

Network Working Group
Internet-Draft
Intended status: Informational
Expires: July 29, 2016

T. Hardjono, Ed.
MIT
E. Maler
ForgeRock
M. Machulak
Cloud Identity
D. Catalano
Oracle
January 26, 2016

User-Managed Access (UMA) Profile of OAuth 2.0
draft-hardjono-oauth-umacore-14

Abstract

User-Managed Access (UMA) is a profile of OAuth 2.0. UMA defines how resource owners can control protected-resource access by clients operated by arbitrary requesting parties, where the resources reside on any number of resource servers, and where a centralized authorization server governs access based on resource owner policies.

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 29, 2016.

Copyright Notice

Copyright (c) 2016 IETF Trust and the persons identified as the document authors. All rights reserved.

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

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

Table of Contents

1. Introduction	2
2. References	3
2.1. Normative References	3
2.2. Informative References	3
Authors' Addresses	3

1. Introduction

User-Managed Access (UMA) is a profile of OAuth 2.0 [OAuth2]. UMA defines how resource owners can control protected-resource access by clients operated by arbitrary requesting parties, where the resources reside on any number of resource servers, and where a centralized authorization server governs access based on resource owner policies. Resource owners configure authorization servers with access policies that serve as asynchronous authorization grants.

UMA serves numerous use cases where a resource owner uses a dedicated service to manage authorization for access to their resources, potentially even without the run-time presence of the resource owner. A typical example is the following: a web user (an end-user resource owner) can authorize a web or native app (a client) to gain one-time or ongoing access to a protected resource containing his home address stored at a "personal data store" service (a resource server), by telling the resource server to respect access entitlements issued by his chosen cloud-based authorization service (an authorization server). The requesting party operating the client might be the resource owner, where the app is run by an e-commerce company that needs to know where to ship a purchased item, or the requesting party might be resource owner's friend who is using an online address book service to collect contact information, or the requesting party might be a survey company that uses an autonomous web service to compile population demographics. A variety of use cases can be found in [UMA-usecases] and [UMA-casestudies].

Please see for the full UMA-Core 1.0 Specification for a complete description of UMA Core.

2. References

2.1. Normative References

- [OAuth2] Hardt, D., "The OAuth 2.0 Authorization Framework", October 2012, <<http://tools.ietf.org/html/rfc6749>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<http://www.rfc-editor.org/info/rfc2119>>.
- [UMAcure] Hardjono, T., Maler, E., Machulak, M., and D. Catalano, "User-Managed Access (UMA) Profile of OAuth 2.0 Version 1.0.1", December 2015, <https://docs.kantarainitiative.org/uma/draft-uma-core-v1_0_1.html>.

2.2. Informative References

- [UMA-casestudies] Maler, E., "UMA Case Studies", April 2014, <<http://kantarainitiative.org/confluence/display/uma/Case+Studies>>.
- [UMA-usecases] Maler, E., "UMA Scenarios and Use Cases", October 2010, <<http://kantarainitiative.org/confluence/display/uma/UMA+Scenarios+and+Use+Cases>>.

Authors' Addresses

Thomas Hardjono (editor)
MIT

Email: hardjono@mit.edu

Eve Maler
ForgeRock

Email: eve.maler@forgerock.com

Maciej Machulak
Cloud Identity

Email: maciej.machulak@cloudidentity.co.uk

Domenico Catalano
Oracle

Email: domenico.catalano@oracle.com

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: May 16, 2015

B. Campbell
Ping Identity
C. Mortimore
Salesforce
M. Jones
Microsoft
November 12, 2014

SAML 2.0 Profile for OAuth 2.0 Client Authentication and Authorization
Grants
draft-ietf-oauth-saml2-bearer-23

Abstract

This specification defines the use of a Security Assertion Markup Language (SAML) 2.0 Bearer Assertion as a means for requesting an OAuth 2.0 access token as well as for use as a means of client authentication.

Status of This Memo

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

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

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

This Internet-Draft will expire on May 16, 2015.

Copyright Notice

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

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

include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
1.1. Notational Conventions	4
1.2. Terminology	4
2. HTTP Parameter Bindings for Transporting Assertions	4
2.1. Using SAML Assertions as Authorization Grants	4
2.2. Using SAML Assertions for Client Authentication	5
3. Assertion Format and Processing Requirements	6
3.1. Authorization Grant Processing	8
3.2. Client Authentication Processing	9
4. Authorization Grant Example	9
5. Interoperability Considerations	11
6. Security Considerations	11
7. Privacy Considerations	12
8. IANA Considerations	12
8.1. Sub-Namespace Registration of urn:ietf:params:oauth: :grant-type:saml2-bearer	12
8.2. Sub-Namespace Registration of urn:ietf:params:oauth: :client-assertion-type:saml2-bearer	12
9. References	13
9.1. Normative References	13
9.2. Informative References	14
Appendix A. Acknowledgements	14
Appendix B. Document History	15
Authors' Addresses	21

1. Introduction

The Security Assertion Markup Language (SAML) 2.0 [OASIS.saml-core-2.0-os] is an XML-based framework that allows identity and security information to be shared across security domains. The SAML specification, while primarily targeted at providing cross domain Web browser single sign-on, was also designed to be modular and extensible to facilitate use in other contexts.

The Assertion, an XML security token, is a fundamental construct of SAML that is often adopted for use in other protocols and specifications. (Some examples include [OASIS.WSS-SAMLTokenProfile] and [OASIS.WS-Fed].) An Assertion is generally issued by an identity provider and consumed by a service provider who relies on its content to identify the Assertion's subject for security related purposes.

The OAuth 2.0 Authorization Framework [RFC6749] provides a method for making authenticated HTTP requests to a resource using an access token. Access tokens are issued to third-party clients by an authorization server (AS) with the (sometimes implicit) approval of the resource owner. In OAuth, an authorization grant is an abstract term used to describe intermediate credentials that represent the resource owner authorization. An authorization grant is used by the client to obtain an access token. Several authorization grant types are defined to support a wide range of client types and user experiences. OAuth also allows for the definition of new extension grant types to support additional clients or to provide a bridge between OAuth and other trust frameworks. Finally, OAuth allows the definition of additional authentication mechanisms to be used by clients when interacting with the authorization server.

The Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants [I-D.ietf-oauth-assertions] specification is an abstract extension to OAuth 2.0 that provides a general framework for the use of Assertions as client credentials and/or authorization grants with OAuth 2.0. This specification profiles the Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants [I-D.ietf-oauth-assertions] specification to define an extension grant type that uses a SAML 2.0 Bearer Assertion to request an OAuth 2.0 access token as well as for use as client credentials. The format and processing rules for the SAML Assertion defined in this specification are intentionally similar, though not identical, to those in the Web Browser SSO Profile defined in the SAML Profiles [OASIS.saml-profiles-2.0-os] specification. This specification is reusing, to the extent reasonable, concepts and patterns from that well-established Profile.

This document defines how a SAML Assertion can be used to request an access token when a client wishes to utilize an existing trust relationship, expressed through the semantics of (and digital signature or keyed message digest calculated over) the SAML Assertion, without a direct user approval step at the authorization server. It also defines how a SAML Assertion can be used as a client authentication mechanism. The use of an Assertion for client authentication is orthogonal to and separable from using an Assertion as an authorization grant. They can be used either in combination or separately. Client assertion authentication is nothing more than an alternative way for a client to authenticate to the token endpoint and must be used in conjunction with some grant type to form a complete and meaningful protocol request. Assertion authorization grants may be used with or without client authentication or identification. Whether or not client authentication is needed in conjunction with an assertion authorization grant, as well as the

supported types of client authentication, are policy decisions at the discretion of the authorization server.

The process by which the client obtains the SAML Assertion, prior to exchanging it with the authorization server or using it for client authentication, is out of scope.

1.1. Notational Conventions

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

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

1.2. Terminology

All terms are as defined in The OAuth 2.0 Authorization Framework [RFC6749], the Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants [I-D.ietf-oauth-assertions], and the Security Assertion Markup Language (SAML) 2.0 [OASIS.saml-core-2.0-os] specifications.

2. HTTP Parameter Bindings for Transporting Assertions

The Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants [I-D.ietf-oauth-assertions] specification defines generic HTTP parameters for transporting Assertions during interactions with a token endpoint. This section defines specific parameters and treatments of those parameters for use with SAML 2.0 Bearer Assertions.

2.1. Using SAML Assertions as Authorization Grants

To use a SAML Bearer Assertion as an authorization grant, the client uses an access token request as defined in Section 4 of the Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants [I-D.ietf-oauth-assertions] specification with the following specific parameter values and encodings.

The value of the "grant_type" parameter is "urn:ietf:params:oauth:grant-type:saml2-bearer".

The value of the "assertion" parameter contains a single SAML 2.0 Assertion. It MUST NOT contain more than one SAML 2.0 assertion. The SAML Assertion XML data MUST be encoded using base64url, where the encoding adheres to the definition in Section 5 of RFC 4648

[RFC4648] and where the padding bits are set to zero. To avoid the need for subsequent encoding steps (by "application/x-www-form-urlencoded" [W3C.REC-html401-19991224], for example), the base64url encoded data MUST NOT be line wrapped and pad characters ("=") MUST NOT be included.

The "scope" parameter may be used, as defined in the Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants [I-D.ietf-oauth-assertions] specification, to indicate the requested scope.

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

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

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

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

2.2. Using SAML Assertions for Client Authentication

To use a SAML Bearer Assertion for client authentication, the client uses the following parameter values and encodings.

The value of the "client_assertion_type" parameter is "urn:ietf:params:oauth:client-assertion-type:saml2-bearer".

The value of the "client_assertion" parameter MUST contain a single SAML 2.0 Assertion. The SAML Assertion XML data MUST be encoded using base64url, where the encoding adheres to the definition in Section 5 of RFC 4648 [RFC4648] and where the padding bits are set to zero. To avoid the need for subsequent encoding steps (by "application/x-www-form-urlencoded" [W3C.REC-html401-19991224], for example), the base64url encoded data SHOULD NOT be line wrapped and pad characters ("=") SHOULD NOT be included.

The following example demonstrates a client authenticating using an assertion during the presentation of an authorization code grant in an Access Token Request (with extra line breaks for display purposes only):

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

grant_type=authorization_code&
code=vAZEIHjQTHuGgaSvyW9hO0RpusLzkvTOww3trZBxZpo&
client_assertion_type=urn%3Aietf%3Aparams%3Aoauth
%3Aclient-assertion-type%3Asaml2-bearer&
client_assertion=PHNhbW...[omitted for brevity]...ZT
```

3. Assertion Format and Processing Requirements

In order to issue an access token response as described in OAuth 2.0 [RFC6749] or to rely on an Assertion for client authentication, the authorization server MUST validate the Assertion according to the criteria below. Application of additional restrictions and policy are at the discretion of the authorization server.

1. The Assertion's <Issuer> element MUST contain a unique identifier for the entity that issued the Assertion. In the absence of an application profile specifying otherwise, compliant applications MUST compare Issuer values using the Simple String Comparison method defined in Section 6.2.1 of RFC 3986 [RFC3986].
2. The Assertion MUST contain a <Conditions> element with an <AudienceRestriction> element with an <Audience> element that identifies the authorization server as an intended audience. Section 2.5.1.4 of Assertions and Protocols for the OASIS Security Assertion Markup Language [OASIS.saml-core-2.0-os] defines the <AudienceRestriction> and <Audience> elements and, in addition to the URI references discussed there, the token endpoint URL of the authorization server MAY be used as a URI that identifies the authorization server as an intended audience. The Authorization Server MUST reject any assertion that does not contain its own identity as the intended audience. In the absence of an application profile specifying otherwise, compliant applications MUST compare the audience values using the Simple String Comparison method defined in Section 6.2.1 of RFC 3986 [RFC3986]. As noted in Section 5, the precise strings to be used as the audience for a given Authorization Server must be configured out-of-band by the Authorization Server and the Issuer of the assertion.
3. The Assertion MUST contain a <Subject> element identifying the principal that is the subject of the Assertion. Additional information identifying the subject/principal MAY be included in an <AttributeStatement>.

- A. For the authorization grant, the Subject typically identifies an authorized accessor for which the access token is being requested (i.e., the resource owner or an authorized delegate), but in some cases, may be a pseudonymous identifier or other value denoting an anonymous user.
 - B. For client authentication, the Subject MUST be the "client_id" of the OAuth client.
- 4. The Assertion MUST have an expiry that limits the time window during which it can be used. The expiry can be expressed either as the NotOnOrAfter attribute of the <Conditions> element or as the NotOnOrAfter attribute of a suitable <SubjectConfirmationData> element.
 - 5. The <Subject> element MUST contain at least one <SubjectConfirmation> element that has a Method attribute with a value of "urn:oasis:names:tc:SAML:2.0:cm:bearer". If the Assertion does not have a suitable NonOnOrAfter attribute on the <Conditions> element, the <SubjectConfirmation> element MUST contain a <SubjectConfirmationData> element. When present, the <SubjectConfirmationData> element MUST have a Recipient attribute with a value indicating the token endpoint URL of the authorization server (or an acceptable alias). The authorization server MUST verify that the value of the Recipient attribute matches the token endpoint URL (or an acceptable alias) to which the Assertion was delivered. The <SubjectConfirmationData> element MUST have a NotOnOrAfter attribute that limits the window during which the Assertion can be confirmed. The <SubjectConfirmationData> element MAY also contain an Address attribute limiting the client address from which the Assertion can be delivered. Verification of the Address is at the discretion of the authorization server.
 - 6. The authorization server MUST reject the entire Assertion if the NotOnOrAfter instant on the <Conditions> element has passed (subject to allowable clock skew between systems). The authorization server MUST reject the <SubjectConfirmation> (but MAY still use the rest of the Assertion) if the NotOnOrAfter instant on the <SubjectConfirmationData> has passed (subject to allowable clock skew). Note that the authorization server may reject Assertions with a NotOnOrAfter instant that is unreasonably far in the future. The authorization server MAY ensure that Bearer Assertions are not replayed, by maintaining the set of used ID values for the length of time for which the Assertion would be considered valid based on the applicable NotOnOrAfter instant.

7. If the Assertion issuer directly authenticated the subject, the Assertion SHOULD contain a single <AuthnStatement> representing that authentication event. If the Assertion was issued with the intention that the client act autonomously on behalf of the subject, an <AuthnStatement> SHOULD NOT be included and the client presenting the assertion SHOULD be identified in the <NameID> or similar element in the <SubjectConfirmation> element, or by other available means like SAML V2.0 Condition for Delegation Restriction [OASIS.saml-deleg-cs].
8. Other statements, in particular <AttributeStatement> elements, MAY be included in the Assertion.
9. The Assertion MUST be digitally signed or have a Message Authentication Code applied by the issuer. The authorization server MUST reject assertions with an invalid signature or Message Authentication Code.
10. Encrypted elements MAY appear in place of their plain text counterparts as defined in [OASIS.saml-core-2.0-os].
11. The authorization server MUST reject an Assertion that is not valid in all other respects per [OASIS.saml-core-2.0-os], such as (but not limited to) all content within the Conditions element including the NotOnOrAfter and NotBefore attributes, unknown condition types, etc.

3.1. Authorization Grant Processing

Assertion authorization grants may be used with or without client authentication or identification. Whether or not client authentication is needed in conjunction with an assertion authorization grant, as well as the supported types of client authentication, are policy decisions at the discretion of the authorization server. However, if client credentials are present in the request, the authorization server MUST validate them.

If the Assertion is not valid (including if its subject confirmation requirements cannot be met), the authorization server constructs an error response as defined in OAuth 2.0 [RFC6749]. The value of the "error" parameter MUST be the "invalid_grant" error code. The authorization server MAY include additional information regarding the reasons the Assertion was considered invalid using the "error_description" or "error_uri" parameters.

For example:

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

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

3.2. Client Authentication Processing

If the client Assertion is not valid (including if its subject confirmation requirements cannot be met), the authorization server constructs an error response as defined in OAuth 2.0 [RFC6749]. The value of the "error" parameter MUST be the "invalid_client" error code. The authorization server MAY include additional information regarding the reasons the Assertion was considered invalid using the "error_description" or "error_uri" parameters.

4. Authorization Grant Example

The following examples illustrate what a conforming Assertion and an access token request would look like.

The example shows an assertion issued and signed by the SAML Identity Provider identified as "https://saml-idp.example.com". The subject of the assertion is identified by email address as "brian@example.com", who authenticated to the Identity Provider by means of a digital signature where the key was validated as part of an X.509 Public Key Infrastructure. The intended audience of the assertion is "https://saml-sp.example.net", which is an identifier for a SAML Service Provider with which the authorization server identifies itself. The assertion is sent as part of an access token request to the authorization server's token endpoint at "https://authz.example.net/token.oauth2".

Below is an example SAML 2.0 Assertion (whitespace formatting is for display purposes only):

```
<Assertion IssueInstant="2010-10-01T20:07:34.619Z"
  ID="eflxsBZxPV2oqjd7HTLRLIBlBb7"
  Version="2.0"
  xmlns="urn:oasis:names:tc:SAML:2.0:assertion">
  <Issuer>https://saml-idp.example.com</Issuer>
  <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    [...omitted for brevity...]
  </ds:Signature>
  <Subject>
    <NameID
      Format="urn:oasis:names:tc:SAML:1.1:nameid-format:emailAddress">
      brian@example.com
    </NameID>
    <SubjectConfirmation
      Method="urn:oasis:names:tc:SAML:2.0:cm:bearer">
      <SubjectConfirmationData
        NotOnOrAfter="2010-10-01T20:12:34.619Z"
        Recipient="https://authz.example.net/token.oauth2"/>
      </SubjectConfirmation>
    </Subject>
    <Conditions>
      <AudienceRestriction>
        <Audience>https://saml-sp.example.net</Audience>
      </AudienceRestriction>
    </Conditions>
    <AuthnStatement AuthnInstant="2010-10-01T20:07:34.371Z">
      <AuthnContext>
        <AuthnContextClassRef>
          urn:oasis:names:tc:SAML:2.0:ac:classes:X509
        </AuthnContextClassRef>
      </AuthnContext>
    </AuthnStatement>
  </Assertion>
```

Figure 1: Example SAML 2.0 Assertion

To present the Assertion shown in the previous example as part of an access token request, for example, the client might make the following HTTPS request (with extra line breaks for display purposes only):

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

grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Asaml2-
bearer&assertion=PEFzc2VydGlvbiBJc3NlZULuc3RhbnQ9IjIwMTEtMDU
[...omitted for brevity...]aG5TdGF0ZW1lbnQ-PC9Bc3NlcnRpb24-
```

Figure 2: Example Request

5. Interoperability Considerations

Agreement between system entities regarding identifiers, keys, and endpoints is required in order to achieve interoperable deployments of this profile. Specific items that require agreement are as follows: values for the issuer and audience identifiers, the location of the token endpoint, the key used to apply and verify the digital signature over the assertion, one-time use restrictions on assertions, maximum assertion lifetime allowed, and the specific subject and attribute requirements of the assertion. The exchange of such information is explicitly out of scope for this specification and typical deployment of it will be done alongside existing SAML Web SSO deployments that have already established a means of exchanging such information. Metadata for the OASIS Security Assertion Markup Language (SAML) V2.0 [OASIS.saml-metadata-2.0-os] is one common method of exchanging SAML related information about system entities.

The RSA-SHA256 algorithm, from [RFC6931], is a mandatory to implement XML signature algorithm for this profile.

6. Security Considerations

The security considerations described within the Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants [I-D.ietf-oauth-assertions], The OAuth 2.0 Authorization Framework [RFC6749], and the Security and Privacy Considerations for the OASIS Security Assertion Markup Language (SAML) V2.0 [OASIS.saml-sec-consider-2.0-os] specifications are all applicable to this document.

The specification does not mandate replay protection for the SAML assertion usage for either the authorization grant or for client

authentication. It is an optional feature, which implementations may employ at their own discretion.

7. Privacy Considerations

A SAML Assertion may contain privacy-sensitive information and, to prevent disclosure of such information to unintended parties, should only be transmitted over encrypted channels, such as TLS. In cases where it is desirable to prevent disclosure of certain information to the client, the Subject and/or individual attributes of a SAML Assertion should be encrypted to the authorization server.

Deployments should determine the minimum amount of information necessary to complete the exchange and include only that information in an Assertion (typically by limiting what information is included in an <AttributeStatement> or omitting it altogether). In some cases, the Subject can be a value representing an anonymous or pseudonymous user, as described in Section 6.3.1 of the Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants [I-D.ietf-oauth-assertions].

8. IANA Considerations

8.1. Sub-Namespace Registration of urn:ietf:params:oauth:grant-type:saml2-bearer

This is a request to IANA to please register the value "grant-type:saml2-bearer" in the registry urn:ietf:params:oauth established in An IETF URN Sub-Namespace for OAuth [RFC6755].

- o URN: urn:ietf:params:oauth:grant-type:saml2-bearer
- o Common Name: SAML 2.0 Bearer Assertion Grant Type Profile for OAuth 2.0
- o Change controller: IESG
- o Specification Document: [[this document]]

8.2. Sub-Namespace Registration of urn:ietf:params:oauth:client-assertion-type:saml2-bearer

This is a request to IANA to please register the value "client-assertion-type:saml2-bearer" in the registry urn:ietf:params:oauth established in An IETF URN Sub-Namespace for OAuth [RFC6755].

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

- o Common Name: SAML 2.0 Bearer Assertion Profile for OAuth 2.0 Client Authentication
- o Change controller: IESG
- o Specification Document: [[this document]]

9. References

9.1. Normative References

[I-D.ietf-oauth-assertions]

Campbell, B., Mortimore, C., Jones, M., and Y. Goland, "Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants", draft-ietf-oauth-assertions (work in progress), October 2014.

[OASIS.saml-core-2.0-os]

Cantor, S., Kemp, J., Philpott, R., and E. Maler, "Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V2.0", OASIS Standard saml-core-2.0-os, March 2005, <<http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf>>.

[OASIS.saml-deleg-cs]

Cantor, S., Ed., "SAML V2.0 Condition for Delegation Restriction", Nov 2009.

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

Hirsch, F., Philpott, R., and E. Maler, "Security and Privacy Considerations for the OASIS Security Markup Language (SAML) V2.0", OASIS Standard saml-sec-consider-2.0-os, March 2005, <<http://docs.oasis-open.org/security/saml/v2.0/saml-sec-consider-2.0-os.pdf>>.

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

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

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.

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

- [RFC6931] Eastlake, D., "Additional XML Security Uniform Resource Identifiers (URIs)", RFC 6931, April 2013.

9.2. Informative References

- [OASIS.WS-Fed]
Goodner, M. and T. Nadalin, "Web Services Federation Language (WS-Federation) Version 1.2", May 2009, <<http://docs.oasis-open.org/wsfed/federation/v1.2/os/ws-federation-1.2-spec-os.html>>.
- [OASIS.WSS-SAMLTOKENProfile]
Monzillo, R., Kaler, C., Nadalin, T., Hallam-Baker, P., and C. Milono, "Web Services Security SAML Token Profile Version 1.1.1", May 2012, <<http://docs.oasis-open.org/wss-m/wss/v1.1.1/wss-SAMLTOKENProfile-v1.1.1.html>>.
- [OASIS.saml-metadata-2.0-os]
Cantor, S., Moreh, J., Philpott, R., and E. Maler, "Metadata for the Security Assertion Markup Language (SAML) V2.0", OASIS Standard saml-metadata-2.0-os, March 2005, <<http://docs.oasis-open.org/security/saml/v2.0/saml-metadata-2.0-os.pdf>>.
- [OASIS.saml-profiles-2.0-os]
Hughes, J., Cantor, S., Hodges, J., Hirsch, F., Mishra, P., Philpott, R., and E. Maler, "Profiles for the OASIS Security Assertion Markup Language (SAML) V2.0", OASIS Standard OASIS.saml-profiles-2.0-os, March 2005, <<http://docs.oasis-open.org/security/saml/v2.0/saml-profiles-2.0-os.pdf>>.
- [RFC6755] Campbell, B. and H. Tschofenig, "An IETF URN Sub-Namespace for OAuth", RFC 6755, October 2012.
- [W3C.REC-html401-19991224]
Raggett, D., Hors, A., and I. Jacobs, "HTML 4.01 Specification", World Wide Web Consortium Recommendation REC-html401-19991224, December 1999, <<http://www.w3.org/TR/1999/REC-html401-19991224>>.

Appendix A. Acknowledgements

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

Appendix B. Document History

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

draft-ietf-oauth-saml2-bearer-23

- o Fix typo per <http://www.ietf.org/mail-archive/web/oauth/current/msg13790.html>

draft-ietf-oauth-saml2-bearer-22

- o Changes/suggestions from IESG reviews.

draft-ietf-oauth-saml2-bearer-21

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

draft-ietf-oauth-saml2-bearer-20

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

draft-ietf-oauth-saml2-bearer-19

- o Updated references.

draft-ietf-oauth-saml2-bearer-18

- o Clean up language around subject per <http://www.ietf.org/mail-archive/web/oauth/current/msg12254.html>.
- o As suggested in <http://www.ietf.org/mail-archive/web/oauth/current/msg12253.html> stated that "In the absence of an application profile specifying otherwise, compliant applications MUST compare the audience/issuer values using the Simple String Comparison method defined in Section 6.2.1 of RFC 3986."
- o Clarify the potentially confusing language about the AS confirming the assertion <http://www.ietf.org/mail-archive/web/oauth/current/msg12255.html>.

- o Combine the two items about AuthnStatement and drop the word presenter as discussed in <http://www.ietf.org/mail-archive/web/oauth/current/msg12257.html>.
- o Added one-time use, maximum lifetime, and specific subject and attribute requirements to Interoperability Considerations based on <http://www.ietf.org/mail-archive/web/oauth/current/msg12252.html>.
- o Reword security considerations and mention that replay protection is not mandated based on <http://www.ietf.org/mail-archive/web/oauth/current/msg12259.html>.

draft-ietf-oauth-saml2-bearer-17

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

draft-ietf-oauth-saml2-bearer-16

- o Changed title from "SAML 2.0 Bearer Assertion Profiles for OAuth 2.0" to "SAML 2.0 Profile for OAuth 2.0 Client Authentication and Authorization Grants" to be more explicit about the scope of the document per <http://www.ietf.org/mail-archive/web/oauth/current/msg11063.html>.
- o Fixed typo in text identifying the presenter from "or similar element, the" to "or similar element in the".
- o Numbered the list of processing rules.
- o Smallish editorial cleanups to try and improve readability and comprehensibility.
- o Cleaner split out of the processing rules in cases where they differ for client authentication and authorization grants.
- o Clarified the parameters that are used/available for authorization grants.
- o Added Interoperability Considerations section and info reference to SAML Metadata.
- o Added more explanatory context to the example in Section 4.

draft-ietf-oauth-saml2-bearer-15

- o Reference RFC 6749 and RFC 6755.

