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.

Table of Contents

1. Introduction

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

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

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

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

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

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

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

1.1. Notational Conventions

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

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

1.2. Terminology

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

2. HTTP Parameter Bindings for Transporting Assertions

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

2.1. Using SAML Assertions as Authorization Grants

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

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

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

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

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

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

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

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

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

2.2. Using SAML Assertions for Client Authentication

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

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

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

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

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

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

3. Assertion Format and Processing Requirements

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

1. The Assertion's <Issuer> element MUST contain a unique identifier for the entity that issued the Assertion. In the absence of an application profile specifying otherwise, compliant applications MUST compare Issuer values using the Simple String Comparison method defined in Section 6.2.1 of RFC 3986 [RFC3986].

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

3. The Assertion MUST contain a <Subject> element identifying the principal that is the subject of the Assertion. Additional information identifying the subject/principal MAY be included in an <AttributeStatement>.

A. For the authorization grant, the Subject typically identifies an authorized accessor for which the access token is being requested (i.e., the resource owner or an authorized delegate), but in some cases, may be a pseudonymous identifier or other value denoting an anonymous user.

B. For client authentication, the Subject MUST be the "client_id" of the OAuth client.

4. The Assertion MUST have an expiry that limits the time window during which it can be used. The expiry can be expressed either as the NotOnOrAfter attribute of the <Conditions> element or as the NotOnOrAfter attribute of a suitable <SubjectConfirmationData> element.

5. The <Subject> element MUST contain at least one <SubjectConfirmation> element that has a Method attribute with a value of "urn:oasis:names:tc:SAML:2.0:cm:bearer". If the Assertion does not have a suitable NonOnOrAfter attribute on the <Conditions> element, the <SubjectConfirmation> element MUST contain a <SubjectConfirmationData> element. When present, the <SubjectConfirmationData> element MUST have a Recipient attribute with a value indicating the token endpoint URL of the authorization server (or an acceptable alias). The authorization server MUST verify that the value of the Recipient attribute matches the token endpoint URL (or an acceptable alias) to which the Assertion was delivered. The <SubjectConfirmationData> element MUST have a NotOnOrAfter attribute that limits the window during which the Assertion can be confirmed. The <SubjectConfirmationData> element MAY also contain an Address attribute limiting the client address from which the Assertion can be delivered. Verification of the Address is at the discretion of the authorization server.

6. The authorization server MUST reject the entire Assertion if the NotOnOrAfter instant on the <Conditions> element has passed (subject to allowable clock skew between systems). The authorization server MUST reject the <SubjectConfirmation> (but MAY still use the rest of the Assertion) if the NotOnOrAfter instant on the <SubjectConfirmationData> has passed (subject to allowable clock skew). Note that the authorization server may reject Assertions with a NotOnOrAfter instant that is unreasonably far in the future. The authorization server MAY ensure that Bearer Assertions are not replayed, by maintaining the set of used ID values for the length of time for which the Assertion would be considered valid based on the applicable NotOnOrAfter instant.

7. If the Assertion issuer directly authenticated the subject, the Assertion SHOULD contain a single <AuthnStatement> representing that authentication event. If the Assertion was issued with the intention that the client act autonomously on behalf of the subject, an <AuthnStatement> SHOULD NOT be included and the client presenting the assertion SHOULD be identified in the <NameID> or similar element in the <SubjectConfirmation> element, or by other available means like SAML V2.0 Condition for Delegation Restriction [OASIS.saml-deleg-cs].

8. Other statements, in particular <AttributeStatement> elements, MAY be included in the Assertion.

9. The Assertion MUST be digitally signed or have a Message Authentication Code applied by the issuer. The authorization server MUST reject assertions with an invalid signature or Message Authentication Code.

10. Encrypted elements MAY appear in place of their plain text counterparts as defined in [OASIS.saml-core-2.0-os].

11. The authorization server MUST reject an Assertion that is not valid in all other respects per [OASIS.saml-core-2.0-os], such as (but not limited to) all content within the Conditions element including the NotOnOrAfter and NotBefore attributes, unknown condition types, etc.

3.1. Authorization Grant Processing

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

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

For example:

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

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

3.2. Client Authentication Processing

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

4. Authorization Grant Example

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

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

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

```
<Assertion IssueInstant="2010-10-01T20:07:34.619Z"
  ID="ef1xsbZxPV2oqjd7HTLRLIBlBb7"
  Version="2.0"
  xmlns="urn:oasis:names:tc:SAML:2.0:assertion">
 <Issuer>https://saml-idp.example.com</Issuer>
 <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
  [...omitted for brevity...]
 </ds:Signature>
 <Subject>
  <NameID
   Format="urn:oasis:names:tc:SAML:1.1:nameid-format:emailAddress">
   brian@example.com
  </NameID>
  <SubjectConfirmation
    Method="urn:oasis:names:tc:SAML:2.0:cm:bearer">
   <SubjectConfirmationData
     NotOnOrAfter="2010-10-01T20:12:34.619Z"
     Recipient="https://authz.example.net/token.oauth2"/>
   </SubjectConfirmation>
  </Subject>
  <Conditions>
    <AudienceRestriction>
      <Audience>https://saml-sp.example.net</Audience>
    </AudienceRestriction>
  </Conditions>
  <AuthnStatement AuthnInstant="2010-10-01T20:07:34.371Z">
    <AuthnContext>
      <AuthnContextClassRef>
        urn:oasis:names:tc:SAML:2.0:ac:classes:X509
      </AuthnContextClassRef>
    </AuthnContext>
  </AuthnStatement>
</Assertion>
```

Figure 1: Example SAML 2.0 Assertion

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

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

grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Asaml2-
bearer&assertion=PEFzc2VydGlvbiBJc3N1ZUluc3RhbnQ9IjIwMTEtMDU
[...omitted for brevity...]aG5TdGF0ZW1lbnQ-PC9Bc3NlcnRpb24-
```

Figure 2: Example Request

5. Interoperability Considerations

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

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

6. Security Considerations

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

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

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

7. Privacy Considerations

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

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

8. IANA Considerations

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

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

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

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

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

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

- Common Name: SAML 2.0 Bearer Assertion Profile for OAuth 2.0 Client Authentication
- Change controller: IESG
- 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

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

draft-ietf-oauth-saml2-bearer-22

- Changes/suggestions from IESG reviews.

draft-ietf-oauth-saml2-bearer-21

- 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

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

draft-ietf-oauth-saml2-bearer-19

- Updated references.

draft-ietf-oauth-saml2-bearer-18

- Clean up language around subject per http://www.ietf.org/mail-archive/web/oauth/current/msg12254.html.

- 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."

- Clarify the potentially confusing language about the AS confirming the assertion http://www.ietf.org/mail-archive/web/oauth/current/msg12255.html.

- 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.

- 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.

- 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

- 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

- 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.

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

- Numbered the list of processing rules.

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

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

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

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

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

draft-ietf-oauth-saml2-bearer-15

- Reference RFC 6749 and RFC 6755.

- Update draft-ietf-oauth-assertions reference to -06.

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

draft-ietf-oauth-saml2-bearer-14

- 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

- Add examples to Sections 2.1 and 2.2

- Update references

draft-ietf-oauth-saml2-bearer-13

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

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

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

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

draft-ietf-oauth-saml2-bearer-12

- 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

- 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.

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

- Added Terminology section.

draft-ietf-oauth-saml2-bearer-10

- fix a spelling mistake

draft-ietf-oauth-saml2-bearer-09

- Attempt to address an ambiguity around validation requirements when the Conditions element contain a NotOnOrAfter and SubjectConfirmation/SubjectConfirmationData does too. Basically it needs to have at least one bearer SubjectConfirmation element but that element can omit SubjectConfirmationData, if Conditions has an expiry on it. Otherwise, a valid SubjectConfirmation must have a SubjectConfirmationData with Recipient and NotOnOrAfter. And any SubjectConfirmationData that has those elements needs to have them checked.

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

- fix a typo

draft-ietf-oauth-saml2-bearer-08

- fix some typos

draft-ietf-oauth-saml2-bearer-07

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

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

draft-ietf-oauth-saml2-bearer-06

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

draft-ietf-oauth-saml2-bearer-05

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

- Rework most of the spec to profile draft-ietf-oauth-assertions for both authn and authz including (but not limited to):

  * remove requirement for issuer to be urn:oasis:names:tc:SAML:2.0:nameid-format:entity

  * change wording on Subject requirements

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

draft-ietf-oauth-saml2-bearer-04

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

draft-ietf-oauth-saml2-bearer-03

- Cleanup of some editorial issues.

draft-ietf-oauth-saml2-bearer-02

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

- 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

- Editorial changes based on feedback

draft-ietf-oauth-saml2-bearer-01

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

- 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

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

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

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

draft-ietf-oauth-saml2-bearer-00

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

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

- 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

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

- Updated examples.

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

- Removed the ’MUST NOT contain a NotBefore attribute’ on SubjectConfirmationData.

- Relaxed wording that ties the subject of the Assertion to the resource owner.
- 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.
- Added a few examples to the language about verifying that the Assertion is valid in all other respects.
- Added some wording to the introduction about the similarities to Web SSO in the format and processing rules
- 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
- Changed title to include "Grant Type" in it.
- Editorial updates based on feedback from the WG and others (including capitalization of Assertion when referring to SAML).

draft-campbell-oauth-saml-00

- 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

Table of Contents

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:

- Third-party applications are required to store the resource owner's credentials for future use, typically a password in clear-text.
- Servers are required to support password authentication, despite the security weaknesses inherent in passwords.
- 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.
- 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.
- 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

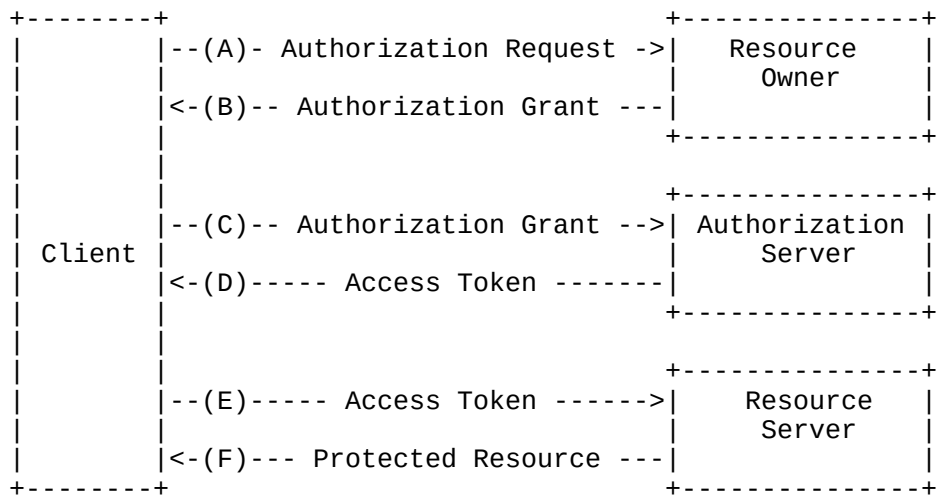


Figure 1: Abstract Protocol Flow

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.

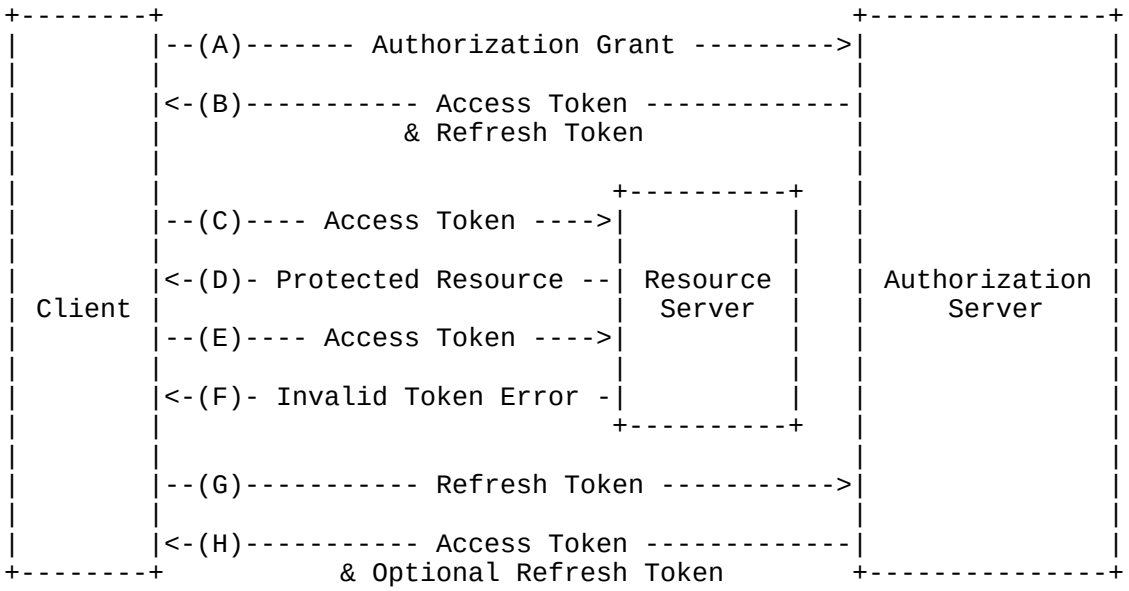


Figure 2: Refreshing an Expired Access Token

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:

- specify the client type as described in Section 2.1,
- provide its client redirection URIs as described in Section 3.1.2, and
- 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):

- Authorization endpoint - used by the client to obtain authorization from the resource owner via user-agent redirection.
- 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:

- 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:

- Public clients.
- 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:

- 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.
- 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.

- 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).

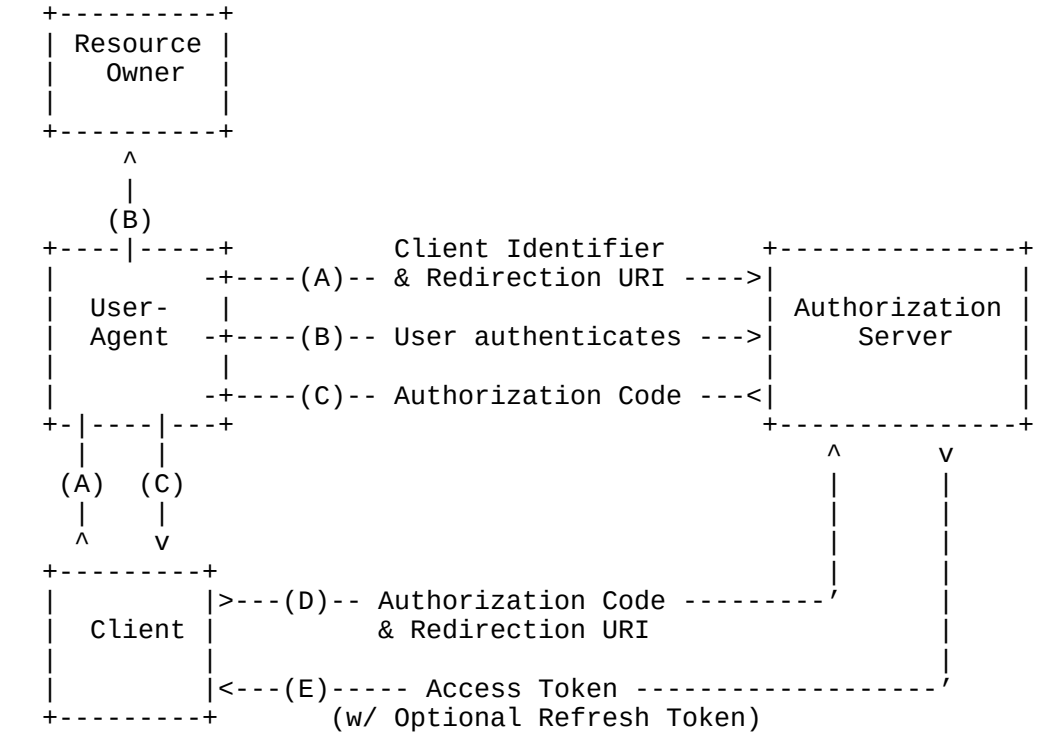
4. Obtaining Authorization

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=SplxlOBeZQQYbYS6WxSbIA
          &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=SplxlOBeZQQYbYS6WxSbIA
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
```

