth Parameters" registry [25] established by [2].

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

6.3. OAuth Extensions Error Registration

This specification registers the following error in the IANA "OAuth Extensions Error Registry" [25] established by [2].

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

7. Acknowledgements

We would like to thank Chuck Mortimore and James Manger for their 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-06 (work in progress), February 2019.
- [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] Jones, M., Nadalin, A., Campbell, B., Bradley, J., and C. Mortimore, "OAuth 2.0 Token Exchange", draft-ietf-oauth-token-exchange-16 (work in progress), October 2018.
- [16] Campbell, B., Bradley, J., and H. Tschofenig, "Resource Indicators for OAuth 2.0", draft-ietf-oauth-resource-indicators-02 (work in progress), January 2019.

8.2. Informative References

- [17] 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>>.
- [18] 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>>.
- [19] 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>>.
- [20] 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>>.

- [21] 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>>.
- [22] 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.
- [23] Richer, J., Ed., "OAuth 2.0 Token Introspection", RFC 7662, DOI 10.17487/RFC7662, October 2015, <<https://www.rfc-editor.org/info/rfc7662>>.
- [24] 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-24 (work in progress), March 2019.
- [25] IANA, "OAuth Parameters", October 2018.
- [26] 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.independentid.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>

Mihaly Meszaros
GITDA
Debrecen 4033
Hungary

Email: bakfitty@gmail.com
URI: <https://github.com/misi>

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

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

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect

to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
1.1. Notational Conventions	5
2. Terminology	5
3. Representations for Proof-of-Possession Keys	6
3.1. Confirmation Claim	6
3.2. Representation of an Asymmetric Proof-of-Possession Key	7
3.3. Representation of an Encrypted Symmetric Proof-of-Possession Key	8
3.4. Representation of a Key ID for a Proof-of-Possession Key	9
3.5. Representation of a URL for a Proof-of-Possession Key	9
3.6. Specifics Intentionally Not Specified	10
4. Security Considerations	10
5. Privacy Considerations	11
6. IANA Considerations	11
6.1. JSON Web Token Claims Registration	12
6.1.1. Registry Contents	12
6.2. JWT Confirmation Methods Registry	12
6.2.1. Registration Template	12
6.2.2. Initial Registry Contents	13
7. References	13
7.1. Normative References	13
7.2. Informative References	14
Appendix A. Acknowledgements	15
Appendix B. Document History	15
Authors' Addresses	17

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.

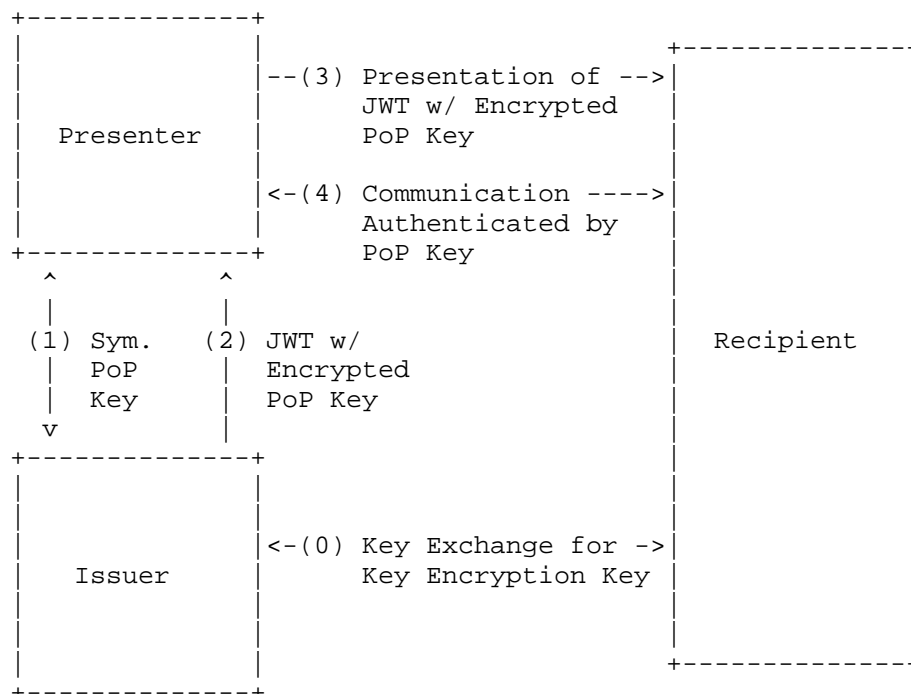


Figure 1: Proof-of-Possession with a Symmetric Key

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.

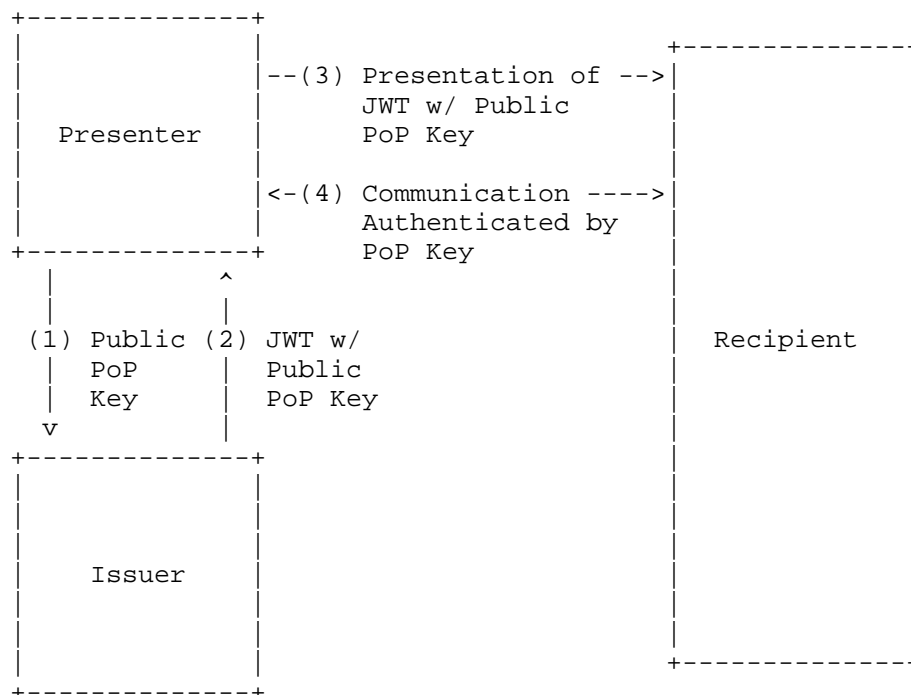


Figure 2: Proof-of-Possession with an Asymmetric Key

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": "18wHLeIgW9wVN6VD1Txgppy2LszYkMf6J8njVAibvhM",
      "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_FzUA5XKMYoVHyzzff5oRJxl-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":
      "eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkJExMjhdQkMtSFMyNTYifQ.
      (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": "dfdlaa97-6d8d-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

- o Claim Name: "cnf"
- o Claim Description: Confirmation
- o Change Controller: IESG
- o 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

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

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

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

- o Confirmation Method Value: "jku"
- o Confirmation Method Description: JWK Set URL
- o Change Controller: IESG
- o 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

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

-10

- o Addressed ballot comments by Barry Leiba.

-09

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

-08

- o Added security consideration about also utilizing audience restriction.

-07

- o Addressed review comments by Hannes Tschofenig, Kathleen Moriarty, and Justin Richer. Changes were:
- o Clarified that symmetric proof-of-possession keys can be generated by either the presenter or the issuer.
- o Clarified that confirmation members that are not understood must be ignored unless otherwise specified by the application.

-06

- o Added diagrams to the introduction.

-05

- o Addressed review comments by Kepeng Li.

-04

- o Allowed the use of "jwk" for symmetric keys when the JWT is encrypted.
- o Added the "jku" (JWK Set URL) member.
- o Added privacy considerations.
- o Reordered sections so that the "cnf" (confirmation) claim is defined before it is used.
- o 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".
- o Applied wording clarifications suggested by Nat Sakimura.

-03

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

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

-02

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

-01

- o Updated references.

-00

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

Authors' Addresses

Michael B. Jones
Microsoft

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

John Bradley
Ping Identity

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

Hannes Tschofenig
ARM Limited
Austria

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

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: May 16, 2015

B. Campbell
Ping Identity
C. Mortimore
Salesforce
M. Jones
Microsoft
November 12, 2014

SAML 2.0 Profile for OAuth 2.0 Client Authentication and Authorization
Grants
draft-ietf-oauth-saml2-bearer-23

Abstract

This specification defines the use of a Security Assertion Markup Language (SAML) 2.0 Bearer Assertion as a means for requesting an OAuth 2.0 access token as well as for use as a means of client authentication.

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 May 16, 2015.

Copyright Notice

Copyright (c) 2014 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must

include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
1.1. Notational Conventions	4
1.2. Terminology	4
2. HTTP Parameter Bindings for Transporting Assertions	4
2.1. Using SAML Assertions as Authorization Grants	4
2.2. Using SAML Assertions for Client Authentication	5
3. Assertion Format and Processing Requirements	6
3.1. Authorization Grant Processing	8
3.2. Client Authentication Processing	9
4. Authorization Grant Example	9
5. Interoperability Considerations	11
6. Security Considerations	11
7. Privacy Considerations	12
8. IANA Considerations	12
8.1. Sub-Namespace Registration of urn:ietf:params:oauth: :grant-type:saml2-bearer	12
8.2. Sub-Namespace Registration of urn:ietf:params:oauth: :client-assertion-type:saml2-bearer	12
9. References	13
9.1. Normative References	13
9.2. Informative References	14
Appendix A. Acknowledgements	14
Appendix B. Document History	15
Authors' Addresses	21

1. Introduction

The Security Assertion Markup Language (SAML) 2.0 [OASIS.saml-core-2.0-os] is an XML-based framework that allows identity and security information to be shared across security domains. The SAML specification, while primarily targeted at providing cross domain Web browser single sign-on, was also designed to be modular and extensible to facilitate use in other contexts.

The Assertion, an XML security token, is a fundamental construct of SAML that is often adopted for use in other protocols and specifications. (Some examples include [OASIS.WSS-SAMLTokenProfile] and [OASIS.WS-Fed].) An Assertion is generally issued by an identity provider and consumed by a service provider who relies on its content to identify the Assertion's subject for security related purposes.

The OAuth 2.0 Authorization Framework [RFC6749] provides a method for making authenticated HTTP requests to a resource using an access token. Access tokens are issued to third-party clients by an authorization server (AS) with the (sometimes implicit) approval of the resource owner. In OAuth, an authorization grant is an abstract term used to describe intermediate credentials that represent the resource owner authorization. An authorization grant is used by the client to obtain an access token. Several authorization grant types are defined to support a wide range of client types and user experiences. OAuth also allows for the definition of new extension grant types to support additional clients or to provide a bridge between OAuth and other trust frameworks. Finally, OAuth allows the definition of additional authentication mechanisms to be used by clients when interacting with the authorization server.

The Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants [I-D.ietf-oauth-assertions] specification is an abstract extension to OAuth 2.0 that provides a general framework for the use of Assertions as client credentials and/or authorization grants with OAuth 2.0. This specification profiles the Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants [I-D.ietf-oauth-assertions] specification to define an extension grant type that uses a SAML 2.0 Bearer Assertion to request an OAuth 2.0 access token as well as for use as client credentials. The format and processing rules for the SAML Assertion defined in this specification are intentionally similar, though not identical, to those in the Web Browser SSO Profile defined in the SAML Profiles [OASIS.saml-profiles-2.0-os] specification. This specification is reusing, to the extent reasonable, concepts and patterns from that well-established Profile.

