OAuth Working Group   B. Campbell
Internet-Draft   Ping Identity
Intended status: Standards Track   C. Mortimore
Expires: April 24, 2015   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).

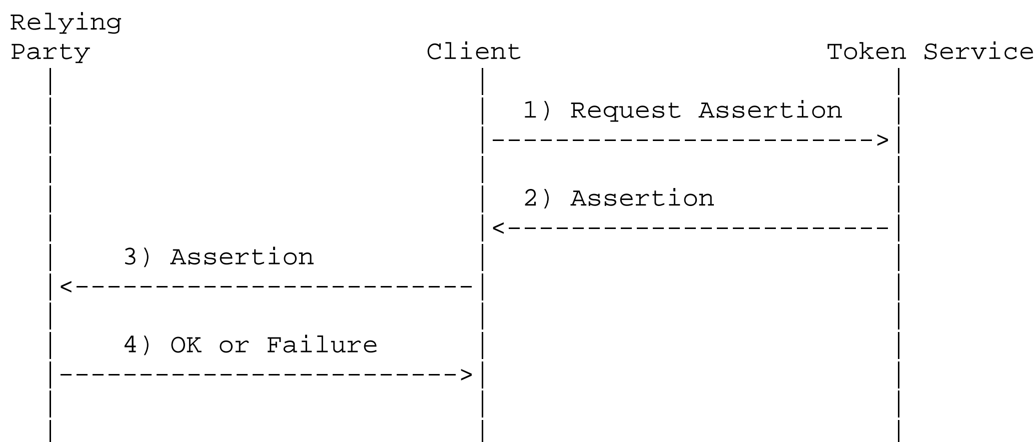


Figure 1: Third Party Created Assertion

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.

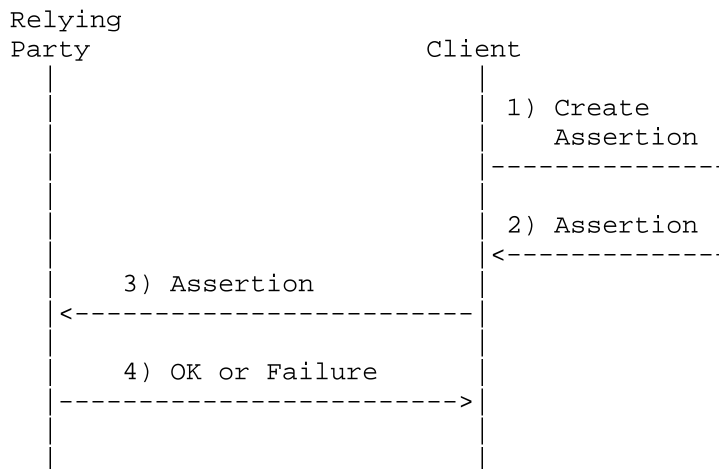


Figure 2: Self-Issued Assertion

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=i1WsRn1uB1&
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".

- 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.

- 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.

- The assertion MAY contain an Issued At entity containing the UTC time at which the assertion was issued.

- 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 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

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

9.2. client_assertion Parameter Registration

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

9.3. client_assertion_type Parameter Registration

- Parameter name: client_assertion_type
- Parameter usage location: token request
- Change controller: IESG
- 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

- Changes/suggestions from IESG reviews.

draft-ietf-oauth-assertions-17

- 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

- 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

- Updated references.

- Improved formatting of hanging lists.

draft-ietf-oauth-assertions-14

- Update reference: draft-iab-privacy-considerations is now RFC 6973

- Update reference: draft-ietf-oauth-dyn-reg from -13 to -15

draft-ietf-oauth-assertions-13

- Clean up language around subject per the subject part of http://www.ietf.org/mail-archive/web/oauth/current/msg12155.html

- Replace "Client Credentials flow" by "Client Credentials _Grant_" as suggested in http://www.ietf.org/mail- archive/web/oauth/current/msg12155.html

- 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."

- Added one-time use, maximum lifetime, and specific subject and attribute requirements to Interoperability Considerations.

draft-ietf-oauth-assertions-12

- 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

- Addressed comments from IESG evaluation https://datatracker.ietf.org/doc/draft-ietf-oauth-assertions/ballot/.

- Reworded Interoperability Considerations to state what identifiers, keys, endpoints, etc. need to be exchanged/agreed upon.

- Added brief description of assertion to the into and included a reference to Section 3 (Framework) where it’s described more.

- Changed such that a self-issued assertion must (was should) have the client id as the issuer.

- 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.

- Refined language around audience, subject, client identifiers, etc. to hopefully be clearer and less redundant.

- 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.

- 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

- Changed term "Principal" to "Subject".

- Added Interoperability Considerations section.

- 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

- Allow audience values to not be URIs.

- Added informative references to draft-ietf-oauth-saml2-bearer and draft-ietf-oauth-jwt-bearer.

- Clarified that the statements about possible issuers are non-normative by using the language "Examples of issuers".

draft-ietf-oauth-assertions-08

- Update reference to RFC 6755 from draft-ietf-oauth-urn-sub-ns

- Tidy up IANA consideration section

draft-ietf-oauth-assertions-07

- Reference RFC 6749.

- Remove extraneous word per http://www.ietf.org/mail-archive/web/oauth/current/msg10029.html

draft-ietf-oauth-assertions-06

- 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

- 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.

- 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.

- 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.

- Remove the MUST from the text prior to the HTTP parameter definitions per http://www.ietf.org/mail-archive/web/oauth/current/msg08920.html.

- 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   J. Richer, Ed.
Internet-Draft
Intended status: Standards Track   M. Jones
Expires: November 29, 2015   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

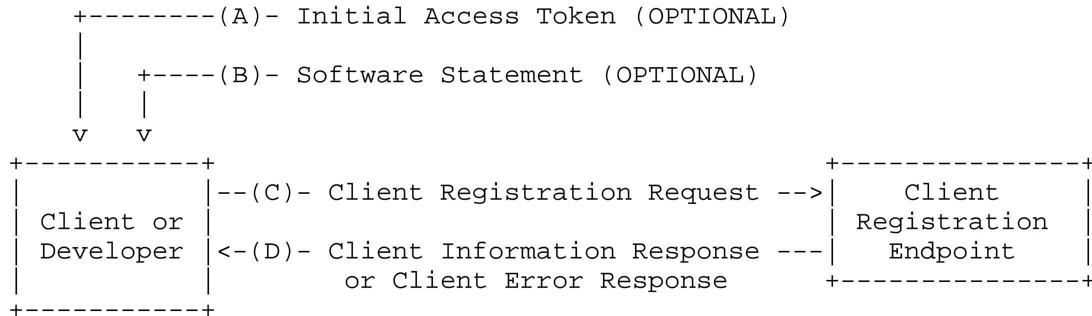


Figure 1: Abstract Dynamic Client Registration Flow

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.

jwks_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 "jwks" parameter, as it allows for easier key rotation. The "jwks_uri" and "jwks" parameters MUST NOT both be present in the same request or response.