The authorization server MUST:

- require client authentication for confidential clients or for any client that was issued client credentials (or with other authentication requirements),
- authenticate the client if client authentication is included,
- 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,
- verify that the authorization code is valid, and
- 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":"2YotnFZFEjr1zCsicMWpAA",
  "token_type":"example",
  "expires_in":3600,
  "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
  "example_parameter":"example_value"
}
```

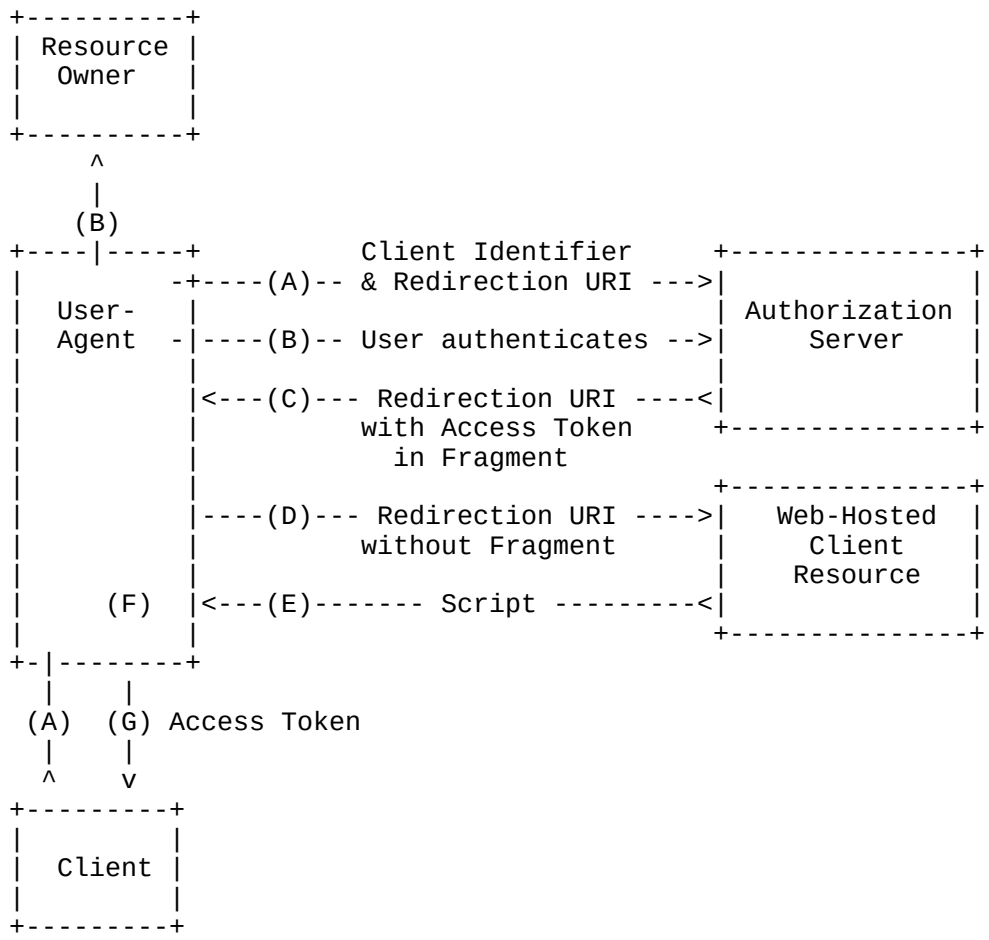
4.2. Implicit Grant

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=2YotnFZFEjr1zCsicMWpAA
          &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.

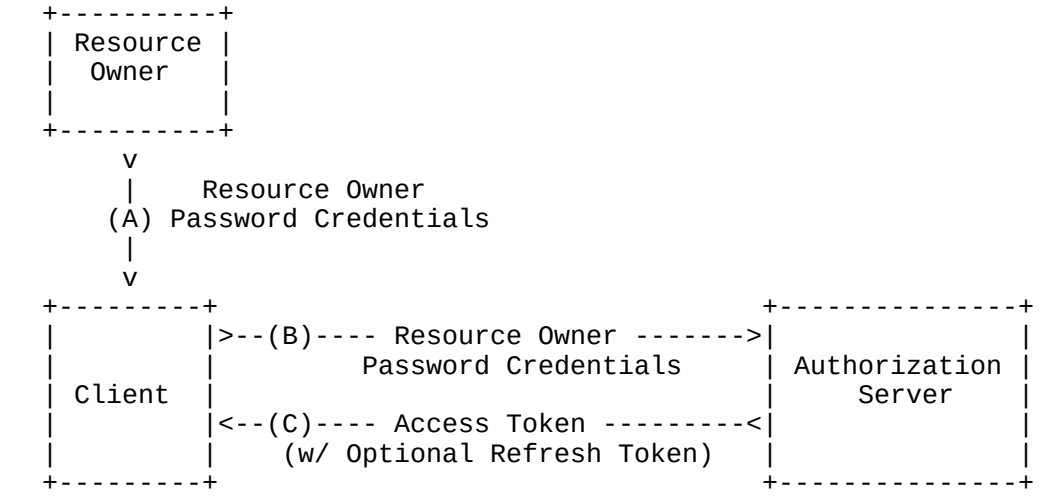


Figure 5: Resource Owner Password Credentials Flow

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:

- require client authentication for confidential clients or for any client that was issued client credentials (or with other authentication requirements),
- authenticate the client if client authentication is included, and
- 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":"2YotnFZFEjr1zCsicMWpAA",
  "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.

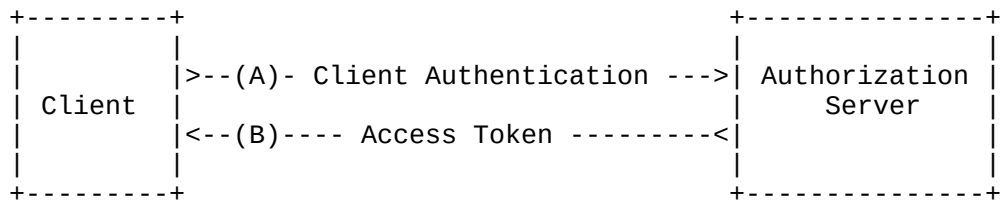


Figure 6: Client Credentials Flow

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":"2YotnFZFEjr1zCsicMWpAA",
  "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=PEFzc2VydGlvbiBJc3N1ZUluc3RhbnQ9IjIwMTEtMDU
[...omitted for brevity...]aG5TdGF0ZW1lbnQ-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":"2YotnFZFEjr1zCsicMWpAA",
  "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 documents 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:

- 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.
- 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:

- 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).
- An embedded user-agent may offer improved usability, as it removes the need to switch context and open new windows.
- 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:

- 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.
- 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:

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

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

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

- Parameter name: response_type
- Parameter usage location: authorization request
- Change controller: IETF
- Specification document(s): [[this document]]

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

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

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

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

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

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

- Parameter name: grant_type
- Parameter usage location: token request

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

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

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

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

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

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

- Parameter name: refresh_token
- Parameter usage location: token request, token response
- Change controller: IETF
- 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:

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

- Response type name: token
- Change controller: IETF
- 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
UNICODECHARNOCRLF = %x09 /%x20-7E / %x80-D7FF /
                    %xE000-FFFD / %x10000-10FFFF
```

(The UNICODECHARNOCRLF 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 Sergent, 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 Suriel, 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

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

-30

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

-29

- 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.
- 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".
- Added Security Considerations section "Misuse of Access Token to Impersonate Resource Owner in Implicit Flow".
- Added references in the "Implicit" and "Implicit Grant" sections to particularly pertinent security considerations.
- Added appendix "Use of application/x-www-form-urlencoded Media Type" and referenced it in places that this encoding is used.
- Deleted ";charset=UTF-8" from examples formerly using "Content-Type: application/x-www-form-urlencoded;charset=UTF-8".
- Added the phrase "with a character encoding of UTF-8" when describing how to send requests using the HTTP request entity-body.
- 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.
- Added "The ABNF below is defined in terms of Unicode code points [W3C.REC-xml-20081126]; these characters are typically encoded in UTF-8".

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

-28

- 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.
- 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

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

Table of Contents

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.

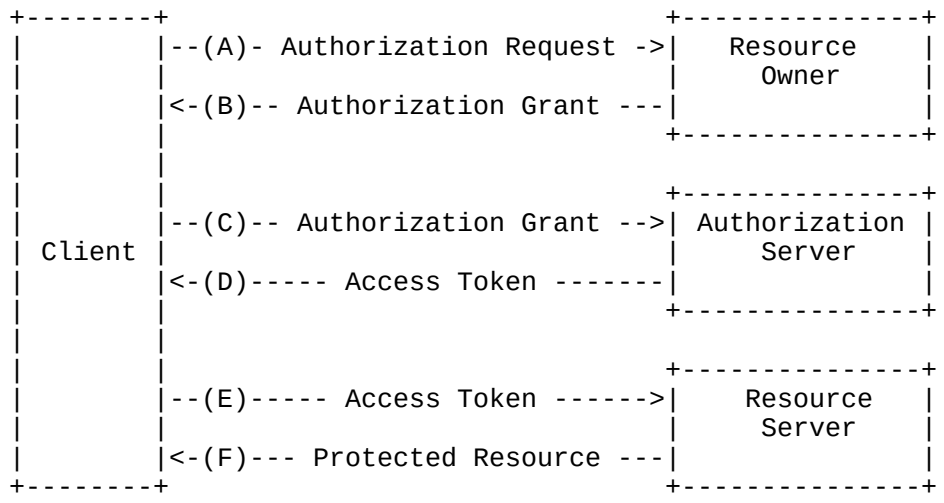


Figure 1: Abstract Protocol Flow

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:

- The HTTP request entity-header includes the "Content-Type" header field set to "application/x-www-form-urlencoded".
- 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].
- The HTTP request entity-body is single-part.
- The content to be encoded in the entity-body MUST consist entirely of ASCII [USASCII] characters.
- 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. Goland (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 Suriel, 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

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

-22

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

-21

- Changed "NOT RECOMMENDED" to "not recommended" in caveat about the URI Query Parameter method.
- 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".
- Changed Acknowledgements to use only ASCII characters, per the RFC style guide.

-20

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

-19

- Addressed DISCUSS issues and comments raised for which resolutions have been agreed to. No normative changes were made. Changes made were:
- Use ABNF from RFC 5234.
- 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.
- In the introduction, state that this document also imposes semantic requirements upon the access token.
- Reference the "scope" definition in the OAuth core spec.
- Added "scope" examples.
- Reference RFC 6265 for security considerations about cookies.

-18

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

- Added example access token response returning a Bearer token.

-17

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

-16

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

-15

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

-14

- Changes made in response to review comments by Security Area Director Stephen Farrell. Specifically:
- Strengthened warnings about passing an access token as a query parameter and more precisely described the limitations placed upon the use of this method.
- 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].
- Normatively stated that "the token integrity protection MUST be sufficient to prevent the token from being modified".
- Added statement that "TLS is mandatory to implement and use with this specification" to the introduction.

- Stated that TLS MUST be used with "a ciphersuite that provides confidentiality and integrity protection".

- 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.

- 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.

- Dropped the confusing phrase "for instance, through the use of TLS" from the sentence about confidentiality protection of the exchanges.

- Reference RFC 6125 for identity verification, rather than RFC 2818.

- 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.

- Stated that bearer tokens MUST NOT be stored in cookies that can be sent in the clear in the Threat Mitigation section.

- Replaced sole remaining reference to [RFC2616] with HTTPbis [I-D.ietf-httpbis-p1-messaging] reference.

- 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]").

- Other on-normative editorial improvements.

-13

- 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

- Made non-normative editorial changes that Hannes Tschofenig requested be applied prior to forwarding the specification to the IESG.
- Added rationale for the choice of the b64token syntax.
- 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.
- Added additional active working group contributors to the Acknowledgements section.

-11

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

-10

- Removed the #auth-param option from Authorization header syntax (leaving only the b64token syntax).
- 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.
- Restricted the character set for "error_description" strings to SP / VCHAR and indicated that they are not meant to be displayed to end users.
- Included more description in the Abstract, since Hannes Tschofenig indicated that the RFC editor would require this.
- Changed "Access Grant" to "Authorization Grant", as was done in the core spec.
- Simplified the introduction to the Authenticated Requests section.

-09

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

- Update credentials definition to conform to [I-D.ietf-httpbis-p7-auth].
- Further clarified that query parameters may occur in any order.
- Specify that error_description is UTF-8 encoded (matching the core specification).
- Registered "Bearer" Authentication Scheme in Authentication Scheme Registry defined by [I-D.ietf-httpbis-p7-auth].
- Updated references to oauth-v2, httpbis-p1-messaging, and httpbis-p7-auth drafts.
- Other wording improvements not introducing normative changes.

-08

- Updated references to oauth-v2 and HTTPbis drafts.

-07

- Added missing comma in error response example.

-06

- Changed parameter name "bearer_token" to "access_token", per working group consensus.
- 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

- Removed OAuth Errors Registry, per design team input.
- 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]].
- Added missing quotation marks in error-uri definition.
- Added note to add language and encoding information to error_description if the core specification does.
- Explicitly reference the Augmented Backus-Naur Form (ABNF) defined in [RFC5234].

- Use auth-param instead of repeating its definition, which is (token "=" (token / quoted-string)).

- 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

- Edits responding to working group last call feedback on -03. Specific edits enumerated below.

- Added Bearer Token definition in Terminology section.

- Changed parameter name "oauth_token" to "bearer_token".

- Added realm parameter to "WWW-Authenticate" response to comply with [RFC2617].

- Removed "[RWS 1#auth-param]" from "credentials" definition since it did not comply with the ABNF in [I-D.ietf-httpbis-p7-auth].