- o Update draft-ietf-oauth-assertions reference to -06.
- o Remove extraneous word per <http://www.ietf.org/mail-archive/web/oauth/current/msg10055.html>

draft-ietf-oauth-saml2-bearer-14

- o Add more text to intro explaining that an assertion grant type can be used with or without client authentication/identification and that client assertion authentication is nothing more than an alternative way for a client to authenticate to the token endpoint
- o Add examples to Sections 2.1 and 2.2
- o Update references

draft-ietf-oauth-saml2-bearer-13

- o Update references: oauth-assertions-04, oauth-urn-sub-ns-05, oauth-28
- o Changed "Description" to "Specification Document" in both registration requests in IANA Considerations per changes to the template in ietf-oauth-urn-sub-ns(-03)
- o Added "(or an acceptable alias)" so that it's in both sentences about Recipient and the token endpoint URL so there's no ambiguity
- o Update area and workgroup (now Security and OAuth was Internet and nothing)

draft-ietf-oauth-saml2-bearer-12

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

draft-ietf-oauth-saml2-bearer-11

- o Removed text about limited lifetime access tokens and the SHOULD NOT on issuing refresh tokens. The text was moved to draft-ietf-oauth-assertions-02 and somewhat modified per <http://www.ietf.org/mail-archive/web/oauth/current/msg08298.html>.
- o Fixed typo/missing word per <http://www.ietf.org/mail-archive/web/oauth/current/msg08733.html>.
- o Added Terminology section.

draft-ietf-oauth-saml2-bearer-10

- o fix a spelling mistake

draft-ietf-oauth-saml2-bearer-09

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

draft-ietf-oauth-saml2-bearer-08

- o fix some typos

draft-ietf-oauth-saml2-bearer-07

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

draft-ietf-oauth-saml2-bearer-06

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

draft-ietf-oauth-saml2-bearer-05

- o Allow for subject confirmation data to be optional when Conditions contain audience and NotOnOrAfter
- o Rework most of the spec to profile draft-ietf-oauth-assertions for both authn and authz including (but not limited to):
 - * remove requirement for issuer to be urn:oasis:names:tc:SAML:2.0:nameid-format:entity
 - * change wording on Subject requirements

- o using a MAY, explicitly say that the Audience can be token endpoint URL of the authorization server
- o Change title to be more generic (allowing for client authn too)
- o added client authentication to the abstract
- o register and use urn:ietf:params:oauth:grant-type:saml2-bearer for grant type rather than http://oauth.net/grant_type/saml/2.0/bearer
- o register urn:ietf:params:oauth:client-assertion-type:saml2-bearer
- o remove scope parameter as it is defined in <http://tools.ietf.org/html/draft-ietf-oauth-assertions>
- o remove assertion param registration because it [should] be in <http://tools.ietf.org/html/draft-ietf-oauth-assertions>
- o fix typo(s) and update/add references

draft-ietf-oauth-saml2-bearer-04

- o Changed the grant_type URI from "http://oauth.net/grant_type/assertion/saml/2.0/bearer" to "http://oauth.net/grant_type/saml/2.0/bearer" - dropping the word assertion from the path. Recent versions of draft-ietf-oauth-v2 no longer refer to extension grants using the word assertion so this URI is more reflective of that. It also more closely aligns with the grant type URI in draft-jones-oauth-jwt-bearer-00 which is "http://oauth.net/grant_type/jwt/1.0/bearer".
- o Added "case sensitive" to scope definition to align with draft-ietf-oauth-v2-15/16.
- o Updated to reference draft-ietf-oauth-v2-16

draft-ietf-oauth-saml2-bearer-03

- o Cleanup of some editorial issues.

draft-ietf-oauth-saml2-bearer-02

- o Added scope parameter with text copied from draft-ietf-oauth-v2-12 (the reorg of draft-ietf-oauth-v2-12 made it so scope wasn't really inherited by this spec anymore)

- o Change definition of the assertion parameter to be more generally applicable per the suggestion near the end of <http://www.ietf.org/mail-archive/web/oauth/current/msg05253.html>

- o Editorial changes based on feedback

draft-ietf-oauth-saml2-bearer-01

- o Update spec name when referencing draft-ietf-oauth-v2 (The OAuth 2.0 Protocol Framework -> The OAuth 2.0 Authorization Protocol)
- o Update wording in Introduction to talk about extension grant types rather than the assertion grant type which is a term no longer used in OAuth 2.0
- o Updated to reference draft-ietf-oauth-v2-12 and denote as work in progress
- o Update Parameter Registration Request to use similar terms as draft-ietf-oauth-v2-12 and remove Related information part
- o Add some text giving discretion to AS on rejecting assertions with unreasonably long validity window.

draft-ietf-oauth-saml2-bearer-00

- o Added Parameter Registration Request for "assertion" to IANA Considerations.
- o Changed document name to draft-ietf-oauth-saml2-bearer in anticipation of becoming an OAUTH WG item.
- o Attempt to move the entire definition of the 'assertion' parameter into this draft (it will no longer be defined in OAuth 2 Protocol Framework).

draft-campbell-oauth-saml-01

- o Updated to reference draft-ietf-oauth-v2-11 and reflect changes from -10 to -11.
- o Updated examples.
- o Relaxed processing rules to allow for more than one SubjectConfirmation element.
- o Removed the 'MUST NOT contain a NotBefore attribute' on SubjectConfirmationData.

- o Relaxed wording that ties the subject of the Assertion to the resource owner.
- o Added some wording about identifying the client when the subject hasn't directly authenticated including an informative reference to SAML V2.0 Condition for Delegation Restriction.
- o Added a few examples to the language about verifying that the Assertion is valid in all other respects.
- o Added some wording to the introduction about the similarities to Web SSO in the format and processing rules
- o Changed the grant_type (was assertion_type) URI from http://oauth.net/assertion_type/saml/2.0/bearer to http://oauth.net/grant_type/assertion/saml/2.0/bearer
- o Changed title to include "Grant Type" in it.
- o Editorial updates based on feedback from the WG and others (including capitalization of Assertion when referring to SAML).

draft-campbell-oauth-saml-00

- o Initial I-D

Authors' Addresses

Brian Campbell
Ping Identity

Email: brian.d.campbell@gmail.com

Chuck Mortimore
Salesforce.com

Email: cmortimore@salesforce.com

Michael B. Jones
Microsoft

Email: mbj@microsoft.com

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

OAuth Working Group
Internet-Draft
Obsoletes: 5849 (if approved)
Intended status: Standards Track
Expires: February 1, 2013

D. Hardt, Ed.
Microsoft
July 31, 2012

The OAuth 2.0 Authorization Framework
draft-ietf-oauth-v2-31

Abstract

The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf. This specification replaces and obsoletes the OAuth 1.0 protocol described in RFC 5849.

Status of this Memo

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

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

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

This Internet-Draft will expire on February 1, 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.

Table of Contents

1.	Introduction	5
1.1.	Roles	6
1.2.	Protocol Flow	7
1.3.	Authorization Grant	8
1.3.1.	Authorization Code	8
1.3.2.	Implicit	8
1.3.3.	Resource Owner Password Credentials	9
1.3.4.	Client Credentials	9
1.4.	Access Token	9
1.5.	Refresh Token	10
1.6.	TLS Version	12
1.7.	HTTP Redirections	12
1.8.	Interoperability	12
1.9.	Notational Conventions	12
2.	Client Registration	13
2.1.	Client Types	13
2.2.	Client Identifier	15
2.3.	Client Authentication	15
2.3.1.	Client Password	15
2.3.2.	Other Authentication Methods	17
2.4.	Unregistered Clients	17
3.	Protocol Endpoints	17
3.1.	Authorization Endpoint	17
3.1.1.	Response Type	18
3.1.2.	Redirection Endpoint	18
3.2.	Token Endpoint	21
3.2.1.	Client Authentication	21
3.3.	Access Token Scope	22
4.	Obtaining Authorization	22
4.1.	Authorization Code Grant	23
4.1.1.	Authorization Request	24
4.1.2.	Authorization Response	25
4.1.3.	Access Token Request	27
4.1.4.	Access Token Response	28
4.2.	Implicit Grant	29
4.2.1.	Authorization Request	31
4.2.2.	Access Token Response	32
4.3.	Resource Owner Password Credentials Grant	35
4.3.1.	Authorization Request and Response	36
4.3.2.	Access Token Request	36
4.3.3.	Access Token Response	37

4.4.	Client Credentials Grant	37
4.4.1.	Authorization Request and Response	38
4.4.2.	Access Token Request	38
4.4.3.	Access Token Response	39
4.5.	Extension Grants	39
5.	Issuing an Access Token	40
5.1.	Successful Response	40
5.2.	Error Response	41
6.	Refreshing an Access Token	43
7.	Accessing Protected Resources	44
7.1.	Access Token Types	44
7.2.	Error Response	45
8.	Extensibility	46
8.1.	Defining Access Token Types	46
8.2.	Defining New Endpoint Parameters	46
8.3.	Defining New Authorization Grant Types	47
8.4.	Defining New Authorization Endpoint Response Types	47
8.5.	Defining Additional Error Codes	47
9.	Native Applications	48
10.	Security Considerations	49
10.1.	Client Authentication	49
10.2.	Client Impersonation	50
10.3.	Access Tokens	50
10.4.	Refresh Tokens	51
10.5.	Authorization Codes	51
10.6.	Authorization Code Redirection URI Manipulation	52
10.7.	Resource Owner Password Credentials	53
10.8.	Request Confidentiality	53
10.9.	Endpoints Authenticity	53
10.10.	Credentials Guessing Attacks	54
10.11.	Phishing Attacks	54
10.12.	Cross-Site Request Forgery	54
10.13.	Clickjacking	55
10.14.	Code Injection and Input Validation	56
10.15.	Open Redirectors	56
10.16.	Misuse of Access Token to Impersonate Resource Owner in Implicit Flow	56
11.	IANA Considerations	57
11.1.	OAuth Access Token Type Registry	57
11.1.1.	Registration Template	58
11.2.	OAuth Parameters Registry	58
11.2.1.	Registration Template	59
11.2.2.	Initial Registry Contents	59
11.3.	OAuth Authorization Endpoint Response Type Registry	61
11.3.1.	Registration Template	62
11.3.2.	Initial Registry Contents	62
11.4.	OAuth Extensions Error Registry	62
11.4.1.	Registration Template	63

12. References	64
12.1. Normative References	64
12.2. Informative References	65
Appendix A. Augmented Backus-Naur Form (ABNF) Syntax	66
A.1. "client_id" Syntax	66
A.2. "client_secret" Syntax	66
A.3. "response_type" Syntax	66
A.4. "scope" Syntax	67
A.5. "state" Syntax	67
A.6. "redirect_uri" Syntax	67
A.7. "error" Syntax	67
A.8. "error_description" Syntax	67
A.9. "error_uri" Syntax	67
A.10. "grant_type" Syntax	68
A.11. "code" Syntax	68
A.12. "access_token" Syntax	68
A.13. "token_type" Syntax	68
A.14. "expires_in" Syntax	68
A.15. "username" Syntax	68
A.16. "password" Syntax	69
A.17. "refresh_token" Syntax	69
A.18. Endpoint Parameter Syntax	69
Appendix B. Use of application/x-www-form-urlencoded Media Type	69
Appendix C. Acknowledgements	70
Appendix D. Document History	71
Author's Address	72

1. Introduction

In the traditional client-server authentication model, the client requests an access restricted resource (protected resource) on the server by authenticating with the server using the resource owner's credentials. In order to provide third-party applications access to restricted resources, the resource owner shares its credentials with the third-party. This creates several problems and limitations:

- o Third-party applications are required to store the resource owner's credentials for future use, typically a password in clear-text.
- o Servers are required to support password authentication, despite the security weaknesses inherent in passwords.
- o Third-party applications gain overly broad access to the resource owner's protected resources, leaving resource owners without any ability to restrict duration or access to a limited subset of resources.
- o Resource owners cannot revoke access to an individual third-party without revoking access to all third-parties, and must do so by changing their password.
- o Compromise of any third-party application results in compromise of the end-user's password and all of the data protected by that password.

OAuth addresses these issues by introducing an authorization layer and separating the role of the client from that of the resource owner. In OAuth, the client requests access to resources controlled by the resource owner and hosted by the resource server, and is issued a different set of credentials than those of the resource owner.

Instead of using the resource owner's credentials to access protected resources, the client obtains an access token - a string denoting a specific scope, lifetime, and other access attributes. Access tokens are issued to third-party clients by an authorization server with the approval of the resource owner. The client uses the access token to access the protected resources hosted by the resource server.

For example, an end-user (resource owner) can grant a printing service (client) access to her protected photos stored at a photo sharing service (resource server), without sharing her username and password with the printing service. Instead, she authenticates directly with a server trusted by the photo sharing service (authorization server), which issues the printing service delegation-specific credentials (access token).

This specification is designed for use with HTTP ([RFC2616]). The

use of OAuth over any other protocol than HTTP is out of scope.

The OAuth 1.0 protocol ([RFC5849]), published as an informational document, was the result of a small ad-hoc community effort. This standards-track specification builds on the OAuth 1.0 deployment experience, as well as additional use cases and extensibility requirements gathered from the wider IETF community. The OAuth 2.0 protocol is not backward compatible with OAuth 1.0. The two versions may co-exist on the network and implementations may choose to support both. However, it is the intention of this specification that new implementation support OAuth 2.0 as specified in this document, and that OAuth 1.0 is used only to support existing deployments. The OAuth 2.0 protocol shares very few implementation details with the OAuth 1.0 protocol. Implementers familiar with OAuth 1.0 should approach this document without any assumptions as to its structure and details.

1.1. Roles

OAuth defines four roles:

resource owner

An entity capable of granting access to a protected resource. When the resource owner is a person, it is referred to as an end-user.

resource server

The server hosting the protected resources, capable of accepting and responding to protected resource requests using access tokens.

client

An application making protected resource requests on behalf of the resource owner and with its authorization. The term client does not imply any particular implementation characteristics (e.g. whether the application executes on a server, a desktop, or other devices).

authorization server

The server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization.

The interaction between the authorization server and resource server is beyond the scope of this specification. The authorization server may be the same server as the resource server or a separate entity. A single authorization server may issue access tokens accepted by multiple resource servers.

1.2. Protocol Flow

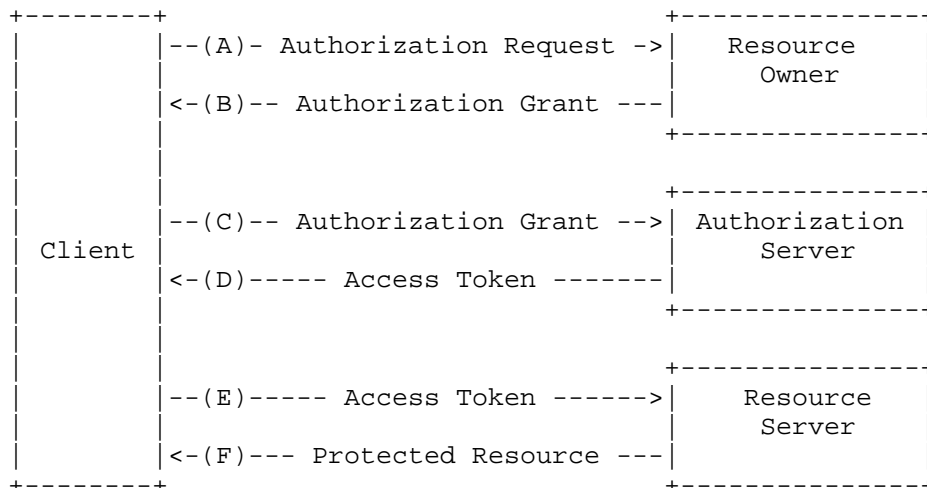


Figure 1: Abstract Protocol Flow

The abstract OAuth 2.0 flow illustrated in Figure 1 describes the interaction between the four roles and includes the following steps:

- (A) The client requests authorization from the resource owner. The authorization request can be made directly to the resource owner (as shown), or preferably indirectly via the authorization server as an intermediary.
- (B) The client receives an authorization grant, which is a credential representing the resource owner's authorization, expressed using one of four grant types defined in this specification or using an extension grant type. The authorization grant type depends on the method used by the client to request authorization and the types supported by the authorization server.
- (C) The client requests an access token by authenticating with the authorization server and presenting the authorization grant.
- (D) The authorization server authenticates the client and validates the authorization grant, and if valid issues an access token.
- (E) The client requests the protected resource from the resource server and authenticates by presenting the access token.
- (F) The resource server validates the access token, and if valid, serves the request.

The preferred method for the client to obtain an authorization grant from the resource owner (depicted in steps (A) and (B)) is to use the authorization server as an intermediary, which is illustrated in

Figure 3.

1.3. Authorization Grant

An authorization grant is a credential representing the resource owner's authorization (to access its protected resources) used by the client to obtain an access token. This specification defines four grant types: authorization code, implicit, resource owner password credentials, and client credentials, as well as an extensibility mechanism for defining additional types.

1.3.1. Authorization Code

The authorization code is obtained by using an authorization server as an intermediary between the client and resource owner. Instead of requesting authorization directly from the resource owner, the client directs the resource owner to an authorization server (via its user-agent as defined in [RFC2616]), which in turn directs the resource owner back to the client with the authorization code.

Before directing the resource owner back to the client with the authorization code, the authorization server authenticates the resource owner and obtains authorization. Because the resource owner only authenticates with the authorization server, the resource owner's credentials are never shared with the client.

The authorization code provides a few important security benefits such as the ability to authenticate the client, and the transmission of the access token directly to the client without passing it through the resource owner's user-agent, potentially exposing it to others, including the resource owner.

1.3.2. Implicit

The implicit grant is a simplified authorization code flow optimized for clients implemented in a browser using a scripting language such as JavaScript. In the implicit flow, instead of issuing the client an authorization code, the client is issued an access token directly (as the result of the resource owner authorization). The grant type is implicit as no intermediate credentials (such as an authorization code) are issued (and later used to obtain an access token).

When issuing an access token during the implicit grant flow, the authorization server does not authenticate the client. In some cases, the client identity can be verified via the redirection URI used to deliver the access token to the client. The access token may be exposed to the resource owner or other applications with access to the resource owner's user-agent.

Implicit grants improve the responsiveness and efficiency of some clients (such as a client implemented as an in-browser application) since it reduces the number of round trips required to obtain an access token. However, this convenience should be weighed against the security implications of using implicit grants, such as those described in Section 10.3 and Section 10.16, especially when the authorization code grant type is available.

1.3.3. Resource Owner Password Credentials

The resource owner password credentials (i.e. username and password) can be used directly as an authorization grant to obtain an access token. The credentials should only be used when there is a high degree of trust between the resource owner and the client (e.g. the client is part of the device operating system or a highly privileged application), and when other authorization grant types are not available (such as an authorization code).

Even though this grant type requires direct client access to the resource owner credentials, the resource owner credentials are used for a single request and are exchanged for an access token. This grant type can eliminate the need for the client to store the resource owner credentials for future use, by exchanging the credentials with a long-lived access token or refresh token.

1.3.4. Client Credentials

The client credentials (or other forms of client authentication) can be used as an authorization grant when the authorization scope is limited to the protected resources under the control of the client, or to protected resources previously arranged with the authorization server. Client credentials are used as an authorization grant typically when the client is acting on its own behalf (the client is also the resource owner), or is requesting access to protected resources based on an authorization previously arranged with the authorization server.

1.4. Access Token

Access tokens are credentials used to access protected resources. An access token is a string representing an authorization issued to the client. The string is usually opaque to the client. Tokens represent specific scopes and durations of access, granted by the resource owner, and enforced by the resource server and authorization server.

The token may denote an identifier used to retrieve the authorization information, or self-contain the authorization information in a

verifiable manner (i.e. a token string consisting of some data and a signature). Additional authentication credentials, which are beyond the scope of this specification, may be required in order for the client to use a token.

The access token provides an abstraction layer, replacing different authorization constructs (e.g. username and password) with a single token understood by the resource server. This abstraction enables issuing access tokens more restrictive than the authorization grant used to obtain them, as well as removing the resource server's need to understand a wide range of authentication methods.

Access tokens can have different formats, structures, and methods of utilization (e.g. cryptographic properties) based on the resource server security requirements. Access token attributes and the methods used to access protected resources are beyond the scope of this specification and are defined by companion specifications.

1.5. Refresh Token

Refresh tokens are credentials used to obtain access tokens. Refresh tokens are issued to the client by the authorization server and are used to obtain a new access token when the current access token becomes invalid or expires, or to obtain additional access tokens with identical or narrower scope (access tokens may have a shorter lifetime and fewer permissions than authorized by the resource owner). Issuing a refresh token is optional at the discretion of the authorization server. If the authorization server issues a refresh token, it is included when issuing an access token (i.e. step (D) in Figure 1).

A refresh token is a string representing the authorization granted to the client by the resource owner. The string is usually opaque to the client. The token denotes an identifier used to retrieve the authorization information. Unlike access tokens, refresh tokens are intended for use only with authorization servers and are never sent to resource servers.

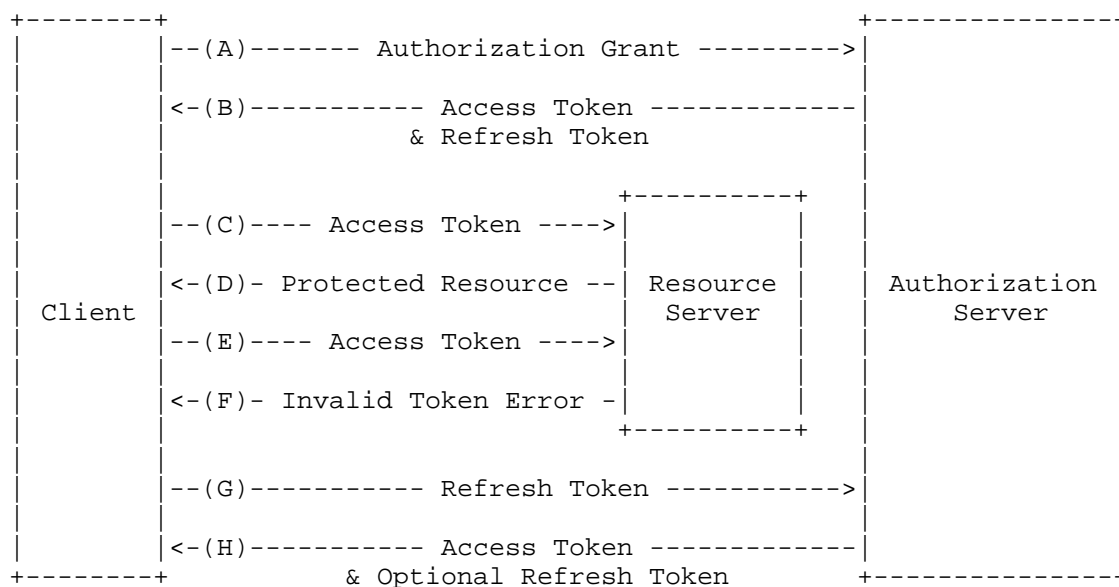


Figure 2: Refreshing an Expired Access Token

The flow illustrated in Figure 2 includes the following steps:

- (A) The client requests an access token by authenticating with the authorization server, and presenting an authorization grant.
- (B) The authorization server authenticates the client and validates the authorization grant, and if valid issues an access token and a refresh token.
- (C) The client makes a protected resource request to the resource server by presenting the access token.
- (D) The resource server validates the access token, and if valid, serves the request.
- (E) Steps (C) and (D) repeat until the access token expires. If the client knows the access token expired, it skips to step (G), otherwise it makes another protected resource request.
- (F) Since the access token is invalid, the resource server returns an invalid token error.
- (G) The client requests a new access token by authenticating with the authorization server and presenting the refresh token. The client authentication requirements are based on the client type and on the authorization server policies.
- (H) The authorization server authenticates the client and validates the refresh token, and if valid issues a new access token (and optionally, a new refresh token).

Steps C, D, E, and F are outside the scope of this specification as

described in Section 7.

1.6. TLS Version

Whenever TLS is used by this specification, the appropriate version (or versions) of TLS will vary over time, based on the widespread deployment and known security vulnerabilities. At the time of this writing, TLS version 1.2 [RFC5246] is the most recent version, but has a very limited deployment base and might not be readily available for implementation. TLS version 1.0 [RFC2246] is the most widely deployed version, and will provide the broadest interoperability.

Implementations MAY also support additional transport-layer security mechanisms that meet their security requirements.

1.7. HTTP Redirections

This specification makes extensive use of HTTP redirections, in which the client or the authorization server directs the resource owner's user-agent to another destination. While the examples in this specification show the use of the HTTP 302 status code, any other method available via the user-agent to accomplish this redirection is allowed and is considered to be an implementation detail.

1.8. Interoperability

OAuth 2.0 provides a rich authorization framework with well-defined security properties. However, as a rich and highly extensible framework with many optional components, on its own, this specification is likely to produce a wide range of non-interoperable implementations.

In addition, this specification leaves a few required components partially or fully undefined (e.g. client registration, authorization server capabilities, endpoint discovery). Without these components, clients must be manually and specifically configured against a specific authorization server and resource server in order to interoperate.

This framework was designed with the clear expectation that future work will define prescriptive profiles and extensions necessary to achieve full web-scale interoperability.

1.9. Notational Conventions

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 [RFC5234]. Additionally, the rule URI-Reference is included from Uniform Resource Identifier (URI) [RFC3986].

Certain security-related terms are to be understood in the sense defined in [RFC4949]. These terms include, but are not limited to, "attack", "authentication", "authorization", "certificate", "confidentiality", "credential", "encryption", "identity", "sign", "signature", "trust", "validate", and "verify".

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

2. Client Registration

Before initiating the protocol, the client registers with the authorization server. The means through which the client registers with the authorization server are beyond the scope of this specification, but typically involve end-user interaction with an HTML registration form.

Client registration does not require a direct interaction between the client and the authorization server. When supported by the authorization server, registration can rely on other means for establishing trust and obtaining the required client properties (e.g. redirection URI, client type). For example, registration can be accomplished using a self-issued or third-party-issued assertion, or by the authorization server performing client discovery using a trusted channel.

When registering a client, the client developer SHALL:

- o specify the client type as described in Section 2.1,
- o provide its client redirection URIs as described in Section 3.1.2, and
- o include any other information required by the authorization server (e.g. application name, website, description, logo image, the acceptance of legal terms).

2.1. Client Types

OAuth defines two client types, based on their ability to authenticate securely with the authorization server (i.e. ability to maintain the confidentiality of their client credentials):

confidential

Clients capable of maintaining the confidentiality of their credentials (e.g. client implemented on a secure server with restricted access to the client credentials), or capable of secure client authentication using other means.

public

Clients incapable of maintaining the confidentiality of their credentials (e.g. clients executing on the device used by the resource owner such as an installed native application or a web browser-based application), and incapable of secure client authentication via any other means.

The client type designation is based on the authorization server's definition of secure authentication and its acceptable exposure levels of client credentials. The authorization server **SHOULD NOT** make assumptions about the client type.

A client may be implemented as a distributed set of components, each with a different client type and security context (e.g. a distributed client with both a confidential server-based component and a public browser-based component). If the authorization server does not provide support for such clients, or does not provide guidance with regard to their registration, the client **SHOULD** register each component as a separate client.