jwks
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 "jwks_uri" parameter, such as native clients that cannot host public URLs. The "jwks_uri" and "jwks" 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 speicification, 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.
eyJzb2Z0d2FyZV9pZCI6IjROUkIxLTBYWkFCWkk5RTYtNVNNM1IiLCJjbGll
bnRfbmFtZSI6IkV4YW1wbGUgU3RhdGVtZW50LWJhc2VkIENsaWVudCIsImNs
aWVudF91cmkiOiJodHRwczovL2NsaWVudC5leGFtcGxlLm5ldC8ifQ.
GHfL4QNIrQwL18BSRdE595T9jbzqa06R9BT8w409x9oIcKaZo_mt15riEXHa
zdISUvDIZhtiyNrSHQ8K4TvqWxH6uJgcmoodZdPwmWRIEYbQDLqPNxREtYn0
5X3AR7ia4FRjQ2ojZjk5fJqJdQ-JcfxyhK-P8BAWBd6I2LLA77IG32xtbhxY
fHX7VhuU5ProJO8uvu3Ayv4XRhLZJY4yKfmyjiiKiPNe-Ia4SMy_d_QSWxsk
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": "nj3YJwsLUFl9BmpAbkOswCNVx17Eh9wMO-_AReZwBqfaWFcfG
HrZXsIV2VMCNVNU8Tpb4obUaSXcRcQ-VMsfQPJm9IzgtRdAY8NN8Xb7PEcYyk
lBjvTtuPbpzIaqyiUepzUXNDFuAOOkrIol3WmflPUUgMKULBN0EUd1fpOD70p
RM0rlp_gg_WNUKoW1V-3keYUJoXH9NztEDm_D2MQXj9eGOJJ8yPgGL8PAZMLe
2R7jb9TxOCPDED7tY_TU4nFPlxptw59A42mldEmViXsKQt60s1SLboazxFKve
qXC_jpLUt22OC6GUG63p-REw-ZOr3r845z50wMuzifQrMI9bQ",
    "kty": "RSA"
 }]},
 "example_extension_parameter": "example_value"
}
```

3.1.1. 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
bnRfbmFtZSI6IkV4YW1wbGUgU3RhdGVtZW50LWJhc2VkIENsaWVudCIsImNs
aWVudF91cmkiOiJodHRwczovL2NsaWVudC5leGFtcGxlLm5ldC8ifQ.
GHfL4QNIrQwL18BSRdE595T9jbzqa06R9BT8w409x9oIcKaZo_mt15riEXHa
zdISUvDIZhtiyNrSHQ8K4TvqWxH6uJgcmoodZdPwmWRIEYbQDLqPNxREtYn0
5X3AR7ia4FRjQ2ojZjk5fJqJdQ-JcfxyhK-P8BAWBd6I2LLA77IG32xtbhxY
fHX7VhuU5ProJO8uvu3Ayv4XRhLZJY4yKfmyjiiKiPNe-Ia4SMy_d_QSWxsk
U5XIQl5Sa2YRPMbDRXttm2TfnZM1xx70DoYi8g6czz-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. 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.

4.1.2. Initial Registry Contents

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

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

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

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

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

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

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

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

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

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

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

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

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

- 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:

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

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

- Token Endpoint Authorization Method Name: "client_secret_basic"
- Change controller: IESG
- 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:

- A remote web site protected by TLS (e.g., https://client.example.com/oauth_redirect)
- A web site hosted on the local machine using an HTTP URI (e.g., http://localhost:8080/oauth_redirect)
- 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 the 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 Matake, 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

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

-29

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

-28

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

-27

- Changed a registry name missed in -26.

-26

- Used consistent registry name.

-25

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

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

-24

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

-23

- Updated author information.

-22

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

-21

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

-20

- Applied minor editorial fixes from working group comments.

-19

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

-18

- Corrected an example HTTP response status code to be 201 Created.
- Said more about who issues and uses initial access tokens and software statements.
- Stated that the use of an initial access token is required when the authorization server limits the parties that can register a client.
- 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.
- Added the "application_type" metadata value, which had somehow been omitted.
- Added missing default metadata values, which had somehow been omitted.
- Clarified that the "software_id" is ultimately asserted by the client developer.
- Clarified that the "error" member is required in error responses, "error_description" member is optional, and other members may be present.
- Added security consideration about registrations with duplicate "software_id" and "software_version" values.

-17

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

-16

- 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.
- Addressed review comments by Phil Hunt and Tony Nadalin.

-15

- 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.
- 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.
- Created the IANA OAuth Registration Client Metadata registry for registering Client Metadata values.
- Defined Client Instance term and stated that multiple instances can use the same client identifier value under certain circumstances.
- Rewrote the introduction.
- Rewrote the Use Cases appendix.

-14

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

-13

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

-12

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

-11

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

-10

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

-09

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

-08

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

-07

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

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

-06

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

-05

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

-04

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

-03

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

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

-02

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

-01

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

-00

- 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 J. Richer, Ed.
Internet-Draft
Intended status: Experimental M. Jones
Expires: November 6, 2015 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:

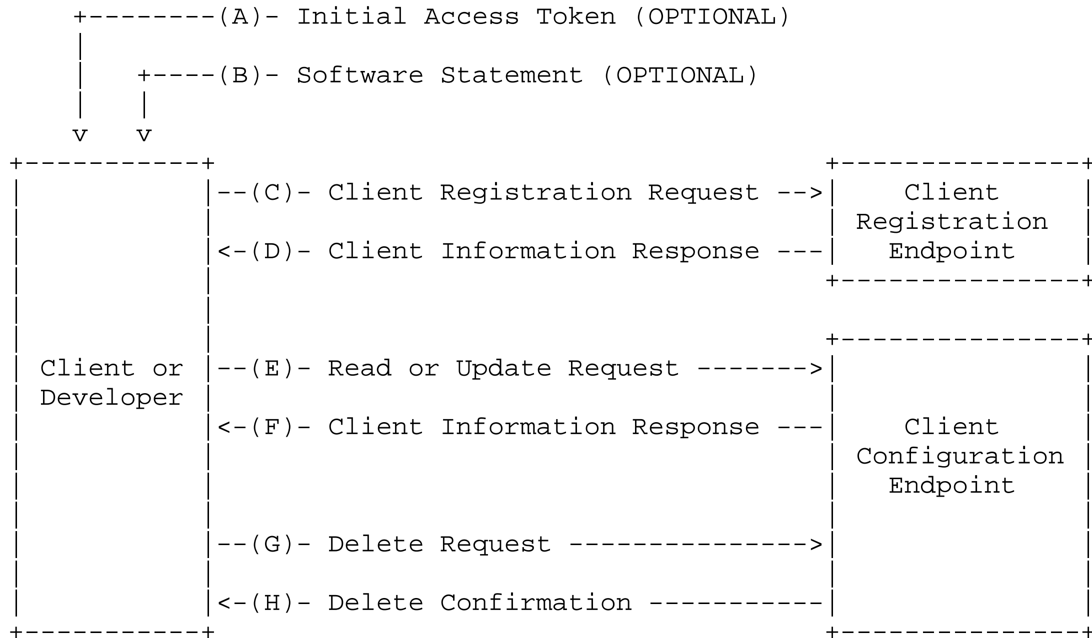


Figure 1: Abstract Extended Dynamic Client Registration Flow

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]:

- Client Metadata Name: "registration_access_token"

- Client Metadata Description: OAuth 2.0 bearer token used to access the client configuration endpoint

- Change controller: IESG

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

- Client Metadata Name: "registration_client_uri"

- Client Metadata Description: Fully qualified URI of the client registration endpoint

- Change controller: IESG

- 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 Matake, 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.

- 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.

- 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.

- 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

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

-14

- Clarified all client metadata as JSON arrays, strings, or numbers.

- Clarified experimental nature of the draft.

-13

- Changed rate-limiting suggestion to a complexity requirement.

-12

- Used consistent registry name.

-11

- 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 J. Richer
Internet-Draft The MITRE Corporation
Intended status: Standards Track M. Jones
Expires: November 23, 2014 Microsoft
J. Bradley
Ping Identity
M. Machulak
Newcastle University
P. Hunt
Oracle Corporation
May 22, 2014

OAuth 2.0 Dynamic Client Registration Metadata
draft-ietf-oauth-dyn-reg-metadata-01

Abstract

This specification is obsolete. Its previous contents have been merged into draft-ietf-oauth-dyn-reg.

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 23, 2014.

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. Security Considerations 3
3. IANA Considerations 3
4. Normative References 3
Appendix A. Document History 3
Authors' Addresses 3

1. Introduction

This specification is obsolete. Its previous contents have been merged into the OAuth 2.0 Dynamic Client Registration Protocol [OAuth.Registration] specification.

2. Security Considerations

This document has no security considerations.

3. IANA Considerations

This document has no IANA considerations.

4. Normative References

[OAuth.Registration]
Richer, J., 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 2014.

Appendix A. Document History

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

-01

- Indicated that this draft is obsolete.

-00

- Created from draft-jones-oauth-dyn-reg-metadata-00.

Authors' Addresses

Justin Richer
The MITRE Corporation

Email: jricher@mitre.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: m.p.machulak@ncl.ac.uk
URI: http://ncl.ac.uk/

Phil Hunt
Oracle Corporation

Email: phil.hunt@yahoo.com

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):

eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly
9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ

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-mB92K27uhbUJU1p1r_wW1gFWFOEjXk

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
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
.
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk

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
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ

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):

eyJhbGciOiJub25lIn0
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
.

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

- Webpage Title: (same as the protocol category)

- Registry Name: JSON Web Token Claims

]]

10.1.1. 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.

10.1.2. Initial Registry Contents

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

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

- Claim Name: "aud"
- 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.MediaTypes]
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.
QR1Owv2ug2WyPBnbQrRARTeEk9kDO2w8qDcjiHnSJflSdv1iNqhWXaKH4MqAkQtM
oNfABIPJaZm0HaA415sv3aeuBWnD8J-Ui7Ah6cWafs3ZwwFKDFUUsWHSK-IPKxLG
TkND09XyjORj_CHAgOPJ-Sd8ONQRnJvWn_hXV1BNMHzUjPyYwEsRhDhzjAD26ima
sOTsgruobpYGoQcXUwFDn7moXPRfDE8-NoQX7N7ZYMmpUDkR-Cx9obNGwJQ3nM52
YCitxoQVPzjbl7WBuB7AohdBoZOdZ24WlN1lVIeh8v1K4krB8xgKvRU8kgFrEn_a
1rZgN5TiysnmzTROF869lQ.
AxY8DCtDaGlsbGljb3RoZQ.
MKOle7UQrG6nSxTLX6Mqwt0orbHvAKeWnDYvpIAeZ72deHxz3roJDXQyhxx0wKaM
HDjUEOKIwrtkHthpqEanSBNYHZgmNOV7sln1Eu9g3J8.
fiK51VwhsxJ-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:

- The Content Encryption Key is encrypted to the recipient using the RSAES-PKCS1-V1_5 algorithm to produce the JWE Encrypted Key.
- 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.
- 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:


```
eyJhbGciOiJSU0ExXzUiLCJlbmMiOiJBMTI4Q0JDLUhTMjU2IiwiY3R5IjoiSldUIn0
```


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:

GawgguFyGrWKav7AX4VKUg

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


```
eyJhbGciOiJSU0ExXzUiLCJlbmMiOiJBMTI4Q0JDLUhTMjU2IiwiY3R5IjoiSldU
In0.
g_hEwksO1Ax8Qn7HoN-BVeBoa8FXe0kpyk_XdcSmxvcM5_P296JXXtoHISr_DD_M
qewaQSH4dZOQHoUgKLeFly-9RI11TG-_Ge1bZFazBPwKC5lJ6OLANLMd0QSL4fYE
b9ERe-epKYE3xb2jfY1AltHqBO-PM6j23Guj2yDKnFv6WO72tteVzm_2n17SBFvh
DuR9a2nHTE67pe0XGBUS_TK7ecA-iVq5COeVdJR4U4VZGGlxRGPLRHvolVLEHx6D
YyLpw30Ay9R6d68YCLi9FYTq3hIXPK_-dmPlOUlKvPr1GgJzRoeC9G5qCvdcHWsq
JGTO_z3Wfo5zsqwkxruxwA.
UmVkbW9uZCBXQSA5ODA1Mg.
VwHERHPvCNcHHpTjkoigx3_ExK0Qc71RMEParpatm0X_qpg-w8kozSjfNIPPXiTB
BLXR65CIPkFqz4l1Ae9w_uowKiwyi9acgVztAi-pSL8GQSXnaamh9kX1mdh3M_TT
-FZGQFQsFhu0Z72gJKGdfGE-OE7hS1zuBD5oEUfk0Dmb0VzWEzpxxiSSBbBAzP10
l56pPfAtrjEYw-7ygeMkwBl6Z_mLS6w6xUgKlvW6ULmkV-uLC4FUiyKECK4e3WZY
Kw1bpgIqGYsw2v_grHjszJZ-_I5uM-9RA8ycX9KqPRp9gc6pXmoU_-27ATs9XCvr
ZXUtK2902AUzqpeEUJYjWWxSNsS-r1TJ1I-FMJ4XyAiGrfmo9hQPcNBYxPz3GQb2
8Y5CLSQfNgKSGt0A4isp1hBUXBHAndgtcslt7ZoQJaKe_nNJgNliWtWpJ_ebuOpE
l8jdhehdccnRMIwAmU1n7SPkmhIl1HlSOpvcvDfhUN5wuqU955vOBvfkBOh5A11U
zBuo2WlgZ6hYi9-e3w29bR0C2-pp3jbqxEDw3iWaf2dc5b-LnR0FEYXvI_tYk5rd
_J9N0mg0tQ6RbpxNEMNoA9QWk5lgdPvbh9BaO195abQ.
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

- Replaced uses of the phrases "JWS object" and "JWE object" with "JWS" and "JWE".

- Applied other minor editorial improvements.

-31

- Updated the example IANA registration request subject line.

-30

- Applied privacy wording supplied by Stephen Farrell.

- Clarified where white space and line breaks may occur in JSON objects by referencing Section 2 of RFC 7159.

- Specified that registration reviews occur on the jwt-reg-review@ietf.org mailing list.

-29

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

-28

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

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

-27

- Removed unused reference to RFC 4648.

- Changed to use the term "authenticated encryption" instead of "encryption", where appropriate.

- Changed the registration review period to three weeks.

- Acknowledged additional contributors.

-26

- Removed an ambiguity in numeric date representations by specifying that leap seconds are handled in the manner specified by POSIX.1.

- Addressed Gen-ART review comments by Russ Housley.

- Addressed secdir review comments by Warren Kumari and Stephen Kent.

- Replaced the terms Plaintext JWS and Plaintext JWT with Unsecured JWS and Unsecured JWT.