This document defines how a SAML Assertion can be used to request an access token when a client wishes to utilize an existing trust relationship, expressed through the semantics of (and digital signature or keyed message digest calculated over) the SAML Assertion, without a direct user approval step at the authorization server. It also defines how a SAML Assertion can be used as a client authentication mechanism. The use of an Assertion for client authentication is orthogonal to and separable from using an Assertion as an authorization grant. They can be used either in combination or separately. Client assertion authentication is nothing more than an alternative way for a client to authenticate to the token endpoint and must be used in conjunction with some grant type to form a complete and meaningful protocol request. Assertion authorization grants may be used with or without client authentication or identification. Whether or not client authentication is needed in conjunction with an assertion authorization grant, as well as the

supported types of client authentication, are policy decisions at the discretion of the authorization server.

The process by which the client obtains the SAML Assertion, prior to exchanging it with the authorization server or using it for client authentication, is out of scope.

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

All terms are as defined in The OAuth 2.0 Authorization Framework [RFC6749], the Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants [I-D.ietf-oauth-assertions], and the Security Assertion Markup Language (SAML) 2.0 [OASIS.saml-core-2.0-os] specifications.

2. HTTP Parameter Bindings for Transporting Assertions

The Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants [I-D.ietf-oauth-assertions] specification defines generic HTTP parameters for transporting Assertions during interactions with a token endpoint. This section defines specific parameters and treatments of those parameters for use with SAML 2.0 Bearer Assertions.

2.1. Using SAML Assertions as Authorization Grants

To use a SAML Bearer Assertion as an authorization grant, the client uses an access token request as defined in Section 4 of the Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants [I-D.ietf-oauth-assertions] specification with the following specific parameter values and encodings.

The value of the "grant_type" parameter is "urn:ietf:params:oauth:grant-type:saml2-bearer".

The value of the "assertion" parameter contains a single SAML 2.0 Assertion. It MUST NOT contain more than one SAML 2.0 assertion. The SAML Assertion XML data MUST be encoded using base64url, where the encoding adheres to the definition in Section 5 of RFC 4648

[RFC4648] and where the padding bits are set to zero. To avoid the need for subsequent encoding steps (by "application/x-www-form-urlencoded" [W3C.REC-html401-19991224], for example), the base64url encoded data MUST NOT be line wrapped and pad characters ("=") MUST NOT be included.

The "scope" parameter may be used, as defined in the Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants [I-D.ietf-oauth-assertions] specification, to indicate the requested scope.

Authentication of the client is optional, as described in Section 3.2.1 of OAuth 2.0 [RFC6749] and consequently, the "client_id" is only needed when a form of client authentication that relies on the parameter is used.

The following example demonstrates an Access Token Request with an assertion as an authorization grant (with extra line breaks for display purposes only):

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

grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Asaml2-bearer&
assertion=PHNhbWxwOl...[omitted for brevity]...ZT4
```

2.2. Using SAML Assertions for Client Authentication

To use a SAML Bearer Assertion for client authentication, the client uses the following parameter values and encodings.

The value of the "client_assertion_type" parameter is "urn:ietf:params:oauth:client-assertion-type:saml2-bearer".

The value of the "client_assertion" parameter MUST contain a single SAML 2.0 Assertion. The SAML Assertion XML data MUST be encoded using base64url, where the encoding adheres to the definition in Section 5 of RFC 4648 [RFC4648] and where the padding bits are set to zero. To avoid the need for subsequent encoding steps (by "application/x-www-form-urlencoded" [W3C.REC-html401-19991224], for example), the base64url encoded data SHOULD NOT be line wrapped and pad characters ("=") SHOULD NOT be included.

The following example demonstrates a client authenticating using an assertion during the presentation of an authorization code grant in an Access Token Request (with extra line breaks for display purposes only):

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

grant_type=authorization_code&
code=vAZEIHjQTHuGgaSvyW9hO0RpusLzkvTOww3trZBxZpo&
client_assertion_type=urn%3Aietf%3Aparams%3Aoauth
%3Aclient-assertion-type%3Asaml2-bearer&
client_assertion=PHNhbW...[omitted for brevity]...ZT
```

3. Assertion Format and Processing Requirements

In order to issue an access token response as described in OAuth 2.0 [RFC6749] or to rely on an Assertion for client authentication, the authorization server MUST validate the Assertion according to the criteria below. Application of additional restrictions and policy are at the discretion of the authorization server.

1. The Assertion's <Issuer> element MUST contain a unique identifier for the entity that issued the Assertion. In the absence of an application profile specifying otherwise, compliant applications MUST compare Issuer values using the Simple String Comparison method defined in Section 6.2.1 of RFC 3986 [RFC3986].
2. The Assertion MUST contain a <Conditions> element with an <AudienceRestriction> element with an <Audience> element that identifies the authorization server as an intended audience. Section 2.5.1.4 of Assertions and Protocols for the OASIS Security Assertion Markup Language [OASIS.saml-core-2.0-os] defines the <AudienceRestriction> and <Audience> elements and, in addition to the URI references discussed there, the token endpoint URL of the authorization server MAY be used as a URI that identifies the authorization server as an intended audience. The Authorization Server MUST reject any assertion that does not contain its own identity as the intended audience. In the absence of an application profile specifying otherwise, compliant applications MUST compare the audience values using the Simple String Comparison method defined in Section 6.2.1 of RFC 3986 [RFC3986]. As noted in Section 5, the precise strings to be used as the audience for a given Authorization Server must be configured out-of-band by the Authorization Server and the Issuer of the assertion.
3. The Assertion MUST contain a <Subject> element identifying the principal that is the subject of the Assertion. Additional information identifying the subject/principal MAY be included in an <AttributeStatement>.

- A. For the authorization grant, the Subject typically identifies an authorized accessor for which the access token is being requested (i.e., the resource owner or an authorized delegate), but in some cases, may be a pseudonymous identifier or other value denoting an anonymous user.
 - B. For client authentication, the Subject MUST be the "client_id" of the OAuth client.
- 4. The Assertion MUST have an expiry that limits the time window during which it can be used. The expiry can be expressed either as the NotOnOrAfter attribute of the <Conditions> element or as the NotOnOrAfter attribute of a suitable <SubjectConfirmationData> element.
 - 5. The <Subject> element MUST contain at least one <SubjectConfirmation> element that has a Method attribute with a value of "urn:oasis:names:tc:SAML:2.0:cm:bearer". If the Assertion does not have a suitable NonOnOrAfter attribute on the <Conditions> element, the <SubjectConfirmation> element MUST contain a <SubjectConfirmationData> element. When present, the <SubjectConfirmationData> element MUST have a Recipient attribute with a value indicating the token endpoint URL of the authorization server (or an acceptable alias). The authorization server MUST verify that the value of the Recipient attribute matches the token endpoint URL (or an acceptable alias) to which the Assertion was delivered. The <SubjectConfirmationData> element MUST have a NotOnOrAfter attribute that limits the window during which the Assertion can be confirmed. The <SubjectConfirmationData> element MAY also contain an Address attribute limiting the client address from which the Assertion can be delivered. Verification of the Address is at the discretion of the authorization server.
 - 6. The authorization server MUST reject the entire Assertion if the NotOnOrAfter instant on the <Conditions> element has passed (subject to allowable clock skew between systems). The authorization server MUST reject the <SubjectConfirmation> (but MAY still use the rest of the Assertion) if the NotOnOrAfter instant on the <SubjectConfirmationData> has passed (subject to allowable clock skew). Note that the authorization server may reject Assertions with a NotOnOrAfter instant that is unreasonably far in the future. The authorization server MAY ensure that Bearer Assertions are not replayed, by maintaining the set of used ID values for the length of time for which the Assertion would be considered valid based on the applicable NotOnOrAfter instant.

7. If the Assertion issuer directly authenticated the subject, the Assertion SHOULD contain a single <AuthnStatement> representing that authentication event. If the Assertion was issued with the intention that the client act autonomously on behalf of the subject, an <AuthnStatement> SHOULD NOT be included and the client presenting the assertion SHOULD be identified in the <NameID> or similar element in the <SubjectConfirmation> element, or by other available means like SAML V2.0 Condition for Delegation Restriction [OASIS.saml-deleg-cs].
8. Other statements, in particular <AttributeStatement> elements, MAY be included in the Assertion.
9. The Assertion MUST be digitally signed or have a Message Authentication Code applied by the issuer. The authorization server MUST reject assertions with an invalid signature or Message Authentication Code.
10. Encrypted elements MAY appear in place of their plain text counterparts as defined in [OASIS.saml-core-2.0-os].
11. The authorization server MUST reject an Assertion that is not valid in all other respects per [OASIS.saml-core-2.0-os], such as (but not limited to) all content within the Conditions element including the NotOnOrAfter and NotBefore attributes, unknown condition types, etc.

3.1. Authorization Grant Processing

Assertion authorization grants may be used with or without client authentication or identification. Whether or not client authentication is needed in conjunction with an assertion authorization grant, as well as the supported types of client authentication, are policy decisions at the discretion of the authorization server. However, if client credentials are present in the request, the authorization server MUST validate them.

If the Assertion is not valid (including if its subject confirmation requirements cannot be met), the authorization server constructs an error response as defined in OAuth 2.0 [RFC6749]. The value of the "error" parameter MUST be the "invalid_grant" error code. The authorization server MAY include additional information regarding the reasons the Assertion was considered invalid using the "error_description" or "error_uri" parameters.

For example:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Cache-Control: no-store

{
  "error": "invalid_grant",
  "error_description": "Audience validation failed"
}
```

3.2. Client Authentication Processing

If the client Assertion is not valid (including if its subject confirmation requirements cannot be met), the authorization server constructs an error response as defined in OAuth 2.0 [RFC6749]. The value of the "error" parameter MUST be the "invalid_client" error code. The authorization server MAY include additional information regarding the reasons the Assertion was considered invalid using the "error_description" or "error_uri" parameters.

4. Authorization Grant Example

The following examples illustrate what a conforming Assertion and an access token request would look like.

The example shows an assertion issued and signed by the SAML Identity Provider identified as "https://saml-idp.example.com". The subject of the assertion is identified by email address as "brian@example.com", who authenticated to the Identity Provider by means of a digital signature where the key was validated as part of an X.509 Public Key Infrastructure. The intended audience of the assertion is "https://saml-sp.example.net", which is an identifier for a SAML Service Provider with which the authorization server identifies itself. The assertion is sent as part of an access token request to the authorization server's token endpoint at "https://authz.example.net/token.oauth2".

Below is an example SAML 2.0 Assertion (whitespace formatting is for display purposes only):

```
<Assertion IssueInstant="2010-10-01T20:07:34.619Z"
  ID="ef1xsbZxPV2oqjd7HTLRLIBlBb7"
  Version="2.0"
  xmlns="urn:oasis:names:tc:SAML:2.0:assertion">
  <Issuer>https://saml-idp.example.com</Issuer>
  <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    [...omitted for brevity...]
  </ds:Signature>
  <Subject>
    <NameID
      Format="urn:oasis:names:tc:SAML:1.1:nameid-format:emailAddress">
      brian@example.com
    </NameID>
    <SubjectConfirmation
      Method="urn:oasis:names:tc:SAML:2.0:cm:bearer">
      <SubjectConfirmationData
        NotOnOrAfter="2010-10-01T20:12:34.619Z"
        Recipient="https://authz.example.net/token.oauth2"/>
      </SubjectConfirmation>
    </Subject>
    <Conditions>
      <AudienceRestriction>
        <Audience>https://saml-sp.example.net</Audience>
      </AudienceRestriction>
    </Conditions>
    <AuthnStatement AuthnInstant="2010-10-01T20:07:34.371Z">
      <AuthnContext>
        <AuthnContextClassRef>
          urn:oasis:names:tc:SAML:2.0:ac:classes:X509
        </AuthnContextClassRef>
      </AuthnContext>
    </AuthnStatement>
  </Assertion>
```

Figure 1: Example SAML 2.0 Assertion

To present the Assertion shown in the previous example as part of an access token request, for example, the client might make the following HTTPS request (with extra line breaks for display purposes only):

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

grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Asaml2-
bearer&assertion=PEFzc2VydGlvbiBJc3NlZULuc3RhbnQ9IjIwMTtMDU
[...omitted for brevity...]aG5TdGF0ZW1lbnQ-PC9Bc3NlcnRpb24-
```

Figure 2: Example Request

5. Interoperability Considerations

Agreement between system entities regarding identifiers, keys, and endpoints is required in order to achieve interoperable deployments of this profile. Specific items that require agreement are as follows: values for the issuer and audience identifiers, the location of the token endpoint, the key used to apply and verify the digital signature over the assertion, one-time use restrictions on assertions, maximum assertion lifetime allowed, and the specific subject and attribute requirements of the assertion. The exchange of such information is explicitly out of scope for this specification and typical deployment of it will be done alongside existing SAML Web SSO deployments that have already established a means of exchanging such information. Metadata for the OASIS Security Assertion Markup Language (SAML) V2.0 [OASIS.saml-metadata-2.0-os] is one common method of exchanging SAML related information about system entities.

