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
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](image)

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.
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).
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. UsingAssertions 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:ietf:params:oauth:grant-type:saml2-bearer&
assertion=PHNhbwWwxOl...[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:

- 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".

- 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 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...
```

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


10.2. Informative References


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
o For consistency with SAML and JWT per http://www.ietf.org/mail-archive/web/oauth/current/msg12251.html and http://www.ietf.org/mail-archive/web/oauth/current/msg12253.html Stated that "In the absence of an application profile specifying otherwise, compliant applications MUST compare the audience values using the Simple String Comparison method defined in Section 6.2.1 of RFC 3986."

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

draft-ietf-oauth-assertions-12

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

draft-ietf-oauth-assertions-11

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

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

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

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

o Changed "Specific Assertion Format and Processing Rules" to "Common Scenarios" and reworded to be more suggestive of common practices, rather than trying to be normative. Also removed lots of repetitive text in that section.

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


o 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
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

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

updated reference to draft-ietf-oauth-v2 from -25 to -26
draft-ietf-oauth-assertions-02

Added text about limited lifetime ATs and RTs per http://www.ietf.org/mail-archive/web/oauth/current/msg08298.html.

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.

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.

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.

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.

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/msg08873.html.

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.

Add ref to Section 3.3 of oauth-v2 for scope definition and remove some then redundant text per http://www.ietf.org/mail-archive/web/oauth/current/msg08890.html.
o Change "The following format and processing rules SHOULD be applied" to "The following format and processing rules apply" in sections 6.x to remove conflicting normative qualification of other normative statements per http://www.ietf.org/mail-archive/web/oauth/current/msg08892.html.

o Add text the client_id must id the client to 4.1 and remove similar text from other places per http://www.ietf.org/mail-archive/web/oauth/current/msg08893.html.

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

o Updated examples to use grant_type and client_assertion_type values from the OAuth SAML Assertion Profiles spec.

Authors’ Addresses

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

Chuck Mortimore
Salesforce.com
Email: cmortimore@salesforce.com

Michael B. Jones
Microsoft
Email: mbj@microsoft.com

Yaron Y. Goland
Microsoft
Email: yarong@microsoft.com
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
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.
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:ietf:params:oauth:grant-type:sAML2-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):
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="ef1xsbZxPV2oqjd7HTLRLIB1Bb7"
    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):

```plaintext
POST /token.oauth2 HTTP/1.1
Host: authz.example.net
Content-Type: application/x-www-form-urlencoded
grant_type=urn:ietf:params:oauth:grant-type:saml2-bearer&assertion=PEFzc2VydGlvc2l0b3I6NiZlcl90aW1lbnQ-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
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].

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]

[OASIS.saml-core-2.0-os]

[OASIS.saml-deleg-cs]

[OASIS.saml-sec-consider-2.0-os]


9.2. Informative References

[OASIS.WS-Fed]

[OASIS.WSS-SAMLTokenProfile]

[OASIS.saml-metadata-2.0-os]

[OASIS.saml-profiles-2.0-os]


[W3C.REC-html401-19991224]

Appendix A. Acknowledgements

The following people contributed wording and concepts to this document: Paul Madsen, Patrick Harding, Peter Motykowski, Eran Hammer, Peter Saint-Andre, Ian Barnett, Eric Fazendin, Torsten Lodderstedt, Susan Harper, Scott Tomilson, Scott Cantor, Hannes Tschofenig, David Waite, Phil Hunt, and Mukesh Bhatnagar.
Appendix B.  Document History

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

draft-ietf-oauth-saml2-bearer-23
  o  Fix typo per http://www.ietf.org/mail-archive/web/oauth/current/msg13790.html

draft-ietf-oauth-saml2-bearer-22
  o  Changes/suggestions from IESG reviews.

draft-ietf-oauth-saml2-bearer-21
  o  Added Privacy Considerations section per AD review discussion
     http://www.ietf.org/mail-archive/web/oauth/current/msg13148.html
     and http://www.ietf.org/mail-archive/web/oauth/current/msg13144.html

draft-ietf-oauth-saml2-bearer-20
  o  Clarified some text around the treatment of subject based on the
     rough rough consensus from the thread staring at
     http://www.ietf.org/mail-archive/web/oauth/current/msg12630.html

draft-ietf-oauth-saml2-bearer-19
  o  Updated references.

draft-ietf-oauth-saml2-bearer-18
  o  Clean up language around subject per http://www.ietf.org/mail-
     archive/web/oauth/current/msg12254.html.
  o  As suggested in http://www.ietf.org/mail-
     archive/web/oauth/current/msg12253.html stated that "In the
     absence of an application profile specifying otherwise, compliant
     applications MUST compare the audience/issuer values using the
     Simple String Comparison method defined in Section 6.2.1 of RFC
     3986."
  o  Clarify the potentially confusing language about the AS confirming
     the assertion http://www.ietf.org/mail-archive/web/oauth/current/
     msg12255.html.
o Combine the two items about AuthnStatement and drop the word presenter as discussed in http://www.ietf.org/mail-archive/web/oauth/current/msg12257.html.

o Added one-time use, maximum lifetime, and specific subject and attribute requirements to Interoperability Considerations based on http://www.ietf.org/mail-archive/web/oauth/current/msg12252.html.

o Reword security considerations and mention that replay protection is not mandated based on http://www.ietf.org/mail-archive/web/oauth/current/msg12259.html.

draft-ietf-oauth-saml2-bearer-17

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

draft-ietf-oauth-saml2-bearer-16

o Changed title from "SAML 2.0 Bearer Assertion Profiles for OAuth 2.0" to "SAML 2.0 Profile for OAuth 2.0 Client Authentication and Authorization Grants" to be more explicit about the scope of the document per http://www.ietf.org/mail-archive/web/oauth/current/msg11063.html.

o Fixed typo in text identifying the presenter from "or similar element, the" to "or similar element in the".

o Numbered the list of processing rules.

o Smallish editorial cleanups to try and improve readability and comprehensibility.

o Cleaner split out of the processing rules in cases where they differ for client authentication and authorization grants.

o Clarified the parameters that are used/available for authorization grants.

o Added Interoperability Considerations section and info reference to SAML Metadata.

o Added more explanatory context to the example in Section 4.

draft-ietf-oauth-saml2-bearer-15

o Reference RFC 6749 and RFC 6755.
o Update draft-ietf-oauth-assertions reference to -06.

o Remove extraneous word per http://www.ietf.org/mail-archive/web/oauth/current/msg10055.html

draft-ietf-oauth-saml2-bearer-14

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

o Add examples to Sections 2.1 and 2.2

o Update references

draft-ietf-oauth-saml2-bearer-13

o Update references: oauth-assertions-04, oauth-urn-sub-ns-05, oauth-28

o Changed "Description" to "Specification Document" in both registration requests in IANA Considerations per changes to the template in ietf-oauth-urn-sub-ns(-03)

o Added "(or an acceptable alias)" so that it’s in both sentences about Recipient and the token endpoint URL so there’s no ambiguity

o Update area and workgroup (now Security and OAuth was Internet and nothing)

draft-ietf-oauth-saml2-bearer-12

o updated reference to draft-ietf-oauth-v2 from -25 to -26 and draft-ietf-oauth-assertions from -02 to -03

draft-ietf-oauth-saml2-bearer-11

o Removed text about limited lifetime access tokens and the SHOULD NOT on issuing refresh tokens. The text was moved to draft-ietf-oauth-assertions-02 and somewhat modified per http://www.ietf.org/mail-archive/web/oauth/current/msg08298.html.

o Fixed typo/missing word per http://www.ietf.org/mail-archive/web/oauth/current/msg08733.html.

o Added Terminology section.
draft-ietf-oauth-saml2-bearer-10
  o fix a spelling mistake

draft-ietf-oauth-saml2-bearer-09
  o Attempt to address an ambiguity around validation requirements when the Conditions element contain a NotOnOrAfter and SubjectConfirmation/SubjectConfirmationData does too. Basically it needs to have at least one bearer SubjectConfirmation element but that element can omit SubjectConfirmationData, if Conditions has an expiry on it. Otherwise, a valid SubjectConfirmation must have a SubjectConfirmationData with Recipient and NotOnOrAfter. And any SubjectConfirmationData that has those elements needs to have them checked.
  o clarified that AudienceRestriction is under Conditions (even though it’s implied by schema)
  o fix a typo

draft-ietf-oauth-saml2-bearer-08
  o fix some typos

draft-ietf-oauth-saml2-bearer-07
  o update reference from draft-campbell-oauth-urn-sub-ns to draft-ietf-oauth-urn-sub-ns
  o Updated to reference draft-ietf-oauth-v2-20

draft-ietf-oauth-saml2-bearer-06
  o Fix three typos NameID->NameID and (2x) Namespace->Namespace

draft-ietf-oauth-saml2-bearer-05
  o Allow for subject confirmation data to be optional when Conditions contain audience and NotOnOrAfter
  o Rework most of the spec to profile draft-ietf-oauth-assertions for both authn and authz including (but not limited to):
    * remove requirement for issuer to be urn:oasis:names:tc:SAML:2.0:nameid-format:entity
    * change wording on Subject requirements
o using a MAY, explicitly say that the Audience can be token endpoint URL of the authorization server

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

o added client authentication to the abstract

o register and use urn:ietf:params:oauth:grant-type:saml2-bearer for grant type rather than http://oauth.net/grant_type/saml/2.0/bearer

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

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

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

o fix typo(s) and update/add references

draft-ietf-oauth-saml2-bearer-04

o Changed the grant_type URI from "http://oauth.net/grant_type/assertion/saml/2.0/bearer" to "http://oauth.net/grant_type/saml/2.0/bearer" - dropping the word assertion from the path. Recent versions of draft-ietf-oauth-v2 no longer refer to extension grants using the word assertion so this URI is more reflective of that. It also more closely aligns with the grant type URI in draft-jones-oauth-jwt-bearer-00 which is "http://oauth.net/grant_type/jwt/1.0/bearer".

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

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

draft-ietf-oauth-saml2-bearer-03

o Cleanup of some editorial issues.

draft-ietf-oauth-saml2-bearer-02

o Added scope parameter with text copied from draft-ietf-oauth-v2-12 (the reorg of draft-ietf-oauth-v2-12 made it so scope wasn’t really inherited by this spec anymore)
o Change definition of the assertion parameter to be more generally applicable per the suggestion near the end of http://www.ietf.org/mail-archive/web/oauth/current/msg05253.html

o Editorial changes based on feedback

draft-ietf-oauth-saml2-bearer-01

o Update spec name when referencing draft-ietf-oauth-v2 (The OAuth 2.0 Protocol Framework -> The OAuth 2.0 Authorization Protocol)

o Update wording in Introduction to talk about extension grant types rather than the assertion grant type which is a term no longer used in OAuth 2.0

o Updated to reference draft-ietf-oauth-v2-12 and denote as work in progress

o Update Parameter Registration Request to use similar terms as draft-ietf-oauth-v2-12 and remove Related information part

o Add some text giving discretion to AS on rejecting assertions with unreasonably long validity window.

draft-ietf-oauth-saml2-bearer-00

o Added Parameter Registration Request for "assertion" to IANA Considerations.

o Changed document name to draft-ietf-oauth-saml2-bearer in anticipation of becoming an OAUTH WG item.

o Attempt to move the entire definition of the ‘assertion’ parameter into this draft (it will no longer be defined in OAuth 2 Protocol Framework).

draft-campbell-oauth-saml-01

o Updated to reference draft-ietf-oauth-v2-11 and reflect changes from -10 to -11.

o Updated examples.

o Relaxed processing rules to allow for more than one SubjectConfirmation element.

o Removed the ‘MUST NOT contain a NotBefore attribute’ on SubjectConfirmationData.
o Relaxed wording that ties the subject of the Assertion to the resource owner.

o Added some wording about identifying the client when the subject hasn’t directly authenticated including an informative reference to SAML V2.0 Condition for Delegation Restriction.

o Added a few examples to the language about verifying that the Assertion is valid in all other respects.

o Added some wording to the introduction about the similarities to Web SSO in the format and processing rules

o Changed the grant_type (was assertion_type) URI from http://oauth.net/assertion_type/saml/2.0/bearer to http://oauth.net/grant_type/assertion/saml/2.0/bearer

o Changed title to include "Grant Type" in it.

o Editorial updates based on feedback from the WG and others (including capitalization of Assertion when referring to SAML).

draft-campbell-oauth-saml-00

o Initial I-D

Authors’ Addresses

Brian Campbell
Ping Identity

Email: brian.d.campbell@gmail.com

Chuck Mortimore
Salesforce.com

Email: cmortimore@salesforce.com

Michael B. Jones
Microsoft

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

This document establishes an IETF URN Sub-namespace for use with OAuth related specifications.

Status of this Memo

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

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

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

This Internet-Draft will expire on January 17, 2013.

Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Various extensions and companion specifications to The OAuth 2.0 Authorization Framework [I-D.ietf-oauth-v2] utilize URIs to identify the extension in use or other relevant context. This document creates and registers an IETF URN Sub-namespace, as documented in RFC 3553 [RFC3553], for use with such specifications. The new 'oauth' sub-namespace is urn:ietf:params:oauth and OAuth relevant parameters will be established underneath it.

2. Registration Template

If a registrant wishes to have a OAuth URI registered, then a URN of the form urn:ietf:params:oauth:<value> will be requested where <value> is a suitable representation of the functionality or concept being registered.

The registration procedure for new entries requires a request in the form of the following template and is Specification Required per RFC 5226 [RFC5226].

URN:
   The URI that identifies the registered functionality.

Common Name:
   The name by which the functionality being registered is generally known.

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 the document that specifies the URI, 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.

The registration request for the urn:ietf:params:oauth URN Sub-namespace is found in the IANA Considerations (Section 4) section of this document.

2.1. Example Registration Request

The following is an example registration request for a URI underneath the urn:ietf:params:oauth sub-namespace. The requested URI represents a new OAuth 2.0 grant type.
This is a request to IANA to please register the value "grant-type:example" in the registry urn:ietf:params:oauth established in An IETF URN Sub-Namespace for OAuth.

- URN: urn:ietf:params:oauth:grant-type:example
- Common Name: An Example Grant Type for OAuth 2.0
- Change controller: IETF
- Specification Document: [[the document URI]]

3. Security Considerations

There are no additional security considerations beyond those already inherent to using URNs. Security considerations for URNs in general can be found in RFC 2141 [RFC2141].

Any work that is related to OAuth would benefit from familiarity with the security considerations of The OAuth 2.0 Authorization Framework [I-D.ietf-oauth-v2].

4. IANA Considerations

This document makes two requests of IANA:

- Registration of a new IANA URN sub-namespace, urn:ietf:params:oauth:, per RFC 3553 [RFC3553]. The registration request can be found in Section 4.1 below.

- Establishment of a new registry for URNs subordinate to urn:ietf:params:oauth. Instructions for a registrant to request the registration of such a URN are in Section 2.