- Removed restriction that the "bearer_token" (formerly "oauth_token") parameter be the last parameter in the entity-body and the HTTP request URI query.

- 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.

- Removed OAuth Parameters registry extension.

- Numerous editorial improvements suggested by working group members.

-03

- Restored the WWW-Authenticate response header functionality deleted from the framework specification in draft 12 based upon the specification text from draft 11.

- Augmented the OAuth Parameters registry by adding two additional parameter usage locations: "resource request" and "resource response".

- Registered the "oauth_token" OAuth parameter with usage location "resource request".

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

-02

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

-01

- 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

- 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/

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: July 8, 2011

M. Jones
Microsoft
D. Balfanz
Google
J. Bradley
independent
Y. Goland
Microsoft
J. Panzer
Google
N. Sakimura
Nomura Research Institute
P. Tarjan
Facebook
January 04, 2011

JSON Web Token (JWT) - Claims and Signing
draft-jones-json-web-token-01

Abstract

JSON Web Token (JWT) is a means of representing signed content using JSON data structures, including claims to be transferred between two parties. The claims in a JWT are encoded as a JSON object that is digitally signed and optionally encrypted. Encryption for JWTs is described in a separate companion specification.

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

Requirements Language

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

Status of this Memo

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

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

Internet-Drafts are draft documents valid for a maximum of six months

and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on July 8, 2011.

Table of Contents

1. Introduction

JSON Web Token (JWT) is a compact token format intended for space constrained environments such as HTTP Authorization headers and URI query parameters. JWTs encode claims to be transmitted as a JSON object (as defined in RFC 4627 [RFC4627]) that is base64url encoded and digitally signed. The JWT signature mechanisms are independent of the type of content being signed, allowing arbitrary content to be signed. Encryption for JWTs is described in a separate companion specification.

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

2. Terminology

JSON Web Token (JWT) A data structure containing three JWT Token Segments: the JWT Header Segment, the JWT Payload Segment, and the JWT Crypto Segment. The JWT Payload Segment typically represents a set of claims convened by the JWT as a JSON object, but in the general case, may represent arbitrary signed content.

JWT Compact Serialization A data structure representing a JWT as a string consisting of three JWT Token Segments: the JWT Header Segment, the JWT Payload Segment, and the JWT Crypto Segment, in that order, with the segments being separated by period ('.') characters.

JWT JSON Serialization A data structure representing a JWT as a JSON object with members for each of three kinds of JWT Token Segments: a "header" member whose value is a non-empty array of JWT Header Segments, a "payload" member whose value is the JWT Payload Segment, and a "signature" member whose value is a non-empty array of JWT Crypto Segments, where the cardinality of both arrays is the same.

JWT Token Segment One of the three parts that make up a JSON Web Token (JWT). JWT Token Segments are always base64url encoded values.

JWT Header Segment A JWT Token Segment containing a base64url encoded JSON object that describes the signature applied to the JWT Header Segment and the JWT Payload Segment.

JWT Payload Segment A JWT Token Segment containing base64url encoded content. This may be a JWT Claims Object.

JWT Crypto Segment A JWT Token Segment containing base64url encoded cryptographic signature material that secures the JWT Header Segment's and the JWT Payload Segment's contents.

Decoded JWT Header Segment A JWT Header Segment that has been base64url decoded back into a JSON object.

Decoded JWT Payload Segment A JWT Payload Segment that has been base64url decoded. If the corresponding JWT Payload Segment is a JWT Claims Object, this will be a Decoded JWT Claims Object.

Decoded JWT Crypto Segment A JWT Crypto Segment that has been base64url decoded back into cryptographic material.

JWT Claims Object A base64url encoded JSON object that represents the claims contained in the JWT.

Decoded JWT Claims Object A JSON object that represents the claims contained in the JWT.

JWT Signing Input The concatenation of the JWT Header Segment, a period ('.') character, and the JWT Payload Segment.

Digital Signature For the purposes of this specification, we use this term to encompass both Hash-based Message Authentication Codes (HMACs), which can provide authenticity but not non-repudiation, and digital signatures using public key algorithms, which can provide both. Readers should be aware of this distinction, despite the decision to use a single term for both concepts to improve readability of the specification.

Base64url Encoding For the purposes of this specification, this term always refers to the he URL- and filename-safe Base64 encoding described in RFC 4648 [RFC4648], Section 5, with the '=' padding characters omitted, as permitted by Section 3.2; see Section 7 for more details.

Header Parameter Names The names of the members within the JSON object represented in a JWT Header Segment.

Header Parameter Values The values of the members within the JSON object represented in a JWT Header Segment.

Claim Names The names of the members of the JSON object represented in a JWT Claims Object.

Claim Values The values of the members of the JSON object represented in a JWT Claims Object.

3. JSON Web Token (JWT) Overview

JWTs represent content that is base64url encoded and digitally signed, and optionally encrypted, using JSON data structures; this content is typically a set of claims represented as a JSON object.

When the JWT payload is a set of claims, the claims are represented as name/value pairs that are members of a JSON object. The JSON object is base64url encoded to produce the JWT Claims Object, which is used as the JWT Payload Segment. An accompanying base64url encoded JSON header - the JWT Header Segment - describes the signature method used.

The names within the header object MUST be unique. The names within the header object are referred to as Header Parameter Names. The corresponding values are referred to as Header Parameter Values. Likewise, if the payload represents a JWT Claims Object, the names within the claims object MUST be unique. The names within the claims object are referred to as Claim Names. The corresponding values are referred to as Claim Values.

JWTs contain a signature that ensures the integrity of the content of the JWT Header Segment and the JWT Payload Segment. This signature value is carried in the JWT Crypto Segment. The JSON Header object MUST contain an "alg" parameter, the value of which is a string that unambiguously identifies the algorithm used to sign the JWT Header Segment and the JWT Payload Segment to produce the JWT Crypto Segment.

3.1. Example JWT

The following is an example of a JSON object that can be encoded to produce a JWT Claims Object:

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

Base64url encoding the UTF-8 representation of the JSON object yields this JWT Claims Object, which is used as the JWT Payload Segment:

```
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19y
b290Ijp0cnVlfQ
```

The following example JSON header object declares that the encoded object is a JSON Web Token (JWT) and the JWT Header Segment and the JWT Payload Segment are signed using the HMAC SHA-256 algorithm:

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

Base64url encoding the UTF-8 representation of the JSON header object yields this JWT Header Segment value:

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
```

Signing the UTF-8 representation of the JWT Signing Input (the concatenation of the JWT Header Segment, a period ('.') character, and the JWT Payload Segment) with the HMAC SHA-256 algorithm and base64url encoding the result, as per Section 8.1, yields this JWT Crypto Segment value:

```
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

Concatenating these segments in the order Header.Payload.Signature with period characters between the segments yields this complete JWT using the JWT Compact Serialization (with line breaks for display purposes only):

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19y
b290Ijp0cnVlfQ
.
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

This computation is illustrated in more detail in Appendix A.1.

4. JWT Claims

If the JWT contains a set of claims represented as a JSON object, then the members of the JSON object represented by the Decoded JWT Claims Object decoded from the JWT Payload Segment contain the claims. Note however, that the set of claims a JWT must contain to be considered valid is context-dependent and is outside the scope of this specification. When used in a security-related context, implementations MUST understand and support all of the claims present; otherwise, the JWT MUST be rejected for processing.

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

4.1. Reserved Claim Names

The following claim names are reserved. None of the claims defined in the table below are intended to be mandatory, but rather, provide

a starting point for a set of useful, interoperable claims. All the names are short because a core goal of JWTs is for the tokens themselves to be short.

Claim Name	JSON Value Type	Claim Syntax	Claim Semantics
exp	integer	IntDate	The "exp" (expiration time) claim identifies the expiration time on or after which the token MUST NOT be accepted for processing. The processing of the "exp" claim requires that the current date/time MUST be before the expiration date/time listed in the "exp" claim. Implementers MAY provide for some small leeway, usually no more than a few minutes, to account for clock skew. This claim is OPTIONAL.
iss	string	StringAndURI	The "iss" (issuer) claim identifies the principal that issued the JWT. The processing of this claim is generally application specific. This claim is OPTIONAL.
aud	string	StringAndURI	The "aud" (audience) claim identifies the audience that the JWT is intended for. The principal intended to process the JWT MUST be identified by the value of the audience claim. If the principal processing the claim does not identify itself with the identifier in the "aud" claim value then the JWT MUST be rejected. The interpretation of the contents of the audience value is generally application specific. This claim is OPTIONAL.

typ	string	String	The "typ" (type) claim is used to declare a type for the contents of this JWT. This claim is OPTIONAL.

Table 1: Reserved Claim Definitions

Additional reserved claim names MAY be defined via the IANA JSON Web Token Claims registry, as per Section 10. The syntax values used above and in Table 3 are defined as follows:

Syntax Name	Syntax Definition
IntDate	The number of seconds from 1970-01-01T0:0:0Z as measured in UTC until the desired date/time. See RFC 3339 [RFC3339] for details regarding date/times in general and UTC in particular.
String	Any string value MAY be used.
StringAndURI	Any string value MAY be used but a value containing a ":" character MUST be a URI as defined in RFC 3986 [RFC3986].
URI	A URI as defined in RFC 3986 [RFC3986].
URL	A URL as defined in RFC 1738 [RFC1738].

Table 2

4.2. Public Claim Names

Claim names can be defined at will by those using JWTs. However, in order to prevent collisions, any new claim name SHOULD either be defined in the IANA JSON Web Token Claims registry or be defined as a URI that contains a collision resistant namespace. Examples of collision resistant namespaces include:

- Domain Names,
- Object Identifiers (OIDs) as defined in the ITU-T X 660 and X 670 Recommendation series or
- Universally Unique IDentifier (UUID) as defined in RFC 4122 [RFC4122].

In each case, the definer of the name or value MUST take reasonable precautions to make sure they are in control of the part of the namespace they use to define the claim name.

4.3. Private Claim Names

A producer and consumer of a JWT may agree to any claim name that is not a Reserved Name Section 4.1 or a Public Name Section 4.2. Unlike Public Names, these private names are subject to collision and should be used with caution.

5. JWT Header

The members of the JSON object represented by the Decoded JWT Header Segment describe the signature applied to the JWT Header Segment and the JWT Payload Segment and optionally additional properties of the JWT. Implementations MUST understand the entire contents of the header; otherwise, the JWT MUST be rejected for processing.

5.1. Reserved Header Parameter Names

The following header parameter names are reserved. All the names are short because a core goal of JWTs is for the tokens themselves to be short.

Header Parameter Name	JSON Value Type	Header Parameter Syntax	Header Parameter Semantics
alg	string	StringAndURI	The "alg" (algorithm) header parameter identifies the cryptographic algorithm used to secure the JWT. A list of reserved alg values is in Table 4. The processing of the "alg" (algorithm) header parameter, if present, requires that the value of the "alg" header parameter MUST be one that is both supported and for which there exists a key for use with that algorithm associated with the issuer of the JWT. This header parameter is REQUIRED.

typ	string	String	The "typ" (type) header parameter is used to declare that this data structure is a JWT. If a "typ" parameter is present, it is RECOMMENDED that its value be "JWT". This header parameter is OPTIONAL.
jku	string	URL	The "jku" (JSON Key URL) header parameter is a URL that points to JSON-encoded public key certificates that can be used to validate the signature. The specification for this encoding is TBD. This header parameter is OPTIONAL.
kid	string	String	The "kid" (key ID) header parameter is a hint indicating which specific key owned by the signer should be used to validate the signature. This allows signers to explicitly signal a change of key to recipients. Omitting this parameter is equivalent to setting it to an empty string. The interpretation of the contents of the "kid" parameter is unspecified. This header parameter is OPTIONAL.
x5u	string	URL	The "x5u" (X.509 URL) header parameter is a URL that points to an X.509 public key certificate that can be used to validate the signature. This certificate MUST conform to RFC 5280 [RFC5280]. This header parameter is OPTIONAL.

x5t	string	String	The "x5t" (x.509 certificate thumbprint) header parameter provides a base64url encoded SHA-256 thumbprint (a.k.a. digest) of the DER encoding of an X.509 certificate that can be used to match a certificate. This header parameter is OPTIONAL.

Table 3: Reserved Header Parameter Definitions

Additional reserved header parameter names MAY be defined via the IANA JSON Web Token Header Parameters registry, as per Section 10. The syntax values used above and in Table 1 are defined in Table 2.

5.2. Public Header Parameter Names

Additional header parameter names can be defined by those using JWTs. However, in order to prevent collisions, any new header parameter name or algorithm value SHOULD either be defined in the IANA JSON Web Token Header Parameters registry or be defined as a URI that contains a collision resistant namespace. In each case, the definer of the name or value MUST take reasonable precautions to make sure they are in control of the part of the namespace they use to define the header parameter name.

New header parameters should be introduced sparingly, as they can result in non-interoperable JWTs. Nonetheless, some extensions needed for some use cases may require them, such as an extension to enable the inclusion of multiple signatures.

5.3. Private Header Parameter Names

A producer and consumer of a JWT may agree to any header parameter name that is not a Reserved Name Section 5.1 or a Public Name Section 5.2. Unlike Public Names, these private names are subject to collision and should be used with caution.

New header parameters should be introduced sparingly, as they can result in non-interoperable JWTs.

6. Rules for Creating and Validating a JWT

To create a JWT one MUST follow these steps:

1. Create the payload content to be encoded as the Decoded JWT Payload Segment. If the payload represents a JWT Claims Object, then these steps for creating the Decoded JWT Payload Segment also apply:

   * Create a JSON object containing the desired claims. Note that white space is explicitly allowed in the representation and no canonicalization is performed before encoding.

   * Translate this JSON object's Unicode code points into UTF-8, as defined in RFC 3629 [RFC3629]. This is the Decoded JWT Payload Segment.

2. Base64url encode the Decoded JWT Payload Segment. This encoding becomes the JWT Payload Segment.

3. Create a JSON object containing a set of desired header parameters. Note that white space is explicitly allowed in the representation and no canonicalization is performed before encoding.

4. Translate this JSON object's Unicode code points into UTF-8, as defined in RFC 3629 [RFC3629].

5. Base64url encode the UTF-8 representation of this JSON object as defined in this specification (without padding). This encoding becomes a JWT Header Segment.

6. Construct a JWT Crypto Segment as defined for the particular algorithm being used. The JWT Signing Input is always the concatenation of a JWT Header Segment, a period ('.') character, and the JWT Payload Segment. The "alg" header parameter MUST be present in the JSON Header Segment, with the algorithm value accurately representing the algorithm used to construct the JWT Crypto Segment.

7. If the JWT Compact Serialization is being used, then:

   * Concatenate the JWT Header Segment, the JWT Payload Segment and then the JWT Crypto Segment in that order, separating each by period characters, to create the JWT.

   Else if the JWT JSON Serialization is being used, then:

   * Create a JSON object with these three members: a "header" member whose value is an array of JWT Header Segments, a "payload" member whose value is the JWT Payload Segment, and a "signature" member whose value is an array of JWT Crypto

Segments.