The RSA-SHA256 algorithm, from [RFC6931], is a mandatory to implement XML signature algorithm for this profile.

6. Security Considerations

The security considerations described within the Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants [I-D.ietf-oauth-assertions], The OAuth 2.0 Authorization Framework [RFC6749], and the Security and Privacy Considerations for the OASIS Security Assertion Markup Language (SAML) V2.0 [OASIS.saml-sec-consider-2.0-os] specifications are all applicable to this document.

The specification does not mandate replay protection for the SAML assertion usage for either the authorization grant or for client

authentication. It is an optional feature, which implementations may employ at their own discretion.

7. Privacy Considerations

A SAML Assertion may contain privacy-sensitive information and, to prevent disclosure of such information to unintended parties, should only be transmitted over encrypted channels, such as TLS. In cases where it is desirable to prevent disclosure of certain information to the client, the Subject and/or individual attributes of a SAML Assertion should be encrypted to the authorization server.

Deployments should determine the minimum amount of information necessary to complete the exchange and include only that information in an Assertion (typically by limiting what information is included in an <AttributeStatement> or omitting it altogether). In some cases, the Subject can be a value representing an anonymous or pseudonymous user, as described in Section 6.3.1 of the Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants [I-D.ietf-oauth-assertions].

8. IANA Considerations

8.1. Sub-Namespace Registration of urn:ietf:params:oauth:grant-type:saml2-bearer

This is a request to IANA to please register the value "grant-type:saml2-bearer" in the registry urn:ietf:params:oauth established in An IETF URN Sub-Namespace for OAuth [RFC6755].

- o URN: urn:ietf:params:oauth:grant-type:saml2-bearer
- o Common Name: SAML 2.0 Bearer Assertion Grant Type Profile for OAuth 2.0
- o Change controller: IESG
- o Specification Document: [[this document]]

8.2. Sub-Namespace Registration of urn:ietf:params:oauth:client-assertion-type:saml2-bearer

This is a request to IANA to please register the value "client-assertion-type:saml2-bearer" in the registry urn:ietf:params:oauth established in An IETF URN Sub-Namespace for OAuth [RFC6755].

- o URN: urn:ietf:params:oauth:client-assertion-type:saml2-bearer

- o Common Name: SAML 2.0 Bearer Assertion Profile for OAuth 2.0 Client Authentication
- o Change controller: IESG
- o Specification Document: [[this document]]

9. References

9.1. Normative References

[I-D.ietf-oauth-assertions]

Campbell, B., Mortimore, C., Jones, M., and Y. Goland, "Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants", draft-ietf-oauth-assertions (work in progress), October 2014.

[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, <<http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf>>.

[OASIS.saml-deleg-cs]

Cantor, S., Ed., "SAML V2.0 Condition for Delegation Restriction", Nov 2009.

[OASIS.saml-sec-consider-2.0-os]

Hirsch, F., Philpott, R., and E. Maler, "Security and Privacy Considerations for the OASIS Security Markup Language (SAML) V2.0", OASIS Standard saml-sec-consider-2.0-os, March 2005, <<http://docs.oasis-open.org/security/saml/v2.0/saml-sec-consider-2.0-os.pdf>>.

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

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.

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

- [RFC6931] Eastlake, D., "Additional XML Security Uniform Resource Identifiers (URIs)", RFC 6931, April 2013.

9.2. Informative References

- [OASIS.WS-Fed]
Goodner, M. and T. Nadalin, "Web Services Federation Language (WS-Federation) Version 1.2", May 2009, <<http://docs.oasis-open.org/wsfed/federation/v1.2/os/ws-federation-1.2-spec-os.html>>.
- [OASIS.WSS-SAMLTOKENProfile]
Monzillo, R., Kaler, C., Nadalin, T., Hallam-Baker, P., and C. Milono, "Web Services Security SAML Token Profile Version 1.1.1", May 2012, <<http://docs.oasis-open.org/wss-m/wss/v1.1.1/wss-SAMLTOKENProfile-v1.1.1.html>>.
- [OASIS.saml-metadata-2.0-os]
Cantor, S., Moreh, J., Philpott, R., and E. Maler, "Metadata for the Security Assertion Markup Language (SAML) V2.0", OASIS Standard saml-metadata-2.0-os, March 2005, <<http://docs.oasis-open.org/security/saml/v2.0/saml-metadata-2.0-os.pdf>>.
- [OASIS.saml-profiles-2.0-os]
Hughes, J., Cantor, S., Hodges, J., Hirsch, F., Mishra, P., Philpott, R., and E. Maler, "Profiles for the OASIS Security Assertion Markup Language (SAML) V2.0", OASIS Standard OASIS.saml-profiles-2.0-os, March 2005, <<http://docs.oasis-open.org/security/saml/v2.0/saml-profiles-2.0-os.pdf>>.
- [RFC6755] Campbell, B. and H. Tschofenig, "An IETF URN Sub-Namespace for OAuth", RFC 6755, October 2012.
- [W3C.REC-html401-19991224]
Raggett, D., Hors, A., and I. Jacobs, "HTML 4.01 Specification", World Wide Web Consortium Recommendation REC-html401-19991224, December 1999, <<http://www.w3.org/TR/1999/REC-html401-19991224>>.

Appendix A. Acknowledgements

The following people contributed wording and concepts to this document: Paul Madsen, Patrick Harding, Peter Motykowski, Eran Hammer, Peter Saint-Andre, Ian Barnett, Eric Fazendin, Torsten Lodderstedt, Susan Harper, Scott Tomilson, Scott Cantor, Hannes Tschofenig, David Waite, Phil Hunt, and Mukesh Bhatnagar.

Appendix B. Document History

[[to be removed by RFC editor before publication as an RFC]]

draft-ietf-oauth-saml2-bearer-23

- o Fix typo per <http://www.ietf.org/mail-archive/web/oauth/current/msg13790.html>

draft-ietf-oauth-saml2-bearer-22

- o Changes/suggestions from IESG reviews.

draft-ietf-oauth-saml2-bearer-21

- o Added Privacy Considerations section per AD review discussion <http://www.ietf.org/mail-archive/web/oauth/current/msg13148.html> and <http://www.ietf.org/mail-archive/web/oauth/current/msg13144.html>

draft-ietf-oauth-saml2-bearer-20

- o Clarified some text around the treatment of subject based on the rough rough consensus from the thread staring at <http://www.ietf.org/mail-archive/web/oauth/current/msg12630.html>

draft-ietf-oauth-saml2-bearer-19

- o Updated references.

draft-ietf-oauth-saml2-bearer-18

- o Clean up language around subject per <http://www.ietf.org/mail-archive/web/oauth/current/msg12254.html>.
- o As suggested in <http://www.ietf.org/mail-archive/web/oauth/current/msg12253.html> stated that "In the absence of an application profile specifying otherwise, compliant applications MUST compare the audience/issuer values using the Simple String Comparison method defined in Section 6.2.1 of RFC 3986."
- o Clarify the potentially confusing language about the AS confirming the assertion <http://www.ietf.org/mail-archive/web/oauth/current/msg12255.html>.

- o Combine the two items about AuthnStatement and drop the word presenter as discussed in <http://www.ietf.org/mail-archive/web/oauth/current/msg12257.html>.
- o Added one-time use, maximum lifetime, and specific subject and attribute requirements to Interoperability Considerations based on <http://www.ietf.org/mail-archive/web/oauth/current/msg12252.html>.
- o Reword security considerations and mention that replay protection is not mandated based on <http://www.ietf.org/mail-archive/web/oauth/current/msg12259.html>.

draft-ietf-oauth-saml2-bearer-17

- o Stated that issuer and audience values SHOULD be compared using the Simple String Comparison method defined in Section 6.2.1 of RFC 3986 unless otherwise specified by the application.

draft-ietf-oauth-saml2-bearer-16

- o Changed title from "SAML 2.0 Bearer Assertion Profiles for OAuth 2.0" to "SAML 2.0 Profile for OAuth 2.0 Client Authentication and Authorization Grants" to be more explicit about the scope of the document per <http://www.ietf.org/mail-archive/web/oauth/current/msg11063.html>.
- o Fixed typo in text identifying the presenter from "or similar element, the" to "or similar element in the".
- o Numbered the list of processing rules.
- o Smallish editorial cleanups to try and improve readability and comprehensibility.
- o Cleaner split out of the processing rules in cases where they differ for client authentication and authorization grants.
- o Clarified the parameters that are used/available for authorization grants.
- o Added Interoperability Considerations section and info reference to SAML Metadata.
- o Added more explanatory context to the example in Section 4.

draft-ietf-oauth-saml2-bearer-15

- o Reference RFC 6749 and RFC 6755.

- o Update draft-ietf-oauth-assertions reference to -06.
- o Remove extraneous word per <http://www.ietf.org/mail-archive/web/oauth/current/msg10055.html>

draft-ietf-oauth-saml2-bearer-14

- o Add more text to intro explaining that an assertion grant type can be used with or without client authentication/identification and that client assertion authentication is nothing more than an alternative way for a client to authenticate to the token endpoint
- o Add examples to Sections 2.1 and 2.2
- o Update references

draft-ietf-oauth-saml2-bearer-13

- o Update references: oauth-assertions-04, oauth-urn-sub-ns-05, oauth-28
- o Changed "Description" to "Specification Document" in both registration requests in IANA Considerations per changes to the template in ietf-oauth-urn-sub-ns(-03)
- o Added "(or an acceptable alias)" so that it's in both sentences about Recipient and the token endpoint URL so there's no ambiguity
- o Update area and workgroup (now Security and OAuth was Internet and nothing)

draft-ietf-oauth-saml2-bearer-12

- o updated reference to draft-ietf-oauth-v2 from -25 to -26 and draft-ietf-oauth-assertions from -02 to -03

draft-ietf-oauth-saml2-bearer-11

- o Removed text about limited lifetime access tokens and the SHOULD NOT on issuing refresh tokens. The text was moved to draft-ietf-oauth-assertions-02 and somewhat modified per <http://www.ietf.org/mail-archive/web/oauth/current/msg08298.html>.
- o Fixed typo/missing word per <http://www.ietf.org/mail-archive/web/oauth/current/msg08733.html>.
- o Added Terminology section.

draft-ietf-oauth-saml2-bearer-10

- o fix a spelling mistake

draft-ietf-oauth-saml2-bearer-09

- o Attempt to address an ambiguity around validation requirements when the Conditions element contain a NotOnOrAfter and SubjectConfirmation/SubjectConfirmationData does too. Basically it needs to have at least one bearer SubjectConfirmation element but that element can omit SubjectConfirmationData, if Conditions has an expiry on it. Otherwise, a valid SubjectConfirmation must have a SubjectConfirmationData with Recipient and NotOnOrAfter. And any SubjectConfirmationData that has those elements needs to have them checked.
- o clarified that AudienceRestriction is under Conditions (even though it's implied by schema)