This specification has been designed around the following client profiles:

web application

A web application is a confidential client running on a web server. Resource owners access the client via an HTML user interface rendered in a user-agent on the device used by the resource owner. The client credentials as well as any access token issued to the client are stored on the web server and are not exposed to or accessible by the resource owner.

user-agent-based application

A user-agent-based application is a public client in which the client code is downloaded from a web server and executes within a user-agent (e.g. web browser) on the device used by the resource owner. Protocol data and credentials are easily accessible (and often visible) to the resource owner. Since such applications reside within the user-agent, they can make seamless use of the user-agent capabilities when requesting authorization.

native application

A native application is a public client installed and executed on the device used by the resource owner. Protocol data and credentials are accessible to the resource owner. It is assumed that any client authentication credentials included in the

application can be extracted. On the other hand, dynamically issued credentials such as access tokens or refresh tokens can receive an acceptable level of protection. At a minimum, these credentials are protected from hostile servers with which the application may interact with. On some platforms these credentials might be protected from other applications residing on the same device.

2.2. Client Identifier

The authorization server issues the registered client a client identifier - a unique string representing the registration information provided by the client. The client identifier is not a secret; it is exposed to the resource owner, and MUST NOT be used alone for client authentication. The client identifier is unique to the authorization server.

The client identifier string size is left undefined by this specification. The client should avoid making assumptions about the identifier size. The authorization server SHOULD document the size of any identifier it issues.

2.3. Client Authentication

If the client type is confidential, the client and authorization server establish a client authentication method suitable for the security requirements of the authorization server. The authorization server MAY accept any form of client authentication meeting its security requirements.

Confidential clients are typically issued (or establish) a set of client credentials used for authenticating with the authorization server (e.g. password, public/private key pair).

The authorization server MAY establish a client authentication method with public clients. However, the authorization server MUST NOT rely on public client authentication for the purpose of identifying the client.

The client MUST NOT use more than one authentication method in each request.

2.3.1. Client Password

Clients in possession of a client password MAY use the HTTP Basic authentication scheme as defined in [RFC2617] to authenticate with the authorization server. The client identifier is encoded using the "application/x-www-form-urlencoded" encoding algorithm per Appendix B

and the encoded value is used as the username; the client password is encoded using the same algorithm and used as the password. The authorization server MUST support the HTTP Basic authentication scheme for authenticating clients that were issued a client password.

For example (with extra line breaks for display purposes only):

```
Authorization: Basic czZCaGRSa3F0Mzo3RmpmcDBaQnIxs3REUmJuZlZkbUl3
```

Alternatively, the authorization server MAY support including the client credentials in the request body using the following parameters:

`client_id`

REQUIRED. The client identifier issued to the client during the registration process described by Section 2.2.

`client_secret`

REQUIRED. The client secret. The client MAY omit the parameter if the client secret is an empty string.

Including the client credentials in the request body using the two parameters is NOT RECOMMENDED, and SHOULD be limited to clients unable to directly utilize the HTTP Basic authentication scheme (or other password-based HTTP authentication schemes). The parameters can only be transmitted in the request body and MUST NOT be included in the request URI.

For example, requesting to refresh an access token (Section 6) using the body parameters (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=refresh_token&refresh_token=tGzv3JOkF0XG5Qx2TlKWIA
&client_id=s6BhdRkqt3&client_secret=7Fjfp0ZBr1KtDRbnfVdmIw
```

The authorization server MUST require the use of TLS as described in Section 1.6 when sending requests using password authentication.

Since this client authentication method involves a password, the authorization server MUST protect any endpoint utilizing it against brute force attacks.

2.3.2. Other Authentication Methods

The authorization server MAY support any suitable HTTP authentication scheme matching its security requirements. When using other authentication methods, the authorization server MUST define a mapping between the client identifier (registration record) and authentication scheme.

2.4. Unregistered Clients

This specification does not exclude the use of unregistered clients. However, the use with such clients is beyond the scope of this specification, and requires additional security analysis and review of its interoperability impact.

3. Protocol Endpoints

The authorization process utilizes two authorization server endpoints (HTTP resources):

- o Authorization endpoint - used by the client to obtain authorization from the resource owner via user-agent redirection.
- o Token endpoint - used by the client to exchange an authorization grant for an access token, typically with client authentication.

As well as one client endpoint:

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

Not every authorization grant type utilizes both endpoints. Extension grant types MAY define additional endpoints as needed.

3.1. Authorization Endpoint

The authorization endpoint is used to interact with the resource owner and obtain an authorization grant. The authorization server MUST first verify the identity of the resource owner. The way in which the authorization server authenticates the resource owner (e.g. username and password login, session cookies) is beyond the scope of this specification.

The means through which the client obtains the location of the authorization endpoint are beyond the scope of this specification, but the location is typically provided in the service documentation.

The endpoint URI MAY include an "application/x-www-form-urlencoded" formatted (per Appendix B) query component ([RFC3986] section 3.4), which MUST be retained when adding additional query parameters. The endpoint URI MUST NOT include a fragment component.

Since requests to the authorization endpoint result in user authentication and the transmission of clear-text credentials (in the HTTP response), the authorization server MUST require the use of TLS as described in Section 1.6 when sending requests to the authorization endpoint.

The authorization server MUST support the use of the HTTP "GET" method [RFC2616] for the authorization endpoint, and MAY support the use of the "POST" method as well.

Parameters sent without a value MUST be treated as if they were omitted from the request. The authorization server MUST ignore unrecognized request parameters. Request and response parameters MUST NOT be included more than once.

3.1.1. Response Type

The authorization endpoint is used by the authorization code grant type and implicit grant type flows. The client informs the authorization server of the desired grant type using the following parameter:

response_type

REQUIRED. The value MUST be one of "code" for requesting an authorization code as described by Section 4.1.1, "token" for requesting an access token (implicit grant) as described by Section 4.2.1, or a registered extension value as described by Section 8.4.

Extension response types MAY contain a space-delimited (%x20) list of values, where the order of values does not matter (e.g. response type "a b" is the same as "b a"). The meaning of such composite response types is defined by their respective specifications.

If an authorization request is missing the "response_type" parameter, or if the response type is not understood, the authorization server MUST return an error response as described in Section 4.1.2.1.

3.1.2. Redirection Endpoint

After completing its interaction with the resource owner, the authorization server directs the resource owner's user-agent back to the client. The authorization server redirects the user-agent to the

client's redirection endpoint previously established with the authorization server during the client registration process or when making the authorization request.

The redirection endpoint URI MUST be an absolute URI as defined by [RFC3986] section 4.3. The endpoint URI MAY include an "application/x-www-form-urlencoded" formatted (per Appendix B) query component ([RFC3986] section 3.4), which MUST be retained when adding additional query parameters. The endpoint URI MUST NOT include a fragment component.

3.1.2.1. Endpoint Request Confidentiality

The redirection endpoint SHOULD require the use of TLS as described in Section 1.6 when the requested response type is "code" or "token", or when the redirection request will result in the transmission of sensitive credentials over an open network. This specification does not mandate the use of TLS because at the time of this writing, requiring clients to deploy TLS is a significant hurdle for many client developers. If TLS is not available, the authorization server SHOULD warn the resource owner about the insecure endpoint prior to redirection (e.g. display a message during the authorization request).

Lack of transport-layer security can have a severe impact on the security of the client and the protected resources it is authorized to access. The use of transport-layer security is particularly critical when the authorization process is used as a form of delegated end-user authentication by the client (e.g. third-party sign-in service).

3.1.2.2. Registration Requirements

The authorization server MUST require the following clients to register their redirection endpoint:

- o Public clients.
- o Confidential clients utilizing the implicit grant type.

The authorization server SHOULD require all clients to register their redirection endpoint prior to utilizing the authorization endpoint.

The authorization server SHOULD require the client to provide the complete redirection URI (the client MAY use the "state" request parameter to achieve per-request customization). If requiring the registration of the complete redirection URI is not possible, the authorization server SHOULD require the registration of the URI scheme, authority, and path (allowing the client to dynamically vary

only the query component of the redirection URI when requesting authorization).

The authorization server MAY allow the client to register multiple redirection endpoints.

Lack of a redirection URI registration requirement can enable an attacker to use the authorization endpoint as open redirector as described in Section 10.15.

3.1.2.3. Dynamic Configuration

If multiple redirection URIs have been registered, if only part of the redirection URI has been registered, or if no redirection URI has been registered, the client MUST include a redirection URI with the authorization request using the "redirect_uri" request parameter.

When a redirection URI is included in an authorization request, the authorization server MUST compare and match the value received against at least one of the registered redirection URIs (or URI components) as defined in [RFC3986] section 6, if any redirection URIs were registered. If the client registration included the full redirection URI, the authorization server MUST compare the two URIs using simple string comparison as defined in [RFC3986] section 6.2.1.

3.1.2.4. Invalid Endpoint

If an authorization request fails validation due to a missing, invalid, or mismatching redirection URI, the authorization server SHOULD inform the resource owner of the error, and MUST NOT automatically redirect the user-agent to the invalid redirection URI.

3.1.2.5. Endpoint Content

The redirection request to the client's endpoint typically results in an HTML document response, processed by the user-agent. If the HTML response is served directly as the result of the redirection request, any script included in the HTML document will execute with full access to the redirection URI and the credentials it contains.

The client SHOULD NOT include any third-party scripts (e.g. third-party analytics, social plug-ins, ad networks) in the redirection endpoint response. Instead, it SHOULD extract the credentials from the URI and redirect the user-agent again to another endpoint without exposing the credentials (in the URI or elsewhere). If third-party scripts are included, the client MUST ensure that its own scripts (used to extract and remove the credentials from the URI) will execute first.

3.2. Token Endpoint

The token endpoint is used by the client to obtain an access token by presenting its authorization grant or refresh token. The token endpoint is used with every authorization grant except for the implicit grant type (since an access token is issued directly).

The means through which the client obtains the location of the token endpoint are beyond the scope of this specification but is typically provided in the service documentation.

The endpoint URI MAY include an "application/x-www-form-urlencoded" formatted (per Appendix B) query component ([RFC3986] section 3.4), which MUST be retained when adding additional query parameters. The endpoint URI MUST NOT include a fragment component.

Since requests to the token endpoint result in the transmission of clear-text credentials (in the HTTP request and response), the authorization server MUST require the use of TLS as described in Section 1.6 when sending requests to the token endpoint.

The client MUST use the HTTP "POST" method when making access token requests.

Parameters sent without a value MUST be treated as if they were omitted from the request. The authorization server MUST ignore unrecognized request parameters. Request and response parameters MUST NOT be included more than once.

3.2.1. Client Authentication

Confidential clients or other clients issued client credentials MUST authenticate with the authorization server as described in Section 2.3 when making requests to the token endpoint. Client authentication is used for:

- o Enforcing the binding of refresh tokens and authorization codes to the client they were issued to. Client authentication is critical when an authorization code is transmitted to the redirection endpoint over an insecure channel, or when the redirection URI has not been registered in full.
- o Recovering from a compromised client by disabling the client or changing its credentials, thus preventing an attacker from abusing stolen refresh tokens. Changing a single set of client credentials is significantly faster than revoking an entire set of refresh tokens.

- o Implementing authentication management best practices, which require periodic credential rotation. Rotation of an entire set of refresh tokens can be challenging, while rotation of a single set of client credentials is significantly easier.

A client MAY use the "client_id" request parameter to identify itself when sending requests to the token endpoint. In the "authorization_code" "grant_type" request to the token endpoint, an unauthenticated client MUST send its "client_id" to prevent itself from inadvertently accepting a code intended for a client with a different "client_id". This protects the client from substitution of the authentication code. (It provides no additional security for the protected resource.)

3.3. Access Token Scope

The authorization and token endpoints allow the client to specify the scope of the access request using the "scope" request parameter. In turn, the authorization server uses the "scope" response parameter to inform the client of the scope of the access token issued.

The value of the scope parameter is expressed as a list of space-delimited, case sensitive strings. The strings are defined by the authorization server. If the value contains multiple space-delimited strings, their order does not matter, and each string adds an additional access range to the requested scope.

```
scope          = scope-token *( SP scope-token )
scope-token    = 1*( %x21 / %x23-5B / %x5D-7E )
```

The authorization server MAY fully or partially ignore the scope requested by the client based on the authorization server policy or the resource owner's instructions. If the issued access token scope is different from the one requested by the client, the authorization server MUST include the "scope" response parameter to inform the client of the actual scope granted.

If the client omits the scope parameter when requesting authorization, the authorization server MUST either process the request using a pre-defined default value, or fail the request indicating an invalid scope. The authorization server SHOULD document its scope requirements and default value (if defined).

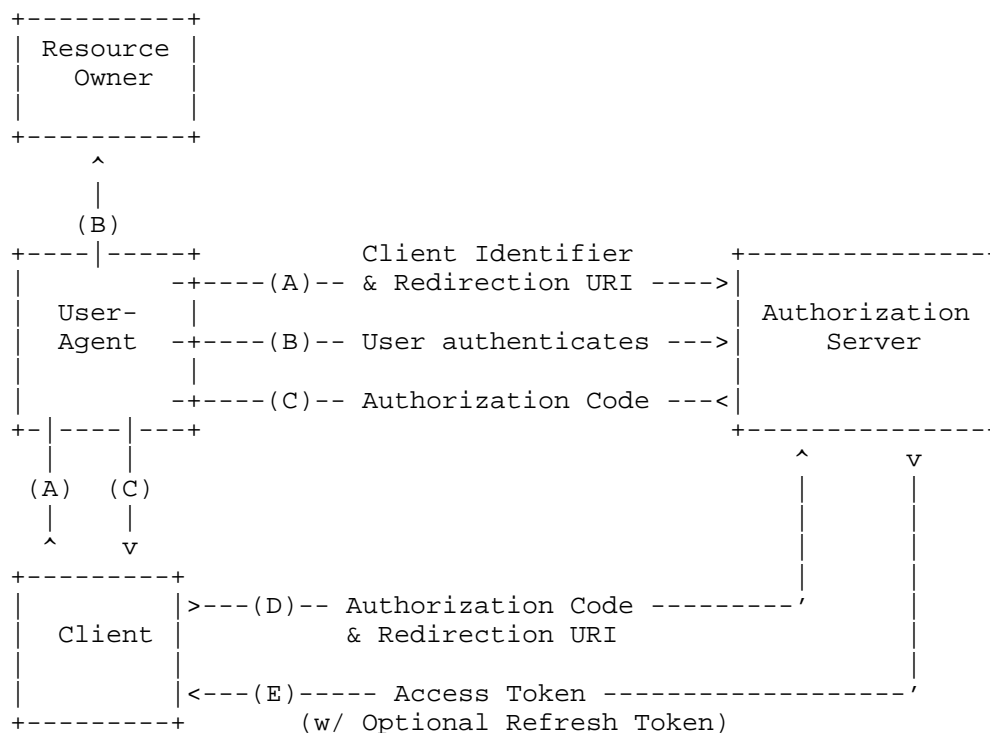
4. Obtaining Authorization

To request an access token, the client obtains authorization from the resource owner. The authorization is expressed in the form of an

authorization grant, which the client uses to request the access token. OAuth defines four grant types: authorization code, implicit, resource owner password credentials, and client credentials. It also provides an extension mechanism for defining additional grant types.

4.1. Authorization Code Grant

The authorization code grant type is used to obtain both access tokens and refresh tokens and is optimized for confidential clients. As a redirection-based flow, the client must be capable of interacting with the resource owner's user-agent (typically a web browser) and capable of receiving incoming requests (via redirection) from the authorization server.



Note: The lines illustrating steps A, B, and C are broken into two parts as they pass through the user-agent.

Figure 3: Authorization Code Flow

The flow illustrated in Figure 3 includes the following steps:

- (A) The client initiates the flow by directing the resource owner's user-agent to the authorization endpoint. The client includes its client identifier, requested scope, local state, and a redirection URI to which the authorization server will send the user-agent back once access is granted (or denied).
- (B) The authorization server authenticates the resource owner (via the user-agent) and establishes whether the resource owner grants or denies the client's access request.
- (C) Assuming the resource owner grants access, the authorization server redirects the user-agent back to the client using the redirection URI provided earlier (in the request or during client registration). The redirection URI includes an authorization code and any local state provided by the client earlier.
- (D) The client requests an access token from the authorization server's token endpoint by including the authorization code received in the previous step. When making the request, the client authenticates with the authorization server. The client includes the redirection URI used to obtain the authorization code for verification.
- (E) The authorization server authenticates the client, validates the authorization code, and ensures the redirection URI received matches the URI used to redirect the client in step (C). If valid, the authorization server responds back with an access token and optionally, a refresh token.

4.1.1. Authorization Request

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

`response_type`
REQUIRED. Value MUST be set to "code".

`client_id`
REQUIRED. The client identifier as described in Section 2.2.

`redirect_uri`
OPTIONAL. As described in Section 3.1.2.

`scope`
OPTIONAL. The scope of the access request as described by Section 3.3.

`state`
RECOMMENDED. An opaque value used by the client to maintain state between the request and callback. The authorization server includes this value when redirecting the user-agent back to the client. The parameter SHOULD be used for preventing cross-site request forgery as described in Section 10.12.

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

For example, the client directs the user-agent to make the following HTTP request using TLS (with extra line breaks for display purposes only):

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

The authorization server validates the request to ensure all required parameters are present and valid. If the request is valid, the authorization server authenticates the resource owner and obtains an authorization decision (by asking the resource owner or by establishing approval via other means).

When a decision is established, the authorization server directs the user-agent to the provided client redirection URI using an HTTP redirection response, or by other means available to it via the user-agent.

4.1.2. Authorization Response

If the resource owner grants the access request, the authorization server issues an authorization code and delivers it to the client by adding the following parameters to the query component of the redirection URI using the "application/x-www-form-urlencoded" format, per Appendix B:

code

REQUIRED. The authorization code generated by the authorization server. The authorization code **MUST** expire shortly after it is issued to mitigate the risk of leaks. A maximum authorization code lifetime of 10 minutes is RECOMMENDED. The client **MUST NOT** use the authorization code more than once. If an authorization code is used more than once, the authorization server **MUST** deny the request and **SHOULD** revoke (when possible) all tokens previously issued based on that authorization code. The authorization code is bound to the client identifier and redirection URI.

state

REQUIRED if the "state" parameter was present in the client authorization request. The exact value received from the client.

For example, the authorization server redirects the user-agent by sending the following HTTP response:

```
HTTP/1.1 302 Found
Location: https://client.example.com/cb?code=Splxl0BeZQQYbYS6WxSbIA
&state=xyz
```

The client **MUST** ignore unrecognized response parameters. The authorization code string size is left undefined by this specification. The client should avoid making assumptions about code value sizes. The authorization server **SHOULD** document the size of any value it issues.

4.1.2.1. Error Response

If the request fails due to a missing, invalid, or mismatching redirection URI, or if the client identifier is missing or invalid, the authorization server **SHOULD** inform the resource owner of the error, and **MUST NOT** automatically redirect the user-agent to the invalid redirection URI.

If the resource owner denies the access request or if the request fails for reasons other than a missing or invalid redirection URI, the authorization server informs the client by adding the following parameters to the query component of the redirection URI using the "application/x-www-form-urlencoded" format, per Appendix B:

error

REQUIRED. A single ASCII [USASCII] error code from the following:

invalid_request

The request is missing a required parameter, includes an invalid parameter value, includes a parameter more than once, or is otherwise malformed.

unauthorized_client

The client is not authorized to request an authorization code using this method.

access_denied

The resource owner or authorization server denied the request.

unsupported_response_type

The authorization server does not support obtaining an authorization code using this method.

invalid_scope

The requested scope is invalid, unknown, or malformed.

server_error
The authorization server encountered an unexpected condition that prevented it from fulfilling the request. (This error code is needed because a 500 Internal Server Error HTTP status code cannot be returned to the client via a HTTP redirect.)

temporarily_unavailable
The authorization server is currently unable to handle the request due to a temporary overloading or maintenance of the server. (This error code is needed because a 503 Service Unavailable HTTP status code cannot be returned to the client via a HTTP redirect.)

Values for the "error" parameter MUST NOT include characters outside the set %x20-21 / %x23-5B / %x5D-7E.

error_description
OPTIONAL. A human-readable ASCII [USASCII] text providing additional information, used to assist the client developer in understanding the error that occurred.
Values for the "error_description" parameter MUST NOT include characters outside the set %x20-21 / %x23-5B / %x5D-7E.

error_uri
OPTIONAL. A URI identifying a human-readable web page with information about the error, used to provide the client developer with additional information about the error.
Values for the "error_uri" parameter MUST conform to the URI-Reference syntax, and thus MUST NOT include characters outside the set %x21 / %x23-5B / %x5D-7E.

state
REQUIRED if a "state" parameter was present in the client authorization request. The exact value received from the client.

For example, the authorization server redirects the user-agent by sending the following HTTP response:

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

4.1.3. Access Token Request

The client makes a request to the token endpoint by sending the following parameters using the "application/x-www-form-urlencoded" format per Appendix B with a character encoding of UTF-8 in the HTTP request entity-body:

grant_type
REQUIRED. Value MUST be set to "authorization_code".

code
REQUIRED. The authorization code received from the authorization server.

redirect_uri
REQUIRED, if the "redirect_uri" parameter was included in the authorization request as described in Section 4.1.1, and their values MUST be identical.

client_id
REQUIRED, if the client is not authenticating with the authorization server as described in Section 3.2.1.

If the client type is confidential or the client was issued client credentials (or assigned other authentication requirements), the client MUST authenticate with the authorization server as described in Section 3.2.1.

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

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

grant_type=authorization_code&code=Splxl0BeZQQYbYS6WxSbIA
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
```

The authorization server MUST:

- o require client authentication for confidential clients or for any client that was issued client credentials (or with other authentication requirements),
- o authenticate the client if client authentication is included,
- o ensure the authorization code was issued to the authenticated confidential client, or if the client is public, ensure the code was issued to "client_id" in the request,
- o verify that the authorization code is valid, and
- o ensure that the "redirect_uri" parameter is present if the "redirect_uri" parameter was included in the initial authorization request as described in Section 4.1.1, and if included ensure their values are identical.

4.1.4. Access Token Response

If the access token request is valid and authorized, the authorization server issues an access token and optional refresh

token as described in Section 5.1. If the request client authentication failed or is invalid, the authorization server returns an error response as described in Section 5.2.

An example successful response:

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

{
  "access_token": "2YotnFZFEjrlzCsicMWpAA",
  "token_type": "example",
  "expires_in": 3600,
  "refresh_token": "tGzv3JOkF0XG5Qx2TlKWIA",
  "example_parameter": "example_value"
}
```

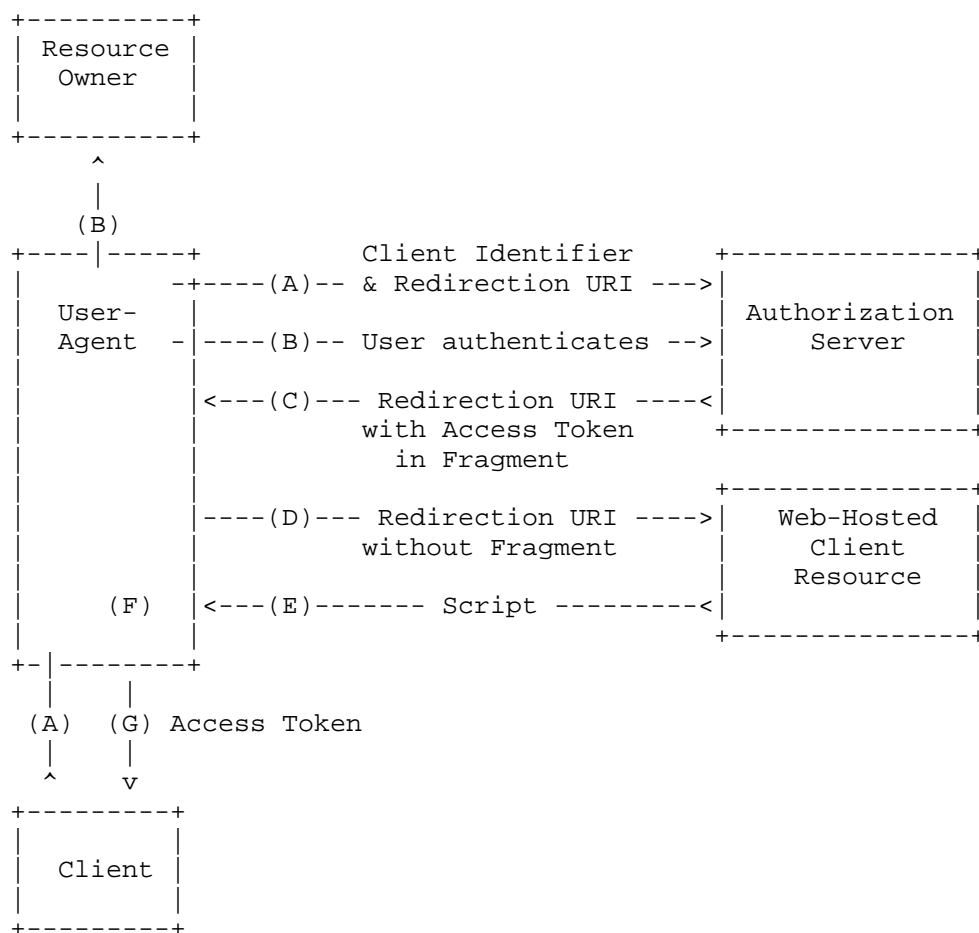
4.2. Implicit Grant

The implicit grant type is used to obtain access tokens (it does not support the issuance of refresh tokens) and is optimized for public clients known to operate a particular redirection URI. These clients are typically implemented in a browser using a scripting language such as JavaScript.

As a redirection-based flow, the client must be capable of interacting with the resource owner's user-agent (typically a web browser) and capable of receiving incoming requests (via redirection) from the authorization server.

Unlike the authorization code grant type in which the client makes separate requests for authorization and access token, the client receives the access token as the result of the authorization request.

The implicit grant type does not include client authentication, and relies on the presence of the resource owner and the registration of the redirection URI. Because the access token is encoded into the redirection URI, it may be exposed to the resource owner and other applications residing on the same device.



Note: The lines illustrating steps A and B are broken into two parts as they pass through the user-agent.

Figure 4: Implicit Grant Flow

The flow illustrated in Figure 4 includes the following steps:

- (A) The client initiates the flow by directing the resource owner's user-agent to the authorization endpoint. The client includes its client identifier, requested scope, local state, and a redirection URI to which the authorization server will send the user-agent back once access is granted (or denied).

- (B) The authorization server authenticates the resource owner (via the user-agent) and establishes whether the resource owner grants or denies the client's access request.
- (C) Assuming the resource owner grants access, the authorization server redirects the user-agent back to the client using the redirection URI provided earlier. The redirection URI includes the access token in the URI fragment.
- (D) The user-agent follows the redirection instructions by making a request to the web-hosted client resource (which does not include the fragment per [RFC2616]). The user-agent retains the fragment information locally.
- (E) The web-hosted client resource returns a web page (typically an HTML document with an embedded script) capable of accessing the full redirection URI including the fragment retained by the user-agent, and extracting the access token (and other parameters) contained in the fragment.
- (F) The user-agent executes the script provided by the web-hosted client resource locally, which extracts the access token and passes it to the client.

See Section 1.3.2 and Section 9 for background on using the implicit grant. See Section 10.3 and Section 10.16 for important security considerations when using the implicit grant.

4.2.1. Authorization Request

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

response_type
REQUIRED. Value MUST be set to "token".

client_id
REQUIRED. The client identifier as described in Section 2.2.

redirect_uri
OPTIONAL. As described in Section 3.1.2.

scope
OPTIONAL. The scope of the access request as described by Section 3.3.

state
RECOMMENDED. An opaque value used by the client to maintain state between the request and callback. The authorization server includes this value when redirecting the user-agent back to the client. The parameter SHOULD be used for preventing cross-site request forgery as described in Section 10.12.

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

user-agent.

For example, the client directs the user-agent to make the following HTTP request using TLS (with extra line breaks for display purposes only):

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

The authorization server validates the request to ensure all required parameters are present and valid. The authorization server **MUST** verify that the redirection URI to which it will redirect the access token matches a redirection URI registered by the client as described in Section 3.1.2.

If the request is valid, the authorization server authenticates the resource owner and obtains an authorization decision (by asking the resource owner or by establishing approval via other means).

When a decision is established, the authorization server directs the user-agent to the provided client redirection URI using an HTTP redirection response, or by other means available to it via the user-agent.

4.2.2. Access Token Response

If the resource owner grants the access request, the authorization server issues an access token and delivers it to the client by adding the following parameters to the fragment component of the redirection URI using the "application/x-www-form-urlencoded" format, per Appendix B:

access_token
REQUIRED. The access token issued by the authorization server.

token_type
REQUIRED. The type of the token issued as described in Section 7.1. Value is case insensitive.

expires_in
RECOMMENDED. The lifetime in seconds of the access token. For example, the value "3600" denotes that the access token will expire in one hour from the time the response was generated. If omitted, the authorization server **SHOULD** provide the expiration time via other means or document the default value.

scope
OPTIONAL, if identical to the scope requested by the client, otherwise REQUIRED. The scope of the access token as described by Section 3.3.

state
REQUIRED if the "state" parameter was present in the client authorization request. The exact value received from the client.

The authorization server MUST NOT issue a refresh token.

For example, the authorization server redirects the user-agent by sending the following HTTP response (with extra line breaks for display purposes only):