* If more than one signature is present, then repeat steps 3 through 6 for each header and crypto segment to produce additional values for the header and signature arrays.

* The header and signature arrays must have the same number of values, with each header value and corresponding signature value being located at the same array index.

When validating a JWT the following steps MUST be taken. If any of the listed steps fails then the token MUST be rejected for processing.

1. If the JWT Compact Serialization is being used, then:

   * The JWT MUST contain two period characters.

   * The JWT MUST be split on the two period characters resulting in three non-empty segments. The first segment is the JWT Header Segment; the second is the JWT Payload Segment; the third is the JWT Crypto Segment.

   Else if the JWT JSON Serialization is being used, then:

   * The JSON MUST contain the three members "header", "payload", and "signature" and MAY contain others, which MUST be ignored. The payload member MUST be a string and the header and signature members MUST be non-empty arrays of strings with equal cardinality.

   * Use a "header" member array value as the JWT Header Segment; use the "payload" member value as the JWT Payload Segment; use a "signature" member array value with the same index as the "header" member array value used as the JWT Crypto Segment.

2. The JWT Payload Segment MUST be successfully base64url decoded following the restriction given in this spec that no padding characters have been used.

3. If the payload represents a JWT Claims Object, then these steps for validating the Decoded JWT Payload Segment also apply:

   * The Decoded JWT Payload Segment, which is the Decoded JWT Claims Object, MUST be completely valid JSON syntax conforming to RFC 4627 [RFC4627].

   * When used in a security-related context, the Decoded JWT Claims Object MUST be validated to only include claims whose syntax and semantics are both understood and supported.

4. The JWT Header Segment MUST be successfully base64url decoded following the restriction given in this spec that no padding characters have been used.

5. The Decoded JWT Header Segment MUST be completely valid JSON syntax conforming to RFC 4627 [RFC4627].

6. The JWT Crypto Segment MUST be successfully base64url decoded following the restriction given in this spec that no padding characters have been used.

7. The JWT Header Segment MUST be validated to only include parameters and values whose syntax and semantics are both understood and supported.

8. The JWT Crypto Segment MUST be successfully validated against the JWT Header Segment and JWT Payload Segment in the manner defined for the algorithm being used, which MUST be accurately represented by the value of the "alg" header parameter, which MUST be present.

9. If the JWT JSON Serialization is being used, then repeat steps 4 to 8 for each element of the header and signature arrays.

Processing a JWT inevitably requires comparing known strings to values in the token. For example, in checking what the algorithm is, the Unicode string encoding "alg" will be checked against the member names in the Decoded JWT Header Segment to see if there is a matching header parameter name. A similar process occurs when determining if the value of the "alg" header parameter represents a supported algorithm. Comparing Unicode strings, however, has significant security implications, as per Section 11.

Comparisons between JSON strings and other Unicode strings MUST be performed as specified below:

1. Remove any JSON applied escaping to produce an array of Unicode code points.

2. Unicode Normalization [USA15] MUST NOT be applied at any point to either the JSON string or to the string it is to be compared against.

3. Comparisons between the two strings MUST be performed as a Unicode code point to code point equality comparison.

7. Base64url encoding as used by JWTs

JWTs make use of the base64url encoding as defined in RFC 4648 [RFC4648]. As allowed by Section 3.2 of the RFC, this specification mandates that base64url encoding when used with JWTs MUST NOT use padding. The reason for this restriction is that the padding character ('=') is not URL safe.

For notes on implementing base64url encoding without padding, see Appendix B.

8. Signing JWTs with Cryptographic Algorithms

JWTs use specific cryptographic algorithms to sign the contents of the JWT Header Segment and the JWT Payload Segment. The use of the following algorithms for producing JWTs is defined in this section. The table below is the list of "alg" header parameter values reserved by this specification, each of which is explained in more detail in the following sections:

Alg Parameter Value	Algorithm
HS256	HMAC using SHA-256 hash algorithm
HS384	HMAC using SHA-384 hash algorithm
HS512	HMAC using SHA-512 hash algorithm
RS256	RSA using SHA-256 hash algorithm
RS384	RSA using SHA-384 hash algorithm
RS512	RSA using SHA-512 hash algorithm
ES256	ECDSA using P-256 curve and SHA-256 hash algorithm
ES384	ECDSA using P-384 curve and SHA-384 hash algorithm
ES512	ECDSA using P-521 curve and SHA-512 hash algorithm

Table 4: JSON Web Token Reserved Algorithm Values

Of these algorithms, only HMAC SHA-256 and RSA SHA-256 MUST be implemented by conforming implementations. It is RECOMMENDED that implementations also support the ECDSA P-256 SHA-256 algorithm.

Support for other algorithms is OPTIONAL.

The portion of a JWT that is signed is the same for all algorithms: the concatenation of the JWT Header Segment, a period ('.') character, and the JWT Payload Segment. This character sequence is referred to as the JWT Signing Input. Note that in the JWT Compact Serialization, this corresponds to the portion of the JWT representation preceding the second period character. The UTF-8 representation of the JWT Signing Input is passed to the respective signing algorithms.

8.1. Signing a JWT with HMAC SHA-256

Hash based Message Authentication Codes (HMACs) enable one to use a secret plus a cryptographic hash function to generate a Message Authentication Code (MAC). This can be used to demonstrate that the MAC matches the hashed content, in this case the JWT Signing Input, which therefore demonstrates that whoever generated the MAC was in possession of the secret.

The algorithm for implementing and validating HMACs is provided in RFC 2104 [RFC2104]. Although any HMAC can be used with JWTs, this section defines the use of the SHA-256 cryptographic hash function as defined in FIPS 180-3 [FIPS.180-3]. The reserved "alg" header parameter value "HS256" is used in the JWT Header Segment to indicate that the JWT Crypto Segment contains a base64url encoded HMAC SHA-256 HMAC value.

The HMAC SHA-256 MAC is generated as follows:

1. Apply the HMAC SHA-256 algorithm to the UTF-8 representation of the JWT Signing Input using the shared key to produce an HMAC.

2. Base64url encode the HMAC as defined in this document.

The output is placed in the JWT Crypto Segment for that JWT.

The HMAC SHA-256 MAC on a JWT is validated as follows:

1. Apply the HMAC SHA-256 algorithm to the UTF-8 representation of the JWT Signing Input of the JWT using the shared key.

2. Base64url encode the previously generated HMAC as defined in this document.

3. If the JWT Crypto Segment and the previously calculated value exactly match, then one has confirmation that the key was used to generate the HMAC on the JWT and that the contents of the JWT

have not be tampered with.

4. If the validation fails, the token MUST be rejected.

Signing with the HMAC SHA-384 and HMAC SHA-512 algorithms is performed identically to the procedure for HMAC SHA-256 - just with correspondingly longer key and result values.

8.2. Signing a JWT with RSA SHA-256

This section defines the use of the RSASSA-PKCS1-v1_5 signature algorithm as defined in RFC 3447 [RFC3447], Section 8.2 (commonly known as PKCS#1), using SHA-256 as the hash function. Note that the use of the RSASSA-PKCS1-v1_5 algorithm is described in FIPS 186-3 [FIPS.186-3], Section 5.5, as is the SHA-256 cryptographic hash function, which is defined in FIPS 180-3 [FIPS.180-3]. The reserved "alg" header parameter value "RS256" is used in the JWT Header Segment to indicate that the JWT Crypto Segment contains an RSA SHA-256 signature.

A 2048-bit or longer key length MUST be used with this algorithm.

The RSA SHA-256 signature is generated as follows:

1. Let K be the signer’s RSA private key and let M be the UTF-8 representation of the JWT Signing Input.

2. Compute the octet string S = RSASSA-PKCS1-V1_5-SIGN (K, M) using SHA-256 as the hash function.

3. Base64url encode the octet string S, as defined in this document.

The output is placed in the JWT Crypto Segment for that JWT.

The RSA SHA-256 signature on a JWT is validated as follows:

1. Take the JWT Crypto Segment and base64url decode it into an octet string S. If decoding fails, then the token MUST be rejected.

2. Let M be the UTF-8 representation of the JWT Signing Input and let (n, e) be the public key corresponding to the private key used by the signer.

3. Validate the signature with RSASSA-PKCS1-V1_5-VERIFY ((n, e), M, S) using SHA-256 as the hash function.

4. If the validation fails, the token MUST be rejected.

Signing with the RSA SHA-384 and RSA SHA-512 algorithms is performed identically to the procedure for RSA SHA-256 - just with correspondingly longer key and result values.

8.3. Signing a JWT with ECDSA P-256 SHA-256

The Elliptic Curve Digital Signature Algorithm (ECDSA) is defined by FIPS 186-3 [FIPS.186-3]. ECDSA provides for the use of Elliptic Curve cryptography, which is able to provide equivalent security to RSA cryptography but using shorter key lengths and with greater processing speed. This means that ECDSA signatures will be substantially smaller in terms of length than equivalently strong RSA Digital Signatures.

This specification defines the use of ECDSA with the P-256 curve and the SHA-256 cryptographic hash function. The P-256 curve is also defined in FIPS 186-3. The reserved "alg" header parameter value "ES256" is used in the JWT Header Segment to indicate that the JWT Crypto Segment contains an ECDSA P-256 SHA-256 signature.

A JWT is signed with an ECDSA P-256 SHA-256 signature as follows:

1. Generate a digital signature of the UTF-8 representation of the JWT Signing Input using ECDSA P-256 SHA-256 with the desired private key. The output will be the EC point (R, S), where R and S are unsigned integers.

2. Turn R and S into byte arrays in big endian order. Each array will be 32 bytes long.

3. Concatenate the two byte arrays in the order R and then S.

4. Base64url encode the 64 byte array as defined in this specification.

The output becomes the JWT Crypto Segment for the JWT.

The following procedure is used to validate the ECDSA signature of a JWT:

1. Take the JWT Crypto Segment and base64url decode it into a byte array. If decoding fails, the token MUST be rejected.

2. The output of the base64url decoding MUST be a 64 byte array.

3. Split the 64 byte array into two 32 byte arrays. The first array will be R and the second S. Remember that the byte arrays are in big endian byte order; please check the ECDSA validator in use to

see what byte order it requires.

4. Submit the UTF-8 representation of the JWT Signing Input, R, S and the public key (x, y) to the ECDSA P-256 SHA-256 validator.

5. If the validation fails, the token MUST be rejected.

The ECDSA validator will then determine if the digital signature is valid, given the inputs. Note that ECDSA digital signature contains a value referred to as K, which is a random number generated for each digital signature instance. This means that two ECDSA digital signatures using exactly the same input parameters will output different signatures because their K values will be different. The consequence of this is that one must validate an ECDSA signature by submitting the previously specified inputs to an ECDSA validator.

Signing with the ECDSA P-384 SHA-384 and ECDSA P-521 SHA-512 algorithms is performed identically to the procedure for ECDSA P-256 SHA-256 - just with correspondingly longer key and result values.

8.4. Additional Algorithms

Additional algorithms MAY be used to protect JWTs with corresponding "alg" header parameter values being defined to refer to them. Like claim names, new "alg" header parameter values SHOULD either be defined in the IANA JSON Web Token Algorithms registry or be a URI that contains a collision resistant namespace. In particular, the use of algorithm identifiers defined in XML DSIG [RFC3275] and related specifications is permitted.

9. JWT Serialization Formats

JSON Web Tokens (JWTs) support two serialization formats: the JWT Compact Serialization, which is more space efficient and intended for uses where the token is passed as a simple string-valued parameter, and the JWT JSON Serialization, which is more general, being able to contain multiple signatures over the same content. The two serialization formats are intended for use in different contexts.

9.1. JWT Compact Serialization

The JWT Compact Serialization represents a JWT as a string consisting of three JWT Token Segments: the JWT Header Segment, the JWT Payload Segment, and the JWT Crypto Segment, in that order, with the segments being separated by period ('.') characters. It is intended for uses where the token is passed as a simple string-valued parameter, including in URLs.

The Compact Serialization contains only one signature to keep this format simple. The example JWT in Section 3.1 uses the Compact Serialization.

9.2. JWT JSON Serialization

The JWT JSON Serialization represents a JWT as a JSON object with members for each of three kinds of JWT Token Segments: a "header" member whose value is a non-empty array of JWT Header Segments, a "payload" member whose value is the JWT Payload Segment, and a "signature" member whose value is a non-empty array of JWT Crypto Segments, where the cardinality of both arrays is the same.

Unlike the Compact Serialization, JWTs using the JSON Serialization MAY contain multiple signatures. Each signature is represented as a JWT Crypto Segment in the "signature" member array. For each signature, there is a corresponding "header" member array element that specifies the signature algorithm for that signature, and potentially other information as well. Therefore, the syntax is:

```
{"header":["<header 1 contents>",...,"<header N contents>"],
 "payload":"<payload contents>",
 "signature":["<signature 1 contents>",...,"<signature N contents>"]
}
```

The i’th signature is computed on the concatenation of <header i contents>.<payload contents>.

Appendix A.4 contains an example JWT using the JSON Serialization.

10. IANA Considerations

This specification calls for:

- A new IANA registry entitled "JSON Web Token Claims" for reserved claim names is defined in Section 4.1. Inclusion in the registry is RFC Required in the RFC 5226 [RFC5226] sense for reserved JWT claim names that are intended to be interoperable between implementations. The registry will just record the reserved claim name and a pointer to the RFC that defines it. This specification defines inclusion of the claim names defined in Table 1.

- A new IANA registry entitled "JSON Web Token Header Parameters" for reserved header parameter names is defined in Section 5.1. Inclusion in the registry is RFC Required in the RFC 5226 [RFC5226] sense for reserved JWT header parameter names that are intended to be interoperable between implementations. The registry will just record the reserved header parameter name and a

pointer to the RFC that defines it. This specification defines inclusion of the header parameter names defined in Table 3.

- o A new IANA registry entitled "JSON Web Token Algorithms" for reserved values used with the "alg" header parameter values is defined in Section 8.4. Inclusion in the registry is RFC Required in the RFC 5226 [RFC5226] sense. The registry will just record the "alg" value and a pointer to the RFC that defines it. This specification defines inclusion of the algorithm values defined in Table 4.

11. Security Considerations

TBD: Lots of work to do here. We need to remember to look into any issues relating to security and JSON parsing. One wonders just how secure most JSON parsing libraries are. Were they ever hardened for security scenarios? If not, what kind of holes does that open up? Also, we need to walk through the JSON standard and see what kind of issues we have especially around comparison of names. For instance, comparisons of claim names and other parameters must occur after they are unescaped. Need to also put in text about: Importance of keeping secrets secret. Rotating keys. Strengths and weaknesses of the different algorithms.

TBD: Need to put in text about why strict JSON validation is necessary. Basically, that if malformed JSON is received then the intent of the sender is impossible to reliably discern. While in non-security contexts it's o.k. to be generous in what one accepts, in security contexts this can lead to serious security holes. For example, malformed JSON might indicate that someone has managed to find a security hole in the issuer's code and is leveraging it to get the issuer to issue "bad" tokens whose content the attacker can control.