- o fix a typo

draft-ietf-oauth-saml2-bearer-08

- o fix some typos

draft-ietf-oauth-saml2-bearer-07

- o update reference from draft-campbell-oauth-urn-sub-ns to draft-ietf-oauth-urn-sub-ns
- o Updated to reference draft-ietf-oauth-v2-20

draft-ietf-oauth-saml2-bearer-06

- o Fix three typos NamseID->NameID and (2x) Namespace->Namespace

draft-ietf-oauth-saml2-bearer-05

- o Allow for subject confirmation data to be optional when Conditions contain audience and NotOnOrAfter
- o Rework most of the spec to profile draft-ietf-oauth-assertions for both authn and authz including (but not limited to):
 - * remove requirement for issuer to be urn:oasis:names:tc:SAML:2.0:nameid-format:entity
 - * change wording on Subject requirements

- o using a MAY, explicitly say that the Audience can be token endpoint URL of the authorization server
- o Change title to be more generic (allowing for client authn too)
- o added client authentication to the abstract
- o register and use urn:ietf:params:oauth:grant-type:saml2-bearer for grant type rather than http://oauth.net/grant_type/saml/2.0/bearer
- o register urn:ietf:params:oauth:client-assertion-type:saml2-bearer
- o remove scope parameter as it is defined in <http://tools.ietf.org/html/draft-ietf-oauth-assertions>
- o remove assertion param registration because it [should] be in <http://tools.ietf.org/html/draft-ietf-oauth-assertions>
- o fix typo(s) and update/add references

draft-ietf-oauth-saml2-bearer-04

- o Changed the grant_type URI from "http://oauth.net/grant_type/assertion/saml/2.0/bearer" to "http://oauth.net/grant_type/saml/2.0/bearer" - dropping the word assertion from the path. Recent versions of draft-ietf-oauth-v2 no longer refer to extension grants using the word assertion so this URI is more reflective of that. It also more closely aligns with the grant type URI in draft-jones-oauth-jwt-bearer-00 which is "http://oauth.net/grant_type/jwt/1.0/bearer".
- o Added "case sensitive" to scope definition to align with draft-ietf-oauth-v2-15/16.
- o Updated to reference draft-ietf-oauth-v2-16

draft-ietf-oauth-saml2-bearer-03

- o Cleanup of some editorial issues.

draft-ietf-oauth-saml2-bearer-02

- o Added scope parameter with text copied from draft-ietf-oauth-v2-12 (the reorg of draft-ietf-oauth-v2-12 made it so scope wasn't really inherited by this spec anymore)

- o Change definition of the assertion parameter to be more generally applicable per the suggestion near the end of <http://www.ietf.org/mail-archive/web/oauth/current/msg05253.html>

- o Editorial changes based on feedback

draft-ietf-oauth-saml2-bearer-01

- o Update spec name when referencing draft-ietf-oauth-v2 (The OAuth 2.0 Protocol Framework -> The OAuth 2.0 Authorization Protocol)
- o Update wording in Introduction to talk about extension grant types rather than the assertion grant type which is a term no longer used in OAuth 2.0
- o Updated to reference draft-ietf-oauth-v2-12 and denote as work in progress
- o Update Parameter Registration Request to use similar terms as draft-ietf-oauth-v2-12 and remove Related information part
- o Add some text giving discretion to AS on rejecting assertions with unreasonably long validity window.

draft-ietf-oauth-saml2-bearer-00

- o Added Parameter Registration Request for "assertion" to IANA Considerations.
- o Changed document name to draft-ietf-oauth-saml2-bearer in anticipation of becoming an OAUTH WG item.
- o Attempt to move the entire definition of the 'assertion' parameter into this draft (it will no longer be defined in OAuth 2 Protocol Framework).

draft-campbell-oauth-saml-01

- o Updated to reference draft-ietf-oauth-v2-11 and reflect changes from -10 to -11.
- o Updated examples.
- o Relaxed processing rules to allow for more than one SubjectConfirmation element.
- o Removed the 'MUST NOT contain a NotBefore attribute' on SubjectConfirmationData.

- o Relaxed wording that ties the subject of the Assertion to the resource owner.
- o Added some wording about identifying the client when the subject hasn't directly authenticated including an informative reference to SAML V2.0 Condition for Delegation Restriction.
- o Added a few examples to the language about verifying that the Assertion is valid in all other respects.
- o Added some wording to the introduction about the similarities to Web SSO in the format and processing rules
- o Changed the grant_type (was assertion_type) URI from http://oauth.net/assertion_type/saml/2.0/bearer to http://oauth.net/grant_type/assertion/saml/2.0/bearer
- o Changed title to include "Grant Type" in it.
- o Editorial updates based on feedback from the WG and others (including capitalization of Assertion when referring to SAML).

draft-campbell-oauth-saml-00

- o Initial I-D

Authors' Addresses

Brian Campbell
Ping Identity

Email: brian.d.campbell@gmail.com

Chuck Mortimore
Salesforce.com

Email: cmortimore@salesforce.com

Michael B. Jones
Microsoft

Email: mbj@microsoft.com

URI: <http://self-issued.info/>

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

J. Richer, Ed.

J. Bradley
Ping Identity
H. Tschofenig
ARM Limited
August 08, 2016

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

Abstract

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

Status of This Memo

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

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

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

This Internet-Draft will expire on February 9, 2017.

Copyright Notice

Copyright (c) 2016 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must

include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

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

1. Introduction

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

Furthermore, it is desirable to bind the signature to the HTTP request. Ideally, this should be done without replicating the information already present in the HTTP request more than required. However, many HTTP application frameworks insert extra headers, query

parameters, and otherwise manipulate the HTTP request on its way from the web server into the application code itself. It is the goal of this draft to have a signature protection mechanism that is sufficiently robust against such deployment constraints while still providing sufficient security benefits.

The key required for this signature calculation is distributed via mechanisms described in companion documents (see [I-D.ietf-oauth-pop-key-distribution] and [I-D.ietf-oauth-pop-architecture]). The JSON Web Signature (JWS) specification [RFC7515] is used for computing a digital signature (which uses asymmetric cryptography) or a keyed message digest (in case of symmetric cryptography).

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

2. Terminology

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

Other terms such as "client", "authorization server", "access token", and "protected resource" are inherited from OAuth 2.0 [RFC6749].

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

3. Generating a JSON Object from an HTTP Request

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

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

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

- ts RECOMMENDED. The timestamp. This integer provides replay protection of the signed JSON object. Its value MUST be a number containing an integer value representing number of whole integer seconds from midnight, January 1, 1970 GMT.
- m OPTIONAL. The HTTP Method used to make this request. This MUST be the uppercase HTTP verb as a JSON string.
- u OPTIONAL. The HTTP URL host component as a JSON string. This MAY include the port separated from the host by a colon in host:port format.
- p OPTIONAL. The HTTP URL path component of the request as an HTTP string.
- q OPTIONAL. The hashed HTTP URL query parameter map of the request as a two-part JSON array. The first part of this array is a JSON array listing all query parameters that were used in the calculation of the hash in the order that they were added to the hashed value as described below. The second part of this array is a JSON string containing the Base64URL encoded hash itself, calculated as described below.
- h OPTIONAL. The hashed HTTP request headers as a two-part JSON array. The first part of this array is a JSON array listing all headers that were used in the calculation of the hash in the order that they were added to the hashed value as described below. The second part of this array is a JSON string containing the Base64URL encoded hash itself, calculated as described below.
- b OPTIONAL. The base64URL encoded hash of the HTTP Request body, calculated as the SHA256 of the byte array of the body

All hashes SHALL be calculated using the SHA256 algorithm. [[Note to WG: do we want crypto agility here? If so how do we signal this]]

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

3.1. Calculating the query parameter list and hash

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

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

1. Adds the name of the query parameter to the end of the list.
2. Percent-encodes the name and value of the parameter as specified in [RFC3986]. Note that if the name and value have already been percent-encoded for transit, they are not re-encoded for this step.
3. Encodes the name and value of the query parameter as "name=value" and appends it to the string buffer separated by the ampersand "&" character.

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

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

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

```
b=bar&a=foo&c=duck
```

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

```
"q": [["b", "a", "c"], "u4LgkGUWhP9MsKrEjA4dizI1lDXluDku6ZqCeyuR-JY"]
```

3.2. Calculating the header list and hash

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

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

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

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

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

For example, the headers "Content-Type: application/json" and "Etag: 742-3u8f34-3r2nv3" are concatenated into the string:

```
content-type: application/json
etag: 742-3u8f34-3r2nv3

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

4. Sending the signed object

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

4.1. HTTP Authorization header

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

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

4.2. HTTP Form body

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

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

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

4.3. HTTP Query parameter

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