4.1. IETF URN Sub-namespace Registration urn:ietf:params:oauth

Per RFC 3553 [RFC3553], IANA is requested to please register a new URN sub-namespace, urn:ietf:params:oauth.

- Registry name: oauth
- Specification: [[this document]]
- Repository: [[The registry created in Section 3.]]
Index value: values subordinate to urn:ietf:params:oauth are of the form urn:ietf:params:oauth:<value> with <value> as the index value. It is suggested that <value> include both a "class" and an "identifier-within-class" component, with the two components being separated by a colon (":"); other compositions of the <value> may also be used.

5. References

5.1. Normative References


5.2. Informative References


Appendix A. Acknowledgements

The authors thank the following for their helpful contributions: Stephen Farrell, Barry Leiba, Peter Saint-Andre, Eran Hammer, John Bradley, Ben Campbell and Michael B. Jones.

Appendix B. Document History

[[ to be removed by RFC editor before publication as an RFC ]]
draft-ietf-oauth-urn-sub-ns-06

Address editorial comments from Gen-ART LC Review:
http://www.ietf.org/mail-archive/web/gen-art/current/msg07576.html
draft-ietf-oauth-urn-sub-ns-05
- Change the registration procedure from Expert Review to Specification Required per WG discussion
  http://www.ietf.org/mail-archive/web/oauth/current/msg09445.html

draft-ietf-oauth-urn-sub-ns-04

- Changed the Index value (and Registration Template into paragraph) from <class><id> to just <value> with a suggestion that it have both a "class" and an "identifier-within-class" parts per
  http://www.ietf.org/mail-archive/web/oauth/current/msg09381.html
so as to be less restrictive

draft-ietf-oauth-urn-sub-ns-03

- Changes to address comments in the message "AD review of draft-ietf-oauth-urn-sub-ns-02" at
  http://www.ietf.org/mail-archive/web/oauth/current/msg09350.html
and subsequent messages in that thread

- Update area and workgroup (now Security and OAuth was Internet and nothing)

- Change from informational to standards-track

- Requesting new URNs now more lightweight by changing from ‘RFC Required’ to ‘Expert Review’ (RFC5226)

- Rework much of the document to be more clear about it registering the urn:ietf:params:oauth URN sub-namespace and separately how other documents are to request URNs under that sub-namespace.

- Removed everything about asking the IANA to generate any part of the URN.

- Added an Example Registration Request

- Added reference to OAuth security considerations in security considerations.

- Added Acknowledgements

draft-ietf-oauth-urn-sub-ns-02

- fix typo: "The registration procedure for new entries to the requires a request ..." --> "The registration procedure for new entries requires a request ..."

draft-ietf-oauth-urn-sub-ns-01
security considerations now points to RFC 2141 rather than RFC 3553 per
http://www.ietf.org/mail-archive/web/oauth/current/msg07880.html
draft-ietf-urn-sub-ns-00

change doc name from draft-campbell-oauth-urn-sub-ns to
draft-ietf-urn-sub-ns per
http://www.ietf.org/mail-archive/web/oauth/current/msg07384.html
draft-campbell-oauth-urn-sub-ns-01

minor editorial changes
draft-campbell-oauth-urn-sub-ns-00

initial draft based on
http://www.ietf.org/mail-archive/web/oauth/current/msg06949.html

Authors’ Addresses

Brian Campbell
Ping Identity Corp.

Email: brian.d.campbell@gmail.com

Hannes Tschofenig
Nokia Siemens Networks

Email: hannes.tschofenig@gmx.net
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
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.