-25

- Reworded the language about JWT implementations ignoring the "typ" parameter, explicitly saying that its processing is performed by JWT applications.

- Added a Privacy Considerations section.

-24

- Cleaned up the reference syntax in a few places.

- Applied minor wording changes to the Security Considerations section.

-23

- 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

- Revised the introduction to the Security Considerations section. Also introduced subsection headings for security considerations items.

- Added text about when applications typically would and would not use the "typ" header parameter.

-21

- Removed unnecessary informative JWK spec reference.

-20

- Changed the RFC 6755 reference to be normative.

- Changed the JWK reference to be informative.

- 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.

- Noted that octet sequences are depicted using JSON array notation.

-19

- Specified that support for Nested JWTs is optional and that applications using this specification can impose additional requirements upon implementations that they use.

- Updated the JSON reference to RFC 7159.

-18

- Clarified that the base64url encoding includes no line breaks, white space, or other additional characters.

- Removed circularity in the audience claim definition.

- Clarified that it is entirely up to applications which claims to use.

- Changed "SHOULD" to "MUST" in "in the absence of such requirements, all claims that are not understood by implementations MUST be ignored".

- 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.

- Removed a JWT creation step that duplicated a step in the underlying JWS or JWE creation.

- Added security considerations about using JWTs in trust decisions.

-17

- Corrected RFC 2119 terminology usage.

- Replaced references to draft-ietf-json-rfc4627bis with RFC 7158.

-16

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

-15

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

-14

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

-13

- Added Claim Description registry field.

- Used Header Parameter Description registry field.

- Removed the phrases "JWA signing algorithms" and "JWA encryption algorithms".

- Removed the term JSON Text Object.

-12

- 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".

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

-11

- Added a Nested JWT example.
- 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

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

-09

- Clarified that the "typ" header parameter is used in an application-specific manner and has no effect upon the JWT processing.
- 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

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

-07

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

- Changed from using the term "byte" to "octet" when referring to 8 bit values.

- Tracked encryption computation changes in the JWE specification.

-06

- 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.

- Allow JWTs to have multiple audiences.

- Applied editorial improvements suggested by Jeff Hodges, Prateek Mishra, and Hannes Tschofenig. Many of these simplified the terminology used.

- Explained why Nested JWTs should be signed and then encrypted.

- Clarified statements of the form "This claim is OPTIONAL" to "Use of this claim is OPTIONAL".

- Referenced String Comparison Rules in JWS.

- Added seriesInfo information to Internet Draft references.

-05

- Updated values for example AES CBC calculations.

-04

- 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.

- Applied changes made by the RFC Editor to RFC 6749's registry language to this specification.

- Reference RFC 6755 -- An IETF URN Sub-Namespace for OAuth.

-03

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

applied".

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

-02

- Added an example of an encrypted JWT.

- 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."

- Applied editorial suggestions.

-01

- 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.

- Moved description of how to determine whether a header is for a JWS or a JWE from the JWT spec to the JWE spec.

- Changed registration requirements from RFC Required to Specification Required with Expert Review.

- Added Registration Template sections for defined registries.

- Added Registry Contents sections to populate registry values.

- Added "Collision Resistant Namespace" to the terminology section.

- Numerous editorial improvements.

-00

- 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 B. Campbell
Internet-Draft Ping Identity
Intended status: Standards Track C. Mortimore
Expires: May 16, 2015 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=PEFzc2VydGlvbiBJc3N1ZUluc3RhbnQ9IjIwMTEtMDU
[...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].

- URN: urn:ietf:params:oauth:grant-type:saml2-bearer
- Common Name: SAML 2.0 Bearer Assertion Grant Type Profile for OAuth 2.0
- Change controller: IESG
- 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].

- 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

- fix a spelling mistake

draft-ietf-oauth-saml2-bearer-09

- 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.

- clarified that AudienceRestriction is under Conditions (even though it's implied by schema)

- fix a typo

draft-ietf-oauth-saml2-bearer-08

- fix some typos

draft-ietf-oauth-saml2-bearer-07

- update reference from draft-campbell-oauth-urn-sub-ns to draft-ietf-oauth-urn-sub-ns

- Updated to reference draft-ietf-oauth-v2-20

draft-ietf-oauth-saml2-bearer-06

- Fix three typos NamseID->NameID and (2x) Namspace->Namespace

draft-ietf-oauth-saml2-bearer-05

- Allow for subject confirmation data to be optional when Conditions contain audience and NotOnOrAfter

- 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

- using a MAY, explicitly say that the Audience can be token endpoint URL of the authorization server

- Change title to be more generic (allowing for client authn too)

- added client authentication to the abstract

- 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

- register urn:ietf:params:oauth:client-assertion-type:saml2-bearer

- remove scope parameter as it is defined in http://tools.ietf.org/html/draft-ietf-oauth-assertions

- remove assertion param registration because it [should] be in http://tools.ietf.org/html/draft-ietf-oauth-assertions

- fix typo(s) and update/add references

draft-ietf-oauth-saml2-bearer-04

- 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".

- Added "case sensitive" to scope definition to align with draft-ietf-oauth-v2-15/16.

- Updated to reference draft-ietf-oauth-v2-16

draft-ietf-oauth-saml2-bearer-03

- Cleanup of some editorial issues.

draft-ietf-oauth-saml2-bearer-02

- 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 J. Richer
Internet-Draft The MITRE Corporation
Intended status: Standards Track W. Mills
Expires: July 19, 2014 Yahoo! Inc.
H. Tschofenig, Ed.

P. Hunt
Oracle Corporation
January 15, 2014

OAuth 2.0 Message Authentication Code (MAC) Tokens
draft-ietf-oauth-v2-http-mac-05.txt

Abstract

This specification describes how to use MAC Tokens in HTTP requests to access OAuth 2.0 protected resources. An OAuth client willing to access a protected resource needs to demonstrate possession of a cryptographic key by using it with a keyed message digest function to the request.

The document also defines a key distribution protocol for obtaining a fresh session 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 July 19, 2014.

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. Terminology 4
3. Architecture 4
4. Key Distribution 6
4.1. Session Key Transport to Client 6
4.2. Session Key Transport to Resource Server 8
5. The Authenticator 9
5.1. The Authenticator 9
5.2. MAC Input String 12
5.3. Keyed Message Digest Algorithms 13
5.3.1. hmac-sha-1 13
5.3.2. hmac-sha-256 14
6. Verifying the Authenticator 14
6.1. Timestamp Verification 15
6.2. Error Handling 15
7. Example 16
8. Security Considerations 16
8.1. Key Distribution 16
8.2. Offering Confidentiality Protection for Access to Protected Resources 16
8.3. Authentication of Resource Servers 17
8.4. Plaintext Storage of Credentials 17
8.5. Entropy of Session Keys 17
8.6. Denial of Service / Resource Exhaustion Attacks 18
8.7. Timing Attacks 18
8.8. CSRF Attacks 19
8.9. Protecting HTTP Header Fields 19
9. IANA Considerations 19
9.1. JSON Web Token Claims 19
9.2. MAC Token Algorithm Registry 20
9.2.1. Registration Template 20
9.2.2. Initial Registry Contents 21
9.3. OAuth Access Token Type Registration 21
9.3.1. The "mac" OAuth Access Token Type 21
9.4. OAuth Parameters Registration 22
9.4.1. The "mac_key" OAuth Parameter 22

9.4.2. The "mac_algorithm" OAuth Parameter 22
9.4.3. The "kid" OAuth Parameter 22
10. Acknowledgments 23
11. References 23
11.1. Normative References 23
11.2. Informative References 25
Appendix A. Background Information 26
A.1. Security and Privacy Threats 26
A.2. Threat Mitigation 27
A.2.1. Confidentiality Protection 28
A.2.2. Sender Constraint 28
A.2.3. Key Confirmation 29
A.2.4. Summary 30
A.3. Requirements 31
A.4. Use Cases 35
A.4.1. Access to an 'Unprotected' Resource 35
A.4.2. Offering Application Layer End-to-End Security . . . 36
A.4.3. Preventing Access Token Re-Use by the Resource Server 36
A.4.4. TLS Channel Binding Support 36
Authors' Addresses 37

1. Introduction

This specification describes how to use MAC Tokens in HTTP requests and responses to access protected resources via the OAuth 2.0 protocol [RFC6749]. An OAuth client willing to access a protected resource needs to demonstrate possession of a symmetric key by using it with a keyed message digest function to the request. The keyed message digest function is computed over a flexible set of parameters from the HTTP message.

The MAC Token mechanism requires the establishment of a shared symmetric key between the client and the resource server. This specification defines a three party key distribution protocol to dynamically distribute this session key from the authorization server to the client and the resource server.

The design goal for this mechanism is to support the requirements outlined in Appendix A. In particular, when a server uses this mechanism, a passive attacker will be unable to use an eavesdropped access token exchanged between the client and the resource server. In addition, this mechanism helps secure the access token against leakage when sent over a secure channel to the wrong resource server if the client provided information about the resource server it wants to interact with in the request to the authorization server.

Since a keyed message digest only provides integrity protection and data-origin authentication confidentiality protection can only be

added by the usage of Transport Layer Security (TLS). This specification provides a mechanism for channel binding is included to ensure that a TLS channel is not terminated prematurely and indeed covers the entire end-to-end communication.

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].

This specification uses the Augmented Backus-Naur Form (ABNF) notation of [I-D.ietf-httpbis-p1-messaging]. Additionally, the following rules are included from [RFC2617]: auth-param.

Session Key:

> The terms mac key, session key, and symmetric key are used interchangeably and refer to the cryptographic keying material established between the client and the resource server. This temporary key used between the client and the resource server, with a lifetime limited to the lifetime of the access token. This session key is generated by the authorization server.

Authenticator:

> A record containing information that can be shown to have been recently generated using the session key known only by the client and the resource server.