```
GET /resource?pop_access_token=eyJ....
```

5. Validating the request

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

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

5.1. Validating the query parameter list and hash

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

1. Fetch the value of the parameter from the HTTP request query parameter map. If a parameter is found in the list of signed parameters but not in the map, the validation fails.
2. Percent-encodes the name and value of the parameter as specified in [RFC3986]. Note that if the name and value have already been percent-encoded for transit, they are not re-encoded for this step.
3. Encode the parameter as "name=value" and concatenate it to the end of the string buffer, separated by an ampersand character.

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

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

5.2. Validating the header list and hash

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

1. Fetch the value of the header from the HTTP request header map. If a header is found in the list of signed parameters but not in the map, the validation fails.
2. Encode the parameter as "name: value" and concatenate it to the end of the string buffer, separated by a newline character.

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

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

6. IANA Considerations

6.1. The 'pop' OAuth Access Token Type

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

Type name: pop

Additional Token Endpoint Response Parameters: (none)

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

Change controller: IETF

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

6.2. JSON Web Signature and Encryption Type Values Registration

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

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

7. Security Considerations

7.1. Offering Confidentiality Protection for Access to Protected Resources

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

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

When TLS is used then confidentiality of the transmission can be ensured between endpoints, including both the request and the

response. The use of TLS in combination with the signed HTTP request mechanism is highly recommended to ensure the confidentiality of the data returned from the protected resource.

7.2. Plaintext Storage of Credentials

The mechanism described in this document works in a similar way to many three-party authentication and key exchange mechanisms. In order to compute the signature over the HTTP request, the client must have access to a key bound to the access token in plaintext form. If an attacker were to gain access to these stored secrets at the client or (in case of symmetric keys) at the resource server they would be able to perform any action on behalf of any client just as if they had stolen a bearer token.

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

7.3. Entropy of Keys

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

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

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

7.4. Denial of Service

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

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

An attacker may exploit this to perform a denial of service attack by sending a large number of invalid requests to the server. The computational overhead of verifying the keyed message digest alone is not likely sufficient to mount a denial of service attack. To help combat this, it is RECOMMENDED that the protected resource validate the access token (contained in the "at" member of the signed structure) before performing any cryptographic verification calculations.

7.5. Validating the integrity of HTTP message

This specification provides flexibility for selectively validating the integrity of the HTTP request, including header fields, query parameters, and message bodies. Since all components of the HTTP request are only optionally validated by this method, and even some components may be validated only in part (e.g., some headers but not others) it is up to protected resource developers to verify that any vital parameters in a request are actually covered by the signature. Failure to do so could allow an attacker to inject vital parameters or headers into the request, outside of the protection of the signature.

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

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

This specification records the order in which query parameters and headers are hashed, but it does not guarantee that order is preserved between the client and protected resource. If the order of

parameters or headers are significant to the underlying application, it MUST confirm their order on its own, apart from the signature and HTTP message validation.

8. Privacy Considerations

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

9. Acknowledgements

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

10. Normative References

- [I-D.ietf-oauth-pop-architecture]
Hunt, P., Richer, J., Mills, W., Mishra, P., and H. Tschofenig, "OAuth 2.0 Proof-of-Possession (PoP) Security Architecture", draft-ietf-oauth-pop-architecture-08 (work in progress), July 2016.
- [I-D.ietf-oauth-pop-key-distribution]
Bradley, J., Hunt, P., Jones, M., and H. Tschofenig, "OAuth 2.0 Proof-of-Possession: Authorization Server to Client Key Distribution", draft-ietf-oauth-pop-key-distribution-02 (work in progress), October 2015.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<http://www.rfc-editor.org/info/rfc2119>>.
- [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <<http://www.rfc-editor.org/info/rfc3986>>.
- [RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <<http://www.rfc-editor.org/info/rfc6749>>.
- [RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, <<http://www.rfc-editor.org/info/rfc6750>>.

- [RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, <<http://www.rfc-editor.org/info/rfc7159>>.
- [RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <<http://www.rfc-editor.org/info/rfc7515>>.
- [RFC7519] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <<http://www.rfc-editor.org/info/rfc7519>>.
- [RFC7662] Richer, J., Ed., "OAuth 2.0 Token Introspection", RFC 7662, DOI 10.17487/RFC7662, October 2015, <<http://www.rfc-editor.org/info/rfc7662>>.

Authors' Addresses

Justin Richer (editor)

Email: ietf@justin.richer.org

John Bradley
Ping Identity

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

Hannes Tschofenig
ARM Limited
Austria

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

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: January 9, 2016

N. Sakimura, Ed.
Nomura Research Institute
J. Bradley
Ping Identity
N. Agarwal
Google
July 8, 2015

Proof Key for Code Exchange by OAuth Public Clients
draft-ietf-oauth-spop-15

Abstract

OAuth 2.0 public clients utilizing the Authorization Code Grant are susceptible to the authorization code interception attack. This specification describes the attack as well as a technique to mitigate against the threat through the use of Proof Key for Code Exchange (PKCE, pronounced "pixy").

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, 2016.

Copyright Notice

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

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must

include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
1.1. Protocol Flow	5
2. Notational Conventions	6
3. Terminology	7
3.1. Abbreviations	7
4. Protocol	7
4.1. Client creates a code verifier	7
4.2. Client creates the code challenge	8
4.3. Client sends the code challenge with the authorization request	8
4.4. Server returns the code	9
4.4.1. Error Response	9
4.5. Client sends the Authorization Code and the Code Verifier to the token endpoint	9
4.6. Server verifies code_verifier before returning the tokens	10
5. Compatibility	10
6. IANA Considerations	10
6.1. OAuth Parameters Registry	10
6.2. PKCE Code Challenge Method Registry	11
6.2.1. Registration Template	11
6.2.2. Initial Registry Contents	12
7. Security Considerations	12
7.1. Entropy of the code_verifier	12
7.2. Protection against eavesdroppers	13
7.3. Salting the code_challenge	13
7.4. OAuth security considerations	14
7.5. TLS security considerations	14
8. Acknowledgements	14
9. Revision History	15
10. References	17
10.1. Normative References	18
10.2. Informative References	18
Appendix A. Notes on implementing base64url encoding without padding	18
Appendix B. Example for the S256 code_challenge_method	19
Authors' Addresses	20

1. Introduction

OAuth 2.0 [RFC6749] public clients are susceptible to the authorization code interception attack.

The attacker thereby intercepts the authorization code returned from the authorization endpoint within communication path not protected by TLS, such as inter-app communication within the operating system of the client.

Once the attacker has gained access to the authorization code it can use it to obtain the access token.

Figure 1 shows the attack graphically. In step (1) the native app running on the end device, such as a smart phone, issues an OAuth 2.0 Authorization Request via the browser/operating system. The Redirection Endpoint URI in this case typically uses a custom URI scheme. Step (1) happens through a secure API that cannot be intercepted, though it may potentially be observed in advanced attack scenarios. The request then gets forwarded to the OAuth 2.0 authorization server in step (2). Because OAuth requires the use of TLS, this communication is protected by TLS, and also cannot be intercepted. The authorization server returns the authorization code in step (3). In step (4), the Authorization Code is returned to the requester via the Redirection Endpoint URI that was provided in step (1).

A malicious app that has been designed to attack this native app has previously registered itself as a handler for the custom URI scheme is now able to intercept the Authorization Code in step (4). This allows the attacker to request and obtain an access token in steps (5) and (6), respectively.

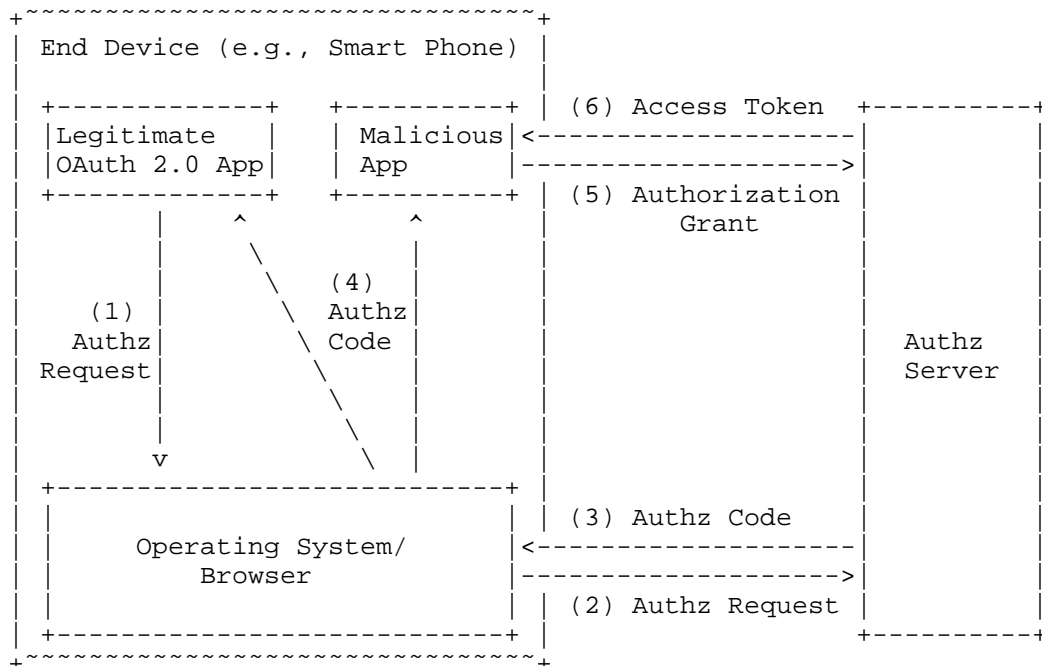


Figure 1: Authorization Code Interception Attack.

A number of pre-conditions need to hold in order for this attack to work:

- 1) The attacker manages to register a malicious application on the client device and registers a custom URI scheme that is also used by another application.
The operating systems must allow a custom URI schemes to be registered by multiple applications.
- 2) The OAuth 2.0 authorization code grant is used.
- 3) The attacker has access to the OAuth 2.0 [RFC6749] client_id and client_secret(if provisioned). All OAuth 2.0 native app client-instances use the same client_id. Secrets provisioned in client binary applications cannot be considered confidential.
- 4a) The attacker (via the installed app) is able to observe only the responses from the authorization endpoint. The plain code_challenge_method mitigates only this attack.
- 4b) A more sophisticated attack scenario allows the attacker to observe requests (in addition to responses) to the authorization endpoint. The attacker is, however, not able to act as a man-in-the-middle. This has been caused by leaking http log information in the OS. To mitigate this the S256 code_challenge_method or

cryptographically secure `code_challenge_method` extension must be used.

While this is a long list of pre-conditions the described attack has been observed in the wild and has to be considered in OAuth 2.0 deployments.

While the OAuth 2.0 Threat Model Section 4.4.1 [RFC6819] describes mitigation techniques they are, unfortunately, not applicable since they rely on a per-client instance secret or a per client instance redirect URI.

To mitigate this attack, this extension utilizes a dynamically created cryptographically random key called "code verifier". A unique code verifier is created for every authorization request and its transformed value, called "code challenge", is sent to the authorization server to obtain the authorization code. The authorization code obtained is then sent to the token endpoint with the "code verifier" and the server compares it with the previously received request code so that it can perform the proof of possession of the "code verifier" by the client. This works as the mitigation since the attacker would not know this one-time key, since it is sent over TLS and cannot be intercepted.

1.1. Protocol Flow

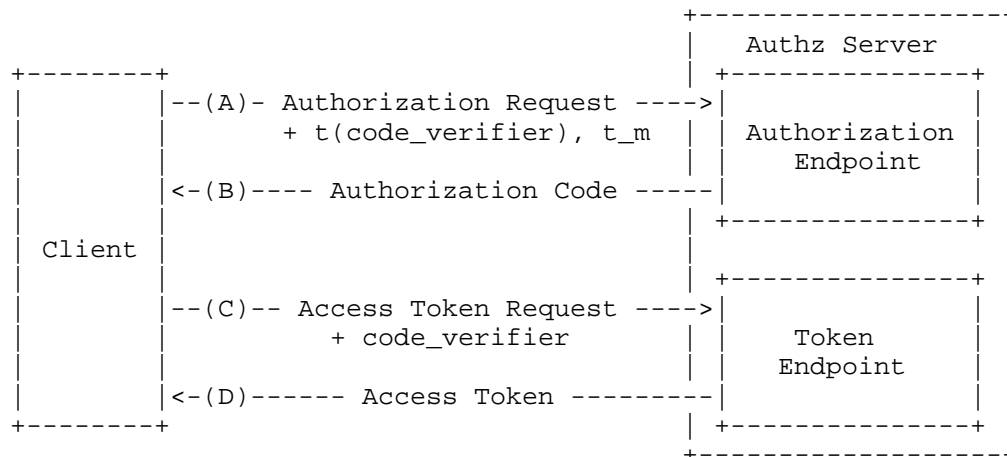


Figure 2: Abstract Protocol Flow

This specification adds additional parameters to the OAuth 2.0 Authorization and Access Token Requests, shown in abstract form in Figure 1.

- A. The client creates and records a secret named the "code_verifier", and derives a transformed version "t(code_verifier)" (referred to as the "code_challenge") which is sent in the OAuth 2.0 Authorization Request, along with the transformation method "t_m".
- B. The Authorization Endpoint responds as usual, but records "t(code_verifier)" and the transformation method.
- C. The client then sends the authorization code in the Access Token Request as usual, but includes the "code_verifier" secret generated at (A).
- D. The authorization server transforms "code_verifier" and compares it to "t(code_verifier)" from (B). Access is denied if they are not equal.

An attacker who intercepts the Authorization Grant at (B) is unable to redeem it for an Access Token, as they are not in possession of the "code_verifier" secret.

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.

This specification uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234].

STRING denotes a sequence of zero or more ASCII [RFC0020] characters.

OCTETS denotes a sequence of zero or more octets.

ASCII(STRING) denotes the octets of the ASCII [RFC0020] representation of STRING where STRING is a sequence of zero or more ASCII characters.

BASE64URL-ENCODE(OCTETS) denotes the base64url encoding of OCTETS, per Section 3 producing a STRING.

BASE64URL-DECODE(STRING) denotes the base64url decoding of STRING, per Section 3, producing a sequence of octets.

SHA256(OCTETS) denotes a SHA2 256bit hash [RFC6234] of OCTETS.

3. Terminology

In addition to the terms defined in OAuth 2.0 [RFC6749], this specification defines the following terms:

code verifier

A cryptographically random string that is used to correlate the authorization request to the token request.

code challenge

A challenge derived from the code verifier that is sent in the authorization request, to be verified against later.

Base64url Encoding

Base64 encoding using the URL- and filename-safe character set defined in Section 5 of [RFC4648], with all trailing '=' characters omitted (as permitted by Section 3.2 of [RFC4648]) and without the inclusion of any line breaks, whitespace, or other additional characters. (See Appendix A for notes on implementing base64url encoding without padding.)

3.1. Abbreviations

ABNF Augmented Backus-Naur Form
Authz Authorization
PKCE Proof Key for Code Exchange
MITM Man-in-the-middle
MTI Mandatory To Implement

4. Protocol

4.1. Client creates a code verifier

The client first creates a code verifier, "code_verifier", for each OAuth 2.0 [RFC6749] Authorization Request, in the following manner:

code_verifier = high entropy cryptographic random STRING using the Unreserved Characters [A-Z] / [a-z] / [0-9] / "-" / "." / "_" / "~" from Sec 2.3 of [RFC3986], with a minimum length of 43 characters and a maximum length of 128 characters.

ABNF for "code_verifier" is as follows.