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.
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.
pwaFh7yJPivLjjPzkC-GeAyHuy7AinGcS51A27TXnwkC800wiaw47kcT_UV54ubo nONbeArwOVuR7shveXnwFmuCrwR_30CCcHarCbeE1HR-Jfme2mF_WR3zUMcwmqmU0R1H kwx9x0o_sKrasi1Xc8RYF-evLCmT1XRKjty5144Gnh0A84hGvVFmXfMFCWXh3biX 2h6JmjQHQQ3mivVu15bf-zzb3qXXxsNO12YoWgs5tP1-T5QYq9i919wodFPWNPKF kY-BgewG-Vmc59JqFprk1O08qhKQeOGCWcOWFC_n_LpGWH6spRm7KGuYdMDMkQ bd4uU0PSLx_eVCdrVzA.
AxV8DcTaGlbGb3bRo2Q.
7M121RCaoyYx1Hcl1VXk9r8DhDoikTMqOp31dEmm4qqBThFkgFqO3ivXZLJTk4M0f laMAdG5_K6K8_B-0E-7ak-0lm-_V03oBUUGTAc-F0A.
OwWNxnC-BMEie-GkFHzVWiNiaV3zUXf6fCOGTwbRckU",
  "token_type": "mac",
  "expires_in": 3600,
  "refresh_token": "8xLOxBtZp8",
  "kid": "22BIjxU93h/IfgEB4zCRu5WF37s=",
  "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":"adijq39jdslaska9asud",
"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`
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 timestamp 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
1361471629
example.com
```

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]:

\[
\text{mac} = \text{HMAC-SHA256} \ (\text{key}, \text{text})
\]

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 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 capacity 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]:

- Claim Name: "kid"
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:

  \texttt{mac}

Additional Token Endpoint Response Parameters:

  \texttt{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 ]]

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]

[I-D.ietf-jose-json-web-encryption]

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

[I-D.richer-oauth-introspection]

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


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]

[NIST-FIPS-180-3]

[NIST800-63]


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 decisions.

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
A 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
Abstract

This document gives additional security considerations for OAuth, beyond those in the OAuth 2.0 specification, based on a comprehensive threat model for the OAuth 2.0 Protocol.

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 9, 2013.

Copyright Notice

Copyright (c) 2012 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
### Table of Contents

1. **Introduction** .................................................. 6  
2. **Overview** ..................................................... 6  
   2.1. **Scope** .................................................... 6  
   2.2. **Attack Assumptions** ...................................... 7  
   2.3. **Architectural assumptions** ................................ 8  
      2.3.1. **Authorization Servers** ................................ 8  
      2.3.2. **Resource Server** .................................... 8  
      2.3.3. **Client** .............................................. 9  
3. **Security Features** ............................................. 9  
   3.1. **Tokens** .................................................. 9  
      3.1.1. **Scope** .............................................. 11  
      3.1.2. **Limited Access Token Lifetime** ...................... 11  
   3.2. **Access Token** ............................................ 11  
   3.3. **Refresh Token** .......................................... 11  
   3.4. **Authorization Code** ...................................... 12  
   3.5. **Redirect URI** ........................................... 13  
   3.6. **State parameter** ........................................ 13  
   3.7. **Client Identifier** ....................................... 13  
4. **Threat Model** .................................................. 15  
   4.1. **Clients** ................................................ 15  
      4.1.1. **Threat: Obtain Client Secrets** ...................... 15  
      4.1.2. **Threat: Obtain Refresh Tokens** ...................... 17  
      4.1.3. **Threat: Obtain Access Tokens** ....................... 19  
      4.1.4. **Threat: End-user credentials phished using**  
              compromised or embedded browser ...................... 19  
      4.1.5. **Threat: Open Redirectors on client** ............... 20  
   4.2. **Authorization Endpoint** ................................ 20  
      4.2.1. **Threat: Password phishing by counterfeit**  
              authorization server ................................ 20  
      4.2.2. **Threat: User unintentionally grants too much**  
              access scope ........................................... 21  
      4.2.3. **Threat: Malicious client obtains existing**  
              authorization by fraud ................................ 21  
      4.2.4. **Threat: Open redirector** ........................... 22  
   4.3. **Token endpoint** .......................................... 22  
      4.3.1. **Threat: Eavesdropping access tokens** .............. 22  
      4.3.2. **Threat: Obtain access tokens from authorization**  
              server database ........................................ 23  
      4.3.3. **Threat: Disclosure of client credentials during**  
              transmission ............................................ 23  
      4.3.4. **Threat: Obtain client secret from authorization**  
              server database ........................................ 23  
      4.3.5. **Threat: Obtain client secret by online guessing**  
              .......................................................... 24
4.4. Obtaining Authorization

4.4.1. Authorization Code

4.4.1.1. Threat: Eavesdropping or leaking authorization codes

4.4.1.2. Threat: Obtain authorization codes from authorization server database

4.4.1.3. Threat: Online guessing of authorization codes

4.4.1.4. Threat: Malicious client obtains authorization

4.4.1.5. Threat: Authorization code phishing

4.4.1.6. Threat: User session impersonation

4.4.1.7. Threat: Authorization code leakage through counterfeit client

4.4.1.8. Threat: CSRF attack against redirect-uri

4.4.1.9. Threat: Clickjacking attack against authorization

4.4.1.10. Threat: Resource Owner Impersonation

4.4.1.11. Threat: DoS, Exhaustion of resources attacks

4.4.1.12. Threat: DoS using manufactured authorization codes

4.4.1.13. Threat: Code substitution (OAuth Login)

4.4.2. Implicit Grant

4.4.2.1. Threat: Access token leak in transport/end-points

4.4.2.2. Threat: Access token leak in browser history

4.4.2.3. Threat: Malicious client obtains authorization

4.4.2.4. Threat: Manipulation of scripts

4.4.2.5. Threat: CSRF attack against redirect-uri

4.4.2.6. Threat: Token substitution (OAuth Login)

4.4.3. Resource Owner Password Credentials

4.4.3.1. Threat: Accidental exposure of passwords at client site

4.4.3.2. Threat: Client obtains scopes without end-user authorization

4.4.3.3. Threat: Client obtains refresh token through automatic authorization

4.4.3.4. Threat: Obtain user passwords on transport

4.4.3.5. Threat: Obtain user passwords from authorization server database

4.4.3.6. Threat: Online guessing

4.4.4. Client Credentials

4.5. Refreshing an Access Token

4.5.1. Threat: Eavesdropping refresh tokens from authorization server

4.5.2. Threat: Obtaining refresh token from authorization server database

4.5.3. Threat: Obtain refresh token by online guessing

4.5.4. Threat: Obtain refresh token phishing by counterfeit authorization server
4.6. Accessing Protected Resources ........................................ 44
4.6.1. Threat: Eavesdropping access tokens on transport ............ 44
4.6.2. Threat: Replay authorized resource server requests .......... 45
4.6.3. Threat: Guessing access tokens ................................... 45
4.6.4. Threat: Access token phishing by counterfeit resource server .................................................. 46
4.6.5. Threat: Abuse of token by legitimate resource server or client .................................................. 46
4.6.6. Threat: Leak of confidential data in HTTP-Proxies .......... 47
4.6.7. Threat: Token leakage via logfiles and HTTP referrers .......... 47
5. Security Considerations .................................................... 48
5.1. General ................................................................. 48
5.1.1. Ensure confidentiality of requests ................................. 48
5.1.2. Utilize server authentication ...................................... 48
5.1.3. Always keep the resource owner informed ......................... 49
5.1.4. Credentials .......................................................... 49
5.1.4.1. Enforce credential storage protection best practices ........ 50
5.1.4.2. Online attacks on secrets ....................................... 51
5.1.5. Tokens (access, refresh, code) ..................................... 52
5.1.5.1. Limit token scope ................................................ 52
5.1.5.2. Expiration time ................................................... 52
5.1.5.3. Use short expiration time ....................................... 53
5.1.5.4. Limit number of usages/ One time usage ...................... 53
5.1.5.5. Bind tokens to a particular resource server (Audience) ........ 54
5.1.5.6. Use endpoint address as token audience ....................... 54
5.1.5.7. Audience and Token scopes ..................................... 54
5.1.5.8. Bind token to client id .......................................... 54
5.1.5.9. Signed tokens .................................................... 55
5.1.5.10. Encryption of token content ................................... 55
5.1.5.11. Assertion formats ............................................... 55
5.1.6. Access tokens ....................................................... 55
5.2. Authorization Server ................................................... 55
5.2.1. Authorization Codes ................................................ 55
5.2.1.1. Automatic revocation of derived tokens if abuse is detected ................................................ 55
5.2.2. Refresh tokens ....................................................... 56
5.2.2.1. Restricted issuance of refresh tokens ......................... 56
5.2.2.2. Binding of refresh token to client_id ........................ 56
5.2.2.3. Refresh Token Rotation ......................................... 56
5.2.2.4. Revoke refresh tokens .......................................... 57
5.2.2.5. Device identification ............................................ 57
5.2.2.6. X-FRAME-OPTION header ........................................ 57
5.2.3. Client authentication and authorization ......................... 57
5.2.3.1. Don’t issue secrets to client with inappropriate security policy ........................................... 58
5.2.3.2. Require user consent for public clients without secret .................................. 59
5.2.3.3. Client_id only in combination with redirect_uri ........................................... 59
5.2.3.4. Installation-specific client secrets ................................................................. 59
5.2.3.5. Validation of pre-registered redirect_uri ........................................................ 60
5.2.3.6. Revoke client secrets ......................................................................................... 61
5.2.3.7. Use strong client authentication (e.g. client_assertion / client_token) .......... 61
5.2.4. End-user authorization .......................................................................................... 61
5.2.4.1. Automatic processing of repeated authorizations requires client validation ..... 61
5.2.4.2. Informed decisions based on transparency ......................................................... 62
5.2.4.3. Validation of client properties by end-user ......................................................... 62
5.2.4.4. Binding of authorization code to client_id ......................................................... 62
5.2.4.5. Binding of authorization code to redirect_uri ..................................................... 62
5.3. Client App Security .................................................................................................. 63
5.3.1. Don’t store credentials in code or resources bundled with software packages ....... 63
5.3.2. Standard web server protection measures (for config files and databases) .......... 63
5.3.3. Store secrets in a secure storage ............................................................................ 63
5.3.4. Utilize device lock to prevent unauthorized device access .................................. 64
5.3.5. Link state parameter to user agent session ......................................................... 64
5.4. Resource Servers .................................................................................................... 64
5.4.1. Authorization headers ......................................................................................... 64
5.4.2. Authenticated requests ....................................................................................... 64
5.4.3. Signed requests ................................................................................................... 65
5.5. A Word on User Interaction and User-Installed Apps ............................................. 65
6. IANA Considerations ................................................................................................. 66
7. Acknowledgements ..................................................................................................... 67
8. References ................................................................................................................ 67
8.1. Informative References .......................................................................................... 67
8.2. Informative References .......................................................................................... 67
Appendix A. Document History .................................................................................... 69
Authors’ Addresses ....................................................................................................... 71
1. Introduction

This document gives additional security considerations for OAuth, beyond those in the OAuth specification, based on a comprehensive threat model for the OAuth 2.0 Protocol [I-D.ietf-oauth-v2]. It contains the following content:

- Documents any assumptions and scope considered when creating the threat model.
- Describes the security features in-built into the OAuth protocol and how they are intended to thwart attacks.
- Gives a comprehensive threat model for OAuth and describes the respective counter measures to thwart those threats.

Threats include any intentional attacks on OAuth tokens and resources protected by OAuth tokens as well as security risks introduced if the proper security measures are not put in place. Threats are structured along the lines of the protocol structure to aid development teams implement each part of the protocol securely. For example all threats for granting access or all threats for a particular grant type or all threats for protecting the resource server.

Note: This document cannot assess the probability nor the risk associated with a particular threat because those aspects strongly depend on the particular application and deployment OAuth is used to protect. Similar, impacts are given on a rather abstract level. But the information given here may serve as a foundation for deployment-specific threat models. Implementors may refine and detail the abstract threat model in order to account for the specific properties of their deployment and to come up with a risk analysis. As this document is based on the base OAuth 2.0 specification, it does not consider proposed extensions, such as client registration or discovery, many of which are still under discussion.

2. Overview

2.1. Scope

The security considerations document only considers clients bound to a particular deployment as supported by [I-D.ietf-oauth-v2]. Such deployments have the following characteristics:
Resource server URLs are static and well-known at development time, authorization server URLs can be static or discovered.

Token scope values (e.g. applicable URLs and methods) are well-known at development time.

Client registration: Since registration of clients is out of scope of the current core spec, this document assumes a broad variety of options from static registration during development time to dynamic registration at runtime.

The following are considered out of scope:

Communication between authorization server and resource server

Token formats

Except for "Resource Owner Password Credentials" (see [I-D.ietf-oauth-v2], section 4.3), the mechanism used by authorization servers to authenticate the user

Mechanism by which a user obtained an assertion and any resulting attacks mounted as a result of the assertion being false.

Clients not bound to a specific deployment: An example could be a mail client with support for contact list access via the portable contacts API (see [portable-contacts]). Such clients cannot be registered upfront with a particular deployment and should dynamically discover the URLs relevant for the OAuth protocol.

2.2. Attack Assumptions

The following assumptions relate to an attacker and resources available to an attacker:

It is assumed the attacker has full access to the network between the client and authorization servers and the client and the resource server, respectively. The attacker may eavesdrop on any communications between those parties. He is not assumed to have access to communication between authorization and resource server.

It is assumed an attacker has unlimited resources to mount an attack.

It is assumed that 2 of the 3 parties involved in the OAuth protocol may collude to mount an attack against the 3rd party. For example, the client and authorization server may be under control of an attacker and collude to trick a user to gain access.
2.3. Architectural assumptions

This section documents the assumptions about the features, limitations, and design options of the different entities of a OAuth deployment along with the security-sensitive data-elements managed by those entity. These assumptions are the foundation of the threat analysis.

The OAuth protocol leaves deployments with a certain degree of freedom how to implement and apply the standard. The core specification defines the core concepts of an authorization server and a resource server. Both servers can be implemented in the same server entity, or they may also be different entities. The later is typically the case for multi-service providers with a single authentication and authorization system, and are more typical in middleware architectures.

2.3.1. Authorization Servers

The following data elements are stored or accessible on the authorization server:

- user names and passwords
- client ids and secrets
- client-specific refresh tokens
- client-specific access tokens (in case of handle-based design - see Section 3.1)
- HTTPS certificate/key
- per-authorization process (in case of handle-based design - Section 3.1): redirect_uri, client_id, authorization code

2.3.2. Resource Server

The following data elements are stored or accessible on the resource server:

- user data (out of scope)
- HTTPS certificate/key
o authorization server credentials (handle-based design - see Section 3.1), or

o authorization server shared secret/public key (assertion-based design - see Section 3.1)

o access tokens (per request)

It is assumed that a resource server has no knowledge of refresh tokens, user passwords, or client secrets.

2.3.3. Client

In OAuth a client is an application making protected resource requests on behalf of the resource owner and with its authorization. There are different types of clients with different implementation and security characteristics, such as web, user-agent-based, and native applications. A full definition of the different client types and profiles is given in [I-D.ietf-oauth-v2], Section 2.1.

The following data elements are stored or accessible on the client:

o client id (and client secret or corresponding client credential)

o one or more refresh tokens (persistent) and access tokens (transient) per end-user or other security-context or delegation context

o trusted CA certificates (HTTPS)

o per-authorization process: redirect_uri, authorization code

3. Security Features

These are some of the security features which have been built into the OAuth 2.0 protocol to mitigate attacks and security issues.

3.1. Tokens

OAuth makes extensive use many kinds of tokens (access tokens, refresh tokens, authorization codes). The information content of a token can be represented in two ways as follows:
Handle (or artifact) a reference to some internal data structure within the authorization server; the internal data structure contains the attributes of the token, such as user id, scope, etc. Handles enable simple revocation and do not require cryptographic mechanisms to protect token content from being modified. On the other hand, handles require communication between issuing and consuming entity (e.g. authorization and resource server) in order to validate the token and obtain token-bound data. This communication might have an negative impact on performance and scalability if both entities reside on different systems. Handles are therefore typically used if the issuing and consuming entity are the same. A 'handle' token is often referred to as an 'opaque' token because the resource server does not need to be able to interpret the token directly, it simply uses the token.

Assertions (aka self-contained token) a parseable token. An assertion typically has a duration, has an audience, and is digitally signed in order to ensure data integrity and origin authentication. It contains information about the user and the client. Examples of assertion formats are SAML assertions [OASIS.saml-core-2.0-os] and Kerberos tickets [RFC4120]. Assertions can typically directly be validated and used by a resource server without interactions with the authorization server. This results in better performance and scalability in deployment where issuing and consuming entity reside on different systems. Implementing token revocation is more difficult with assertions than with handles.