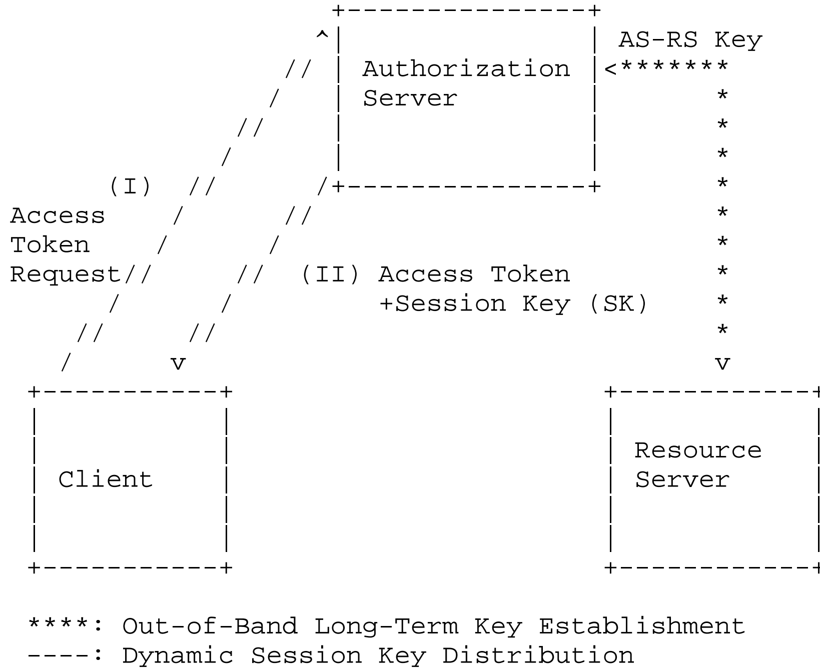
Message Authentication Code (MAC):

> Message authentication codes (MACs) are hash functions that take two distinct inputs, a message and a secret key, and produce a ﬁxed-size output. The design goal is that it is practically infeasible to produce the same output without knowledge of the key. The terms keyed message digest functions and MACs are used interchangeably.

3. Architecture

The architecture of the proposal described in this document assumes that the authorization server acts as a trusted third party that provides session keys to clients and to resource servers. These session keys are used by the client and the resource server as input to a MAC. In order to obtain the session key the client interacts

with the authorization server as part of the a normal grant exchange. This is shown in an abstract way in Figure 1. Together with the access token the authorization server returns a session key (in the mac_key parameter) and several other parameters. The resource server obtains the session key via the access token. Both of these two key distribution steps are described in more detail in Section 4.



****: Out-of-Band Long-Term Key Establishment
----: Dynamic Session Key Distribution

Figure 1: Architecture: Interaction between the Client and the Authorization Server.

Once the client has obtained the necessary access token and the session key (including parameters) it can start to interact with the resource server. To demonstrate possession of the session key it computes a MAC and adds various fields to the outgoing request message. We call this structure the "Authenticator". The server evaluates the request, includes an Authenticator and returns a response back to the client. Since the access token is valid for a period of time the resource server may decide to cache it so that it does not need to be provided in every request from the client. This interaction is shown in Figure 2.

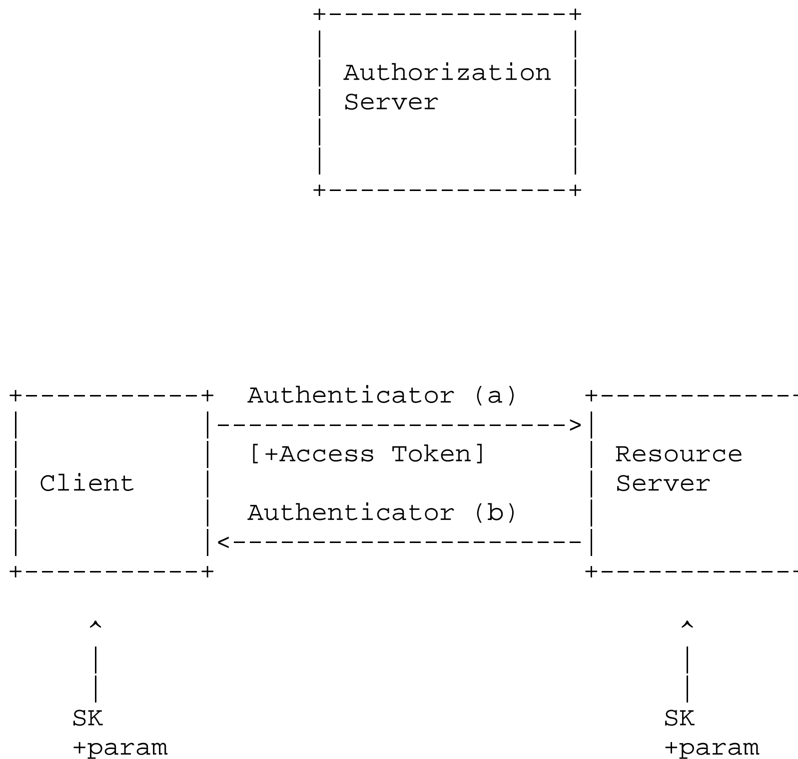


Figure 2: Architecture: Interaction between the Client and the Resource Server.

4. Key Distribution

For this scheme to function a session key must be available to the client and the resource server, which is then used as a parameter in the keyed message digest function. This document describes the key distribution mechanism that uses the authorization server as a trusted third party, which ensures that the session key is transported from the authorization server to the client and the resource server.

4.1. Session Key Transport to Client

Authorization servers issue MAC Tokens based on requests from clients. The request MUST include the audience parameter defined in [I-D.tschofenig-oauth-audience], which indicates the resource server the client wants to interact with. This specification assumes use of the 'Authorization Code' grant. If the request is processed

successfully by the authorization server it MUST return at least the following parameters to the client:

kid

The name of the key (key id), which is an identifier generated by the resource server. It is RECOMMENDED that the authorization server generates this key id by computing a hash over the access_token, for example using SHA-1, and to encode it in a base64 format.

access_token

The OAuth 2.0 access token.

mac_key

The session key generated by the authorization server. Note that the lifetime of the session key is equal to the lifetime of the access token.

mac_algorithm

The MAC algorithm used to calculate the request MAC. The value MUST be one of "hmac-sha-1", "hmac-sha-256", or a registered extension algorithm name as described in Section 9.2. The authorization server is assumed to know the set of algorithms supported by the client and the resource server. It selects an algorithm that meets the security policies and is supported by both nodes.

For example:

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

     {
       "access_token":
   "eyJhbGciOiJSU0ExXzUiLCJlbmMiOiJBMTI4Q0JDK0hTMjU2In0.
   pwaFh7yJPivLjjPkzC-GeAyHuy7AinGcS51AZ7TXnwkC80Ow1aW47kcT_UV54ubo
   nONbeArwOVuR7shveXnwPmucwrk_3OCcHrCbE1HR-Jfme2mF_WR3zUMcwqmU0RlH
   kwx9txo_sKRasjlXc8RYP-evLCmT1XRXKjtY5l44Gnh0A84hGvVfMxMfCWXh38hi
   2h8JMjQHGQ3mivVui5lbf-zzb3qXXxNO1ZYoWgs5tP1-T54QYc9Bi9wodFPWNPKB
   kY-BgewG-Vmc59JqFeprk1O08qhKQeOGCWc0WPC_n_LIpGWH6spRm7KGuYdgDMkQ
   bd4uuB0uPPLx_euVCdrVrA.
   AxY8DCtDaGlsbGljb3RoZQ.
   7MI2lRCaoyYx1HclVXkr8DhmDoikTmOp3IdEmm4qgBThFkqFqOs3ivXLJTku4M0f
   laMAbGG_X6K8_B-0E-7ak-Olm_-_V03oBUUGTAc-F0A.
   OwWNxnC-BMEie-GkFHzVWiNiaV3zUHf6fCOGTwbRckU",
       "token_type":"mac",
       "expires_in":3600,
       "refresh_token":"8xLOxBtZp8",
       "kid":"22BIjxU93h/IgwEb4zCRu5WF37s=",
       "mac_key":"adijq39jdlaska9asud",
       "mac_algorithm":"hmac-sha-256"
     }
```

4.2. Session Key Transport to Resource Server

The transport of the mac_key from the authorization server to the resource server is accomplished by conveying the encrypting mac_key inside the access token. At the time of writing only one standardized format for carrying the access token is defined: the JSON Web Token (JWT) [I-D.ietf-oauth-json-web-token]. Note that the header of the JSON Web Encryption (JWE) structure [I-D.ietf-jose-json-web-encryption], which is a JWT with encrypted content, MUST contain a key id (kid) in the header to allow the resource server to select the appropriate keying material for decryption. This keying material is a symmetric or an asymmetric long-term key established between the resource server and the authorization server, as shown in Figure 1 as AS-RS key. The establishment of this long-term key is outside the scope of this specification.

This document defines two new claims to be carried in the JWT: mac_key, kid. These two parameters match the content of the mac_key and the kid conveyed to the client, as shown in Section 4.1.

kid

The name of the key (key id), which is an identifier generated by the resource server.

mac_key

The session key generated by the authorization server.

This example shows a JWT claim set without header and without encryption:

```
{"iss":"authorization-server.example.com",
"exp":1300819380,
"kid":"22BIjxU93h/IgwEb4zCRu5WF37s=",
"mac_key":"adijq39jdlaska9asud",
"aud":"apps.example.com"
}
```

QUESTIONS: An alternative to the use of a JWT to convey the access token with the encrypted mac_key is use the token introspect [I-D.richer-oauth-introspection]. What mechanism should be described? What should be mandatory to implement?

QUESTIONS: The above description assumes that the entire access token is encrypted but it would be possible to only encrypt the session key and to only apply integrity protection to other fields. Is this desirable?

5. The Authenticator

To access a protected resource the client must be in the possession of a valid set of session key provided by the authorization server. The client constructs the authenticator, as described in Section 5.1.

5.1. The Authenticator

The client constructs the authenticator and adds the resulting fields to the HTTP request using the "Authorization" request header field. The "Authorization" request header field uses the framework defined by [RFC2617]. To include the authenticator in a subsequent response from the authorization server to the client the WWW-Authenticate header is used. For further exchanges a new, yet-to-be-defined header will be used.