```
code-verifier = 43*128unreserved
unreserved = ALPHA / DIGIT / "-" / "." / "_" / "~"
ALPHA = %x41-5A / %x61-7A
DIGIT = %x30-39
```

NOTE: code verifier SHOULD have enough entropy to make it impractical to guess the value. It is RECOMMENDED that the output of a suitable

random number generator be used to create a 32-octet sequence. The Octet sequence is then base64url encoded to produce a 43-octet URL safe string to use as the code verifier.

4.2. Client creates the code challenge

The client then creates a code challenge derived from the code verifier by using one of the following transformations on the code verifier:

```
plain
  code_challenge = code_verifier
S256
  code_challenge = BASE64URL-ENCODE(SHA256(ASCII(code_verifier)))
```

If the client is capable of using "S256", it MUST use "S256", as "S256" is Mandatory To Implement (MTI) on the server. Clients are permitted to use "plain" only if they cannot support "S256" for some technical reason and know via out of band configuration that the server supports "plain".

The plain transformation is for compatibility with existing deployments and for constrained environments that can't use the S256 transformation.

ABNF for "code_challenge" is as follows.

```
code-challenge = 43*128unreserved
unreserved = ALPHA / DIGIT / "-" / "." / "_" / "~"
ALPHA = %x41-5A / %x61-7A
DIGIT = %x30-39
```

4.3. Client sends the code challenge with the authorization request

The client sends the code challenge as part of the OAuth 2.0 Authorization Request (Section 4.1.1 of [RFC6749].) using the following additional parameters:

code_challenge REQUIRED. Code challenge.

code_challenge_method OPTIONAL, defaults to "plain" if not present in the request. Code verifier transformation method, "S256" or "plain".

4.4. Server returns the code

When the server issues the authorization code in the authorization response, it MUST associate the "code_challenge" and "code_challenge_method" values with the authorization code so it can be verified later.

Typically, the "code_challenge" and "code_challenge_method" values are stored in encrypted form in the "code" itself, but could alternatively be stored on the server, associated with the code. The server MUST NOT include the "code_challenge" value in client requests in a form that other entities can extract.

The exact method that the server uses to associate the "code_challenge" with the issued "code" is out of scope for this specification.

4.4.1. Error Response

If the server requires Proof Key for Code Exchange (PKCE) by OAuth Public Clients, and the client does not send the "code_challenge" in the request, the authorization endpoint MUST return the authorization error response with "error" value set to "invalid_request". The "error_description" or the response of "error_uri" SHOULD explain the nature of error, e.g., code challenge required.

If the server supporting PKCE does not support the requested transform, the authorization endpoint MUST return the authorization error response with "error" value set to "invalid_request". The "error_description" or the response of "error_uri" SHOULD explain the nature of error, e.g., transform algorithm not supported.

4.5. Client sends the Authorization Code and the Code Verifier to the token endpoint

Upon receipt of the Authorization Code, the client sends the Access Token Request to the token endpoint. In addition to the parameters defined in the OAuth 2.0 Access Token Request (Section 4.1.3 of [RFC6749]), it sends the following parameter:

code_verifier REQUIRED. Code verifier

The code_challenge_method is bound to the Authorization Code when the Authorization Code is issued. That is the method that the token endpoint MUST use to verify the code_verifier.

4.6. Server verifies code_verifier before returning the tokens

Upon receipt of the request at the Access Token endpoint, the server verifies it by calculating the code challenge from received "code_verifier" and comparing it with the previously associated "code_challenge", after first transforming it according to the "code_challenge_method" method specified by the client.

If the "code_challenge_method" from Section 4.2 was "S256", the received "code_verifier" is hashed by SHA-256, then base64url encoded, and then compared to the "code_challenge". i.e.,

```
BASE64URL-ENCODE(SHA256(ASCII("code_verifier" ))) == "code_challenge"
```

If the "code_challenge_method" from Section 4.2 was "plain", they are compared directly. i.e.,

```
"code_verifier" == "code_challenge".
```

If the values are equal, the Access Token endpoint MUST continue processing as normal (as defined by OAuth 2.0 [RFC6749]). If the values are not equal, an error response indicating "invalid_grant" as described in section 5.2 of [RFC6749] MUST be returned.

5. Compatibility

Server implementations of this specification MAY accept OAuth2.0 Clients that do not implement this extension. If the "code_verifier" is not received from the client in the Authorization Request, servers supporting backwards compatibility revert to a normal OAuth 2.0 [RFC6749] protocol.

As the OAuth 2.0 [RFC6749] server responses are unchanged by this specification, client implementations of this specification do not need to know if the server has implemented this specification or not, and SHOULD send the additional parameters as defined in Section 3. to all servers.

6. IANA Considerations

This specification makes a registration request as follows:

6.1. OAuth Parameters Registry

This specification registers the following parameters in the IANA OAuth Parameters registry defined in OAuth 2.0 [RFC6749].

- o Parameter name: code_verifier

- o Parameter usage location: token request
- o Change controller: IESG
- o Specification document(s): this document

- o Parameter name: code_challenge
- o Parameter usage location: authorization request
- o Change controller: IESG
- o Specification document(s): this document

- o Parameter name: code_challenge_method
- o Parameter usage location: authorization request
- o Change controller: IESG
- o Specification document(s): this document

6.2. PKCE Code Challenge Method Registry

This specification establishes the PKCE Code Challenge Method registry. The new registry should be a sub-registry of OAuth Parameters registry.

Additional code_challenge_method types for use with the authorization endpoint are registered using the Specification Required policy [RFC5226], which includes review of the request by one or more Designated Experts. The DES will ensure there is at least a two-week review of the request on the oauth-ext-review@ietf.org mailing list, and that any discussion on that list converges before they respond to the request. To allow for the allocation of values prior to publication, the Designated Expert(s) may approve registration once they are satisfied that an acceptable specification will be published.

Registration requests and discussion on the oauth-ext-review@ietf.org mailing list should use an appropriate subject, such as "Request for PKCE code_challenge_method: example").

The Designated Expert(s) should consider the discussion on the mailing list, as well as the overall security properties of the challenge Method when evaluating registration requests. New methods should not disclose the value of the code_verifier in the request to the Authorization endpoint. Denials should include an explanation and, if applicable, suggestions as to how to make the request successful.

6.2.1. Registration Template

Code Challenge Method Parameter Name:

The name requested (e.g., "example"). Because a core goal of this specification is for the resulting representations to be compact,

it is RECOMMENDED that the name be short -- not to exceed 8 characters without a compelling reason to do so. This name is case-sensitive. Names may not match other registered names in a case-insensitive manner unless the Designated Expert(s) state that there is a compelling reason to allow an exception in this particular case.

Change Controller:

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

Specification Document(s):

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

6.2.2. Initial Registry Contents

This specification registers the Code Challenge Method Parameter names defined in Section 4.2 in this registry.

- o Code Challenge Method Parameter Name: "plain"
- o Change Controller: IESG
- o Specification Document(s): Section 4.2 of [[this document]]

- o Code Challenge Method Parameter Name: "S256"
- o Change Controller: IESG
- o Specification Document(s): Section 4.2 of [[this document]]

7. Security Considerations

7.1. Entropy of the code_verifier

The security model relies on the fact that the code verifier is not learned or guessed by the attacker. It is vitally important to adhere to this principle. As such, the code verifier has to be created in such a manner that it is cryptographically random and has high entropy that it is not practical for the attacker to guess.

The client SHOULD create a code_verifier with a minimum of 256bits of entropy. This can be done by having a suitable random number generator create a 32-octet sequence. The Octet sequence can then be base64url encoded to produce a 43-octet URL safe string to use as a code_challenge that has the required entropy.

7.2. Protection against eavesdroppers

Clients MUST NOT downgrade to "plain" after trying "S256" method. Servers that support PKCE are required to support "S256", and servers that do not support PKCE will simply ignore the unknown "code_verifier" OAuth 2.0 (see Section 3.2 of [RFC6749]). Because of that, an error when "S256" is presented can only mean that the server is faulty or that a MITM attacker is trying a downgrade attack.

"S256" method protects against eavesdroppers observing or intercepting the "code_challenge", because the challenge cannot be used without the verifier. With the "plain" method, there is a chance that "code_challenge" will be observed by the attacker on the device, or in the http request. Since the code challenge is the same as the code verifier in this case, "plain" method does not protect against the eavesdropping of the initial request.

The use of "S256" protects against disclosure of "code_verifier" value to an attacker.

Because of this, "plain" SHOULD NOT be used, and exists only for compatibility with deployed implementations where the request path is already protected. The "plain" method SHOULD NOT be used in new implementations, unless they cannot support "S256" for some technical reason.

The "S256" code_challenge_method or other cryptographically secure code_challenge_method extension SHOULD be used. The plain code_challenge_method relies on the operating system and transport security not to disclose the request to an attacker.

If the code_challenge_method is plain, and the "code_challenge" is to be returned inside authorization "code" to achieve a stateless server, it MUST be encrypted in such a manner that only the server can decrypt and extract it.

7.3. Salting the code_challenge

In order to reduce implementation complexity Salting is not used in the production of the code_challenge, as the code_verifier contains sufficient entropy to prevent brute force attacks. Concatenating a publicly known value to a code_verifier (containing 256 bits of entropy) and then hashing it with SHA256 to produce a code_challenge would not increase the number of attempts necessary to brute force a valid value for code_verifier.

While the S256 transformation is like hashing a password there are important differences. Passwords tend to be relatively low entropy

words that can be hashed offline and the hash looked up in a dictionary. By concatenating a unique though public value to each password prior to hashing, the dictionary space that an attacker needs to search is greatly expanded.

Modern graphics processors now allow attackers to calculate hashes in real time faster than they could be looked up from a disk. This eliminates the value of the salt in increasing the complexity of a brute force attack for even low entropy passwords.

7.4. OAuth security considerations

All the OAuth security analysis presented in [RFC6819] applies so readers SHOULD carefully follow it.

7.5. 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].

8. Acknowledgements

The initial draft of this specification was created by the OpenID AB/Connect Working Group of the OpenID Foundation.

This specification is the work of the OAuth Working Group, which includes dozens of active and dedicated participants. In particular, the following individuals contributed ideas, feedback, and wording that shaped and formed the final specification:

Anthony Nadalin, Microsoft
Axel Nenker, Deutsche Telekom
Breno de Medeiros, Google
Brian Campbell, Ping Identity
Chuck Mortimore, Salesforce
Dirk Balfanz, Google
Eduardo Gueiros, Jive Communications
Hannes Tschofenig, ARM
James Manger, Telstra
John Bradley, Ping Identity
Justin Richer, MIT Kerberos
Josh Mandel, Boston Children's Hospital
Lewis Adam, Motorola Solutions
Madjid Nakhjiri, Samsung
Michael B. Jones, Microsoft
Nat Sakimura, Nomura Research Institute
Naveen Agarwal, Google

Paul Madsen, Ping Identity
Phil Hunt, Oracle
Prateek Mishra, Oracle
Ryo Ito, mixi
Scott Tomilson, Ping Identity
Sergey Beryozkin
Takamichi Saito
Torsten Lodderstedt, Deutsche Telekom
William Denniss, Google

9. Revision History

-15

- o Addressed Barry's IESG comments around IANA Registration
- o Addressed Barry's IESG comments around Sec 7.2 downgrade attack
- o fix a typo for William and make a small change to Fig 1.1 clarifying t_m
- o more wording changes to sec 7.2 re Barry
- o made the two SHOULD NOT use plain recommendations consistent.
- o slightly cleaned up grammar in Sec 7.2

-14

- o #38. Expanded Section 7.2 to explain why plain should not be used.
- o #39. Modified Section 4.4.1 to discourage the use of plain.
- o #40. Modified Intro text to explain the attack better.
- o #41. Added explanation that the token request is protected in the Last paragraph of the Introduction.
- o #42. Sec 4.2: Removed redundant double quotes caused by spanx.
- o #43. Sec 4.4: Replaced code with authorization code.
- o #44. Sec 4.5: say "code_verifier" rather than "secret"
- o #45. Sec 4.4.1: Expanded PKCE.
- o #46. Sec 5: SHOULD in para 1 removed.
- o Added abbreviations section.

-13

- o Fix the parameter usage locations for the OAuth Parameters Registry per Hannes response.
- o Clarify for IANA that the new registry is a sub-registry of OAuth Parameters registry
- o added text on why the code_challenge_method is not sent to the token endpoint.

-12

- o clarify that the client secret we are talking about in the Introduction is a OAuth 2 client_secret.
- o Update salting security consideration based on Ben's feedback

-11

- o add spanx for plain in sec 4.4 RE Kathleen's comment
- o Add security consideration on TLS and reference BCP195
- o Update to make clearer that plain can only be used for backwards compatibility and constrained environments

-10

- o re #33 specify lower limit to code_verifier in prose
- o remove base64url decode from draft, all steps now use encode only
- o Expanded MTI
- o re #33 change length of 32 octet base64url encoded string back to 43 octets

-09

- o clean up some external references so they don't point at internal sections

-08

- o changed BASE64URL to BASE64URL-ENCODE to be more consistent with appendix A Fixed lowercase base64url in appendix B
- o Added appendix B as an example of S256 processing
- o Change reference for unreserved characters to RFC3986 from base64URL

-07

- o removed unused discovery reference and UTF8
- o re #32 added ASCII(STRING) to make clear that it is the byte array that is being hashed
- o re #2 Remove discovery requirement section.
- o updated Acknowledgement
- o re #32 remove unneeded UTF8(STRING) definition, and define STRING for ASCII(STRING)
- o re #32 remove unneeded utf8 reference from BASE64URL-DECODE(STRING) def
- o resolves #31 unused definition of concatenation
- o re #30 Update figure text call out the endpoints
- o re #30 Update figure to call out the endpoints
- o small wording change to the introduction

-06

- o fix date
- o replace spop with pkce for registry and other references
- o re #29 change name again
- o re #27 removed US-ASCII reference
- o re #27 updated ABNF for code_verifier
- o resolves #24 added security consideration for salting
- o resolves #29 Changed title
- o updated reference to RFC4634 to RFC6234 re #27
- o changed reference for US-ASCII to RFC20 re #27
- o resolves #28 added Acknowledgements
- o resolves #27 updated ABNF
- o resolves #26 updated abstract and added Hannes figure

-05

- o Added IANA registry for code_challenge_method + fixed some broken internal references.

-04

- o Added error response to authorization response.

-03

- o Added an abstract protocol diagram and explanation

-02

- o Copy edits

-01

- o Specified exactly two supported transformations
- o Moved discovery steps to security considerations.
- o Incorporated readability comments by Eduardo Gueiros.
- o Changed MUST in 3.1 to SHOULD.

-00

- o Initial IETF version.

10. References

10.1. 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.
- [RFC0020] Cerf, V., "ASCII format for network interchange", RFC 20, October 1969.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.
- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.
- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.
- [RFC5234] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008.
- [RFC6234] Eastlake, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, May 2011.
- [RFC6749] Hardt, D., "The OAuth 2.0 Authorization Framework", RFC 6749, October 2012.

10.2. Informative References

- [RFC6819] Lodderstedt, T., McGloin, M., and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations", RFC 6819, January 2013.

Appendix A. Notes on implementing base64url encoding without padding

This appendix describes how to implement a base64url encoding function without padding based upon standard base64 encoding function that uses padding.

To be concrete, example C# code implementing these functions is shown below. Similar code could be used in other languages.