Tokens can be used in two ways to invoke requests on resource servers as follows:

bearer token A 'bearer token' is a token that can be used by any client who has received the token (e.g. [I-D.ietf-oauth-v2-bearer]). Because mere possession is enough to use the token it is important that communication between end-points be secured to ensure that only authorized end-points may capture the token. The bearer token is convenient to client applications as it does not require them to do anything to use them (such as a proof of identity). Bearer tokens have similar characteristics to web single-sign-on (SSO) cookies used in browsers.

proof token A 'proof token' is a token that can only be used by a specific client. Each use of the token, requires the client to perform some action that proves that it is the authorized user of the token. Examples of this are MAC tokens, which require the client to digitally sign the resource request with a secret corresponding to the particular token send with the request.
3.1.1. Scope

A Scope represents the access authorization associated with a particular token with respect to resource servers, resources and methods on those resources. Scopes are the OAuth way to explicitly manage the power associated with an access token. A scope can be controlled by the authorization server and/or the end-user in order to limit access to resources for OAuth clients these parties deem less secure or trustworthy. Optionally, the client can request the scope to apply to the token but only for lesser scope than would otherwise be granted, e.g. to reduce the potential impact if this token is sent over non secure channels. A scope is typically complemented by a restriction on a token’s lifetime.

3.1.2. Limited Access Token Lifetime

The protocol parameter expires_in allows an authorization server (based on its policies or on behalf of the end-user) to limit the lifetime of an access token and to pass this information to the client. This mechanism can be used to issue short-living tokens to OAuth clients the authorization server deems less secure or where sending tokens over non secure channels.

3.2. Access Token

An access token is used by a client to access a resource. Access tokens typically have short life-spans (minutes or hours) that cover typical session lifetimes. An access token may be refreshed through the use of a refresh token. The short lifespan of an access token in combination with the usage of refresh tokens enables the possibility of passive revocation of access authorization on the expiry of the current access token.

3.3. Refresh Token

A refresh token represents a long-lasting authorization of a certain client to access resources on behalf of a resource owner. Such tokens are exchanged between client and authorization server, only. Clients use this kind of token to obtain ("refresh") new access tokens used for resource server invocations.

A refresh token, coupled with a short access token lifetime, can be used to grant longer access to resources without involving end user authorization. This offers an advantage where resource servers and authorization servers are not the same entity, e.g. in a distributed environment, as the refresh token is always exchanged at the
authorization server. The authorization server can revoke the refresh token at any time causing the granted access to be revoked once the current access token expires. Because of this, a short access token lifetime is important if timely revocation is a high priority.

The refresh token is also a secret bound to the client identifier and client instance which originally requested the authorization and representing the original resource owner grant. This is ensured by the authorization process as follows:

1. The resource owner and user-agent safely deliver the authorization code to the client instance in first place.

2. The client uses it immediately in secure transport-level communications to the authorization server and then securely stores the long-lived refresh token.

3. The client always uses the refresh token in secure transport-level communications to the authorization server to get an access token (and optionally rollover the refresh token).

So as long as the confidentiality of the particular token can be ensured by the client, a refresh token can also be used as an alternative means to authenticate the client instance itself.

3.4. Authorization Code

An authorization code represents the intermediate result of a successful end-user authorization process and is used by the client to obtain access and refresh token. Authorization codes are sent to the client’s redirection URI instead of tokens for two purposes.

1. Browser-based flows expose protocol parameters to potential attackers via URI query parameters (HTTP referrer), the browser cache, or log file entries and could be replayed. In order to reduce this threat, short-lived authorization codes are passed instead of tokens and exchanged for tokens over a more secure direct connection between client and authorization server.

2. It is much simpler to authenticate clients during the direct request between client and authorization server than in the context of the indirect authorization request. The latter would require digital signatures.
3.5. Redirection URI

A redirection URI helps to detect malicious clients and prevents phishing attacks from clients attempting to trick the user into believing the phisher is the client. The value of the actual redirection URI used in the authorization request has to be presented and is verified when an authorization code is exchanged for tokens. This helps to prevent attacks, where the authorization code is revealed through redirectors and counterfeit web application clients. The authorization server should require public clients and confidential clients using implicit grant type to pre-register their redirect URIs and validate against the registered redirection URI in the authorization request.

3.6. State parameter

The state parameter is used to link requests and callbacks to prevent Cross-Site Request Forgery attacks (see Section 4.4.1.8) where an attacker authorizes access to his own resources and then tricks a users into following a redirect with the attacker’s token. This parameter should bind to the authenticated state in a user agent and, as per the core OAuth spec, the user agent must be capable of keeping it in a location accessible only by the client and user agent, i.e. protected by same-origin policy.

3.7. Client Identifier

Authentication protocols have typically not taken into account the identity of the software component acting on behalf of the end-user. OAuth does this in order to increase the security level in delegated authorization scenarios and because the client will be able to act without the user being present.

OAuth uses the client identifier to collate associated request to the same originator, such as

- a particular end-user authorization process and the corresponding request on the token’s endpoint to exchange the authorization code for tokens or

- the initial authorization and issuance of a token by an end-user to a particular client, and subsequent requests by this client to obtain tokens without user consent (automatic processing of repeated authorization)

This identifier may also be used by the authorization server to display relevant registration information to a user when requesting consent for scope requested by a particular client. The client...
identifier may be used to limit the number of request for a particular client or to charge the client per request. It may further be useful to differentiate access by different clients, e.g. in server log files.

OAuth defines two client types, confidential and public, based on their ability to authenticate with the authorization server (i.e. ability to maintain the confidentiality of their client credentials). Confidential clients are capable of maintaining the confidentiality of client credentials (i.e. a client secret associated with the client identifier) or capable of secure client authentication using other means, such as a client assertion (e.g. SAML) or key cryptography. The latter is considered more secure.

The authorization server should determine whether the client is capable of keeping its secret confidential or using secure authentication. Alternatively, the end-user can verify the identity of the client, e.g. by only installing trusted applications. The redirection URI can be used to prevent delivering credentials to a counterfeit client after obtaining end-user authorization in some cases, but can’t be used to verify the client identifier.

Clients can be categorized as follows based on the client type, profile (e.g. native vs. web application - see [I-D.ietf-oauth-v2], Section 9) and deployment model:

Deployment-independent client_id with pre-registered redirect_uri and without client_secret Such an identifier is used by multiple installations of the same software package. The identifier of such a client can only be validated with the help of the end-user. This is a viable option for native applications in order to identify the client for the purpose of displaying meta information about the client to the user and to differentiate clients in log files. Revocation of the rights associated with such a client identifier will affect ALL deployments of the respective software.

Deployment-independent client_id with pre-registered redirect_uri and with client_secret This is an option for native applications only, since web application would require different redirect URIs. This category is not advisable because the client secret cannot be protected appropriately (see Section 4.1.1). Due to its security weaknesses, such client identities have the same trust level as deployment-independent clients without secret. Revocation will affect ALL deployments.
Deployment-specific client_id with pre-registered redirect_uri and with client_secret. The client registration process ensures the validation of the client’s properties, such as redirection URI, website URL, web site name, contacts. Such a client identifier can be utilized for all relevant use cases cited above. This level can be achieved for web applications in combination with a manual or user-bound registration process. Achieving this level for native applications is much more difficult. Either the installation of the application is conducted by an administrator, who validates the client’s authenticity, or the process from validating the application to the installation of the application on the device and the creation of the client credentials is controlled end-to-end by a single entity (e.g. application market provider). Revocation will affect a single deployment only.

Deployment-specific client_id with client_secret without validated properties. Such a client can be recognized by the authorization server in transactions with subsequent requests (e.g. authorization and token issuance, refresh token issuance and access token refreshment). The authorization server cannot assure any property of the client to end-users. Automatic processing of re-authorizations could be allowed as well. Such client credentials can be generated automatically without any validation of client properties, which makes it another option especially for native applications. Revocation will affect a single deployment only.

4. Threat Model

This section gives a comprehensive threat model of OAuth 2.0. Threats are grouped first by attacks directed against an OAuth component, which are client, authorization server, and resource server. Subsequently, they are grouped by flow, e.g. obtain token or access protected resources. Every countermeasure description refers to a detailed description in Section 5.

4.1. Clients

This section describes possible threats directed to OAuth clients.

4.1.1. Threat: Obtain Client Secrets

The attacker could try to get access to the secret of a particular client in order to:
o replay its refresh tokens and authorization codes, or

o obtain tokens on behalf of the attacked client with the privileges of that client_id acting as an instance of the client.

The resulting impact would be:

o Client authentication of access to authorization server can be bypassed

o Stolen refresh tokens or authorization codes can be replayed

Depending on the client category, the following attacks could be utilized to obtain the client secret.

Attack: Obtain Secret From Source Code or Binary:

This applies for all client types. For open source projects, secrets can be extracted directly from source code in their public repositories. Secrets can be extracted from application binaries just as easily when published source is not available to the attacker. Even if an application takes significant measures to obfuscate secrets in their application distribution one should consider that the secret can still be reverse-engineered by anyone with access to a complete functioning application bundle or binary.

Countermeasures:

o Don’t issue secrets to public clients or clients with inappropriate security policy – Section 5.2.3.1

o Require user consent for public clients – Section 5.2.3.2

o Use deployment-specific client secrets – Section 5.2.3.4

o Revoke client secrets – Section 5.2.3.6

Attack: Obtain a Deployment-Specific Secret:

An attacker may try to obtain the secret from a client installation, either from a web site (web server) or a particular devices (native application).

Countermeasures:
4.1.2. Threat: Obtain Refresh Tokens

Depending on the client type, there are different ways refresh tokens may be revealed to an attacker. The following sub-sections give a more detailed description of the different attacks with respect to different client types and further specialized countermeasures. Before detailing those threats, here are some generally applicable countermeasures:

- The authorization server should validate the client id associated with the particular refresh token with every refresh request - Section 5.2.2.2
- Limit token scope - Section 5.1.5.1
- Revoke refresh tokens - Section 5.2.2.4
- Revoke client secrets - Section 5.2.3.6
- Refresh tokens can automatically be replaced in order to detect unauthorized token usage by another party (Refresh Token Rotation) - Section 5.2.2.3

**Attack: Obtain Refresh Token from Web application:**

An attacker may obtain the refresh tokens issued to a web application by way of overcoming the web server’s security controls. Impact: Since a web application manages the user accounts of a certain site, such an attack would result in an exposure of all refresh tokens on that site to the attacker.

**Countermeasures:**

- Standard web server protection measures - Section 5.3.2
- Use strong client authentication (e.g. client_assertion / client_token), so the attacker cannot obtain the client secret required to exchange the tokens - Section 5.2.3.7
Attack: Obtain Refresh Token from Native clients:

On native clients, leakage of a refresh token typically affects a single user, only.

Read from local file system: The attacker could try get file system access on the device and read the refresh tokens. The attacker could utilize a malicious application for that purpose.

Countermeasures:

- Store secrets in a secure storage - Section 5.3.3
- Utilize device lock to prevent unauthorized device access - Section 5.3.4

Attack: Steal device:

The host device (e.g. mobile phone) may be stolen. In that case, the attacker gets access to all applications under the identity of the legitimate user.

Countermeasures:

- Utilize device lock to prevent unauthorized device access - Section 5.3.4
- Where a user knows the device has been stolen, they can revoke the affected tokens - Section 5.2.2.4

Attack: Clone Device:

All device data and applications are copied to another device. Applications are used as-is on the target device.

Countermeasures:

- Utilize device lock to prevent unauthorized device access - Section 5.3.4
- Combine refresh token request with device identification - Section 5.2.2.5
- Refresh Token Rotation - Section 5.2.2.3
Where a user knows the device has been cloned, they can use this countermeasure - Refresh Token Revocation - Section 5.2.2.4

4.1.3. Threat: Obtain Access Tokens

Depending on the client type, there are different ways access tokens may be revealed to an attacker. Access tokens could be stolen from the device if the application stores them in a storage, which is accessible to other applications.

Impact: Where the token is a bearer token and no additional mechanism is used to identify the client, the attacker can access all resources associated with the token and its scope.

Countermeasures:

- Keep access tokens in transient memory and limit grants: Section 5.1.6
- Limit token scope - Section 5.1.5.1
- Keep access tokens in private memory or apply same protection means as for refresh tokens - Section 5.2.2
- Keep access token lifetime short - Section 5.1.5.3

4.1.4. Threat: End-user credentials phished using compromised or embedded browser

A malicious application could attempt to phish end-user passwords by misusing an embedded browser in the end-user authorization process, or by presenting its own user-interface instead of allowing trusted system browser to render the authorization user interface. By doing so, the usual visual trust mechanisms may be bypassed (e.g. TLS confirmation, web site mechanisms). By using an embedded or internal client application user interface, the client application has access to additional information it should not have access to (e.g. uid/password).

Impact: If the client application or the communication is compromised, the user would not be aware and all information in the authorization exchange could be captured such as username and password.

Countermeasures:
o The OAuth flow is designed so that client applications never need to know user passwords. Client applications should avoid directly asking users for their credentials. In addition, end users could be educated about phishing attacks and best practices, such as only accessing trusted clients, as OAuth does not provide any protection against malicious applications and the end user is solely responsible for the trustworthiness of any native application installed.

o Client applications could be validated prior to publication in an application market for users to access. That validation is out of scope for OAuth but could include validating that the client application handles user authentication in an appropriate way.

o Client developers should not write client applications that collect authentication information directly from users and should instead delegate this task to a trusted system component, e.g. the system-browser.

4.1.5. Threat: Open Redirectors on client

An open redirector is an endpoint using a parameter to automatically redirect a user-agent to the location specified by the parameter value without any validation. If the authorization server allows the client to register only part of the redirection URI, an attacker can use an open redirector operated by the client to construct a redirection URI that will pass the authorization server validation but will send the authorization code or access token to an endpoint under the control of the attacker.

Impact: An attacker could gain access to authorization codes or access tokens

Countermeasure

o require clients to register full redirection URI Section 5.2.3.5

4.2. Authorization Endpoint

4.2.1. Threat: Password phishing by counterfeit authorization server

OAuth makes no attempt to verify the authenticity of the Authorization Server. A hostile party could take advantage of this by intercepting the Client’s requests and returning misleading or otherwise incorrect responses. This could be achieved using DNS or ARP spoofing. Wide deployment of OAuth and similar protocols may cause users to become inured to the practice of being redirected to websites where they are asked to enter their passwords. If users are
not careful to verify the authenticity of these websites before entering their credentials, it will be possible for attackers to exploit this practice to steal Users’ passwords.

Countermeasures:

- Authorization servers should consider such attacks when developing services based on OAuth, and should require the use of transport-layer security for any requests where the authenticity of the authorization server or of request responses is an issue (see Section 5.1.2).

- Authorization servers should attempt to educate Users about the risks phishing attacks pose, and should provide mechanisms that make it easy for users to confirm the authenticity of their sites.

4.2.2. Threat: User unintentionally grants too much access scope

When obtaining end user authorization, the end-user may not understand the scope of the access being granted and to whom or they may end up providing a client with access to resources which should not be permitted.

Countermeasures:

- Explain the scope (resources and the permissions) the user is about to grant in an understandable way - Section 5.2.4.2

- Narrow scope based on client - When obtaining end user authorization and where the client requests scope, the authorization server may want to consider whether to honour that scope based on the client identifier. That decision is between the client and authorization server and is outside the scope of this spec. The authorization server may also want to consider what scope to grant based on the client type, e.g. providing lower scope to public clients. - Section 5.1.5.1

4.2.3. Threat: Malicious client obtains existing authorization by fraud

Authorization servers may wish to automatically process authorization requests from clients which have been previously authorized by the user. When the user is redirected to the authorization server’s end-user authorization endpoint to grant access, the authorization server detects that the user has already granted access to that particular client. Instead of prompting the user for approval, the authorization server automatically redirects the user back to the client.
A malicious client may exploit that feature and try to obtain such an authorization code instead of the legitimate client.

Countermeasures:

- Authorization servers should not automatically process repeat authorizations to public clients unless the client is validated using a pre-registered redirect URI (Section 5.2.3.5)
- Authorization servers can mitigate the risks associated with automatic processing by limiting the scope of Access Tokens obtained through automated approvals - Section 5.1.5.1

4.2.4. Threat: Open redirector

An attacker could use the end-user authorization endpoint and the redirection URI parameter to abuse the authorization server as an open redirector. An open redirector is an endpoint using a parameter to automatically redirect a user-agent to the location specified by the parameter value without any validation.

Impact: An attacker could utilize a user’s trust in your authorization server to launch a phishing attack.

Countermeasure

- require clients to register full redirection URI Section 5.2.3.5
- don’t redirect to redirection URI, if client identifier or redirection URI can’t be verified Section 5.2.3.5

4.3. Token endpoint

4.3.1. Threat: Eavesdropping access tokens

Attackers may attempt to eavesdrop access token in transit from the authorization server to the client.

Impact: The attacker is able to access all resources with the permissions covered by the scope of the particular access token.

Countermeasures:

- As per the core OAuth spec, the authorization servers must ensure that these transmissions are protected using transport-layer mechanisms such as TLS (see Section 5.1.1).
4.3.2. Threat: Obtain access tokens from authorization server database

This threat is applicable if the authorization server stores access tokens as handles in a database. An attacker may obtain access tokens from the authorization server's database by gaining access to the database or launching a SQL injection attack. Impact: disclosure of all access tokens.

Countermeasures:

- Enforce system security measures - Section 5.1.4.1.1
- Store access token hashes only - Section 5.1.4.1.3
- Enforce standard SQL injection Countermeasures - Section 5.1.4.1.2

4.3.3. Threat: Disclosure of client credentials during transmission

An attacker could attempt to eavesdrop the transmission of client credentials between client and server during the client authentication process or during OAuth token requests. Impact: Revelation of a client credential enabling phishing or impersonation of a client service.

Countermeasures:

- The transmission of client credentials must be protected using transport-layer mechanisms such as TLS (see Section 5.1.1).
- Alternative authentication means, which do not require to send plaintext credentials over the wire (e.g. Hash-based Message Authentication Code)

4.3.4. Threat: Obtain client secret from authorization server database

An attacker may obtain valid client_id/secret combinations from the authorization server’s database by gaining access to the database or launching a SQL injection attack. Impact: disclosure of all client_id/secret combinations. This allows the attacker to act on behalf of legitimate clients.

Countermeasures:

- If end-to-end confidentiality cannot be guaranteed, reducing scope (see Section 5.1.5.1) and expiry time (Section 5.1.5.3) for access tokens can be used to reduce the damage in case of leaks.
o Enforce system security measures - Section 5.1.4.1.1
o Enforce standard SQL injection Countermeasures - Section 5.1.4.1.2
o Ensure proper handling of credentials as per Enforce credential storage protection best practices.

4.3.5. Threat: Obtain client secret by online guessing

An attacker may try to guess valid client_id/secret pairs. Impact: disclosure of single client_id/secret pair.

Countermeasures:

o Use high entropy for secrets - Section 5.1.4.2.2
o Lock accounts - Section 5.1.4.2.3
o Use Strong Client Authentication - Section 5.2.3.7

4.4. Obtaining Authorization

This section covers threats which are specific to certain flows utilized to obtain access tokens. Each flow is characterized by response types and/or grant types on the end-user authorization and token endpoint, respectively.

4.4.1. Authorization Code

4.4.1.1. Threat: Eavesdropping or leaking authorization codes

An attacker could try to eavesdrop transmission of the authorization code between authorization server and client. Furthermore, authorization codes are passed via the browser which may unintentionally leak those codes to untrusted web sites and attackers in different ways:

o Referrer headers: browsers frequently pass a "referer" header when a web page embeds content, or when a user travels from one web page to another web page. These referrer headers may be sent even when the origin site does not trust the destination site. The referrer header is commonly logged for traffic analysis purposes.

o Request logs: web server request logs commonly include query parameters on requests.

o Open redirectors: web sites sometimes need to send users to another destination via a redirector. Open redirectors pose a
particular risk to web-based delegation protocols because the redirector can leak verification codes to untrusted destination sites.

- Browser history: web browsers commonly record visited URLs in the browser history. Another user of the same web browser may be able to view URLs that were visited by previous users.

Note: A description of a similar attacks on the SAML protocol can be found at [OASIS.sstc-saml-bindings-1.1], Section 4.1.1.9.1, [gross-sec-analysis], and [OASIS.sstc-gross-sec-analysis-response-01].

Countermeasures:

- As per the core OAuth spec, the authorization server as well as the client must ensure that these transmissions are protected using transport-layer mechanisms such as TLS (see Section 5.1.1).
- The authorization server will require the client to authenticate wherever possible, so the binding of the authorization code to a certain client can be validated in a reliable way (see Section 5.2.4.4).
- Use short expiry time for authorization codes – Section 5.1.5.3
- The authorization server should enforce a one time usage restriction (see Section 5.1.5.4).
- If an Authorization Server observes multiple attempts to redeem an authorization code, the Authorization Server may want to revoke all tokens granted based on the authorization code (see Section 5.2.1.1).
- In the absence of these countermeasures, reducing scope (Section 5.1.5.1) and expiry time (Section 5.1.5.3) for access tokens can be used to reduce the damage in case of leaks.
- The client server may reload the target page of the redirection URI in order to automatically cleanup the browser cache.

4.4.1.2. Threat: Obtain authorization codes from authorization server database

This threat is applicable if the authorization server stores authorization codes as handles in a database. An attacker may obtain authorization codes from the authorization server’s database by gaining access to the database or launching a SQL injection attack.
Impact: disclosure of all authorization codes, most likely along with the respective redirect_uri and client_id values.

Countermeasures:

- Best practices for credential storage protection should be employed - Section 5.1.4.1
- Enforce system security measures - Section 5.1.4.1.1
- Store access token hashes only - Section 5.1.4.1.3
- Standard SQL injection countermeasures - Section 5.1.4.1.2

4.4.1.3. Threat: Online guessing of authorization codes

An attacker may try to guess valid authorization code values and send it using the grant type "code" in order to obtain a valid access token.

Impact: disclosure of single access token, probably also associated refresh token.

Countermeasures:

- Handle-based tokens must use high entropy: Section 5.1.4.2.2
- Assertion-based tokens should be signed: Section 5.1.5.9
- Authenticate the client, adds another value the attacker has to guess - Section 5.2.3.4
- Binding of authorization code to redirection URI, adds another value the attacker has to guess - Section 5.2.4.5
- Use short expiry time for tokens - Section 5.1.5.3

4.4.1.4. Threat: Malicious client obtains authorization

A malicious client could pretend to be a valid client and obtain an access authorization that way. The malicious client could even utilize screen scraping techniques in order to simulate the user consent in the authorization flow.

Assumption: It is not the task of the authorization server to protect the end-user's device from malicious software. This is the responsibility of the platform running on the particular device probably in cooperation with other components of the respective
ecosystem (e.g. an application management infrastructure). The sole responsibility of the authorization server is to control access to the end-user's resources living in resource servers and to prevent unauthorized access to them via the OAuth protocol. Based on this assumption, the following countermeasures are available to cope with the threat.

Countermeasures:

- The authorization server should authenticate the client, if possible (see Section 5.2.3.4). Note: the authentication takes place after the end-user has authorized the access.

- The authorization server should validate the client’s redirection URI against the pre-registered redirection URI, if one exists (see Section 5.2.3.5). Note: An invalid redirect URI indicates an invalid client whereas a valid redirect URI does not necessarily indicate a valid client. The level of confidence depends on the client type. For web applications, the confidence is high since the redirect URI refers to the globally unique network endpoint of this application whose fully qualified domain name (FQDN) is also validated using HTTPS server authentication by the user agent. In contrast for native clients, the redirect URI typically refers to device local resources, e.g. a custom scheme. So a malicious client on a particular device can use the valid redirect URI the legitimate client uses on all other devices.

- After authenticating the end-user, the authorization server should ask him/her for consent. In this context, the authorization server should explain to the end-user the purpose, scope, and duration of the authorization the client asked for. Moreover, the authorization server should show the user any identity information it has for that client. It is up to the user to validate the binding of this data to the particular application (e.g. Name) and to approve the authorization request. (see Section 5.2.4.3).

- The authorization server should not perform automatic re-authorizations for clients it is unable to reliably authenticate or validate (see Section 5.2.4.1).

- If the authorization server automatically authenticates the end-user, it may nevertheless require some user input in order to prevent screen scraping. Examples are CAPTCHAs (Completely Automated Public Turing test to tell Computers and Humans Apart) or other multi-factor authentication techniques such as random questions, token code generators, etc.
4.4.1.5. Threat: Authorization code phishing

A hostile party could impersonate the client site and get access to the authorization code. This could be achieved using DNS or ARP spoofing. This applies to clients, which are web applications, thus the redirect URI is not local to the host where the user’s browser is running.

Impact: This affects web applications and may lead to a disclosure of authorization codes and, potentially, the corresponding access and refresh tokens.

Countermeasures:

It is strongly recommended that one of the following countermeasures is utilized in order to prevent this attack:

- The redirection URI of the client should point to a HTTPS protected endpoint and the browser should be utilized to authenticate this redirection URI using server authentication (see Section 5.1.2).

- The authorization server should require the client to be authenticated, i.e. confidential client, so the binding of the authorization code to a certain client can be validated in a reliable way (see Section 5.2.4.4).

4.4.1.6. Threat: User session impersonation

A hostile party could impersonate the client site and impersonate the user’s session on this client. This could be achieved using DNS or ARP spoofing. This applies to clients, which are web applications, thus the redirect URI is not local to the host where the user’s browser is running.

Impact: An attacker who intercepts the authorization code as it is sent by the browser to the callback endpoint can gain access to protected resources by submitting the authorization code to the client. The client will exchange the authorization code for an access token and use the access token to access protected resources for the benefit of the attacker, delivering protected resources to the attacker, or modifying protected resources as directed by the attacker. If OAuth is used by the client to delegate authentication to a social site (e.g. as in the implementation of "Login" button to...
a third-party social network site), the attacker can use the intercepted authorization code to log in to the client as the user.

Note: Authenticating the client during authorization code exchange will not help to detect such an attack as it is the legitimate client that obtains the tokens.

Countermeasures:

- In order to prevent an attacker from impersonating the end-users session, the redirection URI of the client should point to a HTTPS protected endpoint and the browser should be utilized to authenticate this redirection URI using server authentication (see Section 5.1.2)

4.4.1.7. Threat: Authorization code leakage through counterfeit client

The attack leverages the authorization code grant type in an attempt to get another user (victim) to log-in, authorize access to his/her resources, and subsequently obtain the authorization code and inject it into a client application using the attacker's account. The goal is to associate an access authorization for resources of the victim with the user account of the attacker on a client site.

The attacker abuses an existing client application and combines it with his own counterfeit client web site. The attack depends on the victim expecting the client application to request access to a certain resource server. The victim, seeing only a normal request from an expected application, approves the request. The attacker then uses the victim’s authorization to gain access to the information unknowingly authorized by the victim.

The attacker conducts the following flow:

1. The attacker accesses the client web site (or application) and initiates data access to a particular resource server. The client web site in turn initiates an authorization request to the resource server's authorization server. Instead of proceeding with the authorization process, the attacker modifies the authorization server end-user authorization URL as constructed by the client to include a redirection URI parameter referring to a web site under his control (attacker’s web site).

2. The attacker tricks another user (the victim) to open that modified end-user authorization URI and to authorize access (e.g. an email link, or blog link). The way the attacker achieves that goal is out of scope.
3. Having clicked the link, the victim is requested to authenticate and authorize the client site to have access.

4. After completion of the authorization process, the authorization server redirects the user agent to the attacker’s web site instead of the original client web site.

5. The attacker obtains the authorization code from his web site by means out of scope of this document.

6. He then constructs a redirection URI to the target web site (or application) based on the original authorization request’s redirection URI and the newly obtained authorization code and directs his user agent to this URL. The authorization code is injected into the original client site (or application).

7. The client site uses the authorization code to fetch a token from the authorization server and associates this token with the attacker’s user account on this site.

8. The attacker may now access the victim’s resources using the client site.

Impact: The attacker gains access to the victim’s resources associated with his account on the client site.

Countermeasures:

- The attacker will need to use another redirection URI for its authorization process rather than the target web site because it needs to intercept the flow. So if the authorization server associates the authorization code with the redirection URI of a particular end-user authorization and validates this redirection URI with the redirection URI passed to the token’s endpoint, such an attack is detected (see Section 5.2.4.5).

- The authorization server may also enforce the usage and validation of pre-registered redirect URIs (see Section 5.2.3.5). This will allow for an early recognition of authorization code disclosure to counterfeit clients.

- For native applications, one could also consider to use deployment-specific client ids and secrets (see Section 5.2.3.4, along with the binding of authorization code to client_id (see Section 5.2.4.4), to detect such an attack because the attacker does not have access the deployment-specific secret. Thus he will not be able to exchange the authorization code.
4.4.1.8. Threat: CSRF attack against redirect-uri

Cross-Site Request Forgery (CSRF) is a web-based attack whereby HTTP requests are transmitted from a user that the website trusts or has authenticated (e.g., via HTTP redirects or HTML forms). CSRF attacks on OAuth approvals can allow an attacker to obtain authorization to OAuth protected resources without the consent of the User.

This attack works against the redirection URI used in the authorization code flow. An attacker could authorize an authorization code to their own protected resources on an authorization server. He then aborts the redirect flow back to the client on his device and tricks the victim into executing the redirect back to the client. The client receives the redirect, fetches the token(s) from the authorization server and associates the victim’s client session with the resources accessible using the token.

Impact: The user accesses resources on behalf of the attacker. The effective impact depends on the type of resource accessed. For example, the user may upload private items to an attacker’s resources. Or when using OAuth in 3rd party login scenarios, the user may associate his client account with the attacker’s identity at the external identity provider. This way the attacker could easily access the victim’s data at the client by logging in from another device with his credentials at the external identity provider.

Countermeasures:

- The state parameter should be used to link the authorization request with the redirection URI used to deliver the access token. Section 5.3.5
- The client may consider using other flows, which are not vulnerable to this kind of attack such as "Implicit Grant" or "Resource Owner Password Credentials" (see Section 4.4.2 or Section 4.4.3).

4.4.1.9. Threat: Clickjacking attack against authorization

Clickjacking is a malicious site loads the target site in a transparent iFrame (see [iFrame]) overlayed on top of a set of dummy buttons which are carefully constructed to be placed directly under important buttons on the target site. When a user clicks a visible button, they are actually clicking a button (such as an "Authorize"
button) on the hidden page.

Impact: An attacker can steal a user’s authentication credentials and access their resources.

Countermeasure

- For newer browsers, avoidance of iFrames during authorization can be enforced server side by using the X-FRAME-OPTION header - Section 5.2.2.6

- For older browsers, javascript frame-busting (see [framebusting]) techniques can be used but may not be effective in all browsers.

4.4.1.10. Threat: Resource Owner Impersonation

When a client requests access to protected resources, the authorization flow normally involves the resource owner’s explicit response to the access request, either granting or denying access to the protected resources. A malicious client can exploit knowledge of the structure of this flow in order to gain authorization without the resource owner’s consent, by transmitting the necessary requests programmatically, and simulating the flow against the authorization server. That way, the client may gain access to the victim’s resources without her approval. An authorization server will be vulnerable to this threat, if it uses non-interactive authentication mechanisms or splits the authorization flow across multiple pages.

The malicious client might embed a hidden HTML user agent, interpret the HTML forms sent by the authorization server, and automatically send the corresponding form post requests. As a pre-requisite, the attacker must be able to execute the authorization process in the context of an already authenticated session of the resource owner with the authorization server. There are different ways to achieve this:

- The malicious client could abuse an existing session in an external browser or cross-browser cookies on the particular device.

- The malicious client could also request authorization for an initial scope acceptable to the user and then silently abuse the resulting session in his browser instance to "silently" request another scope.

- Alternatively, the attacker might exploit an authorization server’s ability to authenticate the resource owner automatically and without user interactions, e.g. based on certificates.
In all cases, such an attack is limited to clients running on the victim's device, within the user agent or as native app.

Please note: Such attacks cannot be prevented using CSRF countermeasures, since the attacker just "executes" the URLs as prepared by the authorization server including any nonce etc.

Countermeasures:

Authorization servers should decide, based on an analysis of the risk associated with this threat, whether to detect and prevent this threat.

In order to prevent such an attack, the authorization server may force a user interaction based on non-predictable input values as part of the user consent approval. The authorization server could

- combine password authentication and user consent in a single form,
- make use of CAPTCHAs, or
- or use one-time secrets sent out of band to the resource owner (e.g. via text or instant message).