```
  authenticator  = "MAC" 1*SP #params

  params         = id / ts / seq-nr / access_token / mac / h / cb

  kid            = "kid" "=" string-value
  ts             = "ts" "=" ( <"> timestamp <"> ) / timestamp
  seq-nr         = "seq-nr" "=" string-value
  access_token   = "access_token" "=" b64token
  mac            = "mac" "=" string-value
  cb             = "cb" "=" token
  h              = "h" "=" h-tag
  h-tag          = %x68 [FWS] "=" [FWS] hdr-name
                   *( [FWS] ":" [FWS] hdr-name )
  hdr-name       = token

  timestamp      = 1*DIGIT
  string-value   = ( <"> plain-string <"> ) / plain-string
  plain-string   = 1*( %x20-21 / %x23-5B / %x5D-7E )

  b64token       = 1*( ALPHA / DIGIT /
                       "-" / "." / "_" / "~" / "+" / "/" ) *"="
```

The header attributes are set as follows:

kid

REQUIRED. The key identifier.

ts

REQUIRED. The timestamp. The value MUST be a positive integer set by the client when making each request to the number of milliseconds since 1 January 1970.

The JavaScript getTime() function or the Java System.currentTimeMillis() function, for example, produce such a timestamp.

seq-nr

OPTIONAL. This optional field includes the initial sequence number to be used by the messages exchange between the client and the server when the replay protection provided by the

timestamp is not sufficient enough replay protection. This field specifies the initial sequence number for messages from the client to the server. When included in the response message, the initial sequence number is that for messages from the server to the client. Sequence numbers fall in the range 0 through 2^64 - 1 and wrap to zero following the value 2^64 - 1.

The initial sequence number SHOULD be random and uniformly distributed across the full space of possible sequence numbers, so that it cannot be guessed by an attacker and so that it and the successive sequence numbers do not repeat other sequences. In the event that more than 2^64 messages are to be generated in a series of messages, rekeying MUST be performed before sequence numbers are reused. Rekeying requires a new access token to be requested.

access_token

CONDITIONAL. The access_token MUST be included in the first request from the client to the server but MUST NOT be included in a subsequent response and in a further protocol exchange.

mac

REQUIRED. The result of the keyed message digest computation, as described in Section 5.3.

cb

OPTIONAL. This field carries the channel binding value from RFC 5929 [RFC5929] in the following format: cb= channel-binding-type ":" channel-binding-content. RFC 5929 offers two types of channel bindings for TLS. First, there is the 'tls-server-end-point' channel binding, which uses a hash of the TLS server's certificate as it appears, octet for octet, in the server's Certificate message. The second channel binding is 'tls-unique', which uses the first TLS Finished message sent (note: the Finished struct, not the TLS record layer message containing it) in the most recent TLS handshake of the TLS connection being bound to. As an example, the cb field may contain cb=tls-unique:9382c93673d814579ed1610d3

h

OPTIONAL. This field contains a colon-separated list of header field names that identify the header fields presented to the keyed message digest algorithm. If the 'h' header field is absent then the following value is set by default: h="host". The field MUST contain the complete list of header fields in the order presented to the keyed message digest algorithm. The field MAY contain names of header fields that do not exist at the time of computing the keyed message digest; nonexistent header fields do not contribute to the keyed message digest computation (that is, they are treated as the null input, including the header field name, the separating colon, the header field value, and any CRLF terminator). By including header fields that do not actually exist in the keyed message digest computation, the client can allow the resource server to detect insertion of those header fields by intermediaries. However, since the client cannot possibly know what header fields might be defined in the future, this mechanism cannot be used to prevent the addition of any possible unknown header fields. The field MAY contain multiple instances of a header field name, meaning multiple occurrences of the corresponding header field are included in the header hash. The field MUST NOT include the mac header field. Folding whitespace (FWS) MAY be included on either side of the colon separator. Header field names MUST be compared against actual header field names in a case-insensitive manner. This list MUST NOT be empty. See Section 8 for a discussion of choosing header fields.

Attributes MUST NOT appear more than once. Attribute values are limited to a subset of ASCII, which does not require escaping, as defined by the plain-string ABNF.

5.2. MAC Input String

An HTTP message can either be a request from client to server or a response from server to client. Syntactically, the two types of message differ only in the start-line, which is either a request-line (for requests) or a status-line (for responses).

Two parameters serve as input to a keyed message digest function: a key and an input string. Depending on the communication direction either the request-line or the status-line is used as the first value followed by the HTTP header fields listed in the 'h' parameter. Then, the timestamp field and the seq-nr field (if present) is concatenated.

As an example, consider the HTTP request with the new line separator character represented by "\n" for editorial purposes only. The h parameter is set to h=host, the kid is 314906b0-7c55, and the timstamp is 1361471629.

```
POST /request?b5=%3D%253D&a3=a&c%40=&a2=r%20b&c2&a3=2+q HTTP/1.1
Host: example.com

Hello World!
```

The resulting string is:

```
POST /request?b5=%3D%253D&a3=a&c%40=&a2=r%20b&c2&a3=2+q HTTP/1.1\n
1361471629\n
example.com\n
```

5.3. Keyed Message Digest Algorithms

The client uses a cryptographic algorithm together with a session key to calculate a keyed message digest. This specification defines two algorithms: "hmac-sha-1" and "hmac-sha-256", and provides an extension registry for additional algorithms.

5.3.1. hmac-sha-1

"hmac-sha-1" uses the HMAC-SHA1 algorithm, as defined in [RFC2104]:

```
mac = HMAC-SHA1 (key, text)
```

Where:

text

is set to the value of the input string as described in Section 5.2,

key

is set to the session key provided by the authorization server, and

mac

is used to set the value of the "mac" attribute, after the result string is base64-encoded per Section 6.8 of [RFC2045].

5.3.2. hmac-sha-256

"hmac-sha-256" uses the HMAC algorithm, as defined in [RFC2104], with the SHA-256 hash function, defined in [NIST-FIPS-180-3]:

```
mac = HMAC-SHA256 (key, text)
```

Where:

text

is set to the value of the input string as described in Section 5.2,

key

is set to the session key provided by the authorization server, and

mac

is used to set the value of the "mac" attribute, after the result string is base64-encoded per Section 6.8 of [RFC2045].

6. Verifying the Authenticator

When receiving a message with an authenticator the following steps are performed:

1. When the authorization server receives a message with a new access token (and consequently a new session key) then it obtains the session key by retrieving the content of the access token (which requires decryption of the session key contained inside the token). The content of the access token, in particular the audience field and the scope, MUST be verified as described in Alternatively, the kid parameter is used to look-up a cached session key from a previous exchange.

2. Recalculate the keyed message digest, as described in Section 5.3, and compare the request MAC to the value received from the client via the "mac" attribute.

3. Verify that no replay took place by comparing the value of the ts (timestamp) header with the local time. The processing of authenticators with stale timestamps is described in Section 6.1.

Error handling is described in Section 6.2.

6.1. Timestamp Verification

The timestamp field enables the server to detect replay attacks. Without replay protection, an attacker can use an eavesdropped request to gain access to a protected resource. The following procedure is used to detect replays:

- At the time the first request is received from the client for each key identifier, calculate the difference (in seconds) between the request timestamp and the local clock. The difference is stored locally for later use.

- For each subsequent request, apply the request time delta to the timestamp included in the message to calculate the adjusted request time.

- Verify that the adjusted request time is within the allowed time period defined by the authorization server. If the local time and the calculated time based in the request differ by more than the allowable clock skew (e.g., 5 minutes) a replay has to be assumed.

6.2. Error Handling

If the protected resource request does not include an access token, lacks the keyed message digest, contains an invalid key identifier, or is malformed, the server SHOULD return a 401 (Unauthorized) HTTP status code.

For example:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: MAC
```

The "WWW-Authenticate" request header field uses the framework defined by [RFC2617] as follows:

```
challenge   = "MAC" [ 1*SP #param ]
param       = error / auth-param
error       = "error" "=" ( token / quoted-string)
```

Each attribute MUST NOT appear more than once.

If the protected resource request included a MAC "Authorization" request header field and failed authentication, the server MAY include the "error" attribute to provide the client with a human-readable explanation why the access request was declined to assist the client developer in identifying the problem.