```
static string base64urlencode(byte [] arg)
{
    string s = Convert.ToBase64String(arg); // Regular base64 encoder
    s = s.Split('=')[0]; // Remove any trailing '='s
    s = s.Replace('+', '-'); // 62nd char of encoding
    s = s.Replace('/', '_'); // 63rd char of encoding
    return s;
}
```

An example correspondence between unencoded and encoded values follows. The octet sequence below encodes into the string below, which when decoded, reproduces the octet sequence.

3 236 255 224 193

A-z_4ME

Appendix B. Example for the S256 code_challenge_method

The client uses output of a suitable random number generator to create a 32-octet sequence. The octets representing the value in this example (using JSON array notation) are:"

```
[116, 24, 223, 180, 151, 153, 224, 37, 79, 250, 96, 125, 216, 173,
187, 186, 22, 212, 37, 77, 105, 214, 191, 240, 91, 88, 5, 88, 83,
132, 141, 121]
```

Encoding this octet sequence as a Base64url provides the value of the code_verifier:

dBjftJeZ4CVP-mB92K27uhbUJUlplr_wWlgFWFOEjXk

The code_verifier is then hashed via the SHA256 hash function to produce:

```
[19, 211, 30, 150, 26, 26, 216, 236, 47, 22, 177, 12, 76, 152, 46,
8, 118, 168, 120, 173, 109, 241, 68, 86, 110, 225, 137, 74, 203,
112, 249, 195]
```

Encoding this octet sequence as a base64url provides the value of the code_challenge:

E9Melhoa2OwvFrEMTJguCHaoeKlt8URWbuGJSstw-cM

The authorization request includes:

```
code_challenge=E9Melhoa2OwvFrEMTJguCHaoeKlt8URWbuGJSstw-cM
&code_challenge_method=S256
```

The Authorization server then records the `code_challenge` and `code_challenge_method` along with the code that is granted to the client.

in the request to the `token_endpoint` the client includes the code received in the authorization response as well as the additional paramater:

```
code_verifier=dBjftJeZ4CVP-mB92K27uhbUJU1plr_wW1gFWFOEjXk
```

The Authorization server retrieves the information for the code grant. Based on the recorded `code_challenge_method` being `S256`, it then hashes and `base64url` encodes the value of `code_verifier`.
`BASE64URL-ENCODE(SHA256(ASCII("code_verifier")))`

The calculated value is then compared with the value of `"code_challenge"`:

```
BASE64URL-ENCODE(SHA256(ASCII("code_verifier" ))) == code_challenge
```

If the two values are equal then the Authorization server can provide the tokens as long as there are no other errors in the request. If the values are not equal then the request must be rejected, and an error returned.

Authors' Addresses

Nat Sakimura (editor)
Nomura Research Institute
1-6-5 Marunouchi, Marunouchi Kitaguchi Bldg.
Chiyoda-ku, Tokyo 100-0005
Japan

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

John Bradley
Ping Identity
Casilla 177, Sucursal Talagante
Talagante, RM
Chile

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

Naveen Agarwal
Google
1600 Amphitheatre Pkwy
Mountain View, CA 94043
USA

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

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: January 21, 2020

M. Jones
A. Nadalin
Microsoft
B. Campbell, Ed.
Ping Identity
J. Bradley
Yubico
C. Mortimore
Salesforce
July 20, 2019

OAuth 2.0 Token Exchange
draft-ietf-oauth-token-exchange-19

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 January 21, 2020.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents

carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
1.1. Delegation vs. Impersonation Semantics	4
1.2. Requirements Notation and Conventions	6
1.3. Terminology	6
2. Token Exchange Request and Response	6
2.1. Request	6
2.1.1. Relationship Between Resource, Audience and Scope . .	9
2.2. Response	9
2.2.1. Successful Response	9
2.2.2. Error Response	11
2.3. Example Token Exchange	11
3. Token Type Identifiers	13
4. JSON Web Token Claims and Introspection Response Parameters .	15
4.1. "act" (Actor) Claim	15
4.2. "scope" (Scopes) Claim	17
4.3. "client_id" (Client Identifier) Claim	18
4.4. "may_act" (Authorized Actor) Claim	18
5. Security Considerations	19
6. Privacy Considerations	20
7. IANA Considerations	20
7.1. OAuth URI Registration	20
7.1.1. Registry Contents	20
7.2. OAuth Parameters Registration	21
7.2.1. Registry Contents	21
7.3. OAuth Access Token Type Registration	22
7.3.1. Registry Contents	22
7.4. JSON Web Token Claims Registration	22
7.4.1. Registry Contents	22
7.5. OAuth Token Introspection Response Registration	23
7.5.1. Registry Contents	23
7.6. OAuth Extensions Error Registration	23
7.6.1. Registry Contents	23
8. References	24
8.1. Normative References	24
8.2. Informative References	24
Appendix A. Additional Token Exchange Examples	26
A.1. Impersonation Token Exchange Example	26
A.1.1. Token Exchange Request	26
A.1.2. Subject Token Claims	26
A.1.3. Token Exchange Response	27

A.1.4. Issued Token Claims	27
A.2. Delegation Token Exchange Example	28
A.2.1. Token Exchange Request	28
A.2.2. Subject Token Claims	29
A.2.3. Actor Token Claims	29
A.2.4. Token Exchange Response	29
A.2.5. Issued Token Claims	30
Appendix B. Acknowledgements	31
Appendix C. Document History	31
Authors' Addresses	35

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 security tokens provided to it and issuing new security tokens in response, 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 that 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

One common use case for an STS (as alluded to in the previous section) is to allow a resource server A to make calls to a backend service C on behalf of the requesting user B. Depending on the local site policy and authorization infrastructure, it may be desirable for A to use its own credentials to access C along with an annotation of some form that A is acting on behalf of B ("delegation"), or for A to be granted a limited access credential to C but that continues to

identify B as the authorized entity ("impersonation"). Delegation and impersonation can be useful concepts in other scenarios involving multiple participants as well.

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 insofar 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 within the context of the rights authorized by the token. A's ability to impersonate B could be limited in scope or time, or even with a one-time-use restriction, whether via the contents of the token or an out-of-band mechanism.

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 [RFC6749].

Client authentication to the authorization server is done using the normal mechanisms provided by OAuth 2.0. Section 2.3.1 of [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 bearer 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. Note that omitting client authentication allows for a compromised token to be leveraged via an STS into other tokens by anyone possessing the compromised token. Thus client authentication allows for additional authorization checks by the STS as to which entities are permitted to impersonate or receive delegations from other entities.

The client makes a token exchange request to the token endpoint with an extension grant type using the HTTP "POST" method. The following parameters are included in the HTTP request entity-body using the

"application/x-www-form-urlencoded" format with a character encoding of UTF-8 as described in Appendix B of RFC6749 [RFC6749].

grant_type

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

resource

OPTIONAL. A URI that indicates 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 a URI 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. See [I-D.ietf-oauth-resource-indicators] for additional background and uses of the "resource" parameter.

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 for the target service. 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], are examples of things that might be used as "audience" parameter values. However, "audience" values used with a given authorization server must be unique within that server, to ensure that they are properly interpreted as the intended type of value. 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 resource URIs.

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 the 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 server MUST perform the appropriate validation procedures for the indicated token type and, if the actor token is present, also perform the appropriate validation procedures for its indicated 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 exchange is a one-time event and does not create a tight linkage between the input and output tokens, so that (for example) while the expiration time of the output token may be influenced by that of the input token, renewal or extension of the input token is not expected to be reflected in the output token's properties. It may still be appropriate or desirable to propagate token revocation events. However, doing so is not a general property of the STS protocol and would be specific to a particular implementation, token type or deployment.

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 [RFC8259], 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 issued 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 more typically used with the aforementioned meaning as the structural or syntactical representation of the security token, 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 any target service 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" as discussed in Section 5.2 of [RFC6749].

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 the exchange. It trades an access token, which it received in a protected resource request, for a new token that it will use to call to a backend service (extra line breaks and indentation in the examples are for display purposes only).

Figure 1 shows the resource server receiving a protected resource request containing an OAuth access token in the Authorization 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

In Figure 2, the resource server assumes the role of client for the token exchange and the access token from the request in Figure 1 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
&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 shown in Figure 3 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": "eyJhbGciOiJIJFUzI1NiIsImtpZCI6IjllciJ9.eyJhdWQiOiJodHRwczovL2JhY2t1bmcuZm9udXhhdXBsZS5jb20iLCJpc3MiOiJodHRwczovL2FzLmV4YW1wbGUuY29tIiwiaXNjaXhwIjoiaXNDQXOTE3NTkzLCJpYXQiOiJ0E0NDE5MTc1MzMsInN1YiOi6ImJkY0BleGFtcGxlLmNvbSIsInNjb3BlIjoiaXNjaXhwIn0.40y3ZgQedw6rx  
f59WlwHDD9jryF0r0_Wh3CGozQBihNBhnXEQgU85AI9x3KmsPottVMLPIWvmDCM  
y5-kdXjwhw",
  "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 as illustrated in Figure 4.