11.1. Unicode Comparison Security Issues

Claim names in JWTs are Unicode strings. For security reasons, the representations of these names must be compared verbatim after performing any escape processing (as per RFC 4627 [RFC4627], Section 2.5).

This means, for instance, that these JSON strings must compare as being equal ("JWT", "\u004aWT"), whereas these must all compare as being not equal to the first set or to each other ("jwt", "Jwt", "JW\u0074").

JSON strings MAY contain characters outside the Unicode Basic

Multilingual Plane. For instance, the G clef character (U+1D11E) may be represented in a JSON string as "\uD834\uDD1E". Ideally, JWT implementations SHOULD ensure that characters outside the Basic Multilingual Plane are preserved and compared correctly; alternatively, if this is not possible due to these characters exercising limitations present in the underlying JSON implementation, then input containing them MUST be rejected.

12. Open Issues and Things To Be Done (TBD)

The following items remain to be done in this draft (and related drafts):

- The specification will be a lot clearer if the signature portions are cleanly separated from the claims token format and serialization portions. Having tried it this way and being dissatisfied with the sometimes unwieldy readability of the result, I plan to perform the separation in the next draft.

- Consider whether there is a better term than "Digital Signature" for the concept that includes both HMACs and digital signatures using public keys.

- Consider whether we really want to allow private claim names and header parameters that are not registered with IANA and are not in collision-resistant namespaces. Eventually this could result in interop nightmares where you need to have different code to talk to different endpoints that "knows" about each endpoints' private parameters.

- Clarify the optional ability to provide type information JWTs and/or their segments. Specifically, clarify the intended use of the "typ" Header Parameter and the "typ" claim, whether they convey syntax or semantics, and indeed, whether this is the right approach. Also clarify the relationship between these type values and MIME [RFC2045] types.

- Clarify the semantics of the "kid" (key ID) header parameter. Open issues include: What happens if a kid header is received with an unrecognized value? Is that an error? Should it be treated as if it's empty? What happens if the header has a recognized value but the value doesn't match the key associated with that value, but it does match another key that is associated with the issuer? Is that an error?

- The "x5t" parameter is currently specified as "a base64url encoded SHA-256 thumbprint of the DER encoding of an X.509 certificate".

SHA-1 was traditionally used for certificate digests but collisions are possible to create and can be used for denial of service attacks within multi-tenant services. We need to understand the compatibility issues of using SHA-256 thumbprints instead. We also likely want to specify the digest algorithm explicitly.

- Several people have objected to the requirement for implementing RSA SHA-256, some because they will only be using HMACs and symmetric keys, and others because they only want to use ECDSA when using asymmetric keys, either for security or key length reasons, or both. I believe therefore, that we should consider changing the MUST for RSA SHA-256 to RECOMMENDED.

- Since RFC 3447 Section 8 explicitly calls for people NOT to adopt RSASSA-PKCS1 for new applications and instead requests that people transition to RSASSA-PSS, we probably need some Security Considerations text explaining why RSASSA-PKCS1 is being used (it's what's commonly implemented) and what the potential consequences are.

- Generalize the normative text on signing algorithms so that the descriptions apply equally to the use of various key lengths - not just HMAC SHA-256, RSA SHA-256, and ECDSA P-256 SHA-256.

- Add a table cross-referencing the algorithm name strings used in standard software packages and specifications.

- Add Security Considerations text on timing attacks.

- Finish the Security Considerations section.

- Sort out what to do with the IANA registries if this is first standardized as an OpenID specification.

- Write the related specification for encoding public keys using JSON, as per the agreement documented at http://self-issued.info/?p=390. This will be used by the "jku" (JSON Key URL) header parameter.

- Write the companion encryption specification, per the agreements documented at http://self-issued.info/?p=378.

13. References

13.1. Normative References

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

[FIPS.186-3]
National Institute of Standards and Technology, "Digital Signature Standard (DSS)", FIPS PUB 186-3, June 2009.

[RFC1738] Berners-Lee, T., Masinter, L., and M. McCahill, "Uniform Resource Locators (URL)", RFC 1738, December 1994.

[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.

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

[RFC3447] Jonsson, J. and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1", RFC 3447, February 2003.

[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.

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

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

[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.

[USA15] Davis, M., Whistler, K., and M. Duerst, "Unicode Normalization Forms", Unicode Standard Annex 15, 09 2009.

13.2. Informative References

[CanvasApp]
Facebook, "Canvas Applications", 2010.

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

[MagicSignatures]
Panzer (editor), J., Laurie, B., and D. Balfanz, "Magic Signatures", August 2010.

[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.

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

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

[SWT] Hardt, D. and Y. Goland, "Simple Web Token (SWT)", Version 0.9.5.1, November 2009.

[W3C.CR-xml11-20021015]
Cowan, J., "Extensible Markup Language (XML) 1.1", W3C CR CR-xml11-20021015, October 2002.

Appendix A. JWT Examples

A.1. JWT using HMAC SHA-256

A.1.1. Encoding

The Decoded JWT Payload Segment used in this example is:
{"iss":"joe",
 "exp":1300819380,
 "http://example.com/is_root":true}

Note that white space is explicitly allowed in Decoded JWT Claims Objects and no canonicalization is performed before encoding. The following byte array contains the UTF-8 characters for the Decoded JWT Payload Segment:

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

Base64url encoding the above yields the JWT Payload Segment value:

eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ

The following example JSON header object declares that the data structure is a JSON Web Token (JWT) and the JWT Signing Input is signed using the HMAC SHA-256 algorithm:
{"typ":"JWT",
 "alg":"HS256"}

The following byte array contains the UTF-8 characters for the Decoded JWT Header Segment:

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

Base64url encoding this UTF-8 representation yields this JWT Header Segment value:

eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9

Concatenating the JWT Header Segment, a period character, and the JWT Payload Segment yields this JWT Signing Input value (with line breaks for display purposes only):

eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ

The UTF-8 representation of the JWT Signing Input is the following byte array:

[101, 121, 74, 48, 101, 88, 65, 105, 79, 105, 74, 75, 86, 49, 81,

105, 76, 65, 48, 75, 73, 67, 74, 104, 98, 71, 99, 105, 79, 105, 74, 73, 85, 122, 73, 49, 78, 105, 74, 57, 46, 101, 121, 74, 112, 99, 51, 77, 105, 79, 105, 74, 113, 98, 50, 85, 105, 76, 65, 48, 75, 73, 67, 74, 108, 101, 72, 65, 105, 79, 106, 69, 122, 77, 68, 65, 52, 77, 84, 107, 122, 79, 68, 65, 115, 68, 81, 111, 103, 73, 109, 104, 48, 100, 72, 65, 54, 76, 121, 57, 108, 101, 71, 70, 116, 99, 71, 120, 108, 76, 109, 78, 118, 98, 83, 57, 112, 99, 49, 57, 121, 98, 50, 57, 48, 73, 106, 112, 48, 99, 110, 86, 108, 102, 81]

HMACs are generated using keys. This example used the key represented by the following byte array:

[3, 35, 53, 75, 43, 15, 165, 188, 131, 126, 6, 101, 119, 123, 166, 143, 90, 179, 40, 230, 240, 84, 201, 40, 169, 15, 132, 178, 210, 80, 46, 191, 211, 251, 90, 146, 210, 6, 71, 239, 150, 138, 180, 195, 119, 98, 61, 34, 61, 46, 33, 114, 5, 46, 79, 8, 192, 205, 154, 245, 103, 208, 128, 163]

Running the HMAC SHA-256 algorithm on the UTF-8 representation of the JWT Signing Input with this key yields the following byte array:

[116, 24, 223, 180, 151, 153, 224, 37, 79, 250, 96, 125, 216, 173, 187, 186, 22, 212, 37, 77, 105, 214, 191, 240, 91, 88, 5, 88, 83, 132, 141, 121]

Base64url encoding the above HMAC output yields the JWT Crypto Segment value:
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk

Combining these segments in the order Header.Payload.Signature with period characters between the segments yields this complete JWT using the JWT Compact Serialization (with line breaks for display purposes only):

```
eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19y
b290Ijp0cnVlfQ
.
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

A.1.2. Decoding

Decoding the JWT first requires removing the base64url encoding from the JWT Header Segment, the JWT Payload Segment, and the JWT Crypto Segment. We base64url decode the segments per Section 7 and turn them into the corresponding byte arrays. We translate the header segment byte array containing UTF-8 encoded characters into the Decoded JWT Header Segment string. Likewise, if the payload represents a JWT Claims Object, we translate the payload segment byte

array containing UTF-8 encoded characters into a Decoded JWT Claims Object string.

A.1.3. Validating

Next we validate the decoded results. Since the "alg" parameter in the header is "HS256", we validate the HMAC SHA-256 signature contained in the JWT Crypto Segment. If any of the validation steps fail, the token MUST be rejected.

First, we validate that the decoded JWT Header Segment string is legal JSON.

If the payload represents a JWT Claims Object, we also validate that the decoded JWT Payload Segment string is legal JSON.

To validate the signature, we repeat the previous process of using the correct key and the UTF-8 representation of the JWT Signing Input as input to a SHA-256 HMAC function and then taking the output and determining if it matches the Decoded JWT Crypto Segment. If it matches exactly, the token has been validated.

A.2. JWT using RSA SHA-256

A.2.1. Encoding

The Decoded JWT Payload Segment used in this example is the same as in the previous example:
{"iss":"joe",
 "exp":1300819380,
 "http://example.com/is_root":true}

Since the JWT Payload Segment will therefore be the same, its computation is not repeated here. However, the Decoded JWT Header Segment is different in two ways: First, because a different algorithm is being used, the "alg" value is different. Second, for illustration purposes only, the optional "typ" parameter is not used. (This difference is not related to the signature algorithm employed.) The Decoded JWT Header Segment used is:
{"alg":"RS256"}

The following byte array contains the UTF-8 characters for the Decoded JWT Header Segment:

[123, 34, 97, 108, 103, 34, 58, 34, 82, 83, 50, 53, 54, 34, 125]

Base64url encoding this UTF-8 representation yields this JWT Header Segment value:

eyJhbGciOiJSUzI1NiJ9

Concatenating the JWT Header Segment, a period character, and the JWT Payload Segment yields this JWT Signing Input value (with line breaks for display purposes only):

```
eyJhbGciOiJSUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19y
b290Ijp0cnVlfQ
```

The UTF-8 representation of the JWT Signing Input is the following byte array:

```
[101, 121, 74, 104, 98, 71, 99, 105, 79, 105, 74, 83, 85, 122, 73,
49, 78, 105, 74, 57, 46, 101, 121, 74, 112, 99, 51, 77, 105, 79, 105,
74, 113, 98, 50, 85, 105, 76, 65, 48, 75, 73, 67, 74, 108, 101, 72,
65, 105, 79, 106, 69, 122, 77, 68, 65, 52, 77, 84, 107, 122, 79, 68,
65, 115, 68, 81, 111, 103, 73, 109, 104, 48, 100, 72, 65, 54, 76,
121, 57, 108, 101, 71, 70, 116, 99, 71, 120, 108, 76, 109, 78, 118,
98, 83, 57, 112, 99, 49, 57, 121, 98, 50, 57, 48, 73, 106, 112, 48,
99, 110, 86, 108, 102, 81]
```

The RSA key consists of a public part (n, e), and a private exponent d. The values of the RSA key used in this example, presented as the byte arrays representing big endian integers are:

Parameter Name	Value
n	[161, 248, 22, 10, 226, 227, 201, 180, 101, 206, 141, 45, 101, 98, 99, 54, 43, 146, 125, 190, 41, 225, 240, 36, 119, 252, 22, 37, 204, 144, 161, 54, 227, 139, 217, 52, 151, 197, 182, 234, 99, 221, 119, 17, 230, 124, 116, 41, 249, 86, 176, 251, 138, 143, 8, 154, 220, 75, 105, 137, 60, 193, 51, 63, 83, 237, 208, 25, 184, 119, 132, 37, 47, 236, 145, 79, 228, 133, 119, 105, 89, 75, 234, 66, 128, 211, 44, 15, 85, 191, 98, 148, 79, 19, 3, 150, 188, 110, 155, 223, 110, 189, 210, 189, 163, 103, 142, 236, 160, 198, 104, 247, 1, 179, 141, 191, 251, 56, 200, 52, 44, 226, 254, 109, 39, 250, 222, 74, 90, 72, 116, 151, 157, 212, 185, 207, 154, 222, 196, 199, 91, 5, 133, 44, 44, 15, 94, 248, 165, 193, 117, 3, 146, 249, 68, 232, 237, 100, 193, 16, 198, 182, 71, 96, 154, 164, 120, 58, 235, 156, 108, 154, 215, 85, 49, 48, 80, 99, 139, 131, 102, 92, 111, 111, 122, 130, 163, 150, 112, 42, 31, 100, 27, 130, 211, 235, 242, 57, 34, 25, 73, 31, 182, 134, 135, 44, 87, 22, 245, 10, 248, 53, 141, 154, 139, 157, 23, 195, 64, 114, 143, 127, 135, 216, 154, 24, 216, 252, 171, 103, 173, 132, 89, 12, 46, 207, 117, 147, 57, 54, 60, 7, 3, 77, 111, 96, 111, 158, 33, 224, 84, 86, 202, 229, 233, 161]
e	[1, 0, 1]

d	[18, 174, 113, 164, 105, 205, 10, 43, 195, 126, 82, 108, 69, 0, 87, 31, 29, 97, 117, 29, 100, 233, 73, 112, 123, 98, 89, 15, 157, 11, 165, 124, 150, 60, 64, 30, 63, 207, 47, 44, 211, 189, 236, 136, 229, 3, 191, 198, 67, 155, 11, 40, 200, 47, 125, 55, 151, 103, 31, 82, 19, 238, 216, 193, 90, 37, 216, 213, 206, 160, 2, 94, 227, 171, 46, 139, 127, 121, 33, 111, 198, 59, 234, 86, 39, 83, 180, 6, 68, 198, 161, 81, 39, 217, 178, 149, 69, 64, 160, 187, 225, 163, 5, 86, 152, 45, 78, 159, 222, 95, 100, 37, 241, 77, 75, 113, 52, 65, 181, 93, 199, 59, 155, 74, 237, 204, 146, 172, 227, 146, 126, 55, 245, 125, 12, 253, 94, 117, 129, 250, 81, 44, 143, 73, 97, 169, 235, 11, 128, 248, 168, 7, 70, 114, 138, 85, 255, 70, 71, 31, 52, 37, 6, 59, 157, 83, 100, 47, 94, 222, 30, 132, 214, 19, 8, 26, 250, 92, 34, 208, 81, 40, 91, 214, 59, 148, 59, 86, 93, 137, 138, 5, 104, 84, 19, 229, 60, 60, 108, 101, 37, 255, 31, 227, 78, 61, 220, 112, 240, 213, 100, 80, 253, 164, 139, 161, 46, 16, 78, 157, 235, 159, 184, 24, 129, 225, 196, 189, 242, 93, 146, 71, 244, 80, 200, 101, 146, 121, 104, 231, 115, 52, 244, 65, 79, 117, 167, 80, 225, 57, 84, 110, 58, 138, 115, 157]

The RSA private key (n, d) is then passed to the RSA signing function, which also takes the hash type, SHA-256, and the UTF-8 representation of the JWT Signing Input as inputs. The result of the signature is a byte array S, which represents a big endian integer. In this example, S is:

Result Name	Value
S	[112, 46, 33, 137, 67, 232, 143, 209, 30, 181, 216, 45, 191, 120, 69, 243, 65, 6, 174, 27, 129, 255, 247, 115, 17, 22, 173, 209, 113, 125, 131, 101, 109, 66, 10, 253, 60, 150, 238, 221, 115, 162, 102, 62, 81, 102, 104, 123, 0, 11, 135, 34, 110, 1, 135, 237, 16, 115, 249, 69, 229, 130, 173, 252, 239, 22, 216, 90, 121, 142, 232, 198, 109, 219, 61, 184, 151, 91, 23, 208, 148, 2, 190, 237, 213, 217, 217, 112, 7, 16, 141, 178, 129, 96, 213, 248, 4, 12, 167, 68, 87, 98, 184, 31, 190, 127, 249, 217, 46, 10, 231, 111, 36, 242, 91, 51, 187, 230, 244, 74, 230, 30, 177, 4, 10, 203, 32, 4, 77, 62, 249, 18, 142, 212, 1, 48, 121, 91, 212, 189, 59, 65, 238, 202, 208, 102, 171, 101, 25, 129, 253, 228, 141, 247, 127, 55, 45, 195, 139, 159, 175, 221, 59, 239, 177, 139, 93, 163, 204, 60, 46, 176, 47, 158, 58, 65, 214, 18, 202, 173, 21, 145, 18, 115, 160, 95, 35, 185, 232, 56, 250, 175, 132, 157, 105, 132, 41, 239, 90, 30, 136, 121, 130, 54, 195, 212, 14, 96, 69, 34, 165, 68, 200, 242, 122, 122, 45, 184, 6, 99, 209, 108, 247, 202, 234, 86, 222, 64, 92, 178, 33, 90, 69, 178, 194, 85, 102, 181, 90, 193, 167, 72, 160, 112, 223, 200, 163, 42, 70, 149, 67, 208, 25, 238, 251, 71]

Base64url encoding the signature produces this value for the JWT Crypto Segment:

```
cC4hiUPoj9Eetdgtv3hF80EGrhuB__dzERat0XF9g2VtQgr9PJbu3XOiZj5RZmh7AAuHIm4Bh-0Qc_lF
5YKt_O8W2Fp5jujGbds9uJdbF9CUAr7t1dnZcAcQjbKBYNX4BAynRFdiuB--f_nZLgrnbyTyWzO75vRK
5h6xBArLIARNPvkSjtQBMHlb1L07Qe7K0GarZRmB_eSN9383LcOLn6_dO--xi12jzDwusC-eOkHWEsqt
FZESc6BfI7noOPqvhJ1phCnvWh6IeYI2w9QOYEUipUTI8np6LbgGY9Fs98rqVt5AXLIhWkWywlVmtVrB
p0igcN_IoypGlUPQGe77Rw
```

Combining these segments in the order Header.Payload.Signature with period characters between the segments yields this complete JWT using the JWT Compact Serialization (with line breaks for display purposes only):

```
eyJhbGciOiJSUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19y
b290Ijp0cnVlfQ
.
cC4hiUPoj9Eetdgtv3hF80EGrhuB__dzERat0XF9g2VtQgr9PJbu3XOiZj5RZmh7AAuHIm4Bh-0Qc_lF
5YKt_O8W2Fp5jujGbds9uJdbF9CUAr7t1dnZcAcQjbKBYNX4BAynRFdiuB--f_nZLgrnbyTyWzO75vRK
5h6xBArLIARNPvkSjtQBMHlb1L07Qe7K0GarZRmB_eSN9383LcOLn6_dO--xi12jzDwusC-eOkHWEsqt
FZESc6BfI7noOPqvhJ1phCnvWh6IeYI2w9QOYEUipUTI8np6LbgGY9Fs98rqVt5AXLIhWkWywlVmtVrB
p0igcN_IoypGlUPQGe77Rw
```

A.2.2. Decoding

Decoding the JWT from this example requires processing the JWT Header Segment and JWT Payload Segment exactly as done in the first example.

A.2.3. Validating

Since the "alg" parameter in the header is "RS256", we validate the RSA SHA-256 signature contained in the JWT Crypto Segment. If any of the validation steps fail, the token MUST be rejected.

First, we validate that the decoded JWT Header Segment string is legal JSON.

If the payload represents a JWT Claims Object, we also validate that the decoded JWT Payload Segment string is legal JSON.

Validating the JWT Crypto Segment is a little different from the previous example. First, we base64url decode the JWT Crypto Segment to produce a signature S to check. We then pass (n, e), S and the UTF-8 representation of the JWT Signing Input to an RSA signature verifier that has been configured to use the SHA-256 hash function.

A.3. JWT using ECDSA P-256 SHA-256

A.3.1. Encoding

The Decoded JWT Payload Segment used in this example is the same as in the previous examples:
{"iss":"joe",
 "exp":1300819380,
 "http://example.com/is_root":true}

Since the JWT Payload Segment will therefore be the same, its computation is not repeated here. However, the Decoded JWT Header Segment is differs from the previous example because a different algorithm is being used. The Decoded JWT Header Segment used is:
{"alg":"ES256"}

The following byte array contains the UTF-8 characters for the Decoded JWT Header Segment:

[123, 34, 97, 108, 103, 34, 58, 34, 69, 83, 50, 53, 54, 34, 125]

Base64url encoding this UTF-8 representation yields this JWT Header Segment value:
eyJhbGciOiJFUzI1NiJ9

Concatenating the JWT Header Segment, a period character, and the JWT Payload Segment yields this JWT Signing Input value (with line breaks for display purposes only):

eyJhbGciOiJFUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ

The UTF-8 representation of the JWT Signing Input is the following byte array:

[101, 121, 74, 104, 98, 71, 99, 105, 79, 105, 74, 70, 85, 122, 73, 49, 78, 105, 74, 57, 46, 101, 121, 74, 112, 99, 51, 77, 105, 79, 105, 74, 113, 98, 50, 85, 105, 76, 65, 48, 75, 73, 67, 74, 108, 101, 72, 65, 105, 79, 106, 69, 122, 77, 68, 65, 52, 77, 84, 107, 122, 79, 68, 65, 115, 68, 81, 111, 103, 73, 109, 104, 48, 100, 72, 65, 54, 76, 121, 57, 108, 101, 71, 70, 116, 99, 71, 120, 108, 76, 109, 78, 118, 98, 83, 57, 112, 99, 49, 57, 121, 98, 50, 57, 48, 73, 106, 112, 48, 99, 110, 86, 108, 102, 81]

The ECDSA key consists of a public part, the EC point (x, y), and a private part d. The values of the ECDSA key used in this example, presented as the byte arrays representing big endian integers are:

Parameter Name	Value
x	[127, 205, 206, 39, 112, 246, 196, 93, 65, 131, 203, 238, 111, 219, 75, 123, 88, 7, 51, 53, 123, 233, 239, 19, 186, 207, 110, 60, 123, 209, 84, 69]
y	[199, 241, 68, 205, 27, 189, 155, 126, 135, 44, 223, 237, 185, 238, 185, 244, 179, 105, 93, 110, 169, 11, 36, 173, 138, 70, 35, 40, 133, 136, 229, 173]
d	[142, 155, 16, 158, 113, 144, 152, 191, 152, 4, 135, 223, 31, 93, 119, 233, 203, 41, 96, 110, 190, 210, 38, 59, 95, 87, 194, 19, 223, 132, 244, 178]

The ECDSA private part d is then passed to an ECDSA signing function, which also takes the curve type, P-256, the hash type, SHA-256, and the UTF-8 representation of the JWT Signing Input as inputs. The result of the signature is the EC point (R, S), where R and S are unsigned integers. In this example, the R and S values, given as byte arrays representing big endian integers are:

Result Name	Value
R	[14, 209, 33, 83, 121, 99, 108, 72, 60, 47, 127, 21, 88, 7, 212, 2, 163, 178, 40, 3, 58, 249, 124, 126, 23, 129, 154, 195, 22, 158, 166, 101]
S	[197, 10, 7, 211, 140, 60, 112, 229, 216, 241, 45, 175, 8, 74, 84, 128, 166, 101, 144, 197, 242, 147, 80, 154, 143, 63, 127, 138, 131, 163, 84, 213]

Concatenating the S array to the end of the R array and base64url encoding the result produces this value for the JWT Crypto Segment:

```
DtEhU3ljbEg8L38VWAfUAqOyKAM6-Xx-F4GawxaepmXFCgfTjDxw5djxLa8ISlSApmWQxfKTUJqPP3-K
g6NU1Q
```

Combining these segments in the order Header.Payload.Signature with period characters between the segments yields this complete JWT using the JWT Compact Serialization (with line breaks for display purposes only):

```
eyJhbGciOiJFUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19y
b290Ijp0cnVlfQ
.
DtEhU3ljbEg8L38VWAfUAqOyKAM6-Xx-F4GawxaepmXFCgfTjDxw5djxLa8ISlSApmWQxfKTUJqPP3-K
g6NU1Q
```

A.3.2. Decoding

Decoding the JWT from this example requires processing the JWT Header Segment and JWT Payload Segment exactly as done in the first example.

A.3.3. Validating

Since the "alg" parameter in the header is "ES256", we validate the ECDSA P-256 SHA-256 signature contained in the JWT Crypto Segment. If any of the validation steps fail, the token MUST be rejected.

First, we validate that the decoded JWT Header Segment string is legal JSON.

If the payload represents a JWT Claims Object, we also validate that the decoded JWT Payload Segment string is legal JSON.

Validating the JWT Crypto Segment is a little different from the first example. First, we base64url decode the JWT Crypto Segment as in the previous examples but we then need to split the 64 member byte array that must result into two 32 byte arrays, the first R and the second S. We then pass (x, y), (R, S) and the UTF-8 representation of

the JWT Signing Input to an ECDSA signature verifier that has been configured to use the P-256 curve with the SHA-256 hash function.

As explained in Section 8.3, the use of the k value in ECDSA means that we cannot validate the correctness of the signature in the same way we validated the correctness of the HMAC. Instead, implementations MUST use an ECDSA validator to validate the signature.

A.4. JWT using JSON Serialization

Previous example JWTs shown have used the JWT Compact Serialization. This section contains an example JWT using the JWT JSON Serialization. This example demonstrates the capability for conveying multiple signatures for the same JWT.

A.4.1. Encoding

The Decoded JWT Payload Segment used in this example is the same as in the previous examples:
{"iss":"joe",
 "exp":1300819380,
 "http://example.com/is_root":true}

Two signatures are used in this JWT: an RSA SHA-256 signature, for which the header and signature values are the same as in Appendix A.2, and an ECDSA P-256 SHA-256 signature, for which the header and signature values are the same as in Appendix A.3. The two Decoded JWT Header Segments used are:
{"alg":"RS256"}

and:
{"alg":"ES256"}

Since the computations for all JWT Token Segments used in this example were already presented in previous examples, they are not repeated here.

A JSON Serialization of this JWT is as follows:


```
{"header":[
  "eyJhbGciOiJSUzI1NiJ9",
  "eyJhbGciOiJFUzI1NiJ9"],
 "payload":"eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxl
LmNvbS9pc19yb290Ijp0cnVlfQ",
 "signature":[
  "cC4hiUPoj9Eetdgtv3hF80EGrhuB__dzERat0XF9g2VtQgr9PJbu3XOiZj5RZmh7AAuHIm4Bh-0Qc
_lF5YKt_O8W2Fp5jujGbds9uJdbF9CUAr7t1dnZcAcQjbKBYNX4BAynRFdiuB--f_nZLgrnbyTyWzO75
vRK5h6xBArLIARNPvkSjtQBMHlb1L07Qe7K0GarZRmB_eSN9383LcOLn6_dO--xi12jzDwusC-eOkHWE
sqtFZESc6BfI7noOPqvhJ1phCnvWh6IeYI2w9QOYEUipUTI8np6LbgGY9Fs98rqVt5AXLIhWkWywlVmt
VrBp0igcN_IoypGlUPQGe77Rw",
  "DtEhU3ljbEg8L38VWAfUAqOyKAM6-Xx-F4GawxaepmXFCgfTjDxw5djxLa8ISlSApmWQxfKTUJqPP
3-Kg6NU1Q"]
}
```

A.4.2. Decoding

Decoding the JWT first requires removing the base64url encoding from the array of JWT Header Segments, the JWT Payload Segment, and the array of JWT Crypto Segments. We base64url decode the segments per Section 7 and turn them into the corresponding byte arrays. We translate the header segment byte arrays containing UTF-8 encoded characters into Decoded JWT Header Segment strings. Likewise, if the payload represents a JWT Claims Object, we translate the payload segment byte array into a Decoded JWT Claims Object string.

A.4.3. Validating

If any of the validation steps fail, the token MUST be rejected.

First, we validate that the header and signature arrays contain the same number of elements.

Next, we validate that the Decoded JWT Header Segment strings are all legal JSON.

If the payload represents a JWT Claims Object, we also validate that the decoded JWT Payload Segment string is legal JSON.

Finally, for each Decoded JWT Header Segment, we validate the corresponding signature using the algorithm specified in the "alg" parameter, which must be present.

Appendix B. Notes on implementing base64url encoding without padding

This appendix describes how to implement base64url encoding and decoding functions without padding based upon standard base64 encoding and decoding functions that do use padding.

To be concrete, example C# code implementing these functions is shown below. Similar code could be used in other languages.