```
HTTP/1.1 302 Found
Location: http://example.com/cb#access_token=2YotnFZFEjrlzCsicMWpAA
        &state=xyz&token_type=example&expires_in=3600
```

Developers should note that some user-agents do not support the inclusion of a fragment component in the HTTP "Location" response header field. Such clients will require using other methods for redirecting the client than a 3xx redirection response. For example, returning an HTML page that includes a 'continue' button with an action linked to the redirection URI.

The client MUST ignore unrecognized response parameters. The access token string size is left undefined by this specification. The client should avoid making assumptions about value sizes. The authorization server SHOULD document the size of any value it issues.

4.2.2.1. Error Response

If the request fails due to a missing, invalid, or mismatching redirection URI, or if the client identifier is missing or invalid, the authorization server SHOULD inform the resource owner of the error, and MUST NOT automatically redirect the user-agent to the invalid redirection URI.

If the resource owner denies the access request or if the request fails for reasons other than a missing or invalid redirection URI, the authorization server informs the client by adding the following parameters to the fragment component of the redirection URI using the "application/x-www-form-urlencoded" format, per Appendix B:

error

REQUIRED. A single ASCII [USASCII] error code from the following:

invalid_request

The request is missing a required parameter, includes an invalid parameter value, includes a parameter more than once, or is otherwise malformed.

unauthorized_client

The client is not authorized to request an access token using this method.

access_denied

The resource owner or authorization server denied the request.

unsupported_response_type

The authorization server does not support obtaining an access token using this method.

invalid_scope

The requested scope is invalid, unknown, or malformed.

server_error

The authorization server encountered an unexpected condition that prevented it from fulfilling the request. (This error code is needed because a 500 Internal Server Error HTTP status code cannot be returned to the client via a HTTP redirect.)

temporarily_unavailable

The authorization server is currently unable to handle the request due to a temporary overloading or maintenance of the server. (This error code is needed because a 503 Service Unavailable HTTP status code cannot be returned to the client via a HTTP redirect.)

Values for the "error" parameter MUST NOT include characters outside the set %x20-21 / %x23-5B / %x5D-7E.

error_description

OPTIONAL. A human-readable ASCII [USASCII] text providing additional information, used to assist the client developer in understanding the error that occurred.

Values for the "error_description" parameter MUST NOT include characters outside the set %x20-21 / %x23-5B / %x5D-7E.

error_uri

OPTIONAL. A URI identifying a human-readable web page with information about the error, used to provide the client developer with additional information about the error.

Values for the "error_uri" parameter MUST conform to the URI-Reference syntax, and thus MUST NOT include characters outside the set %x21 / %x23-5B / %x5D-7E.

state

REQUIRED if a "state" parameter was present in the client authorization request. The exact value received from the client.

For example, the authorization server redirects the user-agent by sending the following HTTP response:

HTTP/1.1 302 Found

Location: https://client.example.com/cb#error=access_denied&state=xyz

4.3. Resource Owner Password Credentials Grant

The resource owner password credentials grant type is suitable in cases where the resource owner has a trust relationship with the client, such as the device operating system or a highly privileged application. The authorization server should take special care when enabling this grant type, and only allow it when other flows are not viable.

The grant type is suitable for clients capable of obtaining the resource owner's credentials (username and password, typically using an interactive form). It is also used to migrate existing clients using direct authentication schemes such as HTTP Basic or Digest authentication to OAuth by converting the stored credentials to an access token.

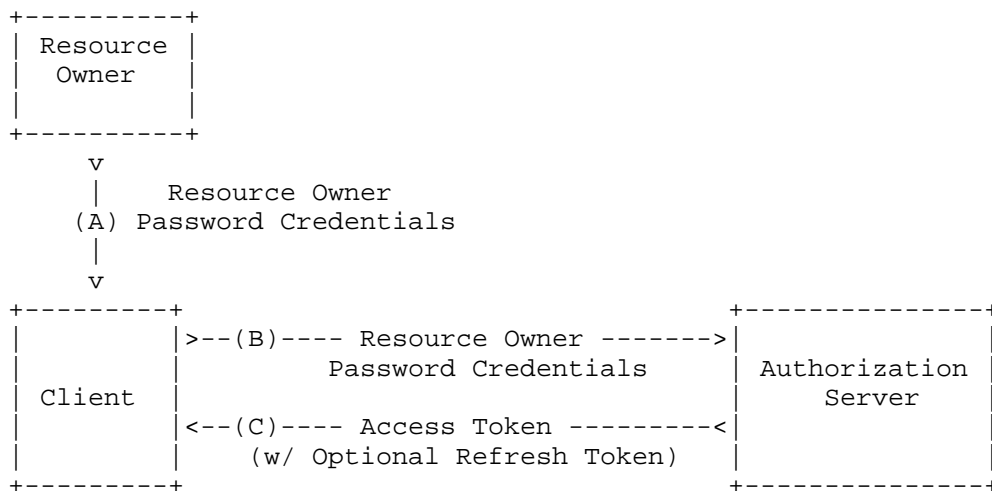


Figure 5: Resource Owner Password Credentials Flow

The flow illustrated in Figure 5 includes the following steps:

- (A) The resource owner provides the client with its username and password.
- (B) The client requests an access token from the authorization server's token endpoint by including the credentials received from the resource owner. When making the request, the client authenticates with the authorization server.
- (C) The authorization server authenticates the client and validates the resource owner credentials, and if valid issues an access token.

4.3.1. Authorization Request and Response

The method through which the client obtains the resource owner credentials is beyond the scope of this specification. The client MUST discard the credentials once an access token has been obtained.

4.3.2. Access Token Request

The client makes a request to the token endpoint by adding the following parameters using the "application/x-www-form-urlencoded" format per Appendix B with a character encoding of UTF-8 in the HTTP request entity-body:

grant_type
REQUIRED. Value MUST be set to "password".
username
REQUIRED. The resource owner username.
password
REQUIRED. The resource owner password.
scope
OPTIONAL. The scope of the access request as described by Section 3.3.

If the client type is confidential or the client was issued client credentials (or assigned other authentication requirements), the client MUST authenticate with the authorization server as described in Section 3.2.1.

For example, the client makes the following HTTP request using transport-layer security (with extra line breaks for display purposes only):

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

grant_type=password&username=johndoe&password=A3ddj3w
```

The authorization server MUST:

- o require client authentication for confidential clients or for any client that was issued client credentials (or with other authentication requirements),
- o authenticate the client if client authentication is included, and
- o validate the resource owner password credentials using its existing password validation algorithm.

Since this access token request utilizes the resource owner's password, the authorization server MUST protect the endpoint against brute force attacks (e.g. using rate-limitation or generating alerts).

4.3.3. Access Token Response

If the access token request is valid and authorized, the authorization server issues an access token and optional refresh token as described in Section 5.1. If the request failed client authentication or is invalid, the authorization server returns an error response as described in Section 5.2.

An example successful response:

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

{
  "access_token": "2YotnFZFEjrlzCsicMWpAA",
  "token_type": "example",
  "expires_in": 3600,
  "refresh_token": "tGzv3JOkF0XG5Qx2TlKWIA",
  "example_parameter": "example_value"
}
```

4.4. Client Credentials Grant

The client can request an access token using only its client credentials (or other supported means of authentication) when the client is requesting access to the protected resources under its control, or those of another resource owner that have been previously arranged with the authorization server (the method of which is beyond the scope of this specification).

The client credentials grant type MUST only be used by confidential clients.

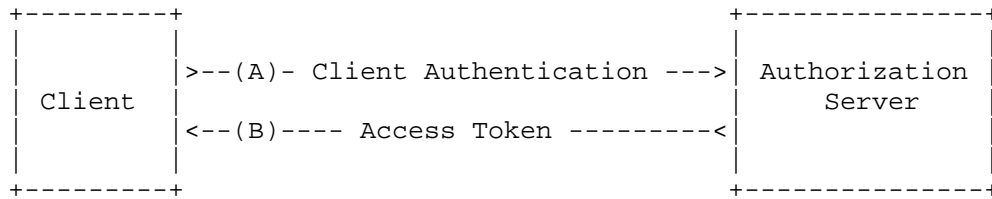


Figure 6: Client Credentials Flow

The flow illustrated in Figure 6 includes the following steps:

- (A) The client authenticates with the authorization server and requests an access token from the token endpoint.
- (B) The authorization server authenticates the client, and if valid issues an access token.

4.4.1. Authorization Request and Response

Since the client authentication is used as the authorization grant, no additional authorization request is needed.

4.4.2. Access Token Request

The client makes a request to the token endpoint by adding the following parameters using the "application/x-www-form-urlencoded" format per Appendix B with a character encoding of UTF-8 in the HTTP request entity-body:

```

grant_type
    REQUIRED. Value MUST be set to "client_credentials".
scope
    OPTIONAL. The scope of the access request as described by
    Section 3.3.
  
```

The client MUST authenticate with the authorization server as described in Section 3.2.1.

For example, the client makes the following HTTP request using transport-layer security (with extra line breaks for display purposes only):

```

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

grant_type=client_credentials
  
```

The authorization server MUST authenticate the client.

4.4.3. Access Token Response

If the access token request is valid and authorized, the authorization server issues an access token as described in Section 5.1. A refresh token SHOULD NOT be included. If the request failed client authentication or is invalid, the authorization server returns an error response as described in Section 5.2.

An example successful response:

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

{
  "access_token":"2YotnFZFEjrlzCsicMWpAA",
  "token_type":"example",
  "expires_in":3600,
  "example_parameter":"example_value"
}
```

4.5. Extension Grants

The client uses an extension grant type by specifying the grant type using an absolute URI (defined by the authorization server) as the value of the "grant_type" parameter of the token endpoint, and by adding any additional parameters necessary.

For example, to request an access token using a SAML 2.0 assertion grant type as defined by [I-D.ietf-oauth-saml2-bearer], the client could make the following HTTP request using TLS (with extra line breaks for display purposes only):

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

grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Asaml2-
bearer&assertion=PEFzc2VydGlvbiBJc3NlZUlu3RhbnQ9IjIwMTFtMDU
[...omitted for brevity...]aG5TdGF0ZWllbnQ-PC9Bc3NlcnRpb24-
```

If the access token request is valid and authorized, the authorization server issues an access token and optional refresh token as described in Section 5.1. If the request failed client authentication or is invalid, the authorization server returns an

error response as described in Section 5.2.

5. Issuing an Access Token

If the access token request is valid and authorized, the authorization server issues an access token and optional refresh token as described in Section 5.1. If the request failed client authentication or is invalid, the authorization server returns an error response as described in Section 5.2.

5.1. Successful Response

The authorization server issues an access token and optional refresh token, and constructs the response by adding the following parameters to the entity body of the HTTP response with a 200 (OK) status code:

access_token
REQUIRED. The access token issued by the authorization server.

token_type
REQUIRED. The type of the token issued as described in Section 7.1. Value is case insensitive.

expires_in
RECOMMENDED. The lifetime in seconds of the access token. For example, the value "3600" denotes that the access token will expire in one hour from the time the response was generated. If omitted, the authorization server SHOULD provide the expiration time via other means or document the default value.

refresh_token
OPTIONAL. The refresh token, which can be used to obtain new access tokens using the same authorization grant as described in Section 6.

scope
OPTIONAL, if identical to the scope requested by the client, otherwise REQUIRED. The scope of the access token as described by Section 3.3.

The parameters are included in the entity body of the HTTP response using the "application/json" media type as defined by [RFC4627]. The parameters are serialized into a JSON structure by adding each parameter at the highest structure level. Parameter names and string values are included as JSON strings. Numerical values are included as JSON numbers. The order of parameters does not matter and can vary.

The authorization server MUST include the HTTP "Cache-Control" response header field [RFC2616] with a value of "no-store" in any response containing tokens, credentials, or other sensitive

information, as well as the "Pragma" response header field [RFC2616] with a value of "no-cache".

For example:

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

```
{
  "access_token":"2YotnFZFEjrlzCsicMWpAA",
  "token_type":"example",
  "expires_in":3600,
  "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
  "example_parameter":"example_value"
}
```

The client **MUST** ignore unrecognized value names in the response. The sizes of tokens and other values received from the authorization server are left undefined. The client should avoid making assumptions about value sizes. The authorization server **SHOULD** document the size of any value it issues.

5.2. Error Response

The authorization server responds with an HTTP 400 (Bad Request) status code (unless specified otherwise) and includes the following parameters with the response:

error

REQUIRED. A single ASCII [USASCII] error code from the following:

invalid_request

The request is missing a required parameter, includes an unsupported parameter value (other than grant type), repeats a parameter, includes multiple credentials, utilizes more than one mechanism for authenticating the client, or is otherwise malformed.

invalid_client

Client authentication failed (e.g. unknown client, no client authentication included, or unsupported authentication method). The authorization server **MAY** return an HTTP 401 (Unauthorized) status code to indicate which HTTP authentication schemes are supported. If the client attempted to authenticate via the "Authorization" request header field, the authorization server **MUST** respond with an HTTP 401 (Unauthorized) status code, and

include the "WWW-Authenticate" response header field matching the authentication scheme used by the client.

invalid_grant
The provided authorization grant (e.g. authorization code, resource owner credentials) or refresh token is invalid, expired, revoked, does not match the redirection URI used in the authorization request, or was issued to another client.

unauthorized_client
The authenticated client is not authorized to use this authorization grant type.

unsupported_grant_type
The authorization grant type is not supported by the authorization server.

invalid_scope
The requested scope is invalid, unknown, malformed, or exceeds the scope granted by the resource owner.

Values for the "error" parameter MUST NOT include characters outside the set %x20-21 / %x23-5B / %x5D-7E.

error_description
OPTIONAL. A human-readable ASCII [USASCII] text providing additional information, used to assist the client developer in understanding the error that occurred.
Values for the "error_description" parameter MUST NOT include characters outside the set %x20-21 / %x23-5B / %x5D-7E.

error_uri
OPTIONAL. A URI identifying a human-readable web page with information about the error, used to provide the client developer with additional information about the error.
Values for the "error_uri" parameter MUST conform to the URI-Reference syntax, and thus MUST NOT include characters outside the set %x21 / %x23-5B / %x5D-7E.

The parameters are included in the entity body of the HTTP response using the "application/json" media type as defined by [RFC4627]. The parameters are serialized into a JSON structure by adding each parameter at the highest structure level. Parameter names and string values are included as JSON strings. Numerical values are included as JSON numbers. The order of parameters does not matter and can vary.

For example:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "error": "invalid_request"
}
```

6. Refreshing an Access Token

If the authorization server issued a refresh token to the client, the client makes a refresh request to the token endpoint by adding the following parameters using the "application/x-www-form-urlencoded" format per Appendix B with a character encoding of UTF-8 in the HTTP request entity-body:

grant_type
REQUIRED. Value MUST be set to "refresh_token".

refresh_token
REQUIRED. The refresh token issued to the client.

scope
OPTIONAL. The scope of the access request as described by Section 3.3. The requested scope MUST NOT include any scope not originally granted by the resource owner, and if omitted is treated as equal to the scope originally granted by the resource owner.

Because refresh tokens are typically long-lasting credentials used to request additional access tokens, the refresh token is bound to the client to which it was issued. If the client type is confidential or the client was issued client credentials (or assigned other authentication requirements), the client MUST authenticate with the authorization server as described in Section 3.2.1.

For example, the client makes the following HTTP request using transport-layer security (with extra line breaks for display purposes only):

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

grant_type=refresh_token&refresh_token=tGzv3JOkF0XG5Qx2TlKWIA
```

The authorization server MUST:

- o require client authentication for confidential clients or for any client that was issued client credentials (or with other authentication requirements),
- o authenticate the client if client authentication is included and ensure the refresh token was issued to the authenticated client, and
- o validate the refresh token.

If valid and authorized, the authorization server issues an access token as described in Section 5.1. If the request failed verification or is invalid, the authorization server returns an error response as described in Section 5.2.

The authorization server MAY issue a new refresh token, in which case the client MUST discard the old refresh token and replace it with the new refresh token. The authorization server MAY revoke the old refresh token after issuing a new refresh token to the client. If a new refresh token is issued, the refresh token scope MUST be identical to that of the refresh token included by the client in the request.

7. Accessing Protected Resources

The client accesses protected resources by presenting the access token to the resource server. The resource server MUST validate the access token and ensure it has not expired and that its scope covers the requested resource. The methods used by the resource server to validate the access token (as well as any error responses) are beyond the scope of this specification, but generally involve an interaction or coordination between the resource server and the authorization server.

The method in which the client utilizes the access token to authenticate with the resource server depends on the type of access token issued by the authorization server. Typically, it involves using the HTTP "Authorization" request header field [RFC2617] with an authentication scheme defined by the access token type specification.

7.1. Access Token Types

The access token type provides the client with the information required to successfully utilize the access token to make a protected resource request (along with type-specific attributes). The client MUST NOT use an access token if it does not understand the token type.

For example, the "bearer" token type defined in [I-D.ietf-oauth-v2-bearer] is utilized by simply including the access token string in the request:

```
GET /resource/1 HTTP/1.1
Host: example.com
Authorization: Bearer mF_9.B5f-4.1JqM
```

while the "mac" token type defined in [I-D.ietf-oauth-v2-http-mac] is utilized by issuing a MAC key together with the access token that is used to sign certain components of the HTTP requests:

```
GET /resource/1 HTTP/1.1
Host: example.com
Authorization: MAC id="h480djs93hd8",
                  nonce="274312:dj83hs9s",
                  mac="kDZvddkndxvhGRXZhvuDjEWhGeE="
```

The above examples are provided for illustration purposes only. Developers are advised to consult the [I-D.ietf-oauth-v2-bearer] and [I-D.ietf-oauth-v2-http-mac] specifications before use.

Each access token type definition specifies the additional attributes (if any) sent to the client together with the "access_token" response parameter. It also defines the HTTP authentication method used to include the access token when making a protected resource request.

7.2. Error Response

If a resource access request fails, the resource server SHOULD inform the client of the error. While the specifics of such error responses are beyond the scope of this specification, this document establishes a common registry in Section 11.4 for error values to be shared among OAuth token authentication schemes.

New authentication schemes designed primarily for OAuth token authentication SHOULD define a mechanism for providing an error status code to the client, in which the error values allowed are registered in the error registry established by this specification. Such schemes MAY limit the set of valid error codes to a subset of the registered values. If the error code is returned using a named parameter, the parameter name SHOULD be "error".

Other schemes capable of being used for OAuth token authentication, but not primarily designed for that purpose, MAY bind their error values to the registry in the same manner.

New authentication schemes MAY choose to also specify the use of the

"error_description" and "error_uri" parameters to return error information in a manner parallel to their usage in this specification.

8. Extensibility

8.1. Defining Access Token Types

Access token types can be defined in one of two ways: registered in the access token type registry (following the procedures in Section 11.1), or by using a unique absolute URI as its name.

Types utilizing a URI name SHOULD be limited to vendor-specific implementations that are not commonly applicable, and are specific to the implementation details of the resource server where they are used.

All other types MUST be registered. Type names MUST conform to the type-name ABNF. If the type definition includes a new HTTP authentication scheme, the type name SHOULD be identical to the HTTP authentication scheme name (as defined by [RFC2617]). The token type "example" is reserved for use in examples.

```
type-name  = 1*name-char
name-char  = "-" / "." / "_" / DIGIT / ALPHA
```

8.2. Defining New Endpoint Parameters

New request or response parameters for use with the authorization endpoint or the token endpoint are defined and registered in the parameters registry following the procedure in Section 11.2.

Parameter names MUST conform to the param-name ABNF and parameter values syntax MUST be well-defined (e.g., using ABNF, or a reference to the syntax of an existing parameter).

```
param-name = 1*name-char
name-char  = "-" / "." / "_" / DIGIT / ALPHA
```

Unregistered vendor-specific parameter extensions that are not commonly applicable, and are specific to the implementation details of the authorization server where they are used SHOULD utilize a vendor-specific prefix that is not likely to conflict with other registered values (e.g. begin with 'companyname_').

8.3. Defining New Authorization Grant Types

New authorization grant types can be defined by assigning them a unique absolute URI for use with the "grant_type" parameter. If the extension grant type requires additional token endpoint parameters, they MUST be registered in the OAuth parameters registry as described by Section 11.2.

8.4. Defining New Authorization Endpoint Response Types

New response types for use with the authorization endpoint are defined and registered in the authorization endpoint response type registry following the procedure in Section 11.3. Response type names MUST conform to the response-type ABNF.

```
response-type = response-name *( SP response-name )
response-name = 1*response-char
response-char = "_" / DIGIT / ALPHA
```

If a response type contains one or more space characters (%x20), it is compared as a space-delimited list of values in which the order of values does not matter. Only one order of values can be registered, which covers all other arrangements of the same set of values.

For example, the response type "token code" is left undefined by this specification. However, an extension can define and register the "token code" response type. Once registered, the same combination cannot be registered as "code token", but both values can be used to denote the same response type.

8.5. Defining Additional Error Codes

In cases where protocol extensions (i.e. access token types, extension parameters, or extension grant types) require additional error codes to be used with the authorization code grant error response (Section 4.1.2.1), the implicit grant error response (Section 4.2.2.1), the token error response (Section 5.2), or the resource access error response (Section 7.2), such error codes MAY be defined.

Extension error codes MUST be registered (following the procedures in Section 11.4) if the extension they are used in conjunction with is a registered access token type, a registered endpoint parameter, or an extension grant type. Error codes used with unregistered extensions MAY be registered.

Error codes MUST conform to the error ABNF, and SHOULD be prefixed by an identifying name when possible. For example, an error identifying

an invalid value set to the extension parameter "example" SHOULD be named "example_invalid".