```
GET /api HTTP/1.1
Host: backend.example.com
Authorization: Bearer eyJhbGciOiJIUzI1NiIsImtpZCI6IjllciJ9.eyJhdWQiOiJodHRwczovL2JhY2t1bmQuZXhhbXBsZS5jb20iLCJpc3MiOiJodHRwczovL2FzLmV4YW1wbGUuYy29tIiwiaXNjaWJoxNDQxOTE3NTkzLCJpYXQiOiJEONDE5MTc1MzMsInN1YiI6ImUuYy29tIiwiaWF0IjoiYXBpLn0.40y3ZgQedw6rxsf59lwhDD9jrYForO_Wh3CGozQBihNBhnXEQgU85AI9x3KmsPottVMLPIWvmDCMy5-kdXjwhw
```

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 refresh 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]).

Note that for tokens which are binary in nature, the URI used for conveying them needs to be associated with the semantics of a base64 or other encoding suitable for usage with HTTP and OAuth.

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 are not used.

Figure 5 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. The nested "act" claims serve as a history trail that connects the initial request and subject through the various delegation steps undertaken before reaching the current actor. In this sense, the current actor is considered to include the entire authorization/delegation history, leading naturally to the nested structure described here.

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 in Figure 6 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 [RFC6749].

Figure 7 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 in Figure 8 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" (Authorized Actor) 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 might be used, for example, when a "subject_token" is presented to the token endpoint in a token exchange request and "may_act" claim in the subject token can be used by the authorization server to determine whether the client (or party identified in the "actor_token") is authorized to engage in the requested delegation or impersonation.

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 are not used.

Figure 9 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: Authorized Actor 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

Much of the guidance from Section 10 of [RFC6749], the Security Considerations in The OAuth 2.0 Authorization Framework, is also applicable here. Furthermore, [RFC6819] provides additional security considerations for OAuth and [I-D.ietf-oauth-security-topics] has updated security guidance based on deployment experience and new threats that have emerged since OAuth 2.0 was originally published.

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 (in addition to other typical constraints such as a limited token lifetime) 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, MUST 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 MUST 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

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

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

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

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

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

- o URN: urn:ietf:params:oauth:token-type:saml2
- o Common Name: Token type URI for a base64url-encoded SAML 2.0 assertion
- o Change Controller: IESG
- o 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

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

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

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

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

- o Parameter name: subject_token_type
- o Parameter usage location: token request
- o Change controller: IESG

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

- o Type name: N_A
- o Additional Token Endpoint Response Parameters: (none)
- o HTTP Authentication Scheme(s): (none)
- o Change controller: IESG
- o 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

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

- o Specification Document(s): Section 4.2 of [[this specification]]
- o Claim Name: "client_id"
- o Claim Description: Client Identifier
- o Change Controller: IESG
- o Specification Document(s): Section 4.3 of [[this specification]]
- o Claim Name: "may_act"
- o Claim Description: Authorized Actor - the party that is authorized to become the actor
- o Change Controller: IESG
- o 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

- o Claim Name: "act"
- o Claim Description: Actor
- o Change Controller: IESG
- o Specification Document(s): Section 4.1 of [[this specification]]
- o Claim Name: "may_act"
- o Claim Description: Authorized Actor - the party that is authorized to become the actor
- o Change Controller: IESG
- o 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

- o Error Name: "invalid_target"
- o Error Usage Location: token error response
- o Related Protocol Extension: OAuth 2.0 Token Exchange
- o Change Controller: IETF
- o 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>>.
- [RFC7662] Richer, J., Ed., "OAuth 2.0 Token Introspection", RFC 7662, DOI 10.17487/RFC7662, October 2015, <<https://www.rfc-editor.org/info/rfc7662>>.
- [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>>.
- [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>>.

8.2. Informative References

- [I-D.ietf-oauth-resource-indicators]
Campbell, B., Bradley, J., and H. Tschofenig, "Resource Indicators for OAuth 2.0", draft-ietf-oauth-resource-indicators-02 (work in progress), January 2019.
- [I-D.ietf-oauth-security-topics]
Lodderstedt, T., Bradley, J., Labunets, A., and D. Fett, "OAuth 2.0 Security Best Current Practice", draft-ietf-oauth-security-topics-13 (work in progress), July 2019.
- [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", November 2014, <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>>.
- [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>>.
- [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>>.

[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 (with only a "subject_token" and no "actor_token", delegation is impossible). 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=eyJhbGciOiJIUzI1NiIsImtpZCI6IjE2In0.eyJhdWQiOiJodHRwczovL2FzLmV4YW1wbGUuY29tIiwiaXNzIjoiaHR0cHM6Ly9vcmlnaW5hbC1pc3N1ZXIuZXhhbXBsZS5uZXQiLCJleHAiOjE0NDE5MTA2MDAsIm5iZiI6MTQ0MTkwOTAwMCwic3ViIjoiYmRjQGV4YW1wbGUubmV0Iiwic2NvcGUiOiJvcmlcnMgcHJvZmlsZSBoaXN0b3J5In0.PRBg-jXn4cJujlgmYXFiGkZzRuzbXZ_sDxdE98ddW44ufsbWLKd3JJ1VZhF64pbTtfjy4VXFVBdaQpKjn5JzAw
&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 ("bdc@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": "bdc@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": "eyJhbGciOiJIUzI1NiIsImtpZCI6IjcyIn0.eyJhdWQiOiJlcm46ZXhhbXBsZTpjb29wZXh0IiwiaXNzIjoiaHR0cHM6Ly9hcy5leGFtcGxlLmNvbSIsImV4cCI6MTQ0MTkxMzYxMCwic3ViIjoiyMjQGV4YW1wbGUubmV0Iiwic2NvcGUiOiJvcmlld2Z2ZWkGLmQeR9ztjb6w2OXraQlkJmGjyiCq24kcB7AI2VqVxl3wSWnVKh85A",
  "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": "bdc@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=eyJhbGciOiJIUzI1NiIsImtpZCI6IjE2In0.eyJhdWQiOiJodHRwczovL2FzLmV4YWlwbGUuY29tIiwiaXNzIjoiaHR0cHM6Ly9vcmlnaW5hbC1pc3N1ZXIuZXBhbXBsZS5uZXQlLCJleHAiOjE0NDE5MTAwNjAsInNjb3BlIjoic3Rh dHVzIGZlZWQiLCJzdWIiOiJlc2VyQG V4YWlwbGUubmV0IiwibWF5X2FjdCI6eyJzdWIiOiJhZGlpbkBLEGFtcGx1Lm5ldCJ9fQ.4rPRSWihQbpMIgAmAoqaJoJAxj-p2X8_fAtAGTXrvMxU-eEZhnXqY0_AOZgLdxw5DyLzua8H_I10MCcckF-Q_g
&subject_token_type=urn%3Aietf%3Aparams%3Aoauth%3Atoken-type%3Ajwt
&actor_token=eyJhbGciOiJIUzI1NiIsImtpZCI6IjE2In0.eyJhdWQiOiJodHRwczovL2FzLmV4YWlwbGUuY29tIiwiaXNzIjoiaHR0cHM6Ly9vcmlnaW5hbC1pc3N1ZXIuZXBhbXBsZS5uZXQlLCJleHAiOjE0NDE5MTAwNjAsInN1YiI6ImFkbWluQG V4YWlwbGUubmV0In0.7YQ-3zpFfhUvzje5oqw8COCvN5uP6NsKik9CVV6cAO f4QKgM-tKfiOwcgZoUuD L2tEs6tqPlcB lMjiSzEjm3yBg
&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": "eyJhbGciOiJIUzI1NiIsImtpZCI6IjcyIn0.eyJhdWQiOiJ1cm46ZXhhbXBzZTpjb29wZXJhdGlvbiljb250ZXh0IiwiaXNzIjoiaHR0cHM6Ly9hcy5leGFtcGxlLmNvbSIsImV4cCI6MTQ0MTkxMzYxMCwic2NvcGUiOiJzdGF0dXMgZmVlZCIsInN1YiI6InVzZXJAZXhhbXBzZS5uZXQlLCJhY3QiOiOnsic3ViIjojYWWRtaW5AZXhhbXBzZS5uZXQifX0.3paKl9UySKYB5ng6_cUtQ2ql08Rc_y7Mea7IwEXTcYbNdwG9-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, Roman Danyliw, and Benjamin Kaduk serving as Security Area Directors. The following individuals contributed ideas, feedback, and wording to this specification:

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

Appendix C. Document History

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

-19

- o Fix-up changes introduced in -18.
- o Fix invalid JSON in the Nested Actor Claim example.
- o Reference figure numbers in text when introducing the examples in Section 2 and 4.
- o Editorial updates from additional IESG evaluation comments.
- o Add an informational reference to ietf-oauth-resource-indicators
- o Update ietf-oauth-security-topics ref to 13

-18

- o Editorial updates based on a few more IESG evaluation comments.

-17

- o Editorial improvements and example fixes resulting from IESG evaluation comments.
- o Added a pointer to RFC6749's Appendix B. on the "Use of application/x-www-form-urlencoded Media Type" as a way of providing a normative citation (by reference) for the media type.
- o Strengthened some of the wording in the privacy considerations to bring it inline with RFC 7519 Sec. 12 and RFC 6749 Sec. 10.8.

-16

- o Fixed typo and added an AD to Acknowledgements.

-15

- o Updated the nested actor claim example to (hopefully) be more straightforward.
- o 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
- o Moved the Security and Privacy Considerations sections to before the IANA Considerations.

-14

- o 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

- o 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.
- o Updated the client identifier claim name to be consistent with the treatment of client id in RFC 7662 OAuth 2.0 Token Introspection.

-12

- o Updated to use the boilerplate from RFC 8174.

-11

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

-10

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

-09

- o 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.

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

-08

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

-07

- o Fixed typo (desecration -> discretion).
- o 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

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

-05

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

-04

- o 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>).
- o 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>).
- o 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>).
- o 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

- o Updated the document editors (adding Campbell, Bradley, and Mortimore).
- o Added to the title.
- o Added to the abstract and introduction.
- o 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.
- o Changed the grant type identifier to urn:ietf:params:oauth:grant-type:token-exchange.
- o 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.
- o Added RFC 6749 registration requests for request/response parameters.
- o Removed the Implementation Considerations and the requirement to support JWTs.
- o Clarified many aspects of the text.
- o 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".
- o 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.
- o 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.

- o Added a "resource" request parameter used to indicate the URLs of resources at which the client intends to use the requested security token.
- o Specified that multiple "audience" and "resource" request parameter values may be used.
- o Defined the JWT claim "act" (actor) to express the current actor or delegation principal.
- o Defined the JWT claim "may_act" to express that one party is authorized to act on behalf of another party.
- o Defined the JWT claim "scp" (scopes) to express OAuth 2.0 scope-token values.
- o 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.
- o Added examples.

-02

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

-01

- o Updated references.

-00

- o Created initial working group draft from draft-jones-oauth-token-exchange-01.

Authors' Addresses

Michael B. Jones
Microsoft

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

Anthony Nadalin
Microsoft

Email: tonynad@microsoft.com

Brian Campbell (editor)
Ping Identity

Email: brian.d.campbell@gmail.com

John Bradley
Yubico

Email: ve7jtb@ve7jtb.com

Chuck Mortimore
Salesforce

Email: cmortimore@salesforce.com