For example:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: MAC error="The MAC credentials expired"
```

7. Example

[Editor’s Note: Full example goes in here.]

8. Security Considerations

As stated in [RFC2617], the greatest sources of risks are usually found not in the core protocol itself but in policies and procedures surrounding its use. Implementers are strongly encouraged to assess how this protocol addresses their security requirements and the security threats they want to mitigate.

8.1. Key Distribution

This specification describes a key distribution mechanism for providing the session key (and parameters) from the authorization server to the client. The interaction between the client and the authorization server requires Transport Layer Security (TLS) with a ciphersuite offering confidentiality protection. The session key MUST NOT be transmitted in clear since this would completely destroy the security benefits of the proposed scheme. Furthermore, the obtained session key MUST be stored so that only the client instance has access to it. Storing the session key, for example, in a cookie allows other parties to gain access to this confidential information and compromises the security of the protocol.

8.2. 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 and responses, it provides no confidentiality protection. Consequently, eavesdroppers will have full access to request content and further messages exchanged between the client and the resource server. This could be problematic when data is exchanged that requires care, such as personal data.

When TLS is used then confidentiality can be ensured and with the use of the TLS channel binding feature it ensures that the TLS channel is cryptographically bound to the used MAC token. TLS in combination with channel bindings bound to the MAC token provide security superior to the OAuth Bearer Token.

The use of TLS in combination with the MAC token is highly recommended to ensure the confidentiality of the user's data.

8.3. Authentication of Resource Servers

This protocol allows clients to verify the authenticity of resource servers in two ways:

1. The resource server demonstrates possession of the session key by computing a keyed message digest function over a number of HTTP fields in the response to the request from the client.

2. When TLS is used the resource server is authenticated as part of the TLS handshake.

8.4. Plaintext Storage of Credentials

The MAC key works in the same way passwords do in traditional authentication systems. In order to compute the keyed message digest, the client and the resource server must have access to the MAC key in plaintext form.

If an attacker were to gain access to these MAC keys - or worse, to the resource server's or the authorization server's database of all such MAC keys - he or she would be able to perform any action on behalf of any client.

It is therefore paramount to the security of the protocol that these session keys are protected from unauthorized access.

8.5. Entropy of Session Keys

Unless TLS is used between the client and the resource server, eavesdroppers will have full access to requests sent by the client. They will thus be able to mount off-line brute-force attacks to recover the session key used to compute the keyed message digest. Authorization servers should be careful to generate fresh and unique session keys with sufficient entropy to resist such attacks for at least the length of time that the session keys are valid.

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

It is equally important that the pseudo-random number generator (PRNG) used to generate these session keys be of sufficiently high quality. Many PRNG implementations generate number sequences that may appear to be random, but which nevertheless exhibit patterns, which make cryptanalysis easier. Implementers are advised to follow the guidance on random number generation in [RFC4086].

8.6. Denial of Service / Resource Exhaustion Attacks

This specification includes a number of features which may make resource exhaustion attacks against resource servers possible. For example, a resource server may need to need to consult back-end databases and the authorization server to verify an incoming request including an access token before granting access to the protected resource.

An attacker may exploit this to perform a denial of service attack by sending a large number of invalid requests to the server. The computational overhead of verifying the keyed message digest alone is, however, not sufficient to mount a denial of service attack since keyed message digest functions belong to the computationally fastest cryptographic algorithms. The usage of TLS does, however, require additional computational capabity to perform the asymmetric cryptographic operations. For a brief discussion about denial of service vulnerabilities of TLS please consult Appendix F.5 of RFC 5246 [RFC5246].

8.7. Timing Attacks

This specification makes use of HMACs, for which a signature verification involves comparing the received MAC string to the expected one. If the string comparison operator operates in observably different times depending on inputs, e.g. because it compares the strings character by character and returns a negative

result as soon as two characters fail to match, then it may be possible to use this timing information to determine the expected MAC, character by character.

Implementers are encouraged to use fixed-time string comparators for MAC verification. This means that the comparison operation is not terminated once a mismatch is found.

8.8. CSRF Attacks

A Cross-Site Request Forgery attack occurs when a site, evil.com, initiates within the victim's browser the loading of a URL from or the posting of a form to a web site where a side-effect will occur, e.g. transfer of money, change of status message, etc. To prevent this kind of attack, web sites may use various techniques to determine that the originator of the request is indeed the site itself, rather than a third party. The classic approach is to include, in the set of URL parameters or form content, a nonce generated by the server and tied to the user's session, which indicates that only the server could have triggered the action.

Recently, the Origin HTTP header has been proposed and deployed in some browsers. This header indicates the scheme, host, and port of the originator of a request. Some web applications may use this Origin header as a defense against CSRF.

To keep this specification simple, HTTP headers are not part of the string to be MACed. As a result, MAC authentication cannot defend against header spoofing, and a web site that uses the Host header to defend against CSRF attacks cannot use MAC authentication to defend against active network attackers. Sites that want the full protection of MAC Authentication should use traditional, cookie-tied CSRF defenses.

8.9. Protecting HTTP Header Fields

This specification provides flexibility for selectively protecting header fields and even the body of the message. At a minimum the following fields are included in the keyed message digest.

9. IANA Considerations

9.1. JSON Web Token Claims

This document adds the following claims to the JSON Web Token Claims registry established with [I-D.ietf-oauth-json-web-token]:

o Claim Name: "kid"

- Change Controller: IETF
- Specification Document(s): [[this document]]
- Claim Name: "mac_key"
- Change Controller: IETF
- Specification Document(s): [[this document]]

9.2. MAC Token Algorithm Registry

This specification establishes the MAC Token Algorithm registry.

Additional keyed message digest algorithms are registered on the advice of one or more Designated Experts (appointed by the IESG or their delegate), with a Specification Required (using terminology from [RFC5226]). 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 should be sent to the [TBD]@ietf.org mailing list for review and comment, with an appropriate subject (e.g., "Request for MAC Algorithm: example"). [[Note to RFC-EDITOR: The name of the mailing list should be determined in consultation with the IESG and IANA. Suggested name: http-mac-ext-review.]]

Within at most 14 days of the request, 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.

Decisions (or lack thereof) made by the Designated Expert can be first appealed to Application Area Directors (contactable using app-ads@tools.ietf.org email address or directly by looking up their email addresses on http://www.iesg.org/ website) and, if the appellant is not satisfied with the response, to the full IESG (using the iesg@iesg.org mailing list).

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

9.2.1. Registration Template

Algorithm name:

The name requested (e.g., "example").

Change controller:

For standards-track RFCs, state "IETF". For others, give the name of the responsible party. Other details (e.g., postal address, e-mail address, home page URI) may also be included.

Specification document(s):

Reference to document that specifies the algorithm, preferably including a URI that can be used to retrieve a copy of the document. An indication of the relevant sections may also be included, but is not required.

9.2.2. Initial Registry Contents

The HTTP MAC authentication scheme algorithm registry's initial contents are:

- Algorithm name: hmac-sha-1

- Change controller: IETF

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

- Algorithm name: hmac-sha-256

- Change controller: IETF

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

9.3. OAuth Access Token Type Registration

This specification registers the following access token type in the OAuth Access Token Type Registry.

9.3.1. The "mac" OAuth Access Token Type

Type name:

mac

Additional Token Endpoint Response Parameters:

secret, algorithm

HTTP Authentication Scheme(s):

MAC

Change controller:

IETF

Specification document(s):

[[this document]]

9.4. OAuth Parameters Registration

This specification registers the following parameters in the OAuth Parameters Registry established by [RFC6749].

9.4.1. The "mac_key" OAuth Parameter

Parameter name: mac_key

Parameter usage location: authorization response, token response

Change controller: IETF

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

Related information: None

9.4.2. The "mac_algorithm" OAuth Parameter

Parameter name: mac_algorithm

Parameter usage location: authorization response, token response

Change controller: IETF

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

Related information: None

9.4.3. The "kid" OAuth Parameter

Parameter name: kid

Parameter usage location: authorization response, token response

Change controller: IETF

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

Related information: None

10. Acknowledgments

This document is based on OAuth 1.0 and we would like to thank Eran Hammer-Lahav for his work on incorporating the ideas into OAuth 2.0. As part of this initial work the following persons provided feedback: Ben Adida, Adam Barth, Rasmus Lerdorf, James Manger, William Mills, Scott Renfro, Justin Richer, Toby White, Peter Wolanin, and Skylar Woodward

Further work in this document was done as part of OAuth working group conference calls late 2012/early 2013 and in design team conference calls February 2013. 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 re-use content from [RFC4962] and the authors would like thank Russ Housely and Bernard Aboba for their work on RFC 4962.

11. References

11.1. Normative References

[I-D.ietf-httpbis-p1-messaging]
Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", draft-ietf-httpbis-p1-messaging-25 (work in progress), November 2013.

[I-D.ietf-jose-json-web-encryption]
Jones, M., Rescorla, E., and J. Hildebrand, "JSON Web Encryption (JWE)", draft-ietf-jose-json-web-encryption-19 (work in progress), December 2013.

[I-D.ietf-oauth-json-web-token]
Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", draft-ietf-oauth-json-web-token-14 (work in progress), December 2013.

[I-D.richer-oauth-introspection]
Richer, J., "OAuth Token Introspection", draft-richer-oauth-introspection-04 (work in progress), May 2013.

[I-D.tschofenig-oauth-audience]

Tschofenig, H., "OAuth 2.0: Audience Information", draft-tschofenig-oauth-audience-00 (work in progress), February 2013.

[RFC2045] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, November 1996.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.

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

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[RFC2617] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication", RFC 2617, June 1999.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.

[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.

[RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, April 2011.

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

[W3C.REC-html401-19991224]
Hors, A., Raggett, D., 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>.

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.

[I-D.tschofenig-oauth-hotk]
Bradley, J., Hunt, P., Nadalin, A., and H. Tschofenig, "The OAuth 2.0 Authorization Framework: Holder-of-the-Key Token Usage", draft-tschofenig-oauth-hotk-02 (work in progress), February 2013.

[NIST-FIPS-180-3]
National Institute of Standards and Technology, "Secure Hash Standard (SHS). FIPS PUB 180-3, October 2008", October 2008.

[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.

[RFC4086] Eastlake, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, June 2005.

[RFC4120] Neuman, C., Yu, T., Hartman, S., and K. Raeburn, "The Kerberos Network Authentication Service (V5)", RFC 4120, July 2005.

[RFC4279] Eronen, P. and H. Tschofenig, "Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)", RFC 4279, December 2005.

[RFC4962] Housley, R. and B. Aboba, "Guidance for Authentication, Authorization, and Accounting (AAA) Key Management", BCP 132, RFC 4962, July 2007.

[RFC5056] Williams, N., "On the Use of Channel Bindings to Secure Channels", RFC 5056, November 2007.

[RFC5849] Hammer-Lahav, E., "The OAuth 1.0 Protocol", RFC 5849, April 2010.

[RFC5929] Altman, J., Williams, N., and L. Zhu, "Channel Bindings for TLS", RFC 5929, July 2010.

[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.

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

Appendix A. Background Information

With the desire to define a security mechanism in addition to bearer tokens a design team was formed to collect threats, explore different threat mitigation techniques, describe use cases, and to derive requirements for the MAC token based security mechanism defined in the body of this document. This appendix provides information about this thought process that should help to motivate design decision.

A.1. Security and Privacy Threats

The following list presents several common threats against protocols utilizing some form of tokens. 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 tokens 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 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 re-use 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.

We excluded one threat from the list, namely '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. We believe that such a property is interesting but most deployments prefer to deal with the violation of this security property through business actions rather than by using cryptography.

A.2. Threat Mitigation

A large range of threats can be mitigated by protecting the content of the token, 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 the subsequent description we assume that the token itself is digitally signed by the Authorization Server and therefore cannot be modified.

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, Transport Layer Security with a ciphersuite that offers confidentiality protection has to be applied. 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.

A.2.1. 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 Appendix A.2.2 and Appendix A.2.3 as well. Furthermore, the Client has to make sure it does not distribute 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.

A.2.2. Sender Constraint

Instead of providing confidentiality protection the Authorization Server could also put the identifier of the Client into the protected token with the following semantic: 'This token is only valid when presented by a Client with the following identifier.' When the access token is then presented to the Resource Server how does it know that it was provided by the Client? It has to authenticate the Client! There are many choices for authenticating the Client to the Resource Server, for example by using client certificates in TLS [RFC5246], or pre-shared secrets within TLS [RFC4279]. The choice of the preferred authentication mechanism and credential type may depend on a number of factors, including

- security properties

- 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.

A.2.3. Key Confirmation

A variation of the mechanism of sender authentication described in Appendix A.2.2 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], Kerberos [RFC4120] when utilizing the AP_REQ/AP_REP exchange (see also [I-D.hardjono-oauth-kerberos] for a comparison between Kerberos and OAuth), the Holder-of-the-Key approach [I-D.tschofenig-oauth-hotk], and also the MAC token approach defined in this document.

To illustrate key confirmation the first examples borrow from Kerberos and use symmetric key cryptography. Assume that the Authorization Server shares a long-term secret with the Resource Server, called K(Authorization Server-Resource Server). This secret would be established between them in an initial registration phase. When the Client requests an access token the Authorization Server creates a fresh and unique session key Ks and places it into the token encrypted with the long term key K(Authorization Server-Resource Server). Additionally, the Authorization Server attaches Ks to the response message to the Client (in addition to the access token itself) over a confidentiality protected channel. When the Client sends a request to the Resource Server it has to use Ks to compute a keyed message digest for the request (in whatever form or whatever layer). The Resource Server, when receiving the message, retrieves the access token, verifies it and extracts K(Authorization Server-Resource Server) to obtain Ks. This key Ks is then used to verify the keyed message digest of the request message.

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

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

A.2.4. Summary

As a high level message, there are various ways how the threats can be mitigated and while the details of each solution is somewhat different they all ultimately accomplish the goal.

The three approaches are:

Confidentiality Protection:

The weak point with this approach, which is briefly described in Appendix A.2.1, 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 Appendix A.2.2, is to setup the authentication infrastructure such that Clients can be authenticated towards Resource Servers. Additionally, 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 Web server load balancing, lack of API access to identity information, etc.

Key Confirmation:

The weak point with this approach, see Appendix A.2.3, 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.

A.3. Requirements

In an attempt to address the threats described in Appendix A.1 the Bearer Token, which corresponds to the description in Appendix A.2.1, was standardized and the work on a JSON-based token format has been started [I-D.ietf-oauth-json-web-token]. The required capability to protected the content of a JSON token using integrity and confidentiality mechanisms is work in progress at the time of writing.

Consequently, the purpose of the remaining document is to provide security that goes beyond the Bearer Token offered security protection.

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. The authorization checking prevents an elevation of privilege attack, and it ensures that an unauthorized authorized is detected.

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. This requires that each Authorization Server obtains different keying material as well as different access tokens with content that does not allow identification of the Resource Owner.

AS-to-RS Relationship Anonymity:

This MAC Token security does not provide AAS-to-RS Relationship Anonymity since the Client has to inform the resource server about the Resource Server it wants to talk to. The Authorization Server needs to know how to encrypt the session key the Client and the Resource Server will be using.

As an additional requirement 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.

Furthermore, there are performance concerns specifically with the usage of asymmetric cryptography. As such, the requirement can be phrases as 'faster is better'. [QUESTION: How are we trading the benefits of asymmetric cryptography against the performance impact?]

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

A.4. Use Cases

This section lists use cases that provide additional requirements and constrain the solution space.

A.4.1. Access to an 'Unprotected' Resource

This use case is for a web client that needs to access a resource where no integrity and confidentiality protection is provided for the exchange of data using TLS following the OAuth-based request. In accessing the resource, the request, which includes the access token, must be protected against replay, and modification.

While it is possible to utilize bearer tokens in this scenario, as described in [RFC6750], with TLS protection when the request to the protected resource is made 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 represent 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.

A.4.2. Offering Application Layer End-to-End Security

In Web deployments resource servers are often placed behind load balancers. Note that the load balancers are deployed by the same organization that operates the resource servers. These load balancers may terminate Transport Layer Security (TLS) and the resulting HTTP traffic may be transmitted in clear from the load balancer to the resource server. With application layer security independent of 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.

A.4.3. Preventing Access Token Re-Use by the Resource Server

Imagine a scenario where a resource server that receives a valid access token re-uses it with other resource server. The reason for re-use may be malicious or may well be legitimate. In a legitimate use case consider a case where the resource server needs to consult third party resource servers to complete the requested operation. In both cases it may be assumed that the scope of the access token is sufficiently large that it allows 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 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.

A.4.4. TLS 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 but the client and the resource server demand that the underlying TLS exchange is bound to additional application layer security to prevent cases where the TLS connection is terminated at a load balancer or a TLS proxy is used that splits the TLS connection into two separate connections.

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

Authors' Addresses

Justin Richer
The MITRE Corporation

Email: jricher@mitre.org

William Mills
Yahoo! Inc.

Email: wmills@yahoo-inc.com

Hannes Tschofenig (editor)
Austria

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

Phil Hunt
Oracle Corporation

Email: phil.hunt@yahoo.com

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: July 18, 2014

J. Bradley
Ping Identity
P. Hunt
Oracle Corporation
T. Nadalin
Microsoft
H. Tschofenig

January 14, 2014

The OAuth 2.0 Authorization Framework: Holder-of-the-Key Token Usage
draft-tschofenig-oauth-hotk-03.txt

Abstract

OAuth 2.0 deployments currently rely on bearer tokens for securing access to protected resources. Bearer tokens require Transport Layer Security to be used between an OAuth client and the resource server when presenting the access token. The security model is based on proof-of-possession: access token storage and transfer has to be done with care to prevent leakage.

There are, however, use cases that require a more active involvement of the OAuth client for an increased level of security, particularly to secure against token leakage. This document specifies an OAuth security framework using the holder-of-the-key concept, which requires the OAuth client when presenting an OAuth access token to also demonstrate knowledge of keying material that is bound to the token.

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 July 18, 2014.

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
2. Terminology 3
3. Protocol Specification 3
3.1. Binding a Key to an Access Token 4
3.1.1. Symmetric Keys 4
3.1.2. Asymmetric Keys 7
3.2. Accessing a Protected Resource 9
3.2.1. Symmetric Keys 9
3.2.2. Asymmetric Keys 11
4. Security Considerations 11
4.1. Security Threats 12
4.2. Threat Mitigation 12
4.3. Summary of Recommendations 13
5. IANA Considerations 14
5.1. OAuth Parameters Registration 14
5.2. The 'hotk' JSON Web Token Claims 15
5.3. The 'hotk' OAuth Access Token Type 15
5.4. Profile Registry 15
6. Acknowledgements 16
7. References 16
7.1. Normative References 16
7.2. Informative References 17
Authors' Addresses 17

1. Introduction

At the time of writing the OAuth 2.0 [3] and accompanying protocols offer one main security mechanism to access protected resources, namely the bearer token. In [12] a bearer token is defined as

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

This specification defines a new security mechanism for usage with OAuth that combines various existing specifications to offer enhanced security properties for OAuth. The incredients for this security solution are:

1. A mechanism for dynamic key distribution.

2. Data elements to bind emphemeral keying material to an access token. For the access token we assume a JSON Web Token (JWT) [6] in this specification to specify a complete solution. Future specifications may make this functionality available to other access token formats as well.

3. A mechanism to allow the OAuth client to demonstrate a proof of possession.

The rest of the document describes how these different components work together.

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].

3. Protocol Specification

To describe the architecture of the proposed security mechanism it is best to start by looking at the main OAuth 2.0 protocol exchange sequence. Figure 1 shows the abstract OAuth 2.0 protocol exchanges graphically. The exchange in this document will focus on two interactions, namely

1. to allow the client to obtain the ephemeral asymmetric credentails in step (D)

2. to use the obtained asymmetric credentials for the interaction with the resource server in step (E)

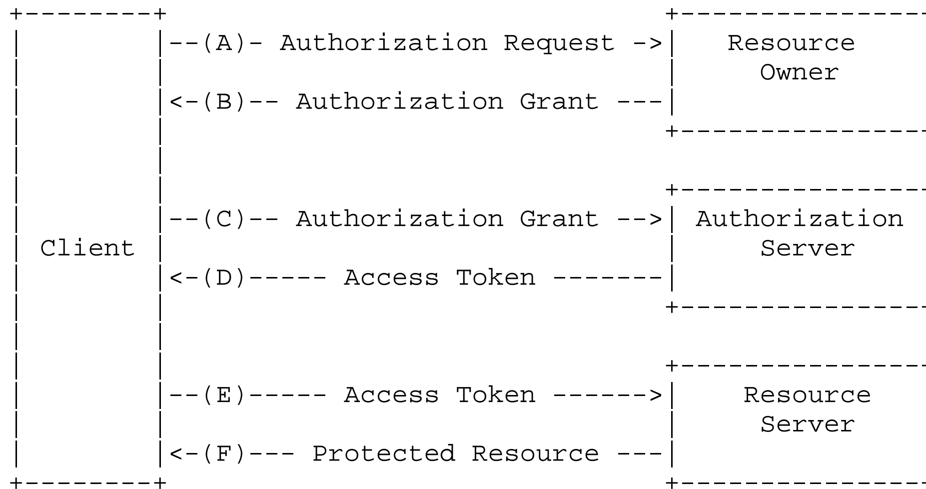


Figure 1: Abstract OAuth 2.0 Protocol Flow

3.1. Binding a Key to an Access Token

OAuth 2.0 offers different ways to obtain an access token, namely using authorization grants and using a refresh token. The core OAuth specification defines four authorization grants, see Section 1.3 of [3], and [11] adds an assertion-based authorization grant to that list.

This document extends the communication with the token endpoint. The token endpoint, which is described in Section 3.2 of [3], is used with every authorization grant except for the implicit grant type. In the implicit grant type the access token is issued directly.

Two types of keying material can be bound to an access token, namely symmetric keys and asymmetric keys, and we explain them in separate sub-sections.

3.1.1. Symmetric Keys

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

token_type: REQUIRED. For the symmetric holder-of-the-key variant the value MUST be set to "hotk-sk".

profile: REQUIRED. The profile parameter provides information about what mechanisms the client supports to provide proof of possession of the key towards a resource server. The value MUST be taken from the algorithm registry created in Section 5.4. Algorithm names are case-sensitive. If the client supports more than one profile then each individual value MUST be separated by a comma.

For example, the client makes the following HTTP request using TLS (extra line breaks are for display purposes only):

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

grant_type=authorization_code&code=SplxlOBeZQQYbYS6WxSbIA
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
&token_type=hotk-sk
&profile=jws,mac
```