```
static string base64urlencode(byte [] arg)
{
  string s = Convert.ToBase64String(arg); // Standard base64 encoder
  s = s.Split('=')[0]; // Remove any trailing '='s
  s = s.Replace('+', '-'); // 62nd char of encoding
  s = s.Replace('/', '_'); // 63rd char of encoding
  return s;
}

static byte [] base64urldecode(string arg)
{
  string s = arg;
  s = s.Replace('-', '+'); // 62nd char of encoding
  s = s.Replace('_', '/'); // 63rd char of encoding
  switch (s.Length % 4) // Pad with trailing '='s
  {
    case 0: break; // No pad chars in this case
    case 2: s += "=="; break; // Two pad chars
    case 3: s += "="; break; // One pad char
    default: throw new System.Exception(
      "Illegal base64url string!");
  }
  return Convert.FromBase64String(s); // Standard base64 decoder
}
```

As per the example code above, the number of '=' padding characters that needs to be added to the end of a base64url encoded string without padding to turn it into one with padding is a deterministic function of the length of the encoded string. Specifically, if the length mod 4 is 0, no padding is added; if the length mod 4 is 2, two '=' padding characters are added; if the length mod 4 is 3, one '=' padding character is added; if the length mod 4 is 1, the input is malformed.

An example correspondence between unencoded and encoded values follows. The byte sequence below encodes into the string below, which when decoded, reproduces the byte sequence.
3 236 255 224 193
A-z_4ME

Appendix C. Relationship of JWTs to SAML Tokens

SAML 2.0 [OASIS.saml-core-2.0-os] provides a standard for creating tokens with much greater expressivity and more security options than supported by JWTs. However, the cost of this flexibility and expressiveness is both size and complexity. In addition, SAML's use of XML [W3C.CR-xml11-20021015] and XML DSIG [RFC3275] only

contributes to the size of SAML tokens.

JWTs are intended to provide a simple token format that is small enough to fit into HTTP headers and query arguments in URIs. It does this by supporting a much simpler token model than SAML and using the JSON [RFC4627] object encoding syntax. It also supports securing tokens using Hash-based Message Authentication Codes (HMACs) and digital signatures using a smaller (and less flexible) format than XML DSIG.

Therefore, while JWTs can do some of the things SAML tokens do, JWTs are not intended as a full replacement for SAML tokens, but rather as a compromise token format to be used when space is at a premium.

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

Both JWTs and Simple Web Tokens SWT [SWT], at their core, enable sets of claims to be communicated between applications. For SWTs, both the claim names and claim values are strings. For JWTs, while claim names are strings, claim values can be any JSON type. Both token types offer cryptographic protection of their content: SWTs with HMAC SHA-256 and JWTs with a choice of algorithms, including HMAC SHA-256, RSA SHA-256, and ECDSA P-256 SHA-256. The signed content of a SWT must be a set of claims, whereas the payload of a JWT, in general, can be any base64url encoded content.

Appendix E. Acknowledgements

The authors acknowledge that the design of JWTs was intentionally influenced by the design and simplicity of Simple Web Tokens [SWT]. Solutions for signing JSON tokens were also previously explored by Magic Signatures [MagicSignatures], JSON Simple Sign [JSS], and Canvas Applications [CanvasApp], all of which influenced this draft.

Appendix F. Document History

-01

- Draft incorporating consensus decisions reached at IIW.

-00

- Public draft published before November 2010 IIW based upon the JSON token convergence proposal incorporating input from several implementers of related specifications.

Authors' Addresses

Michael B. Jones
Microsoft

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

Dirk Balfanz
Google

Email: balfanz@google.com

John Bradley
independent

Email: ve7jtb@ve7jtb.com

Yaron Y. Goland
Microsoft

Email: yarong@microsoft.com

John Panzer
Google

Email: jpanzer@google.com

Nat Sakimura
Nomura Research Institute

Email: n-sakimura@nri.co.jp

Paul Tarjan
Facebook

Email: paul.tarjan@facebook.com

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 29, 2011

M. Jones
Microsoft

March 28, 2011

JSON Web Token (JWT) Bearer Profile for OAuth 2.0
draft-jones-oauth-jwt-bearer-00

Abstract

This specification defines the use of a JSON Web Token (JWT) bearer token as a means of requesting an OAuth 2.0 access token.

Status of this Memo

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

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

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

This Internet-Draft will expire on September 29, 2011.

Table of Contents

1. Introduction

JSON Web Token (JWT) [JWT] is a JSON-based security token encoding that enables identity and security information to be shared across security domains. JWTs utilize JSON data structures, as defined in RFC 4627 [RFC4627].

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 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.

This specification defines an extension grant type that profiles the use of a JSON Web Token (JWT) in requesting an OAuth 2.0 access token.

1.1. Notational Conventions

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

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

1.2. Terminology

All terms are as defined in [I-D.ietf-oauth-v2] and [JWT].

2. JWT Access Token Request

A JSON Web Token (JWT) bearer token can be used to request an access token when a client wishes to utilize an existing trust relationship, expressed through the semantics of the JWT, without a direct user approval step at the authorization server.

The process by which the client obtains the JWT, prior to exchanging it with the authorization server, is out of scope.

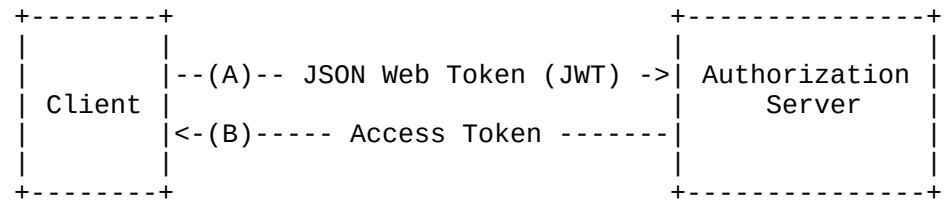


Figure 1: Access Token Request using JWT Bearer Token

The request/response flow illustrated in Figure 1 includes the following steps:

(A) The client sends an access token request to the authorization server that includes a JWT bearer token and a grant_type of "http://oauth.net/grant_type/jwt/1.0/bearer".

(B) The authorization server validates the JWT per the processing rules defined in the JWT specification and in this specification and issues an access token.

2.1. Client Requests Access Token

The client includes the JWT in the access token request, the core details of which are defined in OAuth [I-D.ietf-oauth-v2], by specifying "http://oauth.net/grant_type/jwt/1.0/bearer" as the absolute URI value of the "grant_type" parameter and by adding the following parameter:

jwt
: REQUIRED. The value of the "jwt" parameter MUST be a single JWT that is represented using the Compact Serialization.

scope
: OPTIONAL. The scope of the access request 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.

2.2. JWT Content and Processing Requirements

Prior to issuing an access token response as described in [I-D.ietf-oauth-v2], the authorization server MUST validate the JWT according to the criteria below. If present, the authorization server MUST also validate the client credentials. Application of additional restrictions and policy are at the discretion of the authorization server.

- The JWT MUST contain an "iss" (issuer) claim that contains a unique identifier for the entity that issued the JWT.
- The JWT MUST contain a "prn" (principal) claim. The principal MUST identify the resource owner for whom the access token is being requested.
- The JWT MUST contain an "aud" (audience) claim containing a URI reference that identifies the authorization server as the intended audience. The authorization server MUST verify that it is an intended audience for the JWT.
- The JWT MUST contain an "exp" (expiration) claim that limits the time window during which the JWT 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 MAY reject JWTs with an "exp" claim value that is unreasonably far in the future.
- The JWT MAY contain an "iat" (issued at) claim containing the UTC time at which the JWT was issued. This time is represented as an "IntDate", as defined by [JWT].
- The JWT MAY contain other claims.
- The JWT MUST be digitally signed by the issuer in the manner described in the JWT specification and the authorization server MUST verify the signature.
- The authorization server MUST verify that the JWT is valid in all other respects per [JWT].

2.3. Error Response

If the JWT is not valid or has expired, the authorization server MUST construct an error response as defined in [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 JWT was considered invalid using the "error_description" or "error_uri" parameters.

For example:

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

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

2.4. Example (non-normative)

Though non-normative, the following examples illustrate what a conforming JWT and access token request would look like.

Below is an example JSON object that could be encoded to produce the JWT Claims Object for a JWT:

```
{"iss":"https://jwt-idp.example.com",
 "prn":"mailto:mike@example.com",
 "aud":"https://jwt-rp.example.net",
 "iat":1300815780,
 "exp":1300819380,
 "http://claims.example.com/member":true}
```

Figure 2: Example JWT Claims Object

The following example JSON object, used as the header of a JWT, declares that the JWT is signed with the ECDSA P-256 SHA-256 algorithm.

```
{"alg":"ES256"}
```

Figure 3: Example JWT Header Object

To present a JWT with the claims and header shown above as part of an access token request, for example, the client might make the following HTTPS request (line breaks are for display purposes only):

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

grant_type=http%3A%2F%2Foauth.net%2Fgrant_type%2Fjwt%2F1.0%2Fbearer&
jwt=eyJhbGciOiJFUzI1NiJ9.eyJpc3Mi[...omitted for brevity...].
J9l-ZhwP_2n[...omitted for brevity...]
```

Figure 4: Example Request

3. Security Considerations

Authorization servers SHOULD issue access tokens with a limited lifetime and require clients to refresh them by requesting a new access token using the same JWT, if it is still valid, or with a new JWT. The authorization server SHOULD NOT issue a refresh token.

4. IANA Considerations

4.1. OAuth Parameters Registration

This specification registers the following parameters in the OAuth Parameters Registry established by [I-D.ietf-oauth-v2].

4.1.1. The "jwt" OAuth Parameter

Parameter name: jwt

Parameter usage location: token request

Change controller: IETF

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

Related information: None

5. References

5.1. Normative References

[I-D.ietf-oauth-v2] Hammer-Lahav, E., Recordon, D., and D. Hardt, "The OAuth 2.0 Authorization Protocol", draft-ietf-oauth-v2-13 (work in progress), February 2011.

[JWT] Jones, M., Balfanz, D., Bradley, J., Goland, Y., Panzer, J., Sakimura, N., and P. Tarjan, "JSON Web Token (JWT)", March 2011.

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

[RFC4627] Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)", RFC 4627, July 2006.

5.2. Informative References

[I-D.ietf-oauth-saml2-bearer]
Campbell, B. and C. Mortimore, "SAML 2.0 Bearer Assertion Grant Type Profile for OAuth 2.0", draft-ietf-oauth-saml2-bearer-03 (work in progress), February 2011.

Appendix A. Acknowledgements

This profile was derived from the SAML2 Bearer Assertion Grant Type Profile for OAuth 2.0 [I-D.ietf-oauth-saml2-bearer] by Brian Campbell and Chuck Mortimore.

Appendix B. Document History

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

-00

o Initial draft.

Author’s Address

Michael B. Jones
Microsoft

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

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: July 15, 2011

M. Jones
Y. Goland
Microsoft

January 11, 2011

Simple Web Discovery (SWD)
draft-jones-simple-web-discovery-00

Abstract

Simple Web Discovery (SWD) defines a HTTPS GET based mechanism to discover the location of a given type of service for a given principal starting only with a domain name.

Requirements Language

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

Status of this Memo

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

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

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

This Internet-Draft will expire on July 15, 2011.

Copyright Notice

Table of Contents

1. Introduction

Simple Web Discovery (SWD) defines a HTTPS GET based mechanism to discover the location of a given type of service for a given principal starting only with a domain name. SWD requests use the x-www-form-urlencoded format to specify a URI for the principal and another URI for the type of service being sought. If the request is successful then the response, by default, is a JSON object containing an array of URIs that point to where the principal has instances of services of the requested type.

For example, let us say that a requester wants to discover where Joe keeps his calendar. The requester could take Joe's e-mail address, joe@example.com and take from it its domain to create a HTTPS GET request of the following form:

```
GET /.well-known/simple-web-discovery?principal=mailto:joe@example.com&service=u
rn:adatum.com:calendar HTTP/1.1
Host: example.com

HTTP/1.1 200 O.K.
Content-Type: application/json

{
 "locations":["http://calendars.proseware.com/calendars/joseph"]
}
```

Note: The request-URI is left un-encoded in the above example for the sake of readability in the above example.

2. A Simple Web Discovery Request

Domains that support SWD requests MUST make available a SWD server for their domain at the path .well-known/simple-web-discovery. The syntax and semantics of ".well-known" are defined in RFC 5785 [RFC5785]. "simple-web-discovery" MUST point to a SWD server compliant with this specification.

SWD servers MUST support receiving SWD requests via TLS 1.2 as defined in RFC 5246 [RFC5246] and MAY support other transport layer security mechanisms of equivalent security. SWD servers MUST reject SWD requests sent over plain HTTP or any other transport that does not provide both privacy and validation of the server's identity.

A SWD server is queried using a HTTPS GET request with the previously specified path along with a query segment containing a x-www-form-urlencoded form as defined in HTML 4.01 [W3C.REC-html401-19991224]. The form MUST contain two name/value pairs that MUST appear exactly

once, principal and service. Both name/value pairs MUST have values that are set to URIs (as defined in RFC 3986 [RFC3986] . If any of the previous requirements are not met in a SWD request then the request MUST be rejected with a 400 Bad Request.

The SWD request form MAY contain additional name/value pairs but if those name/value pairs are not recognized by the SWD server then the SWD server MUST ignore them for processing purposes.

The principal query component is a URI that identifies an entity. The service query component is a URI that identifies a service type. The semantics of the SWD query is "Please return the location(s) of instances of the specified service type associated with the specified principal". The definition of URIs used to identify principals and services are outside the scope of this specification.

3. Simple Web Discovery Responses

3.1. A response containing one or more locations

Unless another content-type is negotiated, a 200 O.K. response to a SWD request that contains the information requested MUST return content of type application/json as defined in RFC 4627 [RFC4627]. The JSON response MUST contain a JSON object that contains a member pair whose name is the string "locations" and whose value is an array of strings that are each a URI pointing to a location where the desired service type belonging to the specified principal can be found. There are no semantics associated with the order in which the URIs are listed in the array.

The JSON object MAY contain other members but a receiver of the object MAY ignore any member pairs whose name it does not recognize.

3.2. Redirecting all Simple Web Discovery Requests

SWD requests by definition start off by being issued to the .well-known/simple-web-discovery location. But locating a SWD server at a root location can prove inconvenient. To enable service level redirection a SWD server MAY return a 200 O.k. to a HTTPS request with a content type of application/json (or whatever other content type has been negotiated) that contains a JSON object that contains a member pair whose name is the string "SWD_service_redirect" whose value is a JSON object with a member pair whose name is "location" and whose value is a string that encodes a URI. Optionally the JSON object value of "SWD_service_redirect" MAY also contain a member whose name is "expires" and whose value is a JSON number that encodes an integer.

A SWC compliant client MUST support the SWD_service_redirect response.

The JSON objects MAY contain other members but a receiver of the objects MAY ignore any pairs whose name it does not recognize.

The location member identifies the URI that the caller MUST redirect all SWD requests for that domain to until the expires time is met. SWD requests for the redirected domain MUST be constructed by taking the URI returned in the location and using it as the base URI to which the SWD form arguments are then added as query parameters. The location URI MUST NOT include a query component.

```
GET /.well-known/simple-web-discovery?principal=mailto:joe@example.com&service=u
rn:adatum.com:calendar HTTP/1.1
Host: example.com

HTTP/1.1 200 O.K.
Content-Type: application/json

{
 "SWD_service_redirect":
  {
   "location":"https://swd.proseware.com/swd_server",
   "expires": 1300752001
  }
}

GET /swd_server?principal=mailto:joe@example.com&service=urn:adatum.com:calendar
 HTTP/1.1
Host: swd.proseware.com

HTTP/1.1 200 O.K.
Content-Type: application/json

{
 "locations":["http://calendars.proseware.com/calendars/joseph"]
}
```

Note: The request-URIs are left un-encoded in the above example for the sake of readability in the above example.

The location URI MUST be a HTTPS URL.

The optional expires member identifies the point in time at which the caller MUST NOT redirect its SWD requests for that domain to the previously obtained location and MUST instead return to the .well-known/simple-web-discovery location. The value of the expires member MUST encode the number of seconds from 1970-01-01T0:0:0Z as measured in UTC until the desired date/time. See RFC 3339 [RFC3339] for

details regarding date/times in general and UTC in particular. If the expires value is in the past or if the value is more than one hour in the future then the response MUST be treated as if it didn't contain an expires value.