```
error      = 1*error-char
error-char = %x20-21 / %x23-5B / %x5D-7E
```

9. Native Applications

Native applications are clients installed and executed on the device used by the resource owner (i.e. desktop application, native mobile application). Native applications require special consideration related to security, platform capabilities, and overall end-user experience.

The authorization endpoint requires interaction between the client and the resource owner's user-agent. Native applications can invoke an external user-agent or embed a user-agent within the application. For example:

- o External user-agent - the native application can capture the response from the authorization server using a redirection URI with a scheme registered with the operating system to invoke the client as the handler, manual copy-and-paste of the credentials, running a local web server, installing a user-agent extension, or by providing a redirection URI identifying a server-hosted resource under the client's control, which in turn makes the response available to the native application.
- o Embedded user-agent - the native application obtains the response by directly communicating with the embedded user-agent by monitoring state changes emitted during the resource load, or accessing the user-agent's cookies storage.

When choosing between an external or embedded user-agent, developers should consider:

- o An External user-agent may improve completion rate as the resource owner may already have an active session with the authorization server removing the need to re-authenticate. It provides a familiar end-user experience and functionality. The resource owner may also rely on user-agent features or extensions to assist with authentication (e.g. password manager, 2-factor device reader).
- o An embedded user-agent may offer improved usability, as it removes the need to switch context and open new windows.
- o An embedded user-agent poses a security challenge because resource owners are authenticating in an unidentified window without access to the visual protections found in most external user-agents. An

embedded user-agent educates end-users to trust unidentified requests for authentication (making phishing attacks easier to execute).

When choosing between the implicit grant type and the authorization code grant type, the following should be considered:

- o Native applications that use the authorization code grant type SHOULD do so without using client credentials, due to the native application's inability to keep client credentials confidential.
- o When using the implicit grant type flow, a refresh token is not returned, which requires repeating the authorization process once the access token expires.

10. Security Considerations

As a flexible and extensible framework, OAuth's security considerations depend on many factors. The following sections provide implementers with security guidelines focused on the three client profiles described in Section 2.1: web application, user-agent-based application, and native application.

A comprehensive OAuth security model and analysis, as well as background for the protocol design, is provided by [I-D.ietf-oauth-v2-threatmodel].

10.1. Client Authentication

The authorization server establishes client credentials with web application clients for the purpose of client authentication. The authorization server is encouraged to consider stronger client authentication means than a client password. Web application clients MUST ensure confidentiality of client passwords and other client credentials.

The authorization server MUST NOT issue client passwords or other client credentials to native application or user-agent-based application clients for the purpose of client authentication. The authorization server MAY issue a client password or other credentials for a specific installation of a native application client on a specific device.

When client authentication is not possible, the authorization server SHOULD employ other means to validate the client's identity. For example, by requiring the registration of the client redirection URI or enlisting the resource owner to confirm identity. A valid redirection URI is not sufficient to verify the client's identity

when asking for resource owner authorization, but can be used to prevent delivering credentials to a counterfeit client after obtaining resource owner authorization.

The authorization server must consider the security implications of interacting with unauthenticated clients and take measures to limit the potential exposure of other credentials (e.g. refresh tokens) issued to such clients.

10.2. Client Impersonation

A malicious client can impersonate another client and obtain access to protected resources, if the impersonated client fails to, or is unable to, keep its client credentials confidential.

The authorization server **MUST** authenticate the client whenever possible. If the authorization server cannot authenticate the client due to the client's nature, the authorization server **MUST** require the registration of any redirection URI used for receiving authorization responses, and **SHOULD** utilize other means to protect resource owners from such potentially malicious clients. For example, the authorization server can engage the resource owner to assist in identifying the client and its origin.

The authorization server **SHOULD** enforce explicit resource owner authentication and provide the resource owner with information about the client and the requested authorization scope and lifetime. It is up to the resource owner to review the information in the context of the current client, and authorize or deny the request.

The authorization server **SHOULD NOT** process repeated authorization requests automatically (without active resource owner interaction) without authenticating the client or relying on other measures to ensure the repeated request comes from the original client and not an impersonator.

10.3. Access Tokens

Access token credentials (as well as any confidential access token attributes) **MUST** be kept confidential in transit and storage, and only shared among the authorization server, the resource servers the access token is valid for, and the client to whom the access token is issued. Access token credentials **MUST** only be transmitted using TLS as described in Section 1.6 with server authentication as defined by [RFC2818].

When using the implicit grant type, the access token is transmitted in the URI fragment, which can expose it to unauthorized parties.

The authorization server MUST ensure that access tokens cannot be generated, modified, or guessed to produce valid access tokens by unauthorized parties.

The client SHOULD request access tokens with the minimal scope necessary. The authorization server SHOULD take the client identity into account when choosing how to honor the requested scope, and MAY issue an access token with a less rights than requested.

This specification does not provide any methods for the resource server to ensure that an access token presented to it by a given client was issued to that client by the authorization server.

10.4. Refresh Tokens

Authorization servers MAY issue refresh tokens to web application clients and native application clients.

Refresh tokens MUST be kept confidential in transit and storage, and shared only among the authorization server and the client to whom the refresh tokens were issued. The authorization server MUST maintain the binding between a refresh token and the client to whom it was issued. Refresh tokens MUST only be transmitted using TLS as described in Section 1.6 with server authentication as defined by [RFC2818].

The authorization server MUST verify the binding between the refresh token and client identity whenever the client identity can be authenticated. When client authentication is not possible, the authorization server SHOULD deploy other means to detect refresh token abuse.

For example, the authorization server could employ refresh token rotation in which a new refresh token is issued with every access token refresh response. The previous refresh token is invalidated but retained by the authorization server. If a refresh token is compromised and subsequently used by both the attacker and the legitimate client, one of them will present an invalidated refresh token, which will inform the authorization server of the breach.

The authorization server MUST ensure that refresh tokens cannot be generated, modified, or guessed to produce valid refresh tokens by unauthorized parties.

10.5. Authorization Codes

The transmission of authorization codes SHOULD be made over a secure channel, and the client SHOULD require the use of TLS with its

redirection URI if the URI identifies a network resource. Since authorization codes are transmitted via user-agent redirections, they could potentially be disclosed through user-agent history and HTTP referrer headers.

Authorization codes operate as plaintext bearer credentials, used to verify that the resource owner who granted authorization at the authorization server is the same resource owner returning to the client to complete the process. Therefore, if the client relies on the authorization code for its own resource owner authentication, the client redirection endpoint **MUST** require the use of TLS.

Authorization codes **MUST** be short lived and single use. If the authorization server observes multiple attempts to exchange an authorization code for an access token, the authorization server **SHOULD** attempt to revoke all access tokens already granted based on the compromised authorization code.

If the client can be authenticated, the authorization servers **MUST** authenticate the client and ensure that the authorization code was issued to the same client.

10.6. Authorization Code Redirection URI Manipulation

When requesting authorization using the authorization code grant type, the client can specify a redirection URI via the "redirect_uri" parameter. If an attacker can manipulate the value of the redirection URI, it can cause the authorization server to redirect the resource owner user-agent to a URI under the control of the attacker with the authorization code.

An attacker can create an account at a legitimate client and initiate the authorization flow. When the attacker's user-agent is sent to the authorization server to grant access, the attacker grabs the authorization URI provided by the legitimate client, and replaces the client's redirection URI with a URI under the control of the attacker. The attacker then tricks the victim into following the manipulated link to authorize access to the legitimate client.

Once at the authorization server, the victim is prompted with a normal, valid request on behalf of a legitimate and trusted client, and authorizes the request. The victim is then redirected to an endpoint under the control of the attacker with the authorization code. The attacker completes the authorization flow by sending the authorization code to the client using the original redirection URI provided by the client. The client exchanges the authorization code with an access token and links it to the attacker's client account, which can now gain access to the protected resources authorized by

the victim (via the client).

In order to prevent such an attack, the authorization server **MUST** ensure that the redirection URI used to obtain the authorization code is identical to the redirection URI provided when exchanging the authorization code for an access token. The authorization server **MUST** require public clients and **SHOULD** require confidential clients to register their redirection URIs. If a redirection URI is provided in the request, the authorization server **MUST** validate it against the registered value.

10.7. Resource Owner Password Credentials

The resource owner password credentials grant type is often used for legacy or migration reasons. It reduces the overall risk of storing username and password by the client, but does not eliminate the need to expose highly privileged credentials to the client.

This grant type carries a higher risk than other grant types because it maintains the password anti-pattern this protocol seeks to avoid. The client could abuse the password or the password could unintentionally be disclosed to an attacker (e.g. via log files or other records kept by the client).

Additionally, because the resource owner does not have control over the authorization process (the resource owner involvement ends when it hands over its credentials to the client), the client can obtain access tokens with a broader scope than desired by the resource owner. The authorization server should consider the scope and lifetime of access tokens issued via this grant type.

The authorization server and client **SHOULD** minimize use of this grant type and utilize other grant types whenever possible.

10.8. Request Confidentiality

Access tokens, refresh tokens, resource owner passwords, and client credentials **MUST NOT** be transmitted in the clear. Authorization codes **SHOULD NOT** be transmitted in the clear.

The "state" and "scope" parameters **SHOULD NOT** include sensitive client or resource owner information in plain text as they can be transmitted over insecure channels or stored insecurely.

10.9. Endpoints Authenticity

In order to prevent man-in-the-middle attacks, the authorization server **MUST** require the use of TLS with server authentication as

defined by [RFC2818] for any request sent to the authorization and token endpoints. The client MUST validate the authorization server's TLS certificate as defined by [RFC6125], and in accordance with its requirements for server identity authentication.

10.10. Credentials Guessing Attacks

The authorization server MUST prevent attackers from guessing access tokens, authorization codes, refresh tokens, resource owner passwords, and client credentials.

The probability of an attacker guessing generated tokens (and other credentials not intended for handling by end-users) MUST be less than or equal to $2^{(-128)}$ and SHOULD be less than or equal to $2^{(-160)}$.

The authorization server MUST utilize other means to protect credentials intended for end-user usage.

10.11. Phishing Attacks

Wide deployment of this and similar protocols may cause end-users to become inured to the practice of being redirected to websites where they are asked to enter their passwords. If end-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 resource owners' passwords.

Service providers should attempt to educate end-users about the risks phishing attacks pose, and should provide mechanisms that make it easy for end-users to confirm the authenticity of their sites. Client developers should consider the security implications of how they interact with the user-agent (e.g., external, embedded), and the ability of the end-user to verify the authenticity of the authorization server.

To reduce the risk of phishing attacks, the authorization servers MUST require the use of TLS on every endpoint used for end-user interaction.

10.12. Cross-Site Request Forgery

Cross-site request forgery (CSRF) is an exploit in which an attacker causes the user-agent of a victim end-user to follow a malicious URI (e.g. provided to the user-agent as a misleading link, image, or redirection) to a trusting server (usually established via the presence of a valid session cookie).

A CSRF attack against the client's redirection URI allows an attacker

to 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 **MUST** implement CSRF protection for its redirection URI. This is typically accomplished by requiring any request sent to the redirection URI endpoint to include 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). The client **SHOULD** utilize the "state" request parameter to deliver this value to the authorization server 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. The binding value used for CSRF protection **MUST** contain a non-guessable value (as described in Section 10.10), and the user-agent's authenticated state (e.g. session cookie, HTML5 local storage) **MUST** be kept in a location accessible only to the client and the user-agent (i.e., protected by same-origin policy).

A CSRF attack against the authorization server's authorization endpoint can result in an attacker obtaining end-user authorization for a malicious client without involving or alerting the end-user.

The authorization server **MUST** implement CSRF protection for its authorization endpoint, and ensure that a malicious client cannot obtain authorization without the awareness and explicit consent of the resource owner.

10.13. Clickjacking

In a clickjacking attack, an attacker registers a legitimate client and then constructs a malicious site in which it loads the authorization server's authorization endpoint web page in a transparent iframe overlaid on top of a set of dummy buttons, which are carefully constructed to be placed directly under important buttons on the authorization page. When an end-user clicks a misleading visible button, the end-user is actually clicking an invisible button on the authorization page (such as an "Authorize" button). This allows an attacker to trick a resource owner into granting its client access without their knowledge.

To prevent this form of attack, native applications SHOULD use external browsers instead of embedding browsers within the application when requesting end-user authorization. For most newer browsers, avoidance of iframes can be enforced by the authorization server using the (non-standard) "x-frame-options" header. This header can have two values, "deny" and "sameorigin", which will block any framing, or framing by sites with a different origin, respectively. For older browsers, JavaScript framebusting techniques can be used but may not be effective in all browsers.

10.14. Code Injection and Input Validation

A code injection attack occurs when an input or otherwise external variable is used by an application unsanitized and causes modification to the application logic. This may allow an attacker to gain access to the application device or its data, cause denial of service, or a wide range of malicious side-effects.

The Authorization server and client MUST sanitize (and validate when possible) any value received, in particular, the value of the "state" and "redirect_uri" parameters.

10.15. Open Redirectors

The authorization server authorization endpoint and the client redirection endpoint can be improperly configured and operate as open redirectors. 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.

Open redirectors can be used in phishing attacks, or by an attacker to get end-users to visit malicious sites by making the URI's authority look like a familiar and trusted destination. In addition, 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.

10.16. Misuse of Access Token to Impersonate Resource Owner in Implicit Flow

For public clients using implicit flows, this specification does not provide any method for the client to determine what client an access token was issued to.

A Resource Owner may willingly delegate access to a resource by

granting an access token to an attacker's malicious client. This may be due to Phishing or some other pretext. An attacker may also steal a token via some other mechanism. An attacker may then attempt to impersonate the resource owner by providing the access token to a legitimate public client.

In the implicit flow (`response_type=token`), the attacker can easily switch the token in the response from the authorization server, replacing the real `access_token` with the one previously issued to the attacker.

Servers communicating with native applications that rely on being passed an access token in the back channel to identify the user of the client may be similarly compromised by an attacker creating a compromised application that can inject arbitrary stolen access tokens.

Any public client that makes the assumption that only the resource owner can present them with a valid access token for the resource is vulnerable to this attack.

This attack may expose information about the resource owner at the legitimate client to the attacker (malicious client). This will also allow the attacker to perform operations at the legitimate client with the same permissions as the resource owner who originally granted the access token or authorization code.

Authenticating Resource Owners to clients is out of scope for this specification. Any specification that uses the authorization process as a form of delegated end-user authentication to the client (e.g. third-party sign-in service) MUST NOT use the implicit flow without additional security mechanisms such as audience restricting the access token that enable the client to determine if the access token was issued for its use.

11. IANA Considerations

11.1. OAuth Access Token Type Registry

This specification establishes the OAuth access token type registry.

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

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

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

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

11.1.1. Registration Template

Type name:

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

Additional Token Endpoint Response Parameters:

Additional response parameters returned together with the "access_token" parameter. New parameters MUST be separately registered in the OAuth parameters registry as described by Section 11.2.

HTTP Authentication Scheme(s):

The HTTP authentication scheme name(s), if any, used to authenticate protected resources requests using access tokens of this type.

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 parameter, 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.

11.2. OAuth Parameters Registry

This specification establishes the OAuth parameters registry.

Additional parameters for inclusion in the authorization endpoint request, the authorization endpoint response, the token endpoint request, or the token endpoint response are registered with a Specification Required ([RFC5226]) after a two week review period on the [TBD]@ietf.org mailing list, on the advice of one or more

Designated Experts. However, to allow for the allocation of values prior to publication, the Designated Expert(s) may approve registration once they are satisfied that such a specification will be published.

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

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

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

11.2.1. Registration Template

Parameter name:

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

Parameter usage location:

The location(s) where parameter can be used. The possible locations are: authorization request, authorization response, token request, or token response.

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 parameter, 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.

11.2.2. Initial Registry Contents

The OAuth Parameters Registry's initial contents are:

- o Parameter name: client_id
- o Parameter usage location: authorization request, token request
- o Change controller: IETF

- o Specification document(s): [[this document]]
- o Parameter name: client_secret
- o Parameter usage location: token request
- o Change controller: IETF
- o Specification document(s): [[this document]]
- o Parameter name: response_type
- o Parameter usage location: authorization request
- o Change controller: IETF
- o Specification document(s): [[this document]]
- o Parameter name: redirect_uri
- o Parameter usage location: authorization request, token request
- o Change controller: IETF
- o Specification document(s): [[this document]]
- o Parameter name: scope
- o Parameter usage location: authorization request, authorization response, token request, token response
- o Change controller: IETF
- o Specification document(s): [[this document]]
- o Parameter name: state
- o Parameter usage location: authorization request, authorization response
- o Change controller: IETF
- o Specification document(s): [[this document]]
- o Parameter name: code
- o Parameter usage location: authorization response, token request
- o Change controller: IETF
- o Specification document(s): [[this document]]
- o Parameter name: error_description
- o Parameter usage location: authorization response, token response
- o Change controller: IETF
- o Specification document(s): [[this document]]
- o Parameter name: error_uri
- o Parameter usage location: authorization response, token response
- o Change controller: IETF
- o Specification document(s): [[this document]]
- o Parameter name: grant_type
- o Parameter usage location: token request

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

- o Parameter name: access_token
- o Parameter usage location: authorization response, token response
- o Change controller: IETF
- o Specification document(s): [[this document]]

- o Parameter name: token_type
- o Parameter usage location: authorization response, token response
- o Change controller: IETF
- o Specification document(s): [[this document]]

- o Parameter name: expires_in
- o Parameter usage location: authorization response, token response
- o Change controller: IETF
- o Specification document(s): [[this document]]

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

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

- o Parameter name: refresh_token
- o Parameter usage location: token request, token response
- o Change controller: IETF
- o Specification document(s): [[this document]]

11.3. OAuth Authorization Endpoint Response Type Registry

This specification establishes the OAuth authorization endpoint response type registry.

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

Registration requests must be sent to the [TBD]@ietf.org mailing list for review and comment, with an appropriate subject (e.g., "Request

for response type: example"). [[Note to RFC-EDITOR: The name of the mailing list should be determined in consultation with the IESG and IANA. Suggested name: oauth-ext-review.]]

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

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

11.3.1. Registration Template

Response type 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 the document that specifies the type, 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.

11.3.2. Initial Registry Contents

The OAuth Authorization Endpoint Response Type Registry's initial contents are:

- o Response type name: code
- o Change controller: IETF
- o Specification document(s): [[this document]]

- o Response type name: token
- o Change controller: IETF
- o Specification document(s): [[this document]]

11.4. OAuth Extensions Error Registry

This specification establishes the OAuth extensions error registry.

Additional error codes used together with other protocol extensions (i.e. extension grant types, access token types, or extension parameters) are registered with a Specification Required ([RFC5226])

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

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

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

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

11.4.1. Registration Template

Error name:

The name requested (e.g., "example"). Values for the error name MUST NOT include characters outside the set %x20-21 / %x23-5B / %x5D-7E.

Error usage location:

The location(s) where the error can be used. The possible locations are: authorization code grant error response (Section 4.1.2.1), implicit grant error response (Section 4.2.2.1), token error response (Section 5.2), or resource access error response (Section 7.2).

Related protocol extension:

The name of the extension grant type, access token type, or extension parameter, the error code is used in conjunction with.

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 error code, 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.

12. References

12.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC2246] Dierks, T. and C. Allen, "The TLS Protocol Version 1.0", RFC 2246, January 1999.
- [RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.
- [RFC2617] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication", RFC 2617, June 1999.
- [RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.
- [RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.
- [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.
- [RFC4627] Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)", RFC 4627, July 2006.
- [RFC4949] Shirey, R., "Internet Security Glossary, Version 2", RFC 4949, August 2007.
- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.
- [RFC5234] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.
- [RFC6125] Saint-Andre, P. and J. Hodges, "Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer

Security (TLS)", RFC 6125, March 2011.

[USASCII] American National Standards Institute, "Coded Character Set -- 7-bit American Standard Code for Information Interchange", ANSI X3.4, 1986.

[W3C.REC-html401-19991224]
Hors, A., Raggett, D., and I. Jacobs, "HTML 4.01 Specification", World Wide Web Consortium Recommendation REC-html401-19991224, December 1999, <<http://www.w3.org/TR/1999/REC-html401-19991224>>.

[W3C.REC-xml-20081126]
Sperberg-McQueen, C., Yergeau, F., Paoli, J., Bray, T., and E. Maler, "Extensible Markup Language (XML) 1.0 (Fifth Edition)", World Wide Web Consortium Recommendation REC-xml-20081126, November 2008, <<http://www.w3.org/TR/2008/REC-xml-20081126>>.

12.2. Informative References

[I-D.draft-hardt-oauth-01]
Hardt, D., Ed., Tom, A., Eaton, B., and Y. Goland, "OAuth Web Resource Authorization Profiles", January 2010.

[I-D.ietf-oauth-saml2-bearer]
Campbell, B. and C. Mortimore, "SAML 2.0 Bearer Assertion Profiles for OAuth 2.0", draft-ietf-oauth-saml2-bearer-13 (work in progress), July 2012.

[I-D.ietf-oauth-v2-bearer]
Jones, M., Hardt, D., and D. Recordon, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", draft-ietf-oauth-v2-bearer-22 (work in progress), July 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.

[I-D.ietf-oauth-v2-threatmodel]
Lodderstedt, T., McGloin, M., and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations", draft-ietf-oauth-v2-threatmodel-06 (work in progress), June 2012.

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

April 2010.

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

This section provides Augmented Backus-Naur Form (ABNF) syntax descriptions for the elements defined in this specification using the notation of [RFC5234]. The ABNF below is defined in terms of Unicode code points [W3C.REC-xml-20081126]; these characters are typically encoded in UTF-8. Elements are presented in the order first defined.

Some of the definitions that follow use the "URI-reference" definition from [RFC3986].

Some of the definitions that follow use these common definitions:

```
VSCHAR      = %x20-7E
NQCHAR      = %x21 / %x23-5B / %x5D-7E
NQSCHAR     = %x20-21 / %x23-5B / %x5D-7E
UNICODECHARNOCLRF = %x09 / %x20-7E / %x80-D7FF /
                  %xE000-FFFF / %x10000-10FFFF
```

(The UNICODECHARNOCLRF definition is based upon the Char definition in Section 2.2 of [W3C.REC-xml-20081126], but omitting the Carriage Return and Linefeed characters.)

A.1. "client_id" Syntax

The "client_id" element is defined in Section 2.3.1:

```
client-id    = *VSCHAR
```

A.2. "client_secret" Syntax

The "client_secret" element is defined in Section 2.3.1:

```
client-secret = *VSCHAR
```

A.3. "response_type" Syntax

The "response_type" element is defined in Section 3.1.1 and Section 8.4:

```
response-type = response-name *( SP response-name )
response-name = 1*response-char
response-char = "_" / DIGIT / ALPHA
```

A.4. "scope" Syntax

The "scope" element is defined in Section 3.3:

```
scope          = scope-token *( SP scope-token )
scope-token    = 1*NQCHAR
```

A.5. "state" Syntax

The "state" element is defined in Section 4.1.1, Section 4.1.2, Section 4.1.2.1, Section 4.2.1, Section 4.2.2, and Section 4.2.2.1:

```
state          = 1*VSCHAR
```

A.6. "redirect_uri" Syntax

The "redirect_uri" element is defined in Section 4.1.1, Section 4.1.3, and Section 4.2.1:

```
redirect-uri    = URI-reference
```

A.7. "error" Syntax

The "error" element is defined in Section 4.1.2.1, Section 4.2.2.1, Section 5.2, Section 7.2, and Section 8.5:

```
error           = 1*NQSCHAR
```

A.8. "error_description" Syntax

The "error_description" element is defined in Section 4.1.2.1, Section 4.2.2.1, Section 5.2, and Section 7.2:

```
error-description = 1*NQSCHAR
```

A.9. "error_uri" Syntax

The "error_uri" element is defined in Section 4.1.2.1, Section 4.2.2.1, Section 5.2, and Section 7.2:

```
error-uri       = URI-reference
```

A.10. "grant_type" Syntax

The "grant_type" element is defined in Section 4.1.3, Section 4.3.2, Section 4.4.2, Section 6, and Section 4.5:

```
grant-type = grant-name / URI-reference
grant-name = 1*name-char
name-char  = "-" / "." / "_" / DIGIT / ALPHA
```

A.11. "code" Syntax

The "code" element is defined in Section 4.1.3:

```
code       = 1*VSCHAR
```

A.12. "access_token" Syntax

The "access_token" element is defined in Section 4.2.2 and Section 5.1:

```
access-token = 1*VSCHAR
```

A.13. "token_type" Syntax

The "token_type" element is defined in Section 4.2.2, Section 5.1, and Section 8.1:

```
token-type = type-name / URI-reference
type-name  = 1*name-char
name-char  = "-" / "." / "_" / DIGIT / ALPHA
```

A.14. "expires_in" Syntax

The "expires_in" element is defined in Section 4.2.2 and Section 5.1:

```
expires-in = 1*DIGIT
```

A.15. "username" Syntax

The "username" element is defined in Section 4.3.2:

```
username = *UNICODECHARNOCRLF
```

A.16. "password" Syntax

The "password" element is defined in Section 4.3.2:

```
password = *UNICODECHARNOCRLF
```

A.17. "refresh_token" Syntax

The "refresh_token" element is defined in Section 5.1 and Section 6:

```
refresh-token = 1*VSCHAR
```

A.18. Endpoint Parameter Syntax

The syntax for new endpoint parameters is defined in Section 8.2:

```
param-name = 1*name-char  
name-char  = "-" / "." / "_" / DIGIT / ALPHA
```

Appendix B. Use of application/x-www-form-urlencoded Media Type

At the time of publication of this specification, the "application/x-www-form-urlencoded" media type was defined in Section 17.13.4 of [W3C.REC-html401-19991224], but not registered in the IANA media types registry (<http://www.iana.org/assignments/media-types/index.html>). Furthermore, that definition is incomplete, as it does not consider non-US-ASCII characters.

To address this shortcoming when generating payloads using this media type, names and values MUST be encoded using the UTF-8 character encoding scheme [RFC3629] first; the resulting octet sequence then needs to be further encoded using the escaping rules defined in [W3C.REC-html401-19991224].

When parsing data from a payload using this media type, the names and values resulting from reversing the name/value encoding consequently need to be treated as octet sequences, to be decoded using the UTF-8 character encoding scheme.

For example, the value consisting of the six Unicode code points (1) U+0020 (SPACE), (2) U+0025 (PERCENT SIGN), (3) U+0026 (AMPERSAND), (4) U+002B (PLUS SIGN), (5) U+00A3 (POUND SIGN), and (6) U+20AC (EURO SIGN) would be encoded into the octet sequence below (using hexadecimal notation):