Example Request to the Authorization Server

If the access token request is valid and authorized, the authorization server issues an access token and optionally a refresh token. If the request client authentication failed or is invalid, the authorization server returns an error response as described in Section 5.2 of [3].

The authorization server MUST include the following parameters in a successful response, if it supports any of the profiles listed by the client.

id: REQUIRED. An ephemeral and unique key identifier. The authorization server MUST NOT select the same key identifier twice within the lifetime of the access token, which is indicated by the 'expires_in' parameter.

key: REQUIRED. A fresh and unique shared symmetric secret with sufficient entrophy.

profile: REQUIRED. The profile parameter provides further information about how the client has to provide proof of

possession of the key with the resource server. The authorization server chooses a value from the list of supported mechanisms supported by the client.

For example:

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

{
  "access_token":"SlAV.....32hkKG",
  "token_type":"hotk-sk",
  "expires_in":3600,
  "refresh_token":"8xLOxBtZp8",
  "id":"client12345@example.com",
  "key":"adijq39jdlaska9asud",
  "profile":"jws"
}
```

The content of the 'access_token' MUST contain the key identifier value in the 'hotk' element, as shown in the example below.

```
{"typ":"JWT",
 "alg":"HS256"
}
.
{"iss":"authorization-server-id",
 "exp":1300819380,
 "hotk":"client12345@example.com"
}
.
bbfAAtVT86zwu1RK7aPFFxuhDR1L6tSoc_BJECPebWKRXjBZC
```

DISCUSSION: Should we put the encrypted key into the access token? This would make the mechanism more similar to a Kerberos-based scheme.

The key identifier, the key, and the profile name MUST NOT include characters other than:

```
%x20-21 / %x23-5B / %x5D-7E
; Any printable ASCII character except for <"> and <\>
```

3.1.2. Asymmetric Keys

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

token_type: REQUIRED. For the asymmetric holder-of-the-key variant the value MUST be set to "hotk-pk".

pk_info: REQUIRED. This field contains information about the public key the client would like to bind to the access token in the JSON Web Key format. The public key is "application/x-www-form-urlencoded" encoded.

For example, the client makes the following HTTP request using TLS (extra line breaks are for display purposes only):

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

grant_type=authorization_code&code=SplxlOBeZQQYbYS6WxSbIA
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
&token_type=hotk-pk
&pk_info=eZQQYbYS6WxS...lxlOB
```

whereby the content of the pk_info field represents the following structure:


```
{"keys":
   [
     {"alg":"RSA",
      "mod": "0vx7agoebGcQSuuPiLJXZptN9nndrQmbXEps2aiAFbWhM78LhWx
4cbbfAAtVT86zwu1RK7aPFFxuhDR1L6tSoc_BJECPebWKRXjBZCiFV4n3oknjhMs
tn64tZ_2W-5JsGY4Hc5n9yBXArwl93lqt7_RN5w6Cf0h4QyQ5v-65YGjQR0_FDW2
QvzqY368QQMicAtaSqzs8KJZgnYb9c7d0zgdAZHzu6qMQvRL5hajrn1n91CbOpbI
SD08qNLyrdkt-bFTWhAI4vMQFh6WeZu0fM4lFd2NcRwr3XPksINHaQ-G_xBniIqb
w0Ls1jF44-csFCur-kEgU8awapJzKnqDKgw",
      "exp":"AQAB",
      "kid":"2011-04-29"}
   ]
}
```


Example Request to the Authorization Server

If the access token request is valid and authorized, the authorization server issues an access token and optionally a refresh token. If the request client authentication failed or is invalid, the authorization server returns an error response as described in Section 5.2 of [3].