Alternatively in order to allow the resource owner to detect abuse, the authorization server could notify the resource owner of any approval by appropriate means, e.g. text or instant message or e-Mail.

4.4.1.11. Threat: DoS, Exhaustion of resources attacks

If an authorization server includes a nontrivial amount of entropy in authorization codes or access tokens (limiting the number of possible codes/tokens) and automatically grants either without user intervention and has no limit on code or access tokens per user, an attacker could exhaust the pool of authorization codes by repeatedly directing the user's browser to request code or access tokens.

Countermeasures:

- The authorization server should consider limiting the number of access tokens granted per user. The authorization server should include a nontrivial amount of entropy in authorization codes.
4.4.1.12. Threat: DoS using manufactured authorization codes

An attacker who owns a botnet can locate the redirect URIs of clients that listen on HTTP, access them with random authorization codes, and cause a large number of HTTPS connections to be concentrated onto the authorization server. This can result in a DoS attack on the authorization server.

This attack can still be effective even when CSRF defense/the `state` parameter (see Section 4.4.1.8) is deployed on the client side. With such a defense, the attacker might need to incur an additional HTTP request to obtain a valid CSRF code/ state parameter. This apparently cuts down the effectiveness of the attack by a factor of 2. However, if the HTTPS/HTTP cost ratio is higher than 2 (the cost factor is estimated to be around 3.5x at [ssl-latency]) the attacker still achieves a magnification of resource utilization at the expense of the authorization server.

Impact: There are a few effects that the attacker can accomplish with this OAuth flow that they cannot easily achieve otherwise.

1. Connection laundering: With the clients as the relay between the attacker and the authorization server, the authorization server learns little or no information about the identity of the attacker. Defenses such as rate limiting on the offending attacker machines are less effective due to the difficulty to identify the attacking machines. Although an attacker could also launder its connections through an anonymizing system such as Tor, the effectiveness of that approach depends on the capacity of the anonymizing system. On the other hand, a potentially large number of OAuth clients could be utilized for this attack.

2. Asymmetric resource utilization: The attacker incurs the cost of an HTTP connection and causes an HTTPS connection to be made on the authorization server; and the attacker can co-ordinate the timing of such HTTPS connections across multiple clients relatively easily. Although the attacker could achieve something similar, say, by including an iframe pointing to the HTTPS URL of the authorization server in an HTTP web page and lure web users to visit that page, timing attacks using such a scheme may be more difficult as it seems nontrivial to synchronize a large number of users to simultaneously visit a particular site under the attacker’s control.

Countermeasures
o Though not a complete countermeasure by themselves, CSRF defense and the 'state' parameter created with secure random codes should be deployed on the client side. The client should forward the authorization code to the authorization server only after both the CSRF token and the 'state' parameter are validated.

o If the client authenticates the user, either through a single-sign-on protocol or through local authentication, the client should suspend the access by a user account if the number of invalid authorization codes submitted by this user exceeds a certain threshold.

o The authorization server should send an error response to the client reporting an invalid authorization code and rate limit or disallow connections from clients whose number of invalid requests exceeds a threshold.

4.4.1.13. Threat: Code substitution (OAuth Login)

An attacker could attempt to login to an application or web site using a victim's identity. Applications relying on identity data provided by an OAuth protected service API to login users are vulnerable to this threat. This pattern can be found in so-called "social login" scenarios.

As a pre-requisite, a resource server offers an API to obtain personal information about a user which could be interpreted as having obtained a user identity. In this sense the client is treating the resource server API as an "identity" API. A client utilizes OAuth to obtain an access token for the identity API. It then queries the identity API for an identifier and uses it to look up its internal user account data (login). The client assumes that because it was able to obtain information about the user, that the user has been authenticated.

If the client uses the grant type "code", the attacker needs to gather a valid authorization code of the respective victim from the same identity provider used by the target client application. The attacker tricks the victim into login into a malicious app (which may appear to be legitimate to the Identity Provider) using the same identity provider as the target application. This results in the Identity Provider's authorization server issuing an authorization code for the respective identity API. The malicious app then sends this code to the attacker, which in turn triggers a login process within the target application. The attacker now manipulates the authorization response and substitutes their code (bound to their identity) for the victim's code. This code is then exchanged by the client for an access token, which in turn is accepted by the identity
API since the audience, with respect to the resource server, is correct. But since the identifier returned by the identity API is determined by the identity in the access token (issued based on the victim’s code), the attacker is logged into the target application under the victim’s identity.

Impact: the attacker gains access to an application and user-specific data within the application.

Countermeasures:

- All clients must indicate their client id with every request to exchange an authorization code for an access token. The authorization server must validate whether the particular authorization code has been issued to the particular client. If possible, the client shall be authenticated beforehand.

- Clients should use appropriate protocol, such as OpenID (cf. [openid]) or SAML (cf. [OASIS.sstc-saml-bindings-1.1]) to implement user login. Both support audience restrictions on clients.

4.4.2. Implicit Grant

In the implicit grant type flow, the access token is directly returned to the client as a fragment part of the redirection URI. It is assumed that the token is not sent to the redirection URI target as HTTP user agents do not send the fragment part of URIs to HTTP servers. Thus an attacker cannot eavesdrop the access token on this communication path and it cannot leak through HTTP referee headers.

4.4.2.1. Threat: Access token leak in transport/end-points

This token might be eavesdropped by an attacker. The token is sent from server to client via a URI fragment of the redirection URI. If the communication is not secured or the end-point is not secured, the token could be leaked by parsing the returned URI.

Impact: the attacker would be able to assume the same rights granted by the token.

Countermeasures:

- The authorization server should ensure confidentiality (e.g. using TLS) of the response from the authorization server to the client (see Section 5.1.1).
4.4.2.2. Threat: Access token leak in browser history

An attacker could obtain the token from the browser’s history. Note this means the attacker needs access to the particular device.

Countermeasures:

- Use short expiry time for tokens (see Section 5.1.5.3) and reduced scope of the token may reduce the impact of that attack (see Section 5.1.5.1).
- Make responses non-cachable

4.4.2.3. Threat: Malicious client obtains authorization

A malicious client could attempt to obtain a token by fraud.

The same countermeasures as for Section 4.4.1.4 are applicable, except client authentication.

4.4.2.4. Threat: Manipulation of scripts

A hostile party could act as the client web server and replace or modify the actual implementation of the client (script). This could be achieved using DNS or ARP spoofing. This applies to clients implemented within the Web Browser in a scripting language.

Impact: The attacker could obtain user credential information and assume the full identity of the user.

Countermeasures:

- The authorization server should authenticate the server from which scripts are obtained (see Section 5.1.2).
- The client should ensure that scripts obtained have not been altered in transport (see Section 5.1.1).
- Introduce one time per-use secrets (e.g. client_secret) values that can only be used by scripts in a small time window once loaded from a server. The intention would be to reduce the effectiveness of copying client-side scripts for re-use in an attackers modified code.
4.4.2.5. Threat: CSRF attack against redirect-uri

CSRF attacks (see Section 4.4.1.8) also work against the redirection URI used in the implicit grant flow. An attacker could acquire an access token to their own protected resources. He could then construct a redirection URI and embed their access token in that URI. If he can trick the user into following the redirection URI and the client does not have protection against this attack, the user may have the attacker’s access token authorized within their client.

Impact: The user accesses resources on behalf of the attacker. The effective impact depends on the type of resource accessed. For example, the user may upload private items to an attacker’s resources. Or when using OAuth in 3rd party login scenarios, the user may associate his client account with the attacker’s identity at the external identity provider. This way the attacker could easily access the victim’s data at the client by logging in from another device with his credentials at the external identity provider.

Countermeasures:

o The state parameter should be used to link the authorization request with the redirection URI used deliver the access token. This will ensure the client is not tricked into completing any redirect callback unless it is linked to an authorization request the client initiated. The state parameter should be unguessable and the client should be capable of keeping the state parameter secret.

o Client developers and end-user can be educated not follow untrusted URLs.

4.4.2.6. Threat: Token substitution (OAuth Login)

An attacker could attempt to login to an application or web site using a victim’s identity. Applications relying on identity data provided by an OAuth protected service API to login users are vulnerable to this threat. This pattern can be found in so-called "social login" scenarios.

As a pre-requisite, a resource server offers an API to obtain personal information about a user which could be interpreted as having obtained a user identity. In this sense the client is treating the resource server API as an "identity" API. A client utilizes OAuth to obtain an access token for the identity API. It then queries the identity API for an identifier and uses it to look up its internal user account data (login). The client assumes that because it was able to obtain information about the user, that the
user has been authenticated.

To succeed, the attacker needs to gather a valid access token of the respective victim from the same identity provider used by the target client application. The attacker tricks the victim into login into a malicious app (which may appear to be legitimate to the Identity Provider) using the same identity provider as the target application. This results in the Identity Provider’s authorization server issuing an access token for the respective identity API. The malicious app then sends this access token to the attacker, which in turn triggers a login process within the target application. The attacker now manipulates the authorization response and substitutes their access token (bound to their identity) for the victim’s access token. This token is accepted by the identity API since the audience, with respect to the resource server, is correct. But since the identifier returned by the identity API is determined by the identity in the access token, the attacker is logged into the target application under the victim’s identity.

Impact: the attacker gains access to an application and user-specific data within the application.

Countermeasures:

- Clients should use appropriate protocol, such as OpenID (cf. [openid]) or SAML (cf. [OASIS.sstc-saml-bindings-1.1]) to implement user login. Both support audience restrictions on clients.

4.4.3. Resource Owner Password Credentials

The "Resource Owner Password Credentials" grant type (see [I-D.ietf-oauth-v2], Section 4.3), often used for legacy/migration reasons, allows a client to request an access token using an end-users user-id and password along with its own credential. This grant type has higher risk because it maintains the uid/password anti-pattern. Additionally, because the user does not have control over the authorization process, clients using this grant type are not limited by scope, but instead have potentially the same capabilities as the user themselves. As there is no authorization step, the ability to offer token revocation is bypassed.

Because passwords are often used for more than 1 service, this anti-pattern may also risk whatever else is accessible with the supplied credential. Additionally any easily derived equivalent (e.g. joe@example.com and joe@example.net) might easily allow someone to guess that the same password can be used elsewhere.
Impact: The resource server can only differentiate scope based on the access token being associated with a particular client. The client could also acquire long-living tokens and pass them up to a attacker web service for further abuse. The client, eavesdroppers, or endpoints could eavesdrop user id and password.

Countermeasures:

- Except for migration reasons, minimize use of this grant type
- The authorization server should validate the client id associated with the particular refresh token with every refresh request – Section 5.2.2.2
- As per the core OAuth spec, the authorization server must ensure that these transmissions are protected using transport-layer mechanisms such as TLS (see Section 5.1.1).
- Rather than encouraging users to use a uid and password, service providers should instead encourage users not to use the same password for multiple services.
- Limit use of Resource Owner Password Credential grants to scenarios where the client application and the authorizing service are from the same organization.

4.4.3.1. Threat: Accidental exposure of passwords at client site

If the client does not provide enough protection, an attacker or disgruntled employee could retrieve the passwords for a user.

Countermeasures:

- Use other flows, which do not rely on the client’s cooperation for secure resource owner credential handling
- Use digest authentication instead of plaintext credential processing
- Obfuscate passwords in logs

4.4.3.2. Threat: Client obtains scopes without end-user authorization

All interaction with the resource owner is performed by the client. Thus it might, intentionally or unintentionally, happen that the client obtains a token with scope unknown for or unintended by the resource owner. For example, the resource owner might think the client needs and acquires read-only access to its media storage only
but the client tries to acquire an access token with full access permissions.

Countermeasures:

- Use other flows, which do not rely on the client’s cooperation for resource owner interaction
- The authorization server may generally restrict the scope of access tokens (Section 5.1.5.1) issued by this flow. If the particular client is trustworthy and can be authenticated in a reliable way, the authorization server could relax that restriction. Resource owners may prescribe (e.g. in their preferences) what the maximum scope is for clients using this flow.
- The authorization server could notify the resource owner by an appropriate media, e.g. e-Mail, of the grant issued (see Section 5.1.3).

4.4.3.3. Threat: Client obtains refresh token through automatic authorization

All interaction with the resource owner is performed by the client. Thus it might, intentionally or unintentionally, happen that the client obtains a long-term authorization represented by a refresh token even if the resource owner did not intend so.

Countermeasures:

- Use other flows, which do not rely on the client’s cooperation for resource owner interaction
- The authorization server may generally refuse to issue refresh tokens in this flow (see Section 5.2.2.1). If the particular client is trustworthy and can be authenticated in a reliable way (see client authentication), the authorization server could relax that restriction. Resource owners may allow or deny (e.g. in their preferences) to issue refresh tokens using this flow as well.
- The authorization server could notify the resource owner by an appropriate media, e.g. e-Mail, of the refresh token issued (see Section 5.1.3).
4.4.3.4. Threat: Obtain user passwords on transport

An attacker could attempt to eavesdrop the transmission of end-user credentials with the grant type "password" between client and server.

Impact: disclosure of a single end-users password.

Countermeasures:

- Ensure confidentiality of requests - Section 5.1.1
- alternative authentication means, which do not require to send plaintext credentials over the wire (e.g. Hash-based Message Authentication Code)

4.4.3.5. Threat: Obtain user passwords from authorization server database

An attacker may obtain valid username/password combinations from the authorization server’s database by gaining access to the database or launching a SQL injection attack.

Impact: disclosure of all username/password combinations. The impact may exceed the domain of the authorization server since many users tend to use the same credentials on different services.

Countermeasures:

- Enforce credential storage protection best practices - Section 5.1.4.1

4.4.3.6. Threat: Online guessing

An attacker may try to guess valid username/password combinations using the grant type "password".

Impact: Revelation of a single username/password combination.

Countermeasures:

- Utilize secure password policy - Section 5.1.4.2.1
- Lock accounts - Section 5.1.4.2.3
- Use tar pit - Section 5.1.4.2.4
- Use CAPTCHAs - Section 5.1.4.2.5
4.4.4. Client Credentials

Client credentials (see [I-D.ietf-oauth-v2], Section 3) consist of an identifier (not secret) combined with an additional means (such as a matching client secret) of authenticating a client. The threats to this grant type are similar to Section 4.4.3.

4.5. Refreshing an Access Token

4.5.1. Threat: Eavesdropping refresh tokens from authorization server

An attacker may eavesdrop refresh tokens when they are transmitted from the authorization server to the client.

Countermeasures:

- As per the core OAuth spec, the Authorization servers must ensure that these transmissions are protected using transport-layer mechanisms such as TLS (see Section 5.1.1).
- If end-to-end confidentiality cannot be guaranteed, reducing scope (see Section 5.1.5.1) and expiry time (see Section 5.1.5.3) for issued access tokens can be used to reduce the damage in case of leaks.

4.5.2. Threat: Obtaining refresh token from authorization server database

This threat is applicable if the authorization server stores refresh tokens as handles in a database. An attacker may obtain refresh tokens from the authorization server's database by gaining access to the database or launching a SQL injection attack.

Impact: disclosure of all refresh tokens

Countermeasures:

- Enforce credential storage protection best practices - Section 5.1.4.1
- Bind token to client id, if the attacker cannot obtain the required id and secret - Section 5.1.5.8
4.5.3. Threat: Obtain refresh token by online guessing

An attacker may try to guess valid refresh token values and send it using the grant type "refresh_token" in order to obtain a valid access token.

Impact: exposure of single refresh token and derivable access tokens.

Countermeasures:
- For handle-based designs - Section 5.1.4.2.2
- For assertion-based designs - Section 5.1.5.9
- Bind token to client id, because the attacker would guess the matching client id, too (see Section 5.1.5.8)
- Authenticate the client, adds another element the attacker has to guess (see Section 5.2.3.4)

4.5.4. Threat: Obtain refresh token phishing by counterfeit authorization server

An attacker could try to obtain valid refresh tokens by proxying requests to the authorization server. Given the assumption that the authorization server URL is well-known at development time or can at least be obtained from a well-known resource server, the attacker must utilize some kind of spoofing in order to succeed.

Countermeasures:
- Utilize server authentication (as described in Section 5.1.2)

4.6. Accessing Protected Resources

4.6.1. Threat: Eavesdropping access tokens on transport

An attacker could try to obtain a valid access token on transport between client and resource server. As access tokens are shared secrets between authorization and resource server, they should be treated with the same care as other credentials (e.g. end-user passwords).

Countermeasures:
- Access tokens sent as bearer tokens, should not be sent in the clear over an insecure channel. As per the core OAuth spec, transmission of access tokens must be protected using transport-
layer mechanisms such as TLS (see Section 5.1.1).

- A short lifetime reduces impact in case tokens are compromised (see Section 5.1.5.3).

- The access token can be bound to a client’s identifier and require the client to prove legitimate ownership of the token to the resource server (see Section 5.4.2).

4.6.2. Threat: Replay authorized resource server requests

An attacker could attempt to replay valid requests in order to obtain or to modify/destroy user data.

Countermeasures:

- The resource server should utilize transport security measures (e.g. TLS) in order to prevent such attacks (see Section 5.1.1). This would prevent the attacker from capturing valid requests.