```
20 25 26 2B C2 A3 E2 82 AC
```

and then represented in the payload as:

+%25%26%2B%C2%A3%E2%82%AC

Appendix C. Acknowledgements

The initial OAuth 2.0 protocol specification was edited by David Recordon, based on two previous publications: the OAuth 1.0 community specification [RFC5849], and OAuth WRAP (OAuth Web Resource Authorization Profiles) [I-D.draft-hardt-oauth-01]. Eran Hammer then edited the drafts through draft -26. The Security Considerations section was drafted by Torsten Lodderstedt, Mark McGloin, Phil Hunt, Anthony Nadalin, and John Bradley. The section on use of the application/x-www-form-urlencoded media type was drafted by Julian Reschke. The ABNF section was drafted by Michael B. Jones.

The OAuth 1.0 community specification was edited by Eran Hammer and authored by Mark Atwood, Dirk Balfanz, Darren Bounds, Richard M. Conlan, Blaine Cook, Leah Culver, Breno de Medeiros, Brian Eaton, Kellan Elliott-McCrea, Larry Halff, Eran Hammer, Ben Laurie, Chris Messina, John Panzer, Sam Quigley, David Recordon, Eran Sandler, Jonathan Sergeant, Todd Sieling, Brian Slesinsky, and Andy Smith.

The OAuth WRAP specification was edited by Dick Hardt and authored by Brian Eaton, Yaron Y. Goland, Dick Hardt, and Allen Tom.

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

Michael Adams, Amanda Anganes, Andrew Arnott, Dirk Balfanz, Aiden Bell, John Bradley, Brian Campbell, Scott Cantor, Marcos Caceres, Blaine Cook, Roger Crew, Brian Eaton, Wesley Eddy, Leah Culver, Bill de hOra, Andre DeMarre, Brian Eaton, Wolter Eldering, Brian Ellin, Igor Faynberg, George Fletcher, Tim Freeman, Luca Frosini, Evan Gilbert, Yaron Y. Goland, Brent Goldman, Kristoffer Gronowski, Eran Hammer, Justin Hart, Dick Hardt, Craig Heath, Phil Hunt, Michael B. Jones, Terry Jones, John Kemp, Mark Kent, Raffi Krikorian, Chasen Le Hara, Rasmus Lerdorf, Torsten Lodderstedt, Hui-Lan Lu, Casey Lucas, Paul Madsen, Alastair Mair, Eve Maler, James Manger, Mark McGloin, Laurence Miao, William Mills, Chuck Mortimore, Anthony Nadalin, Julian Reschke, Justin Richer, Peter Saint-Andre, Nat Sakimura, Rob Sayre, Marius Scurtescu, Naitik Shah, Luke Shepard, Vlad Skvortsov, Justin Smith, Haibin Song, Niv Steingarten, Christian Stuebner, Jeremy Surriel, Paul Tarjan, Christopher Thomas, Henry S. Thompson, Allen Tom, Franklin Tse, Nick Walker, Shane Weeden, and Skylar

Woodward.

This document was produced under the chairmanship of Blaine Cook, Peter Saint-Andre, Hannes Tschofenig, Barry Leiba, and Derek Atkins. The area directors included Lisa Dusseault, Peter Saint-Andre, and Stephen Farrell.

Appendix D. Document History

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

-31

- o Clarify that any client can send "client_id" but that sending it is only required when using the code flow if the client is not otherwise authenticated.
- o Removed David Recordon's name from the author list, at his request.

-30

- o Added text explaining why the "server_error" and "temporarily_unavailable" error codes are needed.

-29

- o Added "MUST" to "A public client that was not issued a client password MUST use the "client_id" request parameter to identify itself when sending requests to the token endpoint" and added text explaining why this must be so.
- o Added that the authorization server MUST "ensure the authorization code was issued to the authenticated confidential client or to the public client identified by the "client_id" in the request".
- o Added Security Considerations section "Misuse of Access Token to Impersonate Resource Owner in Implicit Flow".
- o Added references in the "Implicit" and "Implicit Grant" sections to particularly pertinent security considerations.
- o Added appendix "Use of application/x-www-form-urlencoded Media Type" and referenced it in places that this encoding is used.
- o Deleted ";charset=UTF-8" from examples formerly using "Content-Type: application/x-www-form-urlencoded; charset=UTF-8".
- o Added the phrase "with a character encoding of UTF-8" when describing how to send requests using the HTTP request entity-body.
- o For symmetry when using HTTP Basic authentication, also apply the "application/x-www-form-urlencoded" encoding to the client password, just as was already done for the client identifier.
- o Added "The ABNF below is defined in terms of Unicode code points [W3C.REC-xml-20081126]; these characters are typically encoded in UTF-8".

- o Replaced UNICODENOCTRLCHAR in ABNF with UNICODECHARNOCTRLF = %x09 / %x20-7E / %x80-D7FF / %xE000-FFFF / %x10000-10FFFF.
- o Corrected incorrect uses of "which".
- o Reduced multiple blank lines around artwork elements to single blank lines.
- o Removed Eran Hammer's name from the author list, at his request. Dick Hardt is now listed as the editor.

-28

- o Updated the ABNF in the manner discussed by the working group, allowing "username" and "password" to be Unicode and restricting "client_id" and "client_secret" to ASCII.
- o Specified the use of the application/x-www-form-urlencoded content-type encoding method to encode the "client_id" when used as the password for HTTP Basic.

-27

- o Added character set restrictions for error, error_description, and error_uri parameters consistent with the OAuth Bearer spec.
- o Added "resource access error response" as an error usage location in the OAuth Extensions Error Registry.
- o Added an ABNF for all message elements.
- o Corrected editorial issues identified during review.

Author's Address

Dick Hardt (editor)
Microsoft

Email: dick.hardt@gmail.com
URI: <http://dickhardt.org/>

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: February 2, 2013

M. Jones
Microsoft
D. Hardt
independent
August 1, 2012

The OAuth 2.0 Authorization Framework: Bearer Token Usage
draft-ietf-oauth-v2-bearer-23

Abstract

This specification describes how to use bearer tokens in HTTP requests to access OAuth 2.0 protected resources. Any party in possession of a bearer token (a "bearer") can use it to get access to the associated resources (without demonstrating possession of a cryptographic key). To prevent misuse, bearer tokens need to be protected from disclosure in storage and in transport.

Status of this Memo

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

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

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

This Internet-Draft will expire on February 2, 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.

Table of Contents

1. Introduction	3
1.1. Notational Conventions	3
1.2. Terminology	3
1.3. Overview	4
2. Authenticated Requests	5
2.1. Authorization Request Header Field	5
2.2. Form-Encoded Body Parameter	6
2.3. URI Query Parameter	7
3. The WWW-Authenticate Response Header Field	8
3.1. Error Codes	9
4. Example Access Token Response	10
5. Security Considerations	10
5.1. Security Threats	11
5.2. Threat Mitigation	11
5.3. Summary of Recommendations	13
6. IANA Considerations	14
6.1. OAuth Access Token Type Registration	14
6.1.1. The "Bearer" OAuth Access Token Type	14
6.2. OAuth Extensions Error Registration	14
6.2.1. The "invalid_request" Error Value	15
6.2.2. The "invalid_token" Error Value	15
6.2.3. The "insufficient_scope" Error Value	15
7. References	16
7.1. Normative References	16
7.2. Informative References	17
Appendix A. Acknowledgements	17
Appendix B. Document History	18
Authors' Addresses	26

1. Introduction

OAuth enables clients to access protected resources by obtaining an access token, which is defined in OAuth 2.0 Authorization [I-D.ietf-oauth-v2] as "a string representing an access authorization issued to the client", rather than using the resource owner's credentials directly.

Tokens are issued to clients by an authorization server with the approval of the resource owner. The client uses the access token to access the protected resources hosted by the resource server. This specification describes how to make protected resource requests when the OAuth access token is a bearer token.

This specification defines the use of bearer tokens over HTTP/1.1 [RFC2616] using TLS [RFC5246] to access protected resources. TLS is mandatory to implement and use with this specification; other specifications may extend this specification for use with other protocols. While designed for use with access tokens resulting from OAuth 2.0 Authorization [I-D.ietf-oauth-v2] flows to access OAuth protected resources, this specification actually defines a general HTTP authorization method that can be used with bearer tokens from any source to access any resources protected by those bearer tokens. The Bearer authentication scheme is intended primarily for server authentication using the WWW-Authenticate and Authorization HTTP headers, but does not preclude its use for proxy authentication.

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 Key words for use in RFCs to Indicate Requirement Levels [RFC2119].

This document uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234]. Additionally, the following rules are included from HTTP/1.1 [RFC2617]: auth-param and auth-scheme; and from Uniform Resource Identifier (URI) [RFC3986]: URI-Reference.

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

1.2. Terminology

Bearer Token

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

All other terms are as defined in OAuth 2.0 Authorization [I-D.ietf-oauth-v2].

1.3. Overview

OAuth provides a method for clients to access a protected resource on behalf of a resource owner. In the general case, before a client can access a protected resource, it must first obtain an authorization grant from the resource owner and then exchange the authorization grant for an access token. The access token represents the grant's scope, duration, and other attributes granted by the authorization grant. The client accesses the protected resource by presenting the access token to the resource server. In some cases, a client can directly present its own credentials to an authorization server to obtain an access token without having to first obtain an authorization grant from a resource owner.

The access token provides an abstraction, replacing different authorization constructs (e.g., username and password, assertion) for a single token understood by the resource server. This abstraction enables issuing access tokens valid for a short time period, as well as removing the resource server's need to understand a wide range of authentication schemes.

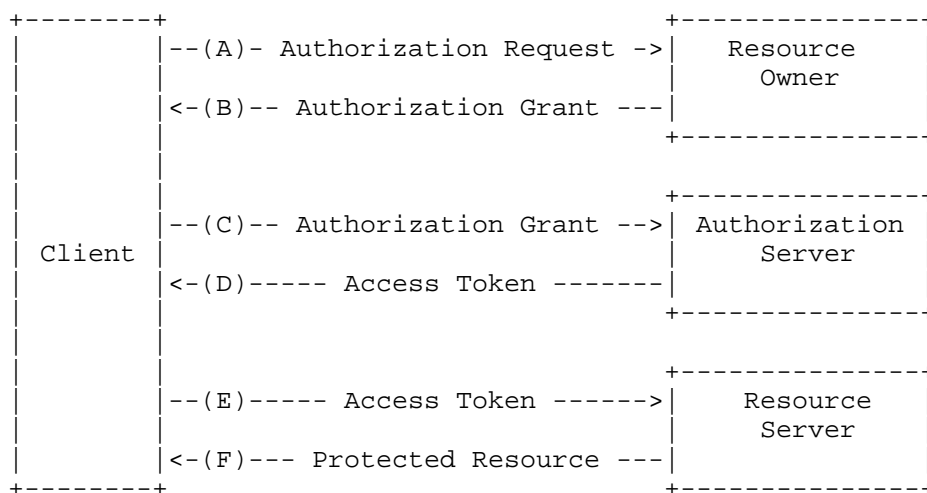


Figure 1: Abstract Protocol Flow

The abstract OAuth 2.0 flow illustrated in Figure 1 describes the interaction between the four roles. The following steps are specified within this document:

E) The client requests the protected resource from the resource server and authenticates by presenting the access token.

F) The resource server validates the access token, and if valid, serves the request.

This document also imposes semantic requirements upon the access token returned in Step D.

2. Authenticated Requests

This section defines three methods of sending bearer access tokens in resource requests to resource servers. Clients MUST NOT use more than one method to transmit the token in each request.

2.1. Authorization Request Header Field

When sending the access token in the "Authorization" request header field defined by HTTP/1.1 [RFC2617], the client uses the "Bearer" authentication scheme to transmit the access token.

For example:

```
GET /resource HTTP/1.1
Host: server.example.com
Authorization: Bearer mF_9.B5f-4.1JqM
```

The "Authorization" header field uses the framework defined by HTTP/1.1 [RFC2617] as follows:

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

Clients SHOULD make authenticated requests with a bearer token using the "Authorization" request header field with the "Bearer" HTTP authorization scheme. Resource servers MUST support this method.

2.2. Form-Encoded Body Parameter

When sending the access token in the HTTP request entity-body, the client adds the access token to the request body using the "access_token" parameter. The client MUST NOT use this method unless all of the following conditions are met:

- o The HTTP request entity-header includes the "Content-Type" header field set to "application/x-www-form-urlencoded".
- o The entity-body follows the encoding requirements of the "application/x-www-form-urlencoded" content-type as defined by HTML 4.01 [W3C.REC-html401-19991224].
- o The HTTP request entity-body is single-part.
- o The content to be encoded in the entity-body MUST consist entirely of ASCII [USASCII] characters.
- o The HTTP request method is one for which the request body has defined semantics. In particular, this means that the "GET" method MUST NOT be used.

The entity-body MAY include other request-specific parameters, in which case, the "access_token" parameter MUST be properly separated from the request-specific parameters using "&" character(s) (ASCII code 38).

For example, the client makes the following HTTP request using transport-layer security:

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

access_token=mF_9.B5f-4.1JqM
```

The "application/x-www-form-urlencoded" method SHOULD NOT be used except in application contexts where participating browsers do not have access to the "Authorization" request header field. Resource servers MAY support this method.

2.3. URI Query Parameter

When sending the access token in the HTTP request URI, the client adds the access token to the request URI query component as defined by Uniform Resource Identifier (URI) [RFC3986] using the "access_token" parameter.

For example, the client makes the following HTTP request using transport-layer security:

```
GET /resource?access_token=mF_9.B5f-4.1JqM HTTP/1.1
Host: server.example.com
```

The HTTP request URI query can include other request-specific parameters, in which case, the "access_token" parameter MUST be properly separated from the request-specific parameters using "&" character(s) (ASCII code 38).

For example:

```
https://server.example.com/resource?access_token=mF_9.B5f-4.1JqM&p=q
```

Clients using the URI Query Parameter method SHOULD also send a Cache-Control header containing the "no-store" option. Server success (2XX status) responses to these requests SHOULD contain a Cache-Control header with the "private" option.

Because of the security weaknesses associated with the URI method (see Section 5), including the high likelihood that the URL containing the access token will be logged, it SHOULD NOT be used unless it is impossible to transport the access token in the "Authorization" request header field or the HTTP request entity-body. Resource servers MAY support this method.

This method is included to document current use; its use is not recommended, both due to its security deficiencies (see Section 5) and because it uses a reserved query parameter name, which is counter to URI namespace best practices, per the Architecture of the World Wide Web [W3C.REC-webarch-20041215].

3. The WWW-Authenticate Response Header Field

If the protected resource request does not include authentication credentials or does not contain an access token that enables access to the protected resource, the resource server MUST include the HTTP "WWW-Authenticate" response header field; it MAY include it in response to other conditions as well. The "WWW-Authenticate" header field uses the framework defined by HTTP/1.1 [RFC2617].

All challenges defined by this specification MUST use the auth-scheme value "Bearer". This scheme MUST be followed by one or more auth-param values. The auth-param attributes used or defined by this specification are as follows. Other auth-param attributes MAY be used as well.

A "realm" attribute MAY be included to indicate the scope of protection in the manner described in HTTP/1.1 [RFC2617]. The "realm" attribute MUST NOT appear more than once.

The "scope" attribute is defined in Section 3.3 of OAuth 2.0 Authorization [I-D.ietf-oauth-v2]. The "scope" attribute is a space-delimited list of case sensitive scope values indicating the required scope of the access token for accessing the requested resource. "scope" values are implementation defined; there is no centralized registry for them; allowed values are defined by the authorization server. The order of "scope" values is not significant. In some cases, the "scope" value will be used when requesting a new access token with sufficient scope of access to utilize the protected resource. Use of the "scope" attribute is OPTIONAL. The "scope" attribute MUST NOT appear more than once. The "scope" value is intended for programmatic use and is not meant to be displayed to end users.

Two example scope values follow; these are taken from the OpenID Connect [OpenID.Messages] and OATC Online Multimedia Authorization Protocol [OMAP] OAuth 2.0 use cases, respectively:

```
scope="openid profile email"
scope="urn:example:channel=HBO&urn:example:rating=G,PG-13"
```

If the protected resource request included an access token and failed

authentication, the resource server SHOULD include the "error" attribute to provide the client with the reason why the access request was declined. The parameter value is described in Section 3.1. In addition, the resource server MAY include the "error_description" attribute to provide developers a human-readable explanation that is not meant to be displayed to end users. It also MAY include the "error_uri" attribute with an absolute URI identifying a human-readable web page explaining the error. The "error", "error_description", and "error_uri" attributes MUST NOT appear more than once.

Values for the "scope" attribute MUST NOT include characters outside the set %x21 / %x23-5B / %x5D-7E specified in Section A.4 of OAuth 2.0 Authorization [I-D.ietf-oauth-v2] for representing scope values and %x20 for delimiters between scope values. Values for the "error" and "error_description" attributes MUST NOT include characters outside the set %x20-21 / %x23-5B / %x5D-7E specified in Sections A.7 and A.8 of OAuth 2.0 Authorization. Values for the "error_uri" attribute MUST conform to the URI-Reference syntax, and thus MUST NOT include characters outside the set %x21 / %x23-5B / %x5D-7E specified in Section A.9 of OAuth 2.0 Authorization.

For example, in response to a protected resource request without authentication:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="example"
```

And in response to a protected resource request with an authentication attempt using an expired access token:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="example",
                  error="invalid_token",
                  error_description="The access token expired"
```

3.1. Error Codes

When a request fails, the resource server responds using the appropriate HTTP status code (typically, 400, 401, 403, or 405), and includes one of the following error codes in the response:

invalid_request

The request is missing a required parameter, includes an unsupported parameter or parameter value, repeats the same parameter, uses more than one method for including an access token, or is otherwise malformed. The resource server SHOULD respond with the HTTP 400 (Bad Request) status code.

invalid_token

The access token provided is expired, revoked, malformed, or invalid for other reasons. The resource SHOULD respond with the HTTP 401 (Unauthorized) status code. The client MAY request a new access token and retry the protected resource request.

insufficient_scope

The request requires higher privileges than provided by the access token. The resource server SHOULD respond with the HTTP 403 (Forbidden) status code and MAY include the "scope" attribute with the scope necessary to access the protected resource.

If the request lacks any authentication information (e.g., the client was unaware authentication is necessary or attempted using an unsupported authentication method), the resource server SHOULD NOT include an error code or other error information.

For example:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="example"
```

4. Example Access Token Response

Typically a bearer token is returned to the client as part of an OAuth 2.0 [I-D.ietf-oauth-v2] access token response. An example of such a response is:

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

{
  "access_token": "mF_9.B5f-4.1JqM",
  "token_type": "Bearer",
  "expires_in": 3600,
  "refresh_token": "tGzv3JOkF0XG5Qx2TlKWIA"
}
```

5. Security Considerations

This section describes the relevant security threats regarding token handling when using bearer tokens and describes how to mitigate these

threats.

5.1. Security Threats

The following list presents several common threats against protocols utilizing some form of tokens. This list of threats is based on NIST Special Publication 800-63 [NIST800-63]. Since this document builds on the OAuth 2.0 Authorization specification, we exclude a discussion of threats that are described there or in related documents.

Token manufacture/modification: An attacker may generate a bogus token or modify the token contents (such as the authentication or attribute statements) of an existing token, causing the resource server to grant inappropriate access to the client. For example, an attacker may modify the token to extend the validity period; a malicious client may modify the assertion to gain access to information that they should not be able to view.

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

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

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

5.2. Threat Mitigation

A large range of threats can be mitigated by protecting the contents of the token by using a digital signature or a Message Authentication Code (MAC). Alternatively, a bearer token can contain a reference to authorization information, rather than encoding the information directly. Such references **MUST** be infeasible for an attacker to guess; using a reference may require an extra interaction between a server and the token issuer to resolve the reference to the authorization information. The mechanics of such an interaction are not defined by this specification.

This document does not specify the encoding or the contents of the token; hence detailed recommendations about the means of guaranteeing token integrity protection are outside the scope of this document. The token integrity protection **MUST** be sufficient to prevent the token from being modified.

To deal with token redirect, it is important for the authorization server to include the identity of the intended recipients (the

audience), typically a single resource server (or a list of resource servers), in the token. Restricting the use of the token to a specific scope is also RECOMMENDED.

The authorization server MUST implement TLS. Which version(s) ought to be implemented will vary over time, and depend on the widespread deployment and known security vulnerabilities at the time of implementation. At the time of this writing, TLS version 1.2 [RFC5246] is the most recent version, but has very limited actual deployment, and might not be readily available in implementation toolkits. TLS version 1.0 [RFC2246] is the most widely deployed version, and will give the broadest interoperability.

To protect against token disclosure, confidentiality protection MUST be applied using TLS [RFC5246] with a ciphersuite that provides confidentiality and integrity protection. This requires that the communication interaction between the client and the authorization server, as well as the interaction between the client and the resource server, utilize confidentiality and integrity protection. Since TLS is mandatory to implement and to use with this specification, it is the preferred approach for preventing token disclosure via the communication channel. For those cases where the client is prevented from observing the contents of the token, token encryption MUST be applied in addition to the usage of TLS protection. As a further defense against token disclosure, the client MUST validate the TLS certificate chain when making requests to protected resources, including checking the Certificate Revocation List (CRL) [RFC5280].

Cookies are typically transmitted in the clear. Thus, any information contained in them is at risk of disclosure. Therefore, bearer tokens MUST NOT be stored in cookies that can be sent in the clear. See HTTP State Management Mechanism [RFC6265] for security considerations about cookies.

In some deployments, including those utilizing load balancers, the TLS connection to the resource server terminates prior to the actual server that provides the resource. This could leave the token unprotected between the front end server where the TLS connection terminates and the back end server that provides the resource. In such deployments, sufficient measures MUST be employed to ensure confidentiality of the token between the front end and back end servers; encryption of the token is one possible such measure.

To deal with token capture and replay, the following recommendations are made: First, the lifetime of the token MUST be limited; one means of achieving this is by putting a validity time field inside the protected part of the token. Note that using short-lived (one hour

or less) tokens reduces the impact of them being leaked. Second, confidentiality protection of the exchanges between the client and the authorization server and between the client and the resource server **MUST** be applied. As a consequence, no eavesdropper along the communication path is able to observe the token exchange. Consequently, such an on-path adversary cannot replay the token. Furthermore, when presenting the token to a resource server, the client **MUST** verify the identity of that resource server, as per Section 3.1 of HTTP Over TLS [RFC2818]. Note that the client **MUST** validate the TLS certificate chain when making these requests to protected resources. Presenting the token to an unauthenticated and unauthorized resource server or failing to validate the certificate chain will allow adversaries to steal the token and gain unauthorized access to protected resources.

5.3. Summary of Recommendations

Safeguard bearer tokens: Client implementations **MUST** ensure that bearer tokens are not leaked to unintended parties, as they will be able to use them to gain access to protected resources. This is the primary security consideration when using bearer tokens and underlies all the more specific recommendations that follow.

Validate TLS certificate chains: The client **MUST** validate the TLS certificate chain when making requests to protected resources. Failing to do so may enable DNS hijacking attacks to steal the token and gain unintended access.

Always use TLS (https): Clients **MUST** always use TLS [RFC5246] (https) or equivalent transport security when making requests with bearer tokens. Failing to do so exposes the token to numerous attacks that could give attackers unintended access.

Don't store bearer tokens in cookies: Implementations **MUST NOT** store bearer tokens within cookies that can be sent in the clear (which is the default transmission mode for cookies). Implementations that do store bearer tokens in cookies **MUST** take precautions against cross site request forgery.

Issue short-lived bearer tokens: Token servers **SHOULD** issue short-lived (one hour or less) bearer tokens, particularly when issuing tokens to clients that run within a web browser or other environments where information leakage may occur. Using short-lived bearer tokens can reduce the impact of them being leaked.

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

Don't pass bearer tokens in page URLs: Bearer tokens SHOULD NOT be passed in page URLs (for example as query string parameters). Instead, bearer tokens SHOULD be passed in HTTP message headers or message bodies for which confidentiality measures are taken. Browsers, web servers, and other software may not adequately secure URLs in the browser history, web server logs, and other data structures. If bearer tokens are passed in page URLs, attackers might be able to steal them from the history data, logs, or other unsecured locations.

6. IANA Considerations

6.1. OAuth Access Token Type Registration

This specification registers the following access token type in the OAuth Access Token Type Registry defined in OAuth 2.0 Authorization [I-D.ietf-oauth-v2].

6.1.1. The "Bearer" OAuth Access Token Type

Type name:
Bearer

Additional Token Endpoint Response Parameters:
(none)

HTTP Authentication Scheme(s):
Bearer

Change controller:
IETF

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

6.2. OAuth Extensions Error Registration

This specification registers the following error values in the OAuth Extensions Error Registry defined in OAuth 2.0 Authorization [I-D.ietf-oauth-v2].

6.2.1. The "invalid_request" Error Value

Error name:
invalid_request

Error usage location:
Resource access error response

Related protocol extension:
Bearer access token type

Change controller:
IETF

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

6.2.2. The "invalid_token" Error Value

Error name:
invalid_token

Error usage location:
Resource access error response

Related protocol extension:
Bearer access token type

Change controller:
IETF

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

6.2.3. The "insufficient_scope" Error Value

Error name:
insufficient_scope

Error usage location:
Resource access error response

Related protocol extension:
Bearer access token type

Change controller:
IETF

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

7. References

7.1. Normative References

- [I-D.ietf-oauth-v2]
Hardt, D., "The OAuth 2.0 Authorization Framework",
draft-ietf-oauth-v2-31 (work in progress), July 2012.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC2246] Dierks, T. and C. Allen, "The TLS Protocol Version 1.0",
RFC 2246, January 1999.
- [RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H.,
Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext
Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.
- [RFC2617] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S.,
Leach, P., Luotonen, A., and L. Stewart, "HTTP
Authentication: Basic and Digest Access Authentication",
RFC 2617, June 1999.
- [RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.
- [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform
Resource Identifier (URI): Generic Syntax", STD 66,
RFC 3986, January 2005.
- [RFC5234] Crocker, D. and P. Overell, "Augmented BNF for Syntax
Specifications: ABNF", STD 68, RFC 5234, January 2008.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security
(TLS) Protocol Version 1.2", RFC 5246, August 2008.
- [RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S.,
Housley, R., and W. Polk, "Internet X.509 Public Key
Infrastructure Certificate and Certificate Revocation List
(CRL) Profile", RFC 5280, May 2008.
- [RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265,

April 2011.

[USASCII] American National Standards Institute, "Coded Character Set -- 7-bit American Standard Code for Information Interchange", ANSI X3.4, 1986.

[W3C.REC-html401-19991224]
Hors, A., Raggett, D., and I. Jacobs, "HTML 4.01 Specification", World Wide Web Consortium Recommendation REC-html401-19991224, December 1999, <<http://www.w3.org/TR/1999/REC-html401-19991224>>.

[W3C.REC-webarch-20041215]
Jacobs, I. and N. Walsh, "Architecture of the World Wide Web, Volume One", World Wide Web Consortium Recommendation REC-webarch-20041215, December 2004, <<http://www.w3.org/TR/2004/REC-webarch-20041215>>.

7.2. Informative References

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

[OMAP] Huff, J., Schlacht, D., Nadalin, A., Simmons, J., Rosenberg, P., Madsen, P., Ace, T., Rickelton-Abdi, C., and B. Boyer, "Online Multimedia Authorization Protocol: An Industry Standard for Authorized Access to Internet Multimedia Resources", April 2012.

[OpenID.Messages]
Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., Mortimore, C., and E. Jay, "OpenID Connect Messages 1.0", June 2012.

Appendix A. Acknowledgements

The following people contributed to preliminary versions of this document: Blaine Cook (BT), Brian Eaton (Google), Yaron Y. Golan (Microsoft), Brent Goldman (Facebook), Raffi Krikorian (Twitter), Luke Shepard (Facebook), and Allen Tom (Yahoo!). The content and concepts within are a product of the OAuth community, the WRAP community, and the OAuth Working Group. David Recordon created a preliminary draft of this specification based upon a preliminary version of OAuth 2.0 draft 11. Michael B. Jones created draft 00 of this specification using portions of David's preliminary draft, and

edited all subsequent versions.

The OAuth Working Group has dozens of very active contributors who proposed ideas and wording for this document, including: Michael Adams, Amanda Anganes, Andrew Arnott, Derek Atkins, Dirk Balfanz, John Bradley, Brian Campbell, Francisco Corella, Leah Culver, Bill de hOra, Breno de Medeiros, Brian Ellin, Stephen Farrell, Igor Faynberg, George Fletcher, Tim Freeman, Evan Gilbert, Yaron Y. Goland, Thomas Hardjono, Justin Hart, Phil Hunt, John Kemp, Eran Hammer, Chasen Le Hara, Dick Hardt, Barry Leiba, Amos Jeffries, Michael B. Jones, Torsten Lodderstedt, Paul Madsen, Eve Maler, James Manger, Laurence Miao, William J. Mills, Chuck Mortimore, Anthony Nadalin, Axel Nennker, Mark Nottingham, David Recordon, Julian Reschke, Rob Richards, Justin Richer, Peter Saint-Andre, Nat Sakimura, Rob Sayre, Marius Scurtescu, Naitik Shah, Justin Smith, Jeremy Surriel, Christian Stuebner, Doug Tangren, Paul Tarjan, Hannes Tschofenig, Franklin Tse, Sean Turner, Paul Walker, Shane Weeden, Skylar Woodward, and Zachary Zeltsan.

Appendix B. Document History

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

-23

- o Removed David Recordon's name from the author list, at his request.

-22

- o Removed uses of HTTPbis in favor of RFC 2616 and RFC 2617, since HTTPbis is not an approved standard.
- o Match formatting of artwork elements with OAuth core specification.

-21

- o Changed "NOT RECOMMENDED" to "not recommended" in caveat about the URI Query Parameter method.
- o Changed "other specifications may extend this specification for use with other transport protocols" to "other specifications may extend this specification for use with other protocols".
- o Changed Acknowledgements to use only ASCII characters, per the RFC style guide.

-20

- o Added caveat about using a reserved query parameter name being counter to URI namespace best practices.
- o Specified use of Cache-Control options when using the URI Query Parameter method.
- o Changed title to "The OAuth 2.0 Authorization Framework: Bearer Token Usage".
- o Referenced syntax definitions for the "scope", "error", "error_description", and "error_uri" parameters in the OAuth 2.0 core spec.
- o Registered the "invalid_request", "invalid_token", and "insufficient_scope" error values in the OAuth Extensions Error Registry.
- o Acknowledged additional individuals.

-19

- o Addressed DISCUSS issues and comments raised for which resolutions have been agreed to. No normative changes were made. Changes made were:
- o Use ABNF from RFC 5234.
- o Added sentence "The Bearer authentication scheme is intended primarily for server authentication using the WWW-Authenticate and Authorization HTTP headers, but does not preclude its use for proxy authentication" to the introduction.
- o In the introduction, state that this document also imposes semantic requirements upon the access token.
- o Reference the "scope" definition in the OAuth core spec.
- o Added "scope" examples.
- o Reference RFC 6265 for security considerations about cookies.