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, called 'hotk-pk', is placed into the 'token_type' parameter.

An example of a successful response is shown below:

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

{
  "access_token":"2YotnFZFE....jr1zCsicMWpAA",
  "token_type":"hotk-pk",
  "expires_in":3600,
  "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA"
}
```

whereby the content of the 'access_token' field, for example, contains an encoded JWT with the following raw structure:


```
{"typ":"JWT",
 "alg":"HS256"}
.
{"iss":"authorization-server-id",
 "exp":1300819380,
 "hotk": {"keys":
 [
   {"alg":"RSA",
    "mod": "0vx7agoebGcQSuuPiLJXZptN9nndrQmbXEps2aiAFbWhM78LhWx
4cbbfAAtVT86zwu1RK7aPFFxuhDR1L6tSoc_BJECPebWKRXjBZCiFV4n3oknjhMs
tn64tZ_2W-5JsGY4Hc5n9yBXArwl93lqt7_RN5w6Cf0h4QyQ5v-65YGjQR0_FDW2
QvzqY368QQMicAtaSqzs8KJZgnYb9c7d0zgdAZHzu6qMQvRL5hajrn1n91CbOpbI
SD08qNLyrdkt-bFTWhAI4vMQFh6WeZu0fM4lFd2NcRwr3XPksINHaQ-G_xBniIqb
w0Ls1jF44-csFCur-kEgU8awapJzKnqDKgw",
    "exp":"AQAB",
    "kid":"2011-04-29"}
 ]
}
}
.
bbfAAtVT86zwu1RK7aPFFxuhDR1L6tSoc_BJECPebWKRXjBZC
```


Example Response from the Authorization Server

3.2. Accessing a Protected Resource

Accessing a protected resource depends on the chosen credential type.

3.2.1. Symmetric Keys

When a symmetric key was used as a holder-of-the-key then the client has to demonstrate possession of the key that corresponds to the key identifier found in the access token.

This specification defines three ways for providing this proof of possession, which are indicated as profiles in Section 3.1.1:

jws: When the 'jws' profile is chosen then the client MUST compute the following string by concatenating together, in order, the following HTTP request elements:

1. The HTTP request method in upper case. For example: "HEAD", "GET", "POST", etc.

2. The HTTP request-URI as defined by Section 5.1.2 of [2].

3. The hostname included in the HTTP request using the "Host" request header field in lower case.

4. The port as included in the HTTP request using the "Host" request header field. If the header field does not include a port, the default value for the scheme MUST be used (e.g., 80 for HTTP and 443 for HTTPS).

5. The value of the "ext" "Authorization" request header field attribute if one was included in the request, otherwise, an empty string.

Each element is followed by a new line character (%x0A) including the last element and even when an element value is an empty string. The resulting value MUST be put into the "request" element of a JSON document that is then subject to JWS processing [7]. The resulting JWS structure is put into the body of the HTTP request. A receiving authorization server MUST use the value in the 'kid' structure to identify the shared key and then use that key to verify the keyed message digest. Additionally, the content of the 'request' field needs to be verified against the HTTP header information. If any of these verification steps fail then the request to the protected resource MUST fail with a "401 Unauthorized" error message back to the OAuth client.

The following example shows and the corresponding encoding in a JWS structure:

1) HTTP Request

```
POST /request?b5=%3D%253D&a3=a&c%40=&a2=r%20b&c2&a3=2+q HTTP/1.1
Host: example.com
```

2) JWS Document

```
{"typ":"HOTK-SK",
 "alg":"HS256",
 "kid":"client12345@example.com",
 "timestamp":"2012-07-15T10:20:00.000-05:00" }
.
{"request":"POST/request?b5=%3D%253D&a3=a&c%40=&a2=r%20b&c2&a3=
 2+qexample.com80"}
.
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

JWS Example

mac: When the 'mac' profile is chosen then the client MUST follow the description in [10].

3.2.2. Asymmetric Keys

The client accesses protected resources by presenting the access token to the resource server. It does so via a Transport Layer Security (TLS) secured channel. Since the client had previously bound a public key to an access token it selects this key for usage with TLS as described in [5].

The resource server validates the access token and ensure it has not expired and that its scope covers the requested resource. Additionally, the resource server verifies that the public key presented during the TLS handshake corresponds to the public key that is contained in the access token.

Note that this step confirms that the client is in possession of the private key corresponding to the public key previously bound to the access token. Information about the client authentication may be contained in the token in case the authorization server added this information when it authenticated the client.

4. Security Considerations

4.1. Security Threats

The following list presents several common threats against protocols utilizing some form of tokens. This list of threats is based on NIST Special Publication 800-63 [14]. We exclude a discussion of threats related to any form of registration and authentication.

Token manufacture/modification: An attacker may craft a fake token or modify the token content (such as the authentication or attribute statements), causing a resource server to grant inappropriate access to the attacker. For example, an attacker may modify the token to extend the validity period or the scope to have extended access to information.

Token disclosure: Tokens may contain authentication and attribute statements that include sensitive information.

Token redirect: An attacker uses a token generated for consumption by one resource server to gain access to a different resource server that mistakenly believes the token to be for it.

Token reuse: An attacker attempts to use a token that has already been used with that resource server in the past.

4.2. Threat Mitigation

A large range of threats can be mitigated by protecting the contents of the access token by using a digital signature or a Message Authentication Code (MAC). Consequently, the token integrity protection MUST be sufficient to prevent the token from being modified.

To deal with token redirect, it is important for the authorization server to include the identity of the intended recipients (the audience), typically a single resource server (or a list of resource servers), in the token. Restricting the use of the token to a specific scope is also RECOMMENDED.

The authorization server MUST implement and use TLS. Which version(s) 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 [8] is the most recent version. The client MUST validate the TLS certificate chain when making requests to protected resources, including checking the Certificate Revocation List (CRL) [9].

For the interaction between the client and the resource server this specification requires a TLS extension for usage with out-of-band

validation [5] to be used that allows clients to present raw public keys for asymmetric holder-of-the-key usage.

With the usage of the holder-of-the-key concept it is not possible for any party other than the legitimate client to use an access token and to re-use it without knowing the corresponding asymmetric key pair. This mechanism prevents against token disclosure.

With the usage of the asymmetric holder-of-the-key concept the following deployment consideration needs to be taken into consideration. In some deployments, including those utilizing load balancers, the TLS connection to the resource server terminates prior to the actual server that provides the resource. This could leave the token unprotected between the front end server where the TLS connection terminates and the back end server that provides the resource.

Client implementations must be carefully implemented to avoid leaking the ephemeral credentials (either the private key from the asymmetric credential or the shared secret).

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.

4.3. Summary of Recommendations

The following three items represent the main recommendations:

Safeguard the private key/shared secret: Client implementations MUST ensure that the ephemeral private key / shared secret is not leaked to third parties, since those will be able to use the access token together with the keying material to gain access to protected resources.

Switch keying material regularly: Clients can at any time create a new ephemeral credential and associate it with an access token. For example, a client presents a new public key when requesting an access token with the help of a refresh token. Nevertheless, the lifetime of these access token may be longer than the lifetime of bearer tokens.

Issue scoped bearer tokens: Token servers SHOULD issue bearer tokens that contain an audience restriction, scoping their use to the intended relying party or set of relying parties.

5. IANA Considerations

This document requires IANA to take the following actions.

5.1. OAuth Parameters Registration

This specification registers the following parameters in the OAuth Parameters Registry established by [3].

Parameter name: pk_info

Parameter usage location: token request

Change controller: IETF

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

Related information: None

Parameter name: token_type

Parameter usage location: token request, token response, authorization response

Change controller: IETF

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

Related information: None

Parameter name: profile

Parameter usage location: token request, token response, authorization response

Change controller: IETF

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

Related information: None

Parameter name: id

Parameter usage location: token response, authorization response

Change controller: IETF

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

Related information: None

Parameter name: key

Parameter usage location: token response, authorization response

Change controller: IETF

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

Related information: None

5.2. The 'hotk' JSON Web Token Claims

[6] established the IANA JSON Web Token Claims registry for reserved JWT Claim Names and this document adds the 'hotk' name to that registry.

5.3. The 'hotk' OAuth Access Token Type

Section 11.1 of [3] defines the OAuth Access Token Type Registry and this document adds another token type to this registry.

Type name: hotk

Additional Token Endpoint Response Parameters: (none)

HTTP Authentication Scheme(s): Holder of the key confirmation using TLS

Change controller: IETF

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

5.4. Profile Registry

This document asks IANA to create a registry for profiles of symmetric key-based holder-of-the-key mechanisms. The policy for adding new entries to the registry is "Specification Required". IANA is asked to populate the registry with the following values:

- Profile name: jws

- Change controller: IETF

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

- Profile name: mac

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

6. Acknowledgements

The author would like to thank the OAuth working group and participants of the Internet Identity Workshop for their discussion input that lead to this document.

7. References

7.1. Normative References

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

[2] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[3] Hardt, D., "The OAuth 2.0 Authorization Framework", draft-ietf-oauth-v2-31 (work in progress), August 2012.

[4] Jones, M., "JSON Web Key (JWK)", draft-ietf-jose-json-web-key-19 (work in progress), December 2013.

[5] Wouters, P., Tschofenig, H., Gilmore, J., Weiler, S., and T. Kivinen, "Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", draft-ietf-tls-oob-pubkey-10 (work in progress), October 2013.

[6] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", draft-ietf-oauth-json-web-token-14 (work in progress), December 2013.

[7] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", draft-ietf-jose-json-web-signature-19 (work in progress), December 2013.

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

[9] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, May 2008.

[10] Richer, J., Mills, W., Tschofenig, H., and P. Hunt, "OAuth 2.0 Message Authentication Code (MAC) Tokens", draft-ietf-oauth-v2-http-mac-04 (work in progress), July 2013.

7.2. Informative References

[11] Campbell, B., Mortimore, C., Jones, M., and Y. Goland, "Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants", draft-ietf-oauth-assertions-13 (work in progress), December 2013.

[12] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", draft-ietf-oauth-v2-bearer-23 (work in progress), August 2012.

[13] Hammer-Lahav, E., "The OAuth 1.0 Protocol", RFC 5849, April 2010.

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

Authors' Addresses

John Bradley
Ping Identity

Email: ve7jtb@ve7jtb.com

Phil Hunt
Oracle Corporation

Email: phil.hunt@yahoo.com

Tony Nadalin
Microsoft

Email: tonynad@microsoft.com

Hannes Tschofenig

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