- Alternatively, the resource server could employ signed requests (see Section 5.4.3) along with nonces and timestamps in order to uniquely identify requests. The resource server should detect and refuse every replayed request.

4.6.3. Threat: Guessing access tokens

Where the token is a handle, the attacker may use attempt to guess the access token values based on knowledge they have from other access tokens.

Impact: Access to a single user’s data.

Countermeasures:

- Handle Tokens should have a reasonable entropy (see Section 5.1.4.2.2) in order to make guessing a valid token value infeasible.

- Assertion (or self-contained token) tokens contents should be protected by a digital signature (see Section 5.1.5.9).

- Security can be further strengthened by using a short access token duration (see Section 5.1.5.2 and Section 5.1.5.3).
4.6.4. Threat: Access token phishing by counterfeit resource server

An attacker may pretend to be a particular resource server and to accept tokens from a particular authorization server. If the client sends a valid access token to this counterfeit resource server, the server in turn may use that token to access other services on behalf of the resource owner.

Countermeasures:

- Clients should not make authenticated requests with an access token to unfamiliar resource servers, regardless of the presence of a secure channel. If the resource server URL is well-known to the client, it may authenticate the resource servers (see Section 5.1.2).
- Associate the endpoint URL of the resource server the client talked to with the access token (e.g. in an audience field) and validate association at legitimate resource server. The endpoint URL validation policy may be strict (exact match) or more relaxed (e.g. same host). This would require to tell the authorization server the resource server endpoint URL in the authorization process.
- Associate an access token with a client and authenticate the client with resource server requests (typically via signature in order to not disclose secret to a potential attacker). This prevents the attack because the counterfeit server is assumed to lack the capability to correctly authenticate on behalf of the legitimate client to the resource server (Section 5.4.2).
- Restrict the token scope (see Section 5.1.5.1) and or limit the token to a certain resource server (Section 5.1.5.5).

4.6.5. Threat: Abuse of token by legitimate resource server or client

A legitimate resource server could attempt to use an access token to access another resource servers. Similarly, a client could try to use a token obtained for one server on another resource server.

Countermeasures:

- Tokens should be restricted to particular resource servers (see Section 5.1.5.5).
4.6.6. Threat: Leak of confidential data in HTTP-Proxies

The HTTP Authorization scheme (OAuth HTTP Authorization Scheme) is optional. However, [RFC2616] relies on the Authorization and WWW-Authenticate headers to distinguish authenticated content so that it can be protected. Proxies and caches, in particular, may fail to adequately protect requests not using these headers. For example, private authenticated content may be stored in (and thus retrievable from) publicly-accessible caches.

Countermeasures:

- Clients and resource servers not using the HTTP Authorization scheme (OAuth HTTP Authorization Scheme - see Section 5.4.1) should take care to use Cache-Control headers to minimize the risk that authenticated content is not protected. Such Clients should send a Cache-Control header containing the "no-store" option [RFC2616]. Resource server success (2XX status) responses to these requests should contain a Cache-Control header with the "private" option [RFC2616].

- Reducing scope (see Section 5.1.5.1) and expiry time (Section 5.1.5.3) for access tokens can be used to reduce the damage in case of leaks.

4.6.7. Threat: Token leakage via logfiles and HTTP referrers

If access tokens are sent via URI query parameters, such tokens may leak to log files and the HTTP "referer".

Countermeasures:

- Use authorization headers or POST parameters instead of URI request parameters (see Section 5.4.1).

- Set logging configuration appropriately

- Prevent unauthorized persons from access to system log files (see Section 5.1.4.1.1)

- Abuse of leaked access tokens can be prevented by enforcing authenticated requests (see Section 5.4.2).

- The impact of token leakage may be reduced by limiting scope (see Section 5.1.5.1) and duration (see Section 5.1.5.3) and enforcing one time token usage (see Section 5.1.5.4).
5. Security Considerations

This section describes the countermeasures as recommended to mitigate the threats as described in Section 4.

5.1. General

The general section covers considerations that apply generally across all OAuth components (client, resource server, token server, and user-agents).

5.1.1. Ensure confidentiality of requests

This is applicable to all requests sent from client to authorization server or resource server. While OAuth provides a mechanism for verifying the integrity of requests, it provides no guarantee of request confidentiality. Unless further precautions are taken, eavesdroppers will have full access to request content and may be able to mount interception or replay attacks through using content of request, e.g. secrets or tokens.

Attacks can be mitigated by using transport-layer mechanisms such as TLS [RFC5246]. A virtual private network (VPN), e.g. based on IPsec VPN [RFC4301], may considered as well.

Note: this document assumes end-to-end TLS protected connections between the respective protocol entities. Deployments deviating from this assumption by offloading TLS in between (e.g. on the data center edge) must refine this threat model in order to account for the additional (mainly insider) threat this may cause.

This is a countermeasure against the following threats:

- Replay of access tokens obtained on tokens endpoint or resource server’s endpoint
- Replay of refresh tokens obtained on tokens endpoint
- Replay of authorization codes obtained on tokens endpoint (redirect?)
- Replay of user passwords and client secrets

5.1.2. Utilize server authentication

HTTPS server authentication or similar means can be used to authenticate the identity of a server. The goal is to reliably bind the fully qualified domain name of the server to the public key
presented by the server during connection establishment (see [RFC2818]).

The client should validate the binding of the server to its domain name. If the server fails to prove that binding, it is considered a man-in-the-middle attack. The security measure depends on the certification authorities the client trusts for that purpose. Clients should carefully select those trusted CAs and protect the storage for trusted CA certificates from modifications.

This is a countermeasure against the following threats:

- Spoofing
- Proxying
- Phishing by counterfeit servers

5.1.3. Always keep the resource owner informed

Transparency to the resource owner is a key element of the OAuth protocol. The user should always be in control of the authorization processes and get the necessary information to meet informed decisions. Moreover, user involvement is a further security countermeasure. The user can probably recognize certain kinds of attacks better than the authorization server. Information can be presented/exchanged during the authorization process, after the authorization process, and every time the user wishes to get informed by using techniques such as:

- User consent forms
- Notification messages (e.g. e-Mail, SMS, ...). Note that notifications can be a phishing vector. Messages should be such that look-alike phishing messages cannot be derived from them.
- Activity/Event logs
- User self-care applications or portals

5.1.4. Credentials

This section describes countermeasures used to protect all kinds of credentials from unauthorized access and abuse. Credentials are long term secrets, such as client secrets and user passwords as well as all kinds of tokens (refresh and access token) or authorization codes.
5.1.4.1. Enforce credential storage protection best practices

Administrators should undertake industry best practices to protect the storage of credentials (see for example [owasp]). Such practices may include but are not limited to the following sub-sections.

5.1.4.1.1. Enforce Standard System Security Means

A server system may be locked down so that no attacker may get access to sensible configuration files and databases.

5.1.4.1.2. Enforce standard SQL Injection Countermeasures

If a client identifier or other authentication component is queried or compared against a SQL Database it may become possible for an injection attack to occur if parameters received are not validated before submission to the database.

- Ensure that server code is using the minimum database privileges possible to reduce the "surface" of possible attacks.
- Avoid dynamic SQL using concatenated input. If possible, use static SQL.
- When using dynamic SQL, parameterize queries using bind arguments. Bind arguments eliminate possibility of SQL injections.
- Filter and sanitize the input. For example, if an identifier has a known format, ensure that the supplied value matches the identifier syntax rules.

5.1.4.1.3. No cleartext storage of credentials

The authorization server should not store credentials in clear text. Typical approaches are to store hashes instead or to encrypt credentials. If the credential lacks a reasonable entropy level (because it is a user password) an additional salt will harden the storage to make offline dictionary attacks more difficult.

Note: Some authentication protocols require the authorization server to have access to the secret in the clear. Those protocols cannot be implemented if the server only has access to hashes. Credentials should strongly encrypted in those cases.

5.1.4.1.4. Encryption of credentials

For client applications, insecurely persisted client credentials are easy targets for attackers to obtain. Store client credentials using
an encrypted persistence mechanism such as a keystore or database. Note that compiling client credentials directly into client code makes client applications vulnerable to scanning as well as difficult to administer should client credentials change over time.

5.1.4.1.5. Use of asymmetric cryptography

Usage of asymmetric cryptography will free the authorization server of the obligation to manage credentials.

5.1.4.2. Online attacks on secrets

5.1.4.2.1. Utilize secure password policy

The authorization server may decide to enforce a complex user password policy in order to increase the user passwords’ entropy to hinder online password attacks. Note that too much complexity can increase the likelihood that users re-use passwords or write them down or otherwise store them insecurely.

5.1.4.2.2. Use high entropy for secrets

When creating secrets not intended for usage by human users (e.g. client secrets or token handles), the authorization server should include a reasonable level of entropy in order to mitigate the risk of guessing attacks. The token value should be >=128 bits long and constructed from a cryptographically strong random or pseudo-random number sequence (see [RFC4086] for best current practice) generated by the Authorization Server.

5.1.4.2.3. Lock accounts

Online attacks on passwords can be mitigated by locking the respective accounts after a certain number of failed attempts.

Note: This measure can be abused to lock down legitimate service users.

5.1.4.2.4. Use tar pit

The authorization server may react on failed attempts to authenticate by username/password by temporarily locking the respective account and delaying the response for a certain duration. This duration may increase with the number of failed attempts. The objective is to slow the attackers attempts on a certain username down.

Note: this may require a more complex and stateful design of the authorization server.
5.1.4.2.5. Use CAPTCHAs

The idea is to prevent programs from automatically checking huge number of passwords by requiring human interaction.

Note: this has a negative impact on user experience.

5.1.5. Tokens (access, refresh, code)

5.1.5.1. Limit token scope

The authorization server may decide to reduce or limit the scope associated with a token. The basis of this decision is out of scope, examples are:

- a client-specific policy, e.g. issue only less powerful tokens to public clients,
- a service-specific policy, e.g. it a very sensitive service,
- a resource-owner specific setting, or
- combinations of such policies and preferences.

The authorization server may allow different scopes dependent on the grant type. For example, end-user authorization via direct interaction with the end-user (authorization code) might be considered more reliable than direct authorization via grant type username/password. This means will reduce the impact of the following threats:

- token leakage
- token issuance to malicious software
- unintended issuance of to powerful tokens with resource owner credentials flow

5.1.5.2. Expiration time

Tokens should generally expire after a reasonable duration. This complements and strengthens other security measures (such as signatures) and reduces the impact of all kinds of token leaks. Depending on the risk associated with a token leakage, tokens may expire after a few minutes (e.g. for payment transactions) or stay valid for hours (e.g. read access to contacts).

The expiration time is determined by a couple of factors, including:
o risk associated to a token leakage

o duration of the underlying access grant,

o duration until the modification of an access grant should take effect, and

o time required for an attacker to guess or produce valid token.

5.1.5.3. Use short expiration time

A short expiration time for tokens is a protection means against the following threats:

o replay

o reduce impact of token leak

o reduce likelihood of successful online guessing

Note: Short token duration requires more precise clock synchronisation between authorization server and resource server. Furthermore, shorter duration may require more token refreshes (access token) or repeated end-user authorization processes (authorization code and refresh token).

5.1.5.4. Limit number of usages/ One time usage

The authorization server may restrict the number of requests or operations which can be performed with a certain token. This mechanism can be used to mitigate the following threats:

o replay of tokens

o guessing

For example, if an Authorization Server observes more than one attempt to redeem an authorization code, the Authorization Server may want to revoke all access tokens granted based on the authorization code as well as reject the current request.

As with the authorization code, access tokens may also have a limited number of operations. This forces client applications to either re-authenticate and use a refresh token to obtain a fresh access token, or it forces the client to re-authorize the access token by involving the user.
5.1.5.5. Bind tokens to a particular resource server (Audience)

Authorization servers in multi-service environments may consider issuing tokens with different content to different resource servers and to explicitly indicate in the token the target server a token is intended to be sent to. SAML Assertions (see [OASIS.saml-core-2.0-os]) use the Audience element for this purpose. This countermeasure can be used in the following situations:

- It reduces the impact of a successful replay attempt, since the token is applicable to a single resource server, only.
- It prevents abuse of a token by a rogue resource server or client, since the token can only be used on that server. It is rejected by other servers.
- It reduces the impact of a leakage of a valid token to a counterfeit resource server.

5.1.5.6. Use endpoint address as token audience

This may be used to indicate to a resource server, which endpoint URL has been used to obtain the token. This measure will allow to detect requests from a counterfeit resource server, since such token will contain the endpoint URL of that server.

5.1.5.7. Audience and Token scopes

Deployments may consider only using tokens with explicitly defined scope, where every scope is associated with a particular resource server. This approach can be used to mitigate attacks, where a resource server or client uses a token for a different then the intended purpose.

5.1.5.8. Bind token to client id

An authorization server may bind a token to a certain client identifier. This identifier should be validated for every request with that token. This means can be used, to

- detect token leakage and
- prevent token abuse.

Note: Validating the client identifier may require the target server to authenticate the client’s identifier. This authentication can be based on secrets managed independent of the token (e.g. pre-registered client id/secret on authorization server) or sent with the
token itself (e.g. as part of the encrypted token content).

5.1.5.9. Signed tokens

Self-contained tokens should be signed in order to detect any attempt
to modify or produce faked tokens (e.g. Hash-based Message
Authentication Code or digital signatures)

5.1.5.10. Encryption of token content

Self-contained tokens may be encrypted for confidentiality reasons or
to protect system internal data. Depending on token format, keys
(e.g. symmetric keys) may have to be distributed between server
nodes. The method of distribution should be defined by the token and
encryption used.

5.1.5.11. Assertion formats

For service providers intending to implement an assertion-based token
design it is highly recommended to adopt a standard assertion format
(such as SAML [OASIS.saml-core-2.0-os] or JWT
[I-D.ietf-oauth-json-web-token].

5.1.6. Access tokens

The following measures should be used to protect access tokens

- keep them in transient memory (accessible by the client
  application only)
- Pass tokens securely using secure transport (TLS)
- Ensure client applications do not share tokens with 3rd parties

5.2. Authorization Server

This section describes considerations related to the OAuth
Authorization Server end-point.

5.2.1. Authorization Codes

5.2.1.1. Automatic revocation of derived tokens if abuse is detected

If an Authorization Server observes multiple attempts to redeem an
authorization grant (e.g. such as an authorization code), the
Authorization Server may want to revoke all tokens granted based on
the authorization grant.
5.2.2. Refresh tokens

5.2.2.1. Restricted issuance of refresh tokens

The authorization server may decide based on an appropriate policy not to issue refresh tokens. Since refresh tokens are long term credentials, they may be subject theft. For example, if the authorization server does not trust a client to securely store such tokens, it may refuse to issue such a client a refresh token.

5.2.2.2. Binding of refresh token to client_id

The authorization server should match every refresh token to the identifier of the client to whom it was issued. The authorization server should check that the same client_id is present for every request to refresh the access token. If possible (e.g. confidential clients), the authorization server should authenticate the respective client.

This is a countermeasure against refresh token theft or leakage.

Note: This binding should be protected from unauthorized modifications.

5.2.2.3. Refresh Token Rotation

Refresh token rotation is intended to automatically detect and prevent attempts to use the same refresh token in parallel from different apps/devices. This happens if a token gets stolen from the client and is subsequently used by the attacker and the legitimate client. The basic idea is to change the refresh token value with every refresh request in order to detect attempts to obtain access tokens using old refresh tokens. Since the authorization server cannot determine whether the attacker or the legitimate client is trying to access, in case of such an access attempt the valid refresh token and the access authorization associated with it are both revoked.

The OAuth specification supports this measure in that the tokens response allows the authorization server to return a new refresh token even for requests with grant type "refresh_token".

Note: This measure may cause problems in clustered environments since usage of the currently valid refresh token must be ensured. In such an environment, other measures might be more appropriate.
5.2.2.4. Revoke refresh tokens

The authorization server may allow clients or end-users to explicitly request the invalidation of refresh tokens. A mechanism to revoke tokens is specified in [I-D.ietf-oauth-revocation].

This is a countermeasure against:

- device theft,
- impersonation of resource owner, or
- suspected compromised client applications.

5.2.2.5. Device identification

The authorization server may require to bind authentication credentials to a device identifier. The _International Mobile Station Equipment Identity_ [IMEI] is one example of such an identifier, there are also operating system specific identifiers. The authorization server could include such an identifier when authenticating user credentials in order to detect token theft from a particular device.

Note: Any implementation should consider potential privacy implications of using device identifiers.

5.2.2.6. X-FRAME-OPTION header

For newer browsers, avoidance of iFrames can be enforced server side by using the X-FRAME-OPTION header (see [I-D.gondrom-x-frame-options]). This header can have two values, "DENY" and "SAMEORIGIN", which will block any framing or framing by sites with a different origin, respectively. The value "ALLOW-FROM" allows iFrames for a list of trusted origins.

This is a countermeasure against the following threats:

- Clickjacking attacks

5.2.3. Client authentication and authorization

As described in Section 3 (Security Features), clients are identified, authenticated and authorized for several purposes, such as a
o Collate requests to the same client,

o Indicate to the user the client is recognized by the authorization server,

o Authorize access of clients to certain features on the authorization or resource server, and