-18

- o Changed example bearer token value from vF9dft4qmT to mF_9.B5f-4.1JqM.

- o Added example access token response returning a Bearer token.

-17

- o Restore RFC 2818 reference for server identity verification and add RFC 5280 reference for certificate revocation lists, per Gen-ART review comments.

-16

- o Use the HTTPbis auth-param syntax for Bearer challenge attributes.
- o Dropped the sentence "The "realm" value is intended for programmatic use and is not meant to be displayed to end users".
- o Reordered form-encoded body parameter description bullets for better readability.
- o Added [USASCII] reference.

-15

- o Clarified that form-encoded content must consist entirely of ASCII characters.
- o Added TLS version requirements.
- o Applied editorial improvements suggested by Mark Nottingham during the APPS area review.

-14

- o Changes made in response to review comments by Security Area Director Stephen Farrell. Specifically:
- o Strengthened warnings about passing an access token as a query parameter and more precisely described the limitations placed upon the use of this method.
- o Clarified that the "realm" attribute MAY included to indicate the scope of protection in the manner described in HTTP/1.1, Part 7 [I-D.ietf-httpbis-p7-auth].
- o Normatively stated that "the token integrity protection MUST be sufficient to prevent the token from being modified".
- o Added statement that "TLS is mandatory to implement and use with this specification" to the introduction.

- o Stated that TLS MUST be used with "a ciphersuite that provides confidentiality and integrity protection".
- o Added "As a further defense against token disclosure, the client MUST validate the TLS certificate chain when making requests to protected resources" to the Threat Mitigation section.
- o Clarified that putting a validity time field inside the protected part of the token is one means, but not the only means, of limiting the lifetime of the token.
- o Dropped the confusing phrase "for instance, through the use of TLS" from the sentence about confidentiality protection of the exchanges.
- o Reference RFC 6125 for identity verification, rather than RFC 2818.
- o Stated that the token MUST be protected between front end and back end servers when the TLS connection terminates at a front end server that is distinct from the actual server that provides the resource.
- o Stated that bearer tokens MUST NOT be stored in cookies that can be sent in the clear in the Threat Mitigation section.
- o Replaced sole remaining reference to [RFC2616] with HTTPbis [I-D.ietf-httpbis-pl-messaging] reference.
- o Replaced all references where the reference is used as if it were part of the sentence (such as "defined by [I-D.whatever]") with ones where the specification name is used, followed by the reference (such as "defined by Whatever [I-D.whatever]").
- o Other on-normative editorial improvements.

-13

- o At the request of Hannes Tschofenig, made ABNF changes to make it clear that no special WWW-Authenticate response header field parsers are needed. The "scope", "error-description", and "error-uri" parameters are all now defined as quoted-string in the ABNF (as "error" already was). Restrictions on these values that were formerly described in the ABNFs are now described in normative text instead.

-12

- o Made non-normative editorial changes that Hannes Tschofenig requested be applied prior to forwarding the specification to the IESG.
- o Added rationale for the choice of the b64token syntax.
- o Added rationale stating that receivers are free to parse the "scope" attribute using a standard quoted-string parser, since it will correctly process all legal "scope" values.
- o Added additional active working group contributors to the Acknowledgements section.

-11

- o Replaced uses of "<" with DQUOTE to pass ABNF syntax check.

-10

- o Removed the #auth-param option from Authorization header syntax (leaving only the b64token syntax).
- o Restricted the "scope" value character set to %x21 / %x23-5B / %x5D-7E (printable ASCII characters excluding double-quote and backslash). Indicated that scope is intended for programmatic use and is not meant to be displayed to end users.
- o Restricted the character set for "error_description" strings to SP / VCHAR and indicated that they are not meant to be displayed to end users.
- o Included more description in the Abstract, since Hannes Tschofenig indicated that the RFC editor would require this.
- o Changed "Access Grant" to "Authorization Grant", as was done in the core spec.
- o Simplified the introduction to the Authenticated Requests section.

-09

- o Incorporated working group last call comments. Specific changes were:
- o Use definitions from [I-D.ietf-httpbis-p7-auth] rather than [RFC2617].

- o Update credentials definition to conform to [I-D.ietf-httpbis-p7-auth].
- o Further clarified that query parameters may occur in any order.
- o Specify that error_description is UTF-8 encoded (matching the core specification).
- o Registered "Bearer" Authentication Scheme in Authentication Scheme Registry defined by [I-D.ietf-httpbis-p7-auth].
- o Updated references to oauth-v2, httpbis-pl-messaging, and httpbis-p7-auth drafts.
- o Other wording improvements not introducing normative changes.

-08

- o Updated references to oauth-v2 and HTTPbis drafts.

-07

- o Added missing comma in error response example.

-06

- o Changed parameter name "bearer_token" to "access_token", per working group consensus.
- o Changed HTTP status code for "invalid_request" error code from HTTP 401 (Unauthorized) back to HTTP 400 (Bad Request), per input from HTTP working group experts.

-05

- o Removed OAuth Errors Registry, per design team input.
- o Changed HTTP status code for "invalid_request" error code from HTTP 400 (Bad Request) to HTTP 401 (Unauthorized) to match HTTP usage [[change pending working group consensus]].
- o Added missing quotation marks in error-uri definition.
- o Added note to add language and encoding information to error_description if the core specification does.
- o Explicitly reference the Augmented Backus-Naur Form (ABNF) defined in [RFC5234].

- o Use auth-param instead of repeating its definition, which is (token "=" (token / quoted-string)).
- o Clarify security considerations about including an audience restriction in the token and include a recommendation to issue scoped bearer tokens in the summary of recommendations.

-04

- o Edits responding to working group last call feedback on -03. Specific edits enumerated below.
- o Added Bearer Token definition in Terminology section.
- o Changed parameter name "oauth_token" to "bearer_token".
- o Added realm parameter to "WWW-Authenticate" response to comply with [RFC2617].
- o Removed "[RWS 1#auth-param]" from "credentials" definition since it did not comply with the ABNF in [I-D.ietf-httpbis-p7-auth].
- o Removed restriction that the "bearer_token" (formerly "oauth_token") parameter be the last parameter in the entity-body and the HTTP request URI query.
- o Do not require WWW-Authenticate Response in a reply to a malformed request, as an HTTP 400 Bad Request response without a WWW-Authenticate header is likely the right response in some cases of malformed requests.
- o Removed OAuth Parameters registry extension.
- o Numerous editorial improvements suggested by working group members.

-03

- o Restored the WWW-Authenticate response header functionality deleted from the framework specification in draft 12 based upon the specification text from draft 11.
- o Augmented the OAuth Parameters registry by adding two additional parameter usage locations: "resource request" and "resource response".
- o Registered the "oauth_token" OAuth parameter with usage location "resource request".

- o Registered the "error" OAuth parameter.
- o Created the OAuth Error registry and registered errors.
- o Changed the "OAuth2" OAuth access token type name to "Bearer".

-02

- o Incorporated feedback received on draft 01. Most changes were to the security considerations section. No normative changes were made. Specific changes included:
- o Changed terminology from "token reuse" to "token capture and replay".
- o Removed sentence "Encrypting the token contents is another alternative" from the security considerations since it was redundant and potentially confusing.
- o Corrected some references to "resource server" to be "authorization server" in the security considerations.
- o Generalized security considerations language about obtaining consent of the resource owner.
- o Broadened scope of security considerations description for recommendation "Don't pass bearer tokens in page URLs".
- o Removed unused reference to OAuth 1.0.
- o Updated reference to framework specification and updated David Recordon's e-mail address.
- o Removed security considerations text on authenticating clients.
- o Registered the "OAuth2" OAuth access token type and "oauth_token" parameter.

-01

- o First public draft, which incorporates feedback received on -00 including enhanced Security Considerations content. This version is intended to accompany OAuth 2.0 draft 11.

-00

- o Initial draft based on preliminary version of OAuth 2.0 draft 11.

Authors' Addresses

Michael B. Jones
Microsoft

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

Dick Hardt
independent

Email: dick.hardt@gmail.com
URI: <http://dickhardt.org/>

OAuth
Internet-Draft
Intended status: Standards Track
Expires: July 19, 2014

J. Richer
The MITRE Corporation
W. Mills
Yahoo! Inc.
H. Tschofenig, Ed.

P. Hunt
Oracle Corporation
January 15, 2014

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

Abstract

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

The document also defines a key distribution protocol for obtaining a fresh session key.

Status of This Memo

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

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

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

This Internet-Draft will expire on July 19, 2014.

Copyright Notice

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

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

Table of Contents

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

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

1. Introduction

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

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

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

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

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

2. Terminology

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

This specification uses the Augmented Backus-Naur Form (ABNF) notation of [I-D.ietf-httpbis-pl-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.

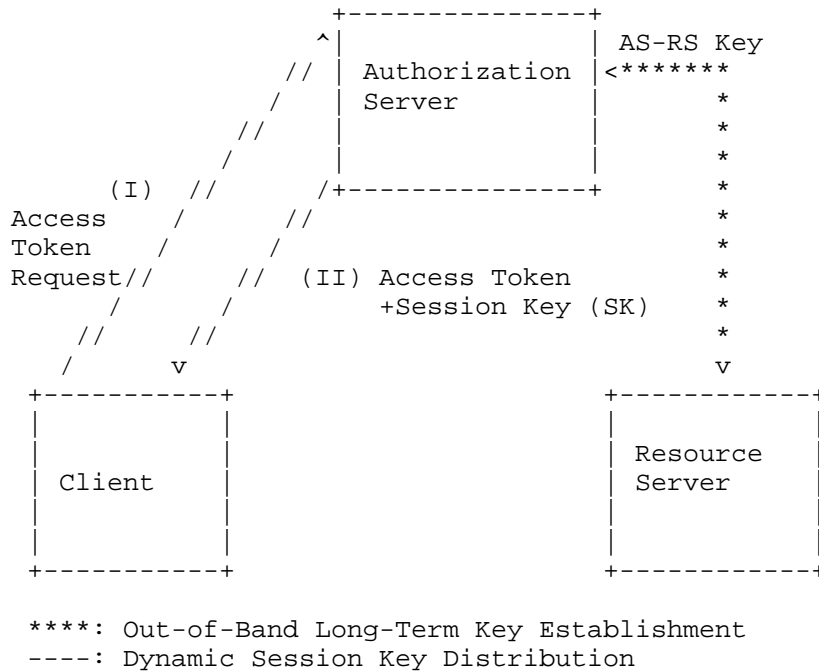


Figure 1: Architecture: Interaction between the Client and the Authorization Server.

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.

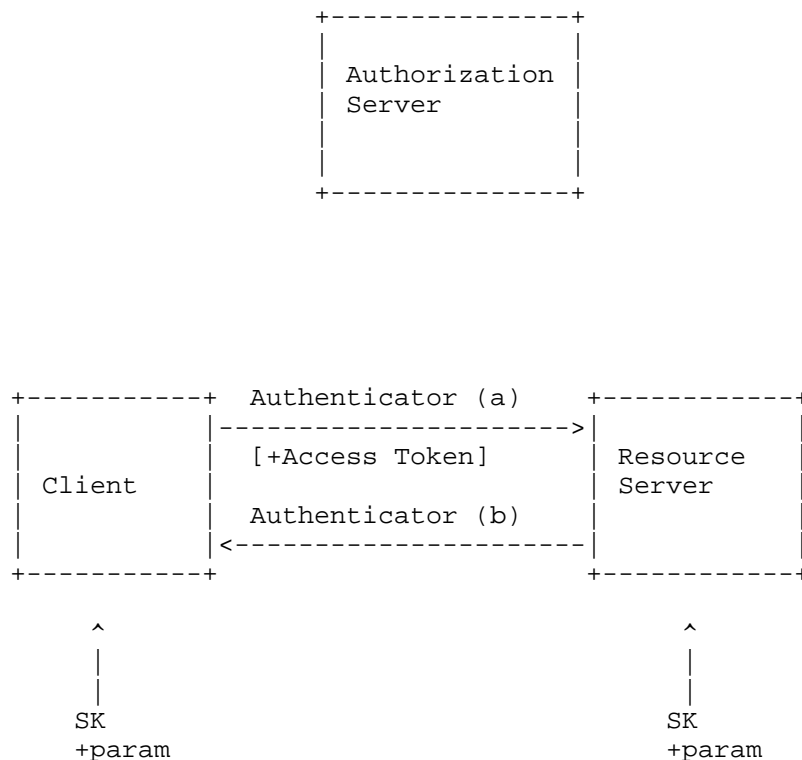


Figure 2: Architecture: Interaction between the Client and the Resource Server.

4. Key Distribution

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

4.1. Session Key Transport to Client

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

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

`kid`

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

`access_token`

The OAuth 2.0 access token.

`mac_key`

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

`mac_algorithm`

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

For example:

```

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

```

```

{
  "access_token":
    "eyJhbGciOiJSU0ExXzUiLCJlbmMiOiJBMTI4Q0JDK0hTMjU2In0.
    pwaFh7yJPivLjjPkzC-GeAyHuy7AinGcS5lAZ7TXnwkc80OwlaW47kcT_UV54ubo
    nONbeArwOVuR7shveXnwPmucwrk_3OCcHrCbE1HR-Jfme2mF_WR3zUMcwqmU0RlH
    kwx9txo_sKRasjlXc8RYP-evLCmT1XRKXjtY5l44Gnh0A84hGvVfMxMfCWXh38hi
    2h8JMjQHGG3mivVui5lbf-zzb3qXXxN0lZYOWgs5tP1-T54QYc9Bi9wodFPWNPKB
    kY-BgewG-Vmc59JqFeprk1008qhKQeOGCwc0WPC_n_LIpGWH6spRm7KGuYdgDMkQ
    bd4uuB0uPPLx_euVCdrVrA.
    AxY8DcTdaGlsbGljb3RoZQ.
    7MI2lRCaoyYx1HclVXkr8DhmDoikTmOp3IdEmm4qgBThFkqFqOs3ivXLJTku4M0f
    laMAbGG_X6K8_B-0E-7ak-Olm_-_V03oBUUGTAc-F0A.
    OwWNxnC-BMEie-GkFHzVWiNiaV3zUHf6fCOGTwbRckU",
    "token_type": "mac",
    "expires_in": 3600,
    "refresh_token": "8xLOxBtZp8",
    "kid": "22BIjxU93h/IgwEb4zCRu5WF37s=",
    "mac_key": "adijq39jdlaska9asud",
    "mac_algorithm": "hmac-sha-256"
}

```

4.2. Session Key Transport to Resource Server

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

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

`kid`

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

mac_key

The session key generated by the authorization server.

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

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

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

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

5. The Authenticator

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

5.1. The Authenticator

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

```

authenticator  = "MAC" 1*SP #params

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

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

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

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

```

The header attributes are set as follows:

kid

REQUIRED. The key identifier.

ts

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

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

seq-nr

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

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

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

access_token

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

mac

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

cb

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

h

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

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

5.2. MAC Input String

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

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

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

5.3. Keyed Message Digest Algorithms

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

5.3.1. hmac-sha-1

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

$$\text{mac} = \text{HMAC-SHA1}(\text{key}, \text{text})$$

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].

5.3.2. hmac-sha-256

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

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

Where:

text

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

key

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

mac

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

6. Verifying the Authenticator

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

1. When the authorization server receives a message with a new access token (and consequently a new session key) then it obtains the session key by retrieving the content of the access token (which requires decryption of the session key contained inside the token). The content of the access token, in particular the audience field and the scope, MUST be verified as described in Alternatively, the kid parameter is used to look-up a cached session key from a previous exchange.
2. Recalculate the keyed message digest, as described in Section 5.3, and compare the request MAC to the value received from the client via the "mac" attribute.

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

Error handling is described in Section 6.2.

6.1. Timestamp Verification

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

- o 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.
- o For each subsequent request, apply the request time delta to the timestamp included in the message to calculate the adjusted request time.
- o 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 capability to perform the asymmetric cryptographic operations. For a brief discussion about denial of service vulnerabilities of TLS please consult Appendix F.5 of RFC 5246 [RFC5246].

8.7. Timing Attacks

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

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

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

8.8. CSRF Attacks

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

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

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

8.9. Protecting HTTP Header Fields

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

9. IANA Considerations

9.1. JSON Web Token Claims

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

- o Claim Name: "kid"

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

9.2. MAC Token Algorithm Registry

This specification establishes the MAC Token Algorithm registry.

Additional keyed message digest algorithms are registered on the advice of one or more Designated Experts (appointed by the IESG or their delegate), with a Specification Required (using terminology from [RFC5226]). However, to allow for the allocation of values prior to publication, the Designated Expert(s) may approve registration once they are satisfied that such a specification will be published.

Registration requests should be sent to the [TBD]@ietf.org mailing list for review and comment, with an appropriate subject (e.g., "Request for MAC Algorithm: example"). [[Note to RFC-EDITOR: The name of the mailing list should be determined in consultation with the IESG and IANA. Suggested name: http-mac-ext-review.]]

Within at most 14 days of the request, the Designated Expert(s) will either approve or deny the registration request, communicating this decision to the review list and IANA. Denials should include an explanation and, if applicable, suggestions as to how to make the request successful.

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

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

9.2.1. Registration Template

Algorithm name:

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

Change controller:

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

Specification document(s):

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

9.2.2. Initial Registry Contents

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

- o Algorithm name: hmac-sha-1
- o Change controller: IETF
- o Specification document(s): [[this document]]
- o Algorithm name: hmac-sha-256
- o Change controller: IETF
- o Specification document(s): [[this document]]

9.3. OAuth Access Token Type Registration

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

9.3.1. The "mac" OAuth Access Token Type

Type name:

mac

Additional Token Endpoint Response Parameters:

secret, algorithm

HTTP Authentication Scheme(s):

MAC

Change controller:

IETF

Specification document(s):

[[this document]]

9.4. OAuth Parameters Registration

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

9.4.1. The "mac_key" OAuth Parameter

Parameter name: mac_key

Parameter usage location: authorization response, token response

Change controller: IETF

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

Related information: None

9.4.2. The "mac_algorithm" OAuth Parameter

Parameter name: mac_algorithm

Parameter usage location: authorization response, token response

Change controller: IETF

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

Related information: None

9.4.3. The "kid" OAuth Parameter

Parameter name: kid

Parameter usage location: authorization response, token response

Change controller: IETF

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

Related information: None

10. Acknowledgments

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

Further work in this document was done as part of OAuth working group conference calls late 2012/early 2013 and in design team conference calls February 2013. The following persons (in addition to the OAuth WG chairs, Hannes Tschofenig, and Derek Atkins) provided their input during these calls: Bill Mills, Justin Richer, Phil Hunt, Prateek Mishra, Mike Jones, George Fletcher, Leif Johansson, Lucy Lynch, John Bradley, Tony Nadalin, Klaas Wierenga, Thomas Hardjono, Brian Campbell

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

11. References

11.1. Normative References

- [I-D.ietf-httpbis-pl-messaging]
Fielding, R. and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", draft-ietf-httpbis-pl-messaging-25 (work in progress), November 2013.
- [I-D.ietf-jose-json-web-encryption]
Jones, M., Rescorla, E., and J. Hildebrand, "JSON Web Encryption (JWE)", draft-ietf-jose-json-web-encryption-19 (work in progress), December 2013.
- [I-D.ietf-oauth-json-web-token]
Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", draft-ietf-oauth-json-web-token-14 (work in progress), December 2013.
- [I-D.richer-oauth-introspection]
Richer, J., "OAuth Token Introspection", draft-richer-oauth-introspection-04 (work in progress), May 2013.
- [I-D.tschofenig-oauth-audience]

- Tschofenig, H., "OAuth 2.0: Audience Information", draft-tschofenig-oauth-audience-00 (work in progress), February 2013.
- [RFC2045] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, November 1996.
- [RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.
- [RFC2617] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication", RFC 2617, June 1999.
- [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.
- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.
- [RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, April 2011.
- [RFC6749] Hardt, D., "The OAuth 2.0 Authorization Framework", RFC 6749, October 2012.
- [W3C.REC-html401-19991224]
Hors, A., Raggett, D., and I. Jacobs, "HTML 4.01 Specification", World Wide Web Consortium Recommendation REC-html401-19991224, December 1999,
<<http://www.w3.org/TR/1999/REC-html401-19991224>>.

11.2. Informative References

- [I-D.hardjono-oauth-kerberos]
Hardjono, T., "OAuth 2.0 support for the Kerberos V5 Authentication Protocol", draft-hardjono-oauth-kerberos-01 (work in progress), December 2010.
- [I-D.tschofenig-oauth-hotk]
Bradley, J., Hunt, P., Nadalin, A., and H. Tschofenig, "The OAuth 2.0 Authorization Framework: Holder-of-the-Key Token Usage", draft-tschofenig-oauth-hotk-02 (work in progress), February 2013.
- [NIST-FIPS-180-3]
National Institute of Standards and Technology, "Secure Hash Standard (SHS). FIPS PUB 180-3, October 2008", October 2008.
- [NIST800-63]
Burr, W., Dodson, D., Perlner, R., Polk, T., Gupta, S., and E. Nabbus, "NIST Special Publication 800-63-1, INFORMATION SECURITY", December 2008.
- [RFC4086] Eastlake, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, June 2005.
- [RFC4120] Neuman, C., Yu, T., Hartman, S., and K. Raeburn, "The Kerberos Network Authentication Service (V5)", RFC 4120, July 2005.
- [RFC4279] Eronen, P. and H. Tschofenig, "Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)", RFC 4279, December 2005.
- [RFC4962] Housley, R. and B. Aboba, "Guidance for Authentication, Authorization, and Accounting (AAA) Key Management", BCP 132, RFC 4962, July 2007.
- [RFC5056] Williams, N., "On the Use of Channel Bindings to Secure Channels", RFC 5056, November 2007.
- [RFC5849] Hammer-Lahav, E., "The OAuth 1.0 Protocol", RFC 5849, April 2010.
- [RFC5929] Altman, J., Williams, N., and L. Zhu, "Channel Bindings for TLS", RFC 5929, July 2010.

- [RFC6125] Saint-Andre, P. and J. Hodges, "Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)", RFC 6125, March 2011.
- [RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, October 2012.

Appendix A. Background Information

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

A.1. Security and Privacy Threats

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

Token manufacture/modification:

An attacker may generate a bogus tokens or modify the token content (such as authentication or attribute statements) of an existing token, causing Resource Server to grant inappropriate access to the Client. For example, an attacker may modify the token to extend the validity period. A Client may modify the token to have access to information that they should not be able to view.

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

Token redirect:

An attacker uses the token generated for consumption by the Resource Server to obtain access to another Resource Server.

Token reuse:

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

We excluded one threat from the list, namely 'token repudiation'. Token repudiation refers to a property whereby a Resource Server is given an assurance that the Authorization Server cannot deny to have created a token for the Client. We believe that such a property is interesting but most deployments prefer to deal with the violation of this security property through business actions rather than by using cryptography.

A.2. Threat Mitigation

A large range of threats can be mitigated by protecting the content of the token, using a digital signature or a keyed message digest. Alternatively, the content of the token could be passed by reference rather than by value (requiring a separate message exchange to resolve the reference to the token content). To simplify the subsequent description we assume that the token itself is digitally signed by the Authorization Server and therefore cannot be modified.

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

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

To deal with token reuse more choices are available.

A.2.1. Confidentiality Protection

In this approach confidentiality protection of the exchange is provided on the communication interfaces between the Client and the Resource Server, and between the Client and the Authorization Server. No eavesdropper on the wire is able to observe the token exchange. Consequently, a replay by a third party is not possible. An Authorization Server wants to ensure that it only hands out tokens to Clients it has authenticated first and who are authorized. For this purpose, authentication of the Client to the Authorization Server will be a requirement to ensure adequate protection against a range of attacks. This is, however, true for the description in Appendix A.2.2 and Appendix A.2.3 as well. Furthermore, the Client has to make sure it does not distribute the access token to entities other than the intended the Resource Server. For that purpose the Client will have to authenticate the Resource Server before transmitting the access token.

A.2.2. Sender Constraint

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

- o security properties

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

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

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 $K(\text{Authorization Server-Resource Server})$. 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 K_s and places it into the token encrypted with the long term key $K(\text{Authorization Server-Resource Server})$. Additionally, the Authorization Server attaches K_s to the response message to the Client (in addition to the access token itself) over a confidentiality protected channel. When the Client sends a request to the Resource Server it has to use K_s to compute a keyed message digest for the request (in whatever form or whatever layer). The Resource Server, when receiving the message, retrieves the access token, verifies it and extracts $K(\text{Authorization Server-Resource Server})$ to obtain K_s . This key K_s is then used to verify the keyed message digest of the request message.

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

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

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 AS-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 phrased 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

OAuth Working Group
Internet-Draft
Intended status: Informational
Expires: April 9, 2013

T. Lodderstedt, Ed.
Deutsche Telekom AG
M. McGloin
IBM
P. Hunt
Oracle Corporation
October 6, 2012

OAuth 2.0 Threat Model and Security Considerations
draft-ietf-oauth-v2-threatmodel-08

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

described in the Simplified BSD License.

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. Redirection 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	24
4.4.1.	Authorization Code	24
4.4.1.1.	Threat: Eavesdropping or leaking authorization codes	24
4.4.1.2.	Threat: Obtain authorization codes from authorization server database	25
4.4.1.3.	Threat: Online guessing of authorization codes	26
4.4.1.4.	Threat: Malicious client obtains authorization	26
4.4.1.5.	Threat: Authorization code phishing	28
4.4.1.6.	Threat: User session impersonation	28
4.4.1.7.	Threat: Authorization code leakage through counterfeit client	29
4.4.1.8.	Threat: CSRF attack against redirect-uri	31
4.4.1.9.	Threat: Clickjacking attack against authorization	31
4.4.1.10.	Threat: Resource Owner Impersonation	32
4.4.1.11.	Threat: DoS, Exhaustion of resources attacks	33
4.4.1.12.	Threat: DoS using manufactured authorization codes	34
4.4.1.13.	Threat: Code substitution (OAuth Login)	35
4.4.2.	Implicit Grant	36
4.4.2.1.	Threat: Access token leak in transport/end-points	36
4.4.2.2.	Threat: Access token leak in browser history	37
4.4.2.3.	Threat: Malicious client obtains authorization	37
4.4.2.4.	Threat: Manipulation of scripts	37
4.4.2.5.	Threat: CSRF attack against redirect-uri	38
4.4.2.6.	Threat: Token substitution (OAuth Login)	38
4.4.3.	Resource Owner Password Credentials	39
4.4.3.1.	Threat: Accidental exposure of passwords at client site	40
4.4.3.2.	Threat: Client obtains scopes without end-user authorization	40
4.4.3.3.	Threat: Client obtains refresh token through automatic authorization	41
4.4.3.4.	Threat: Obtain user passwords on transport	42
4.4.3.5.	Threat: Obtain user passwords from authorization server database	42
4.4.3.6.	Threat: Online guessing	42
4.4.4.	Client Credentials	43
4.5.	Refreshing an Access Token	43
4.5.1.	Threat: Eavesdropping refresh tokens from authorization server	43
4.5.2.	Threat: Obtaining refresh token from authorization server database	43
4.5.3.	Threat: Obtain refresh token by online guessing	44
4.5.4.	Threat: Obtain refresh token phishing by counterfeit authorization server	44

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.	Utiliize 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:

- o Documents any assumptions and scope considered when creating the threat model.
- o Describes the security features in-built into the OAuth protocol and how they are intended to thwart attacks.
- o 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:

- o Resource server URLs are static and well-known at development time, authorization server URLs can be static or discovered.
- o Token scope values (e.g. applicable URLs and methods) are well-known at development time.
- o 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 :

- o Communication between authorization server and resource server
- o Token formats
- o 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
- o Mechanism by which a user obtained an assertion and any resulting attacks mounted as a result of the assertion being false.
- o 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:

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

to resources.