o Log a client identifier to log files for analysis or statistics.

Due to the different capabilities and characteristics of the different client types, there are different ways to support these objectives, which will be described in this section. Authorization server providers should be aware of the security policy and deployment of a particular clients and adapt its treatment accordingly. For example, one approach could be to treat all clients as less trustworthy and unsecure. On the other extreme, a service provider could activate every client installation individually by an administrator and that way gain confidence in the identity of the software package and the security of the environment the client is installed in. And there are several approaches in between.

5.2.3.1. Don’t issue secrets to client with inappropriate security policy

Authorization servers should not issue secrets to clients that cannot protect secrets ("public" clients). This reduces probability of the server treating the client as strongly authenticated.

For example, it is of limited benefit to create a single client id and secret which is shared by all installations of a native application. Such a scenario requires that this secret must be transmitted from the developer via the respective distribution channel, e.g. an application market, to all installations of the application on end-user devices. A secret, burned into the source code of the application or a associated resource bundle, is not protected from reverse engineering. Secondly, such secrets cannot be revoked since this would immediately put all installations out of work. Moreover, since the authorization server cannot really trust the client’s identifier, it would be dangerous to indicate to end-users the trustworthiness of the client.

There are other ways to achieve a reasonable security level, as described in the following sections.
5.2.3.2. Require user consent for public clients without secret

Authorization servers should not allow automatic authorization for public clients. The authorization may issue an individual client id, but should require that all authorizations are approved by the end-user. This is a countermeasure for clients without secret against the following threats:

- Impersonation of public client applications

5.2.3.3. Client_id only in combination with redirect_uri

The authorization may issue a client_id and bind the client_id to a certain pre-configured redirect_uri. Any authorization request with another redirection URI is refused automatically. Alternatively, the authorization server should not accept any dynamic redirection URI for such a client_id and instead always redirect to the well-known pre-configured redirection URI. This is a countermeasure for clients without secrets against the following threats:

- Cross-site scripting attacks
- Impersonation of public client applications

5.2.3.4. Installation-specific client secrets

An authorization server may issue separate client identifiers and corresponding secrets to the different installations of a particular client (i.e. software package). The effect of such an approach would be to turn otherwise "public" clients back into "confidential" clients.

For web applications, this could mean to create one client_id and client_secret per web site a software package is installed on. So the provider of that particular site could request client id and secret from the authorization server during setup of the web site. This would also allow to validate some of the properties of that web site, such as redirection URI, website URL, and whatever proofs useful. The web site provider has to ensure the security of the client secret on the site.

For native applications, things are more complicated because every copy of a particular application on any device is a different installation. Installation-specific secrets in this scenario will require
1. Either to obtain a client_id and client_secret during download process from the application market, or

2. During installation on the device.

Either approach will require an automated mechanism for issuing client ids and secrets, which is currently not defined by OAuth.

The first approach would allow to achieve a certain level of trust in the authenticity of the application, whereas the second option only allows to authenticate the installation but not to validate properties of the client. But this would at least help to prevent several replay attacks. Moreover, installation-specific client_id and secret allow to selectively revoke all refresh tokens of a specific installation at once.

5.2.3.5. Validation of pre-registered redirect_uri

An authorization server should require all clients to register their redirect_uri and the redirect_uri should be the full URI as defined in [I-D.ietf-oauth-v2]. The way this registration is performed is out of scope of this document. As per the core spec, every actual redirection URI sent with the respective client_id to the end-user authorization endpoint must match the registered redirection URI. Where it does not match, the authorization server should assume the inbound GET request has been sent by an attacker and refuse it.

Note: the authorization server should not redirect the user agent back to the redirection URI of such an authorization request. Validating the pre-registered redirect_uri is a countermeasure against the following threats:

- Authorization code leakage through counterfeit web site: allows to detect attack attempts already after first redirect to end-user authorization endpoint (Section 4.4.1.7).

- Open Redirector attack via client redirection endpoint. (Section 4.1.5.)

- Open Redirector phishing attack via authorization server redirection endpoint (Section 4.2.4)

The underlying assumption of this measure is that an attacker will need to use another redirection URI in order to get access to the authorization code. Deployments might consider the possibility of an attacker using spoofing attacks to a victims device to circumvent this security measure.

Note: Pre-registering clients might not scale in some deployments.
(manual process) or require dynamic client registration (not specified yet). With the lack of dynamic client registration, pre-registered "redirect_uri" only works for clients bound to certain deployments at development/configuration time. As soon as dynamic resource server discovery is required, the pre-registered redirect_uri may be no longer feasible.

5.2.3.6. Revoke client secrets

An authorization server may revoke a client’s secret in order to prevent abuse of a revealed secret.

Note: This measure will immediately invalidate any authorization code or refresh token issued to the respective client. This might be unintentionally impact client identifiers and secrets used across multiple deployments of a particular native or web application.

This a countermeasure against:

- Abuse of revealed client secrets for private clients

5.2.3.7. Use strong client authentication (e.g. client_assertion / client_token)

By using an alternative form of authentication such as client assertion [I-D.ietf-oauth-assertions], the need to distribute a client_secret is eliminated. This may require the use of a secure private key store or other supplemental authentication system as specified by the client assertion issuer in its authentication process.

5.2.4. End-user authorization

This section involves considerations for authorization flows involving the end-user.

5.2.4.1. Automatic processing of repeated authorizations requires client validation

Authorization servers should NOT automatically process repeat authorizations where the client is not authenticated through a client secret or some other authentication mechanism such as a signed authentication assertion certificate (Section 5.2.3.7 Use strong client authentication (e.g. client_assertion / client_token)) or validation of a pre-registered redirect URI (Section 5.2.3.5 Validation of pre-registered redirection URI).
5.2.4.2. Informed decisions based on transparency

The authorization server should clearly explain to the end-user what happens in the authorization process and what the consequences are. For example, the user should understand what access he is about to grant to which client for what duration. It should also be obvious to the user, whether the server is able to reliably certify certain client properties (web site URL, security policy).

5.2.4.3. Validation of client properties by end-user

In the authorization process, the user is typically asked to approve a client’s request for authorization. This is an important security mechanism by itself because the end-user can be involved in the validation of client properties, such as whether the client name known to the authorization server fits the name of the web site or the application the end-user is using. This measure is especially helpful in situations where the authorization server is unable to authenticate the client. It is a countermeasure against:

- Malicious application
- A client application masquerading as another client

5.2.4.4. Binding of authorization code to client_id

The authorization server should bind every authorization code to the id of the respective client which initiated the end-user authorization process. This measure is a countermeasure against:

- replay of authorization codes with different client credentials since an attacker cannot use another client_id to exchange an authorization code into a token
- Online guessing of authorization codes

Note: This binding should be protected from unauthorized modifications (e.g. using protected memory and/or a secure database).

5.2.4.5. Binding of authorization code to redirect_uri

The authorization server should be able to bind every authorization code to the actual redirection URI used as redirect target of the client in the end-user authorization process. This binding should be validated when the client attempts to exchange the respective authorization code for an access token. This measure is a countermeasure against authorization code leakage through counterfeit web sites since an attacker cannot use another redirection URI to
exchange an authorization code into a token.

5.3. Client App Security

This section deals with considerations for client applications.

5.3.1. Don’t store credentials in code or resources bundled with software packages

Because of the numbers of copies of client software, there is limited benefit to create a single client id and secret which is shared by all installations of an application. Such an application by itself would be considered a "public" client as it cannot be presumed to be able to keep client secrets. A secret, burned into the source code of the application or an associated resource bundle, cannot be protected from reverse engineering. Secondly, such secrets cannot be revoked since this would immediately put all installations out of work. Moreover, since the authorization server cannot really trust the client’s identifier, it would be dangerous to indicate to end-users the trustworthiness of the client.

5.3.2. Standard web server protection measures (for config files and databases)

Use standard web server protection measures — Section 5.3.2

5.3.3. Store secrets in a secure storage

The are different way to store secrets of all kinds (tokens, client secrets) securely on a device or server.

Most multi-user operating systems segregate the personal storage of the different system users. Moreover, most modern smartphone operating systems even support to store app-specific data in separate areas of the file systems and protect it from access by other applications. Additionally, applications can implements confidential data itself using a user-supplied secret, such as PIN or password.

Another option is to swap refresh token storage to a trusted backend server. This mean in turn requires a resilient authentication mechanisms between client and backend server. Note: Applications should ensure that confidential data is kept confidential even after reading from secure storage, which typically means to keep this data in the local memory of the application.
5.3.4. Utilize device lock to prevent unauthorized device access

On a typical modern phone, there are many "device lock" options which can be utilized to provide additional protection where a device is stolen or misplaced. These include PINs, passwords and other biomtric features such as "face recognition". These are not equal in the level of security they provide.

5.3.5. Link state parameter to user agent session

The state parameter is used to link client requests and prevent CSRF attacks, for example against the redirection URI. An attacker could inject their own authorization code or access token, which can result in the client using an access token associated with the attacker’s protected resources rather than the victim’s (e.g. save the victim’s bank account information to a protected resource controlled by the attacker).

The client should utilize the "state" request parameter to send the authorization server a value that binds the request to the user-agent’s authenticated state (e.g. a hash of the session cookie used to authenticate the user-agent) when making an authorization request. Once authorization has been obtained from the end-user, the authorization server redirects the end-user’s user-agent back to the client with the required binding value contained in the "state" parameter.

The binding value enables the client to verify the validity of the request by matching the binding value to the user-agent’s authenticated state.

5.4. Resource Servers

The following section details security considerations for resource servers.

5.4.1. Authorization headers

Authorization headers are recognized and specially treated by HTTP proxies and servers. Thus the usage of such headers for sending access tokens to resource servers reduces the likelihood of leakage or unintended storage of authenticated requests in general and especially Authorization headers.

5.4.2. Authenticated requests

An authorization server may bind tokens to a certain client identifier and enable resource servers to be able to validate that
association on resource access. This will require the resource server to authenticate the originator of a request as the legitimate owner of a particular token. There are a couple of options to implement this countermeasure:

- The authorization server may associate the client identifier with the token (either internally or in the payload of an self-contained token). The client then uses client certificate-based HTTP authentication on the resource server’s endpoint to authenticate its identity and the resource server validates the name with the name referenced by the token.

- same as before, but the client uses his private key to sign the request to the resource server (public key is either contained in the token or sent along with the request)

- Alternatively, the authorization server may issue a token-bound secret, which the client uses to MAC (message authentication code) the request (see [I-D.ietf-oauth-v2-http-mac]). The resource server obtains the secret either directly from the authorization server or it is contained in an encrypted section of the token. That way the resource server does not "know" the client but is able to validate whether the authorization server issued the token to that client

Authenticated requests are a countermeasure against abuse of tokens by counterfeit resource servers.

5.4.3. Signed requests

A resource server may decide to accept signed requests only, either to replace transport level security measures or to complement such measures. Every signed request should be uniquely identifiable and should not be processed twice by the resource server. This countermeasure helps to mitigate:

- modifications of the message and

- replay attempts

5.5. A Word on User Interaction and User-Installed Apps

OAuth, as a security protocol, is distinctive in that its flow usually involves significant user interaction, making the end user a part of the security model. This creates some important difficulties in defending against some of the threats discussed above. Some of these points have already been made, but it’s worth repeating and highlighting them here.
End users must understand what they are being asked to approve (see Section 5.2.4.1). Users often do not have the expertise to understand the ramifications of saying "yes" to an authorization request, and are likely not to be able to see subtle differences in wording of requests. Malicious software can confuse the user, tricking the user into approving almost anything.

End-user devices are prone to software compromise. This has been a long-standing problem, with frequent attacks on web browsers and other parts of the user’s system. But with increasing popularity of user-installed "apps", the threat posed by compromised or malicious end-user software is very strong, and is one that is very difficult to mitigate.

Be aware that users will demand to install and run such apps, and that compromised or malicious ones can steal credentials at many points in the data flow. They can intercept the very user login credentials that OAuth is designed to protect. They can request authorization far beyond what they have led the user to understand and approve. They can automate a response on behalf of the user, hiding the whole process. No solution is offered here, because none is known; this remains in the space between better security and better usability.

Addressing these issues by restricting the use of user-installed software may be practical in some limited environments, and can be used as a countermeasure in those cases. Such restrictions are not practical in the general case, and mechanisms for after-the-fact recovery should be in place.

While end users are mostly incapable of properly vetting applications they load onto their devices, those who deploy Authorization Servers might have tools at their disposal to mitigate malicious Clients. For example, a well run Authorization Server must only assert client properties to the end-user it is effectively capable of validating, explicitly point out which properties it cannot validate, and indicate to the end-user the risk associated with granting access to the particular client.

6. IANA Considerations

This document makes no request of IANA.

Note to RFC Editor: this section may be removed on publication as an RFC.
7. Acknowledgements

We would like to thank Stephen Farrell, Barry Leiba, Hui-Lan Lu, Francisco Corella, Peifung E Lam, Shane B Weeden, Skylar Woodward, Niv Steingarten, Tim Bray, and James H. Manger for their comments and contributions.

8. References

8.1. Informative References

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

[I-D.ietf-oauth-v2-bearer]
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.

8.2. Informative References

[I-D.gondrom-x-frame-options]
Ross, D. and T. Gondrom, "HTTP Header X-Frame-Options",
draft-gondrom-x-frame-options-00 (work in progress),
March 2012.

[I-D.ietf-oauth-assertions]
Campbell, B., Mortimore, C., Jones, M., and Y. Goland,
"Assertion Framework for OAuth 2.0",
draft-ietf-oauth-assertions-06 (work in progress),
September 2012.

[I-D.ietf-oauth-json-web-token]
Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token
(JWT)", draft-ietf-oauth-json-web-token-03 (work in
progress), July 2012.

[I-D.ietf-oauth-revocation]
Lodderstedt, T., Dronia, S., and M. Scurtescu, "Token
Revocation", draft-ietf-oauth-revocation-01 (work in
progress), October 2012.

[I-D.ietf-oauth-v2-http-mac]
Hammer-Lahav, E., "HTTP Authentication: MAC Access
Authentication", draft-ietf-oauth-v2-http-mac-01 (work in
progress), February 2012.

[IMEI]  3GPP, "International Mobile station Equipment Identities (IMEI)", 3GPP TS 22.016 3.3.0, July 2002.


Appendix A.  Document History

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

draft-lodderstedt-oauth-security-01

o  section 4.4.1.2 - changed "resource server" to "client" in countermeasures description.

o  section 4.4.1.6 - changed "client shall authenticate the server" to "The browser shall be utilized to authenticate the redirection URI of the client"

o  section 5 - general review and alignment with public/confidential client terms

o  all sections - general clean-up and typo corrections

draft-ietf-oauth-v2-threatmodel-00

o  section 3.4 - added the purposes for using authorization codes.

o  extended section 4.4.1.1

o  merged 4.4.1.5 into 4.4.1.2

o  corrected some typos
o reformulated "session fixation", renamed respective sections into 
"authorization code disclosure through counterfeit client"
o added new section "User session impersonation"
o worked out or reworked sections 2.3.3, 4.4.2.4, 4.4.4, 5.1.4.1.2,
5.1.4.1.4, 5.2.3.5
o added new threat "DoS using manufactured authorization codes" as
proposed by Peifung E Lam
o added XSRF and clickjacking (incl. state parameter explanation)
o changed sub-section order in section 4.4.1
o incorporated feedback from Skylar Woodward (client secrets) and
Shane B Weeden (refresh tokens as client instance secret)
o aligned client section with core draft’s client type definition
o converted I-D into WG document
draft-ietf-oauth-v2-threatmodel-01
o Alignment of terminology with core draft 22 (private/public
client, redirect URI validation policy, replaced definition of the
client categories by reference to respective core section)
o Synchronisation with the core’s security consideration section
(UPDATE 10.12 CSRF, NEW 10.14/15)
o Added Resource Owner Impersonation
o Improved section 5
o Renamed Refresh Token Replacement to Refresh Token Rotation
draft-ietf-oauth-v2-threatmodel-02
o Incoporated Tim Bray’s review comments (e.g. removed all normative
language)
draft-ietf-oauth-v2-threatmodel-03
o removed 2119 boilerplate and normative reference
o incorporated shepherd review feedback
draft-ietf-oauth-v2-threatmodel-06
 o incorporated AD review feedback

draft-ietf-oauth-v2-threatmodel-07
 o added new section on token substitution
 o made references to core and bearer normative

Authors’ Addresses

Torsten Lodderstedt (editor)
Deutsche Telekom AG
Email: torsten@lodderstedt.net

Mark McGloin
IBM
Email: mark.mcgloin@ie.ibm.com

Phil Hunt
Oracle Corporation
Email: phil.hunt@yahoo.com