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:

- o user names and passwords
- o client ids and secrets
- o client-specific refresh tokens
- o client-specific access tokens (in case of handle-based design - see Section 3.1)
- o HTTPS certificate/key
- o 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:

- o user data (out of scope)
- o 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 a 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

(e.g.[I-D.ietf-oauth-v2-http-mac]).

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

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

- o Web server: apply standard web server protection measures (for config files and databases) - Section 5.3.2
- o Native applications: Store secrets in a secure local storage - Section 5.3.3
- o Revoke client secrets - Section 5.2.3.6

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:

- o The authorization server should validate the client id associated with the particular refresh token with every refresh request - Section 5.2.2.2
- o Limit token scope - Section 5.1.5.1
- o Revoke refresh tokens - Section 5.2.2.4
- o Revoke client secrets - Section 5.2.3.6
- o 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:

- o Standard web server protection measures - Section 5.3.2
- o 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:

- o Store secrets in a secure storage - Section 5.3.3
- o 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:

- o Utilize device lock to prevent unauthorized device access - Section 5.3.4
- o 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:

- o Utilize device lock to prevent unauthorized device access - Section 5.3.4
- o Combine refresh token request with device identification - Section 5.2.2.5
- o Refresh Token Rotation - Section 5.2.2.3

- o 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:

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

- o 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).
- o 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:

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

- o 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)
- o 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

- o require clients to register full redirection URI Section 5.2.3.5
- o 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:

- o 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).

- o 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.

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:

- o Enforce system security measures - Section 5.1.4.1.1
- o Store access token hashes only - Section 5.1.4.1.3
- o 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:

- o The transmission of client credentials must be protected using transport-layer mechanisms such as TLS (see Section 5.1.1).
- o 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:

- 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 "referrer" 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.

- o 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:

- o 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).
- o 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).
- o Use short expiry time for authorization codes - Section 5.1.5.3
- o The authorization server should enforce a one time usage restriction (see Section 5.1.5.4).
- o 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).
- o 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.
- o 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:

- o Best practices for credential storage protection should be employed - Section 5.1.4.1
- o Enforce system security measures - Section 5.1.4.1.1
- o Store access token hashes only - Section 5.1.4.1.3
- o 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:

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

- o 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.
- o 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.
- o 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).
- o 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).
- o 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.

- o The authorization server may also limit the scope of tokens it issues to clients it cannot reliably authenticate (see Section 5.1.5.1).

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:

- o 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).
- o 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:

- o 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 as associated with his account on the client site.

Countermeasures:

- o 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).
- o 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.
- o 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.

- o 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).

4.4.1.8. 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:

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

4.4.1.9. Threat: Clickjacking attack against authorization

With Clickjacking, a malicious site loads the target site in a transparent iFrame (see [iFrame]) overlaid 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

- o 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
- o 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:

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

- o combine password authentication and user consent in a single form,
- o make use of CAPTCHAs, or
- o 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:

- o 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:

- o 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.
- o 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:

- o 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:

- o 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).
- o 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:

- o The authorization server should authenticate the server from which scripts are obtained (see Section 5.1.2).
- o The client should ensure that scripts obtained have not been altered in transport (see Section 5.1.1).
- o 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:

- o 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 end-points could eavesdrop user id and password.

Countermeasures:

- o Except for migration reasons, minimize use of this grant type
- o The authorization server should validate the client id associated with the particular refresh token with every refresh request - Section 5.2.2.2
- o 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).
- o 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.
- o 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:

- o Use other flows, which do not rely on the client's cooperation for secure resource owner credential handling
- o Use digest authentication instead of plaintext credential processing
- o 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:

- o Use other flows, which do not rely on the client's cooperation for resource owner interaction
- o 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.
- o 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:

- o Use other flows, which do not rely on the client's cooperation for resource owner interaction
- o 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.
- o 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:

- o Ensure confidentiality of requests - Section 5.1.1
- o 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:

- o 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:

- o Utilize secure password policy - Section 5.1.4.2.1
- o Lock accounts - Section 5.1.4.2.3
- o Use tar pit - Section 5.1.4.2.4
- o Use CAPTCHAs - Section 5.1.4.2.5

- o Consider not to use grant type "password"
- o Client authentication (see Section 5.2.3) will provide another authentication factor and thus hinder the attack.

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:

- o 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).
- o 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:

- o Enforce credential storage protection best practices - Section 5.1.4.1
- o 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:

- o For handle-based designs - Section 5.1.4.2.2
- o For assertion-based designs - Section 5.1.5.9
- o Bind token to client id, because the attacker would guess the matching client id, too (see Section 5.1.5.8)
- o 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:

- o 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:

- o 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).

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

- o 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.
- o 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:

- o Handle Tokens should have a reasonable entropy (see Section 5.1.4.2.2) in order to make guessing a valid token value infeasible.
- o Assertion (or self-contained token) tokens contents should be protected by a digital signature (see Section 5.1.5.9).
- o 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:

- o 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).
- o 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.
- o 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).
- o 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:

- o 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:

- o 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].
- o 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 "referrer".

Countermeasures:

- o Use authorization headers or POST parameters instead of URI request parameters (see Section 5.4.1).
- o Set logging configuration appropriately
- o Prevent unauthorized persons from access to system log files (see Section 5.1.4.1.1)
- o Abuse of leaked access tokens can be prevented by enforcing authenticated requests (see Section 5.4.2).
- o 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 be 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:

- o Replay of access tokens obtained on tokens endpoint or resource server's endpoint
- o Replay of refresh tokens obtained on tokens endpoint
- o Replay of authorization codes obtained on tokens endpoint (redirect?)
- o 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:

- o Spoofing
- o Proxying
- o 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:

- o User consent forms
- o 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.
- o Activity/Event logs
- o User self-care applications or portals

5.1.4. Credentials

This sections 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.

- o Ensure that server code is using the minimum database privileges possible to reduce the "surface" of possible attacks.
- o Avoid dynamic SQL using concatenated input. If possible, use static SQL.
- o When using dynamic SQL, parameterize queries using bind arguments. Bind arguments eliminate possibility of SQL injections.
- o 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. Usa 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:

- o a client-specific policy, e.g. issue only less powerful tokens to public clients,
- o a service-specific policy, e.g. it a very sensitive service,
- o a resource-owner specific setting, or
- o 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:

- o token leakage
- o token issuance to malicious software
- o 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:

- o It reduces the impact of a successful replay attempt, since the token is applicable to a single resource server, only.
- o 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.
- o 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

- o detect token leakage and
- o 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

- o keep them in transient memory (accessible by the client application only)
- o Pass tokens securely using secure transport (TLS)
- o 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:

- o device theft,
- o impersonation of resource owner, or
- o 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:

- o 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:

- o 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:

- o Cross-site scripting attacks
- o 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:

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

- o 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:

- o Malicious application
- o 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:

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

There are different ways 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 implement 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 in turn requires a resilient authentication mechanism 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 biometric 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:

- o 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.
- o 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)
- o 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:

- o modifications of the message and
- o 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.

- o 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.
- o 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.
- o 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.
- o 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.
- o 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.
- [OASIS.saml-core-2.0-os]
Cantor, S., Kemp, J., Philpott, R., and E. Maler,
"Assertions and Protocol for the OASIS Security Assertion
Markup Language (SAML) V2.0", OASIS Standard saml-core-
2.0-os, March 2005.
- [OASIS.sstc-gross-sec-analysis-response-01]
Linn, J., Ed. and P. Mishra, Ed., "SSTC Response to
"Security Analysis of the SAML Single Sign-on Browser/
Artifact Profile"", January 2005.
- [OASIS.sstc-saml-bindings-1.1]
Maler, E., Ed., Mishra, P., Ed., and R. Philpott, Ed.,
"Bindings and Profiles for the OASIS Security Assertion
Markup Language (SAML) V1.1", September 2003.
- [RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H.,
Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext
Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.
- [RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.
- [RFC4086] Eastlake, D., Schiller, J., and S. Crocker, "Randomness
Requirements for Security", BCP 106, RFC 4086, June 2005.
- [RFC4120] Neuman, C., Yu, T., Hartman, S., and K. Raeburn, "The
Kerberos Network Authentication Service (V5)", RFC 4120,
July 2005.
- [RFC4301] Kent, S. and K. Seo, "Security Architecture for the
Internet Protocol", RFC 4301, December 2005.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security
(TLS) Protocol Version 1.2", RFC 5246, August 2008.
- [framebusting]
Rydstedt, G., Bursztein, Boneh, D., and C. Jackson,
"Busting Frame Busting: a Study of Clickjacking
Vulnerabilities on Popular Sites", IEEE 3rd Web 2.0
Security and Privacy Workshop, 2010.
- [gross-sec-analysis]
Gross, T., "Security Analysis of the SAML Single Sign-on

Browser/Artifact Profile, 19th Annual Computer Security Applications Conference, Las Vegas", December 2003.

- [iFrame] World Wide Web Consortium, "Frames in HTML documents", W3C HTML 4.01, Dec 1999.
- [openid] "OpenID Foundation Home Page", <<http://openid.net/>>.
- [owasp] "Open Web Application Security Project Home Page", <<https://www.owasp.org/>>.
- [portable-contacts] Smarr, J., "Portable Contacts 1.0 Draft C", August 2008, <<http://portablecontacts.net/>>.
- [ssl-latency] Sissel, J., Ed., "SSL handshake latency and HTTPS optimizations", June 2010.

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 Incorporated 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.mcglain@ie.ibm.com

Phil Hunt
Oracle Corporation

Email: phil.hunt@yahoo.com

Internet-Draft
Intended status: Standards Track
Expires: December 18, 2011

C. Mortimore, Ed.
Salesforce
M. Jones
MSFT
B. Campbell
Ping
Y. Goland
MSFT
June 16, 2011

OAuth 2.0 Assertion Profile
draft-mortimore-oauth-assertions-00

Abstract

This specification provides a general framework for the use of assertions as client credentials and/or authorization grants with OAuth 2.0. It includes a generic mechanism for transporting assertions during interactions with a token endpoint, as well as rules for the content and processing of those assertions. The intent is to provide an enhanced security profile by using derived values such as signatures or HMACs, as well as 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 flow and assertion content. Actual use requires implementation of a companion protocol binding specification. Additional profile documents provide standard representations in formats such as SAML and JWT.

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 December 18, 2011.

Copyright Notice

Copyright (c) 2011 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. Requirements Notation and Conventions	3
2. Overview	3
3. Authentication vs Authorization	4
4. Transporting Assertions	4
4.1. Using Assertions for Client Authentication	4
4.2. Using Assertions as Authorization Grants	5
5. Assertion Content and Processing	6
5.1. Assertion Metamodel	7
5.2. General Assertion Format and Processing Rules	8
6. Specific Assertion Format and Processing Rules	8
6.1. Client authentication	9
6.2. Client acting on behalf of itself	9
6.3. Client acting on behalf of a user	11
6.4. Client acting on behalf of an anonymous user	12
7. Error Responses	13
8. Security Considerations	13
9. Acknowledgements	14
10. Normative References	14
Authors' Addresses	14

1. Requirements Notation and Conventions

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

2. Overview

The OAuth 2.0 Authorization Protocol [I-D.ietf.oauth-v2] provides a method for making authenticated HTTP requests to a resource using an access token. 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 representing authorization by the resource owner or privileged administrator. 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.

In scenarios where security is at a premium one wants to avoid sending secrets across the Internet, even on encrypted connections. Instead one wants to send values derived from the secret that prove to the receiver that the sender is in possession of the secret without actually sending the secret. Typically the way derived values are created is by generating an assertion that is then either HMAC'd or digitally signed using an agreed upon secret. By validating the HMAC or digital signature on the assertion, the receiver can prove to themselves that the entity that generated the assertion was in possession of the secret without actually communicating the secret directly.

This specification provides a general framework for the use of assertions as client credentials and/or authorization grants with OAuth 2.0. It includes a generic mechanism for transporting assertions during interactions with a token endpoint, as well as rules for the content and processing of those assertions. The intent is to provide an enhanced security profile by using derived values such as signatures or HMACs, as well as 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 flow and assertion content. Actual use requires implementation of a companion protocol binding specification. Additional profile documents provide standard representations in formats such as SAML and JWT.

3. Authentication vs Authorization

This specification provides a model for using assertions for authentication of an OAuth client during interactions with an Authorization Server, as well as the use of assertions as authorization grants. It is important to note that the use of assertions for client authentication is orthogonal and separable from using assertions as an authorization grant and can be used either in combination or in isolation. In addition, in scenarios when assertion based authentication and authorization are used in combination, the assertion format and processing may be redundant; under such circumstances, the protocol may be optimized to present a single assertion.

4. Transporting Assertions

This section defines generic HTTP parameters for transporting assertions during interactions with a token endpoint.

4.1. Using Assertions for Client Authentication

In scenarios where one wants to avoid sending secrets, one wants to send values derived from the secret that prove to the receiver that the sender is in possession of the secret without actually sending the secret.

For example, a client can establish a secret using an out-of-band mechanism with a resource server. As part of this out-of-band communication the client and resource server agree that the client will authenticate itself using an assertion with agreed upon parameters that will be signed by the provisioned secret. Later on, the client might send an access token request to the token endpoint for the resource server that includes an authorization code, as well as a client_assertion that is signed with the previously agreed key and parameters. The client_assertion proves to the token endpoint the identity of the client and the authorization code provides the link to the end-user authorization.

The following section defines the use of assertions as client credentials as an extension of Section 3.2 of OAuth 2.0 [I-D.ietf.oauth-v2]. When using assertions as client credentials,

the client MUST include the assertion using the following HTTP request parameters:

`client_id` REQUIRED. The client identifier as described in Section 3 of OAuth 2.0 [I-D.ietf.oauth-v2].

`client_assertion_type` REQUIRED. The format of the assertion as defined by the authorization server. The value MUST 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. The serialization MUST be encoded for transport within HTTP forms. It is RECOMMENDED that base64url be used.

The following non-normative example demonstrates a client authenticating using an assertion during a Authorization Code Access Token Request as defined in Section 4.1.3 of OAuth 2.0 [I-D.ietf.oauth-v2]. (line breaks are for display purposes only):

```
POST /token HTTP/1.1
```

```
Host: server.example.com
```

```
Content-Type: application/x-www-form-urlencoded
```

```
grant_type=authorization_code&
```

```
code=i1WsRnluB1&
```

```
client_id=s6BhdRkqt3&
```

```
client_assertion_type=urn%3Aoasis%3Anames%3Atc%3ASAML%3A2.0%3Aassertion&
```

```
client_assertion=PHNhbWxwOl...[omitted for brevity]...ZT
```

The client MUST NOT include the `client_credential` using more than one mechanism. Token endpoints can differentiate between client assertion credentials and other client credential types by looking for the presence of the `client_assertion` and `client_assertion_type` attributes which will only be present with client assertion credentials. See section 7 for more details

4.2. Using Assertions as Authorization Grants

An assertion can be used to request an access token when a client wishes to utilize an existing trust relationship. This may be done through the semantics of (and a digital signature/HMAC calculated over) the assertion, without direct user approval at the authorization server, and expressed through an extension authorization grant type. The processes by which authorization is previously granted, and by which the client obtains the assertion prior to exchanging it with the authorization server, are out of scope.

The following defines the use of assertions as authorization grants as an extension of OAuth 2.0 [I-D.ietf.oauth-v2], section 4.5. When using assertions as authorization grants, the client MUST include the assertion using the following HTTP request parameters:

`client_id` REQUIRED. The client identifier as described in Section 3 of OAuth 2.0 [I-D.ietf.oauth-v2].

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

`assertion` REQUIRED. The assertion being used as an authorization grant. Specific serialization of the assertion is defined by profile documents. The serialization MUST be encoded for transport within HTTP forms. It is RECOMMENDED that base64url be used.

`scope` OPTIONAL. The request MAY contain a "scope" parameter. The scope of the access request is expressed as a list of space-delimited strings. The value is defined by the authorization server. If the value contains multiple space-delimited strings, their order does not matter, and each string adds an additional access range to the requested scope. When exchanging assertions for access_tokens, the authorization for the token has been previously granted through some other mechanism. As such, the requested scope SHOULD be equal or lesser than the scope originally granted to the authorized accessor. If the scope parameter and/or value is omitted, the scope SHOULD be treated as equal to the scope originally granted to the authorized accessor. The Authorization Server SHOULD limit the scope of the issued access token to be equal or lesser than the scope originally granted to the authorized accessor.

The following non-normative example demonstrates an assertion being used as an authorization grant. (line breaks are for display purposes only):

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

```
client_id=s6BhdRkqt3&
grant_type=urn%3Aoasis%3Anames%3Atc%3ASAML%3A2.0%3Aassertion&
assertion=PHNhbwXwOl...[omitted for brevity]...ZT4
```

5. Assertion Content and Processing

This section provides a general content and processing model for the

use of assertions in OAuth 2.0 [I-D.ietf.oauth-v2].

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 The unique identifier for the entity that issued the assertion. Generally this is the entity that holds the keying material used to generate the assertion. In some use-cases Issuers may be either OAuth Clients (when assertions are self-asserted) or a Security Token Service (an entity capable of issuing, renewing, transforming and validating of security tokens).

Principal A unique identifier for the subject of the assertion. When using assertions for client authentication, the Principal SHOULD be the client_id of the OAuth client. When using assertions as an authorization grant, the Principal MUST identify an authorized accessor for whom the access token is being requested (typically the resource owner, or an authorized delegate).

Audience A URI that identifies the Authorization Server as the intended audience. The audience SHOULD be the URL of the Token Endpoint as defined in section 2.2 of OAuth 2.0 [I-D.ietf.oauth-v2].

Issued At The time at which the assertion was issued. While the serialization may differ by assertion format, this is always 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, this is always expressed in UTC with no time zone component.

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

5.2. General Assertion Format and Processing Rules

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

- o The assertion MUST contain an Issuer. The Issuer MUST identify the entity that issued the assertion as recognized by the Authorization Server. If an assertion is self-asserted, the Issuer SHOULD be the client_id.
- o The assertion SHOULD contain a Principal. The Principal MUST identify an authorized accessor for whom the access token is being requested (typically the resource owner, or an authorized delegate) When the client is acting on behalf of itself, the Principal SHOULD be the client_id.
- o The assertion MUST contain an Audience that identifies the Authorization Server as the intended audience. The Authorization Server MUST verify that it is an intended audience for the assertion. The Audience SHOULD be the URL of the Authorization Server's Token Endpoint.
- o The assertion MUST contain an Expires At entity that limits the time window during which the assertion can be used. The authorization server MUST verify that the expiration time has not passed, subject to allowable clock skew between systems. The authorization server SHOULD reject assertions with an Expires At attribute value that is unreasonably far in the future.
- o The assertion MAY contain an Issued At entity containing the UTC time at which the assertion was issued.
- o The assertion MAY contain a Assertion ID. An Authorization Server MAY dictate that Assertion ID is mandatory.
- o The Authorization Server MUST validate the assertion in order to establish a mapping between the Issuer and the secret used to generate the assertion. The algorithm used to validate the assertion, and the mechanism for designating the secret used to generate assertion is out-of-scope for this specification.

6. Specific Assertion Format and Processing Rules

The following clarifies the format and processing rules defined in section 4 and section 5 for a number of common use-cases:

6.1. Client authentication

When a client authenticates to a token service using an assertion, it SHOULD do so according to section 4.1. The following format and processing rules SHOULD be applied:

- o The client_id HTTP parameter MUST identify the client to the authorization server.
- o The client_assertion_type HTTP parameter MUST identify the assertion format being used for authentication.
- o The client_assertion HTTP parameter MUST contain the serialized assertion as in a format indicated by the client_assertion_type parameter.
- o The Issuer of the assertion MUST identify the entity that issued the assertion as recognized by the Authorization Server. If the assertion is self-asserted, the Issuer SHOULD be the client_id.
- o The Principal MUST identify an authorized accessor. If the assertion is self-issued, the Principal SHOULD be the client_id.
- o The Audience of the assertion MUST identify the Authorization Server and SHOULD be the URL of the Token Endpoint.
- o The Authorization Server MUST validate the assertion in order to establish a mapping between the Issuer and the secret used to generate the assertion.

The following non-normative example demonstrates the use of a client authenticating using an assertion during a Authorization Code Access Token Request as defined in Section 4.1.3 of OAuth 2.0

[I-D.ietf.oauth-v2]. (line breaks are for display purposes only):

```
POST /token HTTP/1.1
```

```
Host: server.example.com
```

```
Content-Type: application/x-www-form-urlencoded
```

```
grant_type=authorization_code&
```

```
code=i1WsRnluB1&
```

```
client_id=s6BhdRkqt3&
```

```
client_assertion_type=urn%3Aoasis%3Anames%3Atc%3ASAML%3A2.0%3Aassertion&
```

```
client_assertion=PHNhbWxwOl...[omitted for brevity]...ZT4
```

6.2. Client acting on behalf of itself

When a client is accessing resources on behalf of itself, it SHOULD do so in a manner analagous to the Client Credentials flow defined in

Section 4.4 of OAuth 2.0 [I-D.ietf.oauth-v2]. 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 4.1, with the addition of a grant_type parameter. The following format and processing rules SHOULD be applied.

- o The client_id HTTP parameter MUST identify the client to the authorization server.
- o The grant_type HTTP request parameter MUST be "client_credentials".
- o The client_assertion_type HTTP parameter MUST identify the assertion format.
- o The client_assertion HTTP parameter MUST contain the serialized assertion as in a format indicated by the client_assertion_type parameter.
- o The Issuer of the assertion MUST identify the entity that issued the assertion as recognized by the Authorization Server. If the assertion is self-asserted, the Issuer SHOULD be the client_id. If the assertion was issued by a Security Token Service, the Issuer SHOULD identify the STS as recognized by the Authorization Server.
- o The Principal SHOULD be the client_id.
- o The Audience of the assertion MUST identify the Authorization Server and SHOULD be the URL of the Token Endpoint.
- o The Authorization Server MUST validate the assertion in order to establish a mapping between the Issuer and the secret used to generate the assertion.

The following non-normative example demonstrates the use of a sample assertion being used for a Client Credentials Access Token Request as defined in Section 4.4.2 of OAuth 2.0 [I-D.ietf.oauth-v2]. (line breaks are for display purposes only):

POST /token HTTP/1.1

Host: server.example.com

Content-Type: application/x-www-form-urlencoded

client_id=s6BhdRkqt3&

grant_type=client_credentials&

client_assertion_type=urn%3Aoasis%3Anames%5Atc%3ASAML%3A2.0%3Aassertion&

client_assertion=PHNhbWxwOl...[omitted for brevity]...ZT4%3D

6.3. Client acting on behalf of a user

When a client is accessing resources on behalf of a user, it SHOULD be treated as using an assertion as an Authorization Grant according to section 4.2. The following format and processing rules SHOULD be applied:

- o The client_id HTTP parameter MUST identify the client to the authorization server.
- o The grant_type HTTP request parameter MUST indicate the assertion format.
- o The assertion HTTP parameter MUST contain the serialized assertion as in a format indicated by the grant_type parameter.
- o The Issuer of the assertion MUST identify the entity that issued the assertion as recognized by the Authorization Server. If the assertion is self-asserted, the Issuer SHOULD be the client_id. If the assertion was issued by a STS, the Issuer SHOULD identify the STS as recognized by the Authorization Server.
- o The Principal MUST identify an authorized accessor for whom the access token is being requested (typically the resource owner, or an authorized delegate).
- o The Audience of the assertion MUST identify the Authorization Server and MAY be the URL of the Token Endpoint.
- o The Authorization Server MUST validate the assertion in order to establish a mapping between the Issuer and the secret used to generate the assertion.

The following non-normative example demonstrates the use of a client authenticating using an assertion during a Authorization Code Access Token Request as defined in Section 4.1.3 of OAuth 2.0 [I-D.ietf.oauth-v2]. (line breaks are for display purposes only):

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

client_id=s6BhdRkqt3&
grant_type=urn%3Aoasis%3Anames%3Atc%3ASAML%3A2.0%3Aassertion&
assertion=PHNhbWxwOl...[omitted for brevity]...ZT4%3D
```

6.4. Client acting on behalf of an anonymous user

When a client is accessing resources on behalf of an anonymous user, the following format and processing rules SHOULD be applied:

- o The `client_id` HTTP parameter MUST identify the client to the authorization server.
- o The `grant_type` HTTP request parameter MUST indicate the assertion format.
- o The `assertion` HTTP parameter MUST contain the serialized assertion as in a format indicated by the `grant_type` parameter.
- o The Issuer of the assertion MUST identify the entity that issued the assertion as recognized by the Authorization Server. If the assertion is self-asserted, the Issuer SHOULD be the `client_id`. If the assertion was issued by a Security Token Service, the Issuer SHOULD identify the STS as recognized by the Authorization Server.
- o The Principal SHOULD indicate to the Authorization Server that the client is acting on-behalf of an anonymous user as defined by the Authorization Server. It is implied that authorization is based upon additional criteria, such as additional attributes or claims provided in the assertion. For example, a client may 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.
- o The Audience of the assertion MUST identify the Authorization Server and MAY be the URL of the Token Endpoint.
- o The Authorization Server MUST validate the assertion in order to establish a mapping between the Issuer and the secret used to generate the assertion.

7. Error Responses

If an assertion is not valid or has expired, the Authorization Server MUST construct an error response as defined in OAuth 2.0 [I-D.ietf.oauth-v2]. 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"
}
```

A client MUST NOT include client credentials using more than one mechanism. 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 attributes which will only be present when using assertions for client authentication. If more than one mechanism is used, the Authorization Server MUST construct an error response as defined in OAuth 2.0 [I-D.ietf.oauth-v2]. 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 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": "Multiple Credentials Not Allowed"
}
```

8. Security Considerations

No additional considerations beyond those described within the OAuth 2.0 Protocol Framework [I-D.ietf.oauth-v2].

9. Acknowledgements

The authors wish to thank the following people that have influenced or contributed this specification: Paul Madsen, Eric Sachs, Jian Cai, Tony Nadlin, the authors of OAuth WRAP, and those in the OAuth 2 working group.

10. Normative References

- [I-D.ietf.oauth-v2]
Hammer-Lahav, E., "The OAuth 2.0 Authorization Protocol",
April 2011.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
Requirement Levels", BCP 14, RFC 2119, March 1997.

Authors' Addresses

Chuck Mortimore (editor)
Salesforce.com

Email: cmortimore@salesforce.com

Michael B. Jones
Microsoft

Email: mbj@microsoft.com

Brian Campbell
Ping Identity

Email: bcampbell@pingidentity.com

Yaron Goland
Microsoft

Email: yarong@microsoft.com