If the expires value is omitted or if its value is incorrect then the expires value MUST be treated as having a value of exactly one hour into the future.

If a JSON response is received that contains both a member pair with the name "SWD_service_redirect" and a member pair with the name "locations" as children of the object root then the "SWD_service_redirect" member pair MUST be ignored.

3.3. 401 Unauthorized Response

A SWD server MAY respond to a request with a 401 Unauthorized Response, as described in RFC 2616 [RFC2616], Section 10. Per the RFC, the request MAY be repeated with a suitable Authorization header field. Authorization information may be communicated in this manner, including a JSON Web Token [JWT].

3.4. Other HTTP 1.1 Responses

A SWD server MAY return other HTTP 1.1 responses, including 404 Not Found, 400 Bad Request, and 403 Forbidden. SWD implementations MUST correctly handle these responses.

4. IANA Considerations

Per RFC 5785 [RFC5785] the following registration template is offered:

URI suffix simple-web-discovery

Change controller IETF

Specification document This RFC

5. Security Considerations

SWD responses can contain confidential information. Therefore a, general approach is used to require TLS in all cases. But TLS can only provide for privacy and server validation, it cannot validate that the requester is authorized to see the results of a query. The exact mechanism used to determine if the requester is authorized to

see the result of the query is outside the scope of this specification.

Because SWD responses can contain confidential information, the requestor may need authorization to receive them. Standard HTTP authorization mechanisms MAY be employed to request authorized access, including the use of an HTTP Authorization header field in requests, which in turn, may contain a JSON Web Token [JWT], among other authorization data formats.

The ability to redirect an entire SWD server as defined in this document is an obvious attack point. This is another reason why we have mandated TLS, so as to be sure that the redirect can only be received over a secure connection. We have also put in the upper limit of 60 minutes for a redirect so as to provide a path for regaining control over queries should a successful attack be launched to return false redirects.

The SWD_service_redirect capability may cause unanticipated failures in cases where a requestor may have permissions to discover content at the original SWD endpoint but not the one redirected to, or vice-versa.

6. References

6.1. Normative References

[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.

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

[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.

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

[RFC5785] Nottingham, M. and E. Hammer-Lahav, "Defining Well-Known Uniform Resource Identifiers (URIs)", RFC 5785, April 2010.

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

6.2. Informative References

[JWT] Jones (editor), M., Balfanz, D., Bradley, J., Goland, Y., Panzer, J., and N. Sakimura, "JSON Web Token (JWT)", October 2010.

Authors' Addresses

Michael B. Jones
Microsoft

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

Yaron Y. Goland
Microsoft

Email: yarong@microsoft.com

Open Authentication Protocol
Internet-Draft
Intended status: Standards Track
Expires: September 15, 2011

T. Lodderstedt, Ed.
Deutsche Telekom AG
M. McGloin
IBM
P. Hunt
Oracle Corporation

March 14, 2011

OAuth 2.0 Threat Model and Security Considerations
draft-lodderstedt-oauth-security-01

Abstract

This document gives security considerations based on a comprehensive threat model for the OAuth 2.0 Protocol.

Requirements Language

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

Status of this Memo

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

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

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

This Internet-Draft will expire on September 15, 2011.

Copyright Notice

Table of Contents

1. Introduction

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

- Documents any assumptions and scope considered when creating the threat model
- Describe the security features in-built into the OAuth protocol and how they are intended to thwart attacks
- Give 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 client profile or all threats for protecting the resource server.

2. Overview

2.1. Scope

The security considerations document only considers clients bound to a particular deployment as supported by [I-D.ietf-oauth-v2]. Such deployments have the following characteristics:

- Resource server URLs are static and well-known at development time, authorization server URLs can be static or discovered.
- Token scope values (e.g. applicable URLs and methods) are well-known at development time.
- Client registration: Since registration of clients is out of scope of the current core spec, this document assumes a broad variety of options from static registration during development time to dynamic registration at runtime.

The following are considered out of scope :

- Communication between authorization server and resource server
- Token formats
- Except for "Resource Owner Password Credentials" (see [I-D.ietf-oauth-v2], section 4.3), the mechanism used by authorization servers to authenticate the user
- Mechanism by which a user obtained an assertion and any resulting attacks mounted as a result of the assertion being false.
- Clients are not bound to a specific deployment: An example could by 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 must dynamically discover the URLs relevant for the Oauth protocol.

2.2. Attack Assumptions

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

- It is assumed the attacker has full access to the network between the client and service provider and may eaves drop on any communications between the two.
- It is assumed an attacker has unlimited resources to mount an attack.
- It is assumed that 2 parties involved in the OAuth 3 legged 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 treat 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 an 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 MAY be stored or accessible on the authorization server:

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

2.3.2. Resource Server

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

- user data (out of scope)
- HTTPS certificate/key
- authz server credentials (handle-based design), or
- authz server shared secret/public key (assertion-based design)
- 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

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

- client id (and secret)

- refresh tokens (persistent) access tokens (transient)
- trusted CA certs (HTTPS)
- per authorization process: redirect_uri, authorization code

2.3.3.1. Web Server

Such clients typically represent a web site with its own user management and login mechanism and have the following characteristics:

- Tokens are bound to a single user identity at the site
- Web servers are able to protect client secrets
- The potential number of tokens affected by a security breach depends on number of site users.

Such clients are implemented using the authorization code flow (see Section 4.4.1).

2.3.3.2. Native Applications

This class of OAuth clients represent apps running on a user-controlled device, such as a notebook, PC, Tablet, Smartphone, or Gaming Console.

Massively distributed applications such as these cannot reliably keep secrets confidential, which are issued per software package. This is because such secrets would need to be transferred to the user device as part of the installation process. An attacker could reverse engineer any secret from the binary or accompanying resources. Native Applications are able to protect per installation/instance secrets (e.g. refresh tokens) to some extent.

Device platforms typically allow users to lock the device with a pin and to segregate different apps or users (multi-user operation systems).

Some devices can be identified/authenticated (to varying degrees of assurance):

- Handsets and smart phones by its International Mobile Equipment Identity (IMEI)
- Set top boxes, gaming consoles, others by using certificates and TPM module - Note: This does not help to identify client apps but

may be used to bound tokens to devices and to detect token theft

Mobile devices, such as handsets or smart phones have the following special characteristics:

- Limited input capabilities, therefore such clients typically obtain a refresh token in order to provide automatic login for sub-sequent application sessions
- As mobile and small devices, they can get cloned, stolen or lost easier than other devices.
- Security breach will affect single user (or a few users) only.

For the purposes of this document, the scenario of attackers who control a smartphone device entirely is out of scope.

There are several implementation options for native applications:

- The authorization code flow in combination with an embedded or external browser (Section 4.4.1)
- The implicte grant flow in combination with an embedded or external browser (Section 4.4.2)
- The resource owner password credentials flow can be used as well (Section 4.4.3)

Different threats exists for those implementation options, which are discussed in the respective sections of the threat model.

2.3.3.3. User Agent

[TBD]

Such client are implemented using the implicite grant flow (Section 4.4.2).

2.3.3.4. Autonomous

Autonomous clients access service providers using rights grants by client credentials only. Thus the autonomous client becomes the "user". Authenticating autonomous clients is conceptually similar to end-user authentication since the issued tokens refer to the client's identity. Autonomous clients shall always be required to use a secret or some other form of authentication (e.g. client assertion in the form of a SAML assertion or STS token) acceptable to the authorization/token services. The client must ensure the

confidentiality of client_secret or other credential.

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 of tokens. Tokens can be implemented in 2 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 typically require a communication between resource server and token server in order to validate the token and obtain token-bound data. Handles enable simple revocation and do not require cryptographic mechanisms to protected token content from being modified. As a disadvantage, they require additional resource/ token server communication impacting on performance and scalability. An authorization code (OAuth Section 4.1.2) is an example of a 'handle' token. An access token may also be implemented as a handle token. 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, an audience, and is digitally signed containing information about the user and the client. Examples of assertion formats are SAML assertions and Kerberos tickets. 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. Implementing token revocation is more difficult with assertions than with handles.

Tokens can be sent to resource server in two ways:

bearer token A 'bearer token' is a token that can be used by any client who has received the token (cf. [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 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 (mutual authentication) and HoK (holder-of-key) tokens (cf. [I-D.hammer-oauth-v2-mac-token].

3.2. 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.

3.3. Expires_In

Expires_In allows an authorization server (based on its policies or on behalf of the end-user) to limit the lifetime of the access token. This mechanisms 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.

3.4. Authorization Code

An Authorization Code represents the intermediary 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 redirect_uri instead of tokens for two purposes.

3.5. Redirect-URI

A Redirect-uri helps to identify clients and prevents phishing attacks from other clients attempting to trick the user into believing the phisher is the client. The redirect URI is pre-registered as requests with authorization code or token will be directed to that URI. Moreover, the value of the actual redirect_uri has to be presented and is verified when an authorization code is exchanged for tokens. This helps to prevent session fixation attacks.

3.6. Access Token

An Access Token is used by a client to access a resource. An access token must be acquired using a HTTP POST operation to ensure no logging or caching of requests. 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 enables the possibility of revocation by requiring the client to refresh their access token at regular intervals.

3.7. Refresh Token

A Refresh Token is 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 must always be 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.

3.8. Client Authentication

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 security level in delegated authorization scenarios and because the client will be able to act without the user’s presence. By authenticating a client when requesting an access token, the token service is able to assess whether a given client and authorization code meets appropriate security requirements and binds the authorization code approved by the user to the client making the request.

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

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

- the initial authorization and issuance of a tokens by an end-user to a particular client and sub-sequent requests by this client to obtain tokens w/o user consent (automatic processing of repeated authorization)

The client identity 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 identity may be used to limit the number of request for a particular client or to charge the client per request. Client Identity may furthermore be useful to differentiate (e.g. in server log files) between accesses by end-user, and delegated accesses by client on behalf of a user.

The _client_secret_ is used to verify the client identifier. This should only be used where the client is capable of keeping its secret confidential. The client identity can also be verified using the _redirect_uri_ or by the _end-user_.

Clients (and the trustworthiness of its identity) can be classifed by using the following parameters:

- Deployment-specific or -independent client_id (Note: for native apps, every installation of a particular app on a certain device is considered a deployment.)
- Validated properties, such as app name or redirect_uri
- Client_secret available

Typical client categories are:

Deployment-independent client_id with pre-registered redirect_uri and without client_secret Such an identity is used by multiple installations of the same software package. The identity of such a client can only be validated with the help of the end-user. This is a viable option for native apps 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 such an identity will affect ALL deployments of the respective software.

Deployment-independent client_id with pre-registered redirect_uri and with client_secre 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 (cf. Section 4.1.1). Due to its security weaknesses, such client identities have the same trustlevel 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 insures the validation of the client's properties, such as redirect_uri, website address, web site name, contacts. Such a client identity 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 app is conducted by an administrator, who validates the clients authenticity, or the process from validating the app to the installation of the app on the device and the creation of the client credentials is controlled end-to-end by a single entity (e.g. app 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). 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 apps. Revocation will affect a single deployment only.

Use of the client secret is considered a relatively weak form of credential for the client. Use of stronger mechanisms such as a client assertion (e.g. SAML), key cryptography, are preferred.

4. Security Threat Model

This sections gives a comprehensive threat model of OAuth 2.0. Threats are grouped first by attackes 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:

- replay its tokens and authorization codes, or
- obtain tokens on behalf of the attacked client with the privileges of that client.

The resulting impact would be:

- Client authentication of access to authorization server can be bypassed
- Stolen refresh tokens or authorization codes can be replayed

Depending on the client category, there are the following approaches an attacker could utilize to obtain the client secret.

Attack: Obtain Secret From Source Code or Binary. This applies for all client profiles and especially for open source projects, where the source code is public accessible. Even if the attacker does not has access to the source code, it could reverse engineer secrets from the binary of native apps.

Countermeasures:

- Don’t issue secrets to clients with inappropriate security policy - Section 5.2.3.1
- Client_id only in combination with forced user consent - Section 5.2.3.2
- Deployment-specific client secrets - Section 5.2.3.4
- Client secret revocation - 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 app).

Countermeasures:

- Web server: apply standard web server protection measures (for config files and databases) - Section 5.3.2
- Native apps: Store secrets in a secure local storage - Section 5.3.3

- Client secret revocation - 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. Some generally applicable countermeasure to mitigate such attacks shall be given in advance:

- The authorization server must validate the client id associated with the particular refresh token with every refresh request - Section 5.2.2.2
- Limited scope tokens - Section 5.1.5.1
- Refresh token revocation - Section 5.2.2.4
- Client secret revocation - Section 5.2.3.6
- Refresh tokens can automatically be replaced in order to detect unauthorized token usage by another party (Refresh Token Replacement) - Section 5.2.2.3

**

Attack: Obtain Refresh Token from Web application. An attack may obtain the refresh tokens issued to a web server client. Impact: Exposure of all refresh tokens on that side.

Countermeasures:

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

**

Attack: Obtain Refresh Token from Native clients. On native clients, leakage of a refresh token typically affects a single user, only.

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

Countermeasures:

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

__

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

Countermeasures:

- Utilize device lock to prevent unauthorized device access - Section 5.3.4
- Where a user knows the device has been cloned, they can use this countermeasure (Refresh Token Revocation) - Section 5.2.2.4

__

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

Countermeasures:

- Combine refresh token request with device identification - Section 5.2.2.6
- Combine refresh token requests with user-provided secret - Section 5.2.2.5
- Refresh Token Replacement - Section 5.2.2.3
- Where a user knows the device has been cloned, they can use this countermeasure - Refresh Token Revocation - Section 5.2.2.4

__

Obtain refresh tokens from backup: A refresh token could be obtained from the backup of a client application, or device.

Countermeasures:

- tbd

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 app 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.

Countermeasures:

- Keep access tokens in transient memory and limit grants: Section 5.1.6
- Limited scope tokens - Section 5.1.5.1
- Combine refresh token requests with user-provided secret - Section 5.2.2.5
- Client secret revocation - Section 5.2.3.6
- Keep access tokens in private memory or apply same protection means as for refresh tokens - Section 5.2.2
- Keep access token lifetime short - Section 5.1.5.3

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

A malicious app 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 UI. 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 app UI, the client app has access to additional information it should not have access to (e.g. uid/password).

Impact: If the client app 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:

- Client developers and end-user can be educated to trust an external System-Browser only.

- Client apps could be validated prior publication in a app market.

- Client developers should not collect authentication information directly from users and should instead use redirects to and back from a trusted external system-browser.

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:

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

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

Countermeasures:

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

- Narrow scope based on client-specific policy - When obtaining end user authorization and where the client requests scope, the service provider may want to consider whether to honour that scope based on who the client is. That decision is between the client and service provider and is outside the scope of this spec. The service provider may also want to consider what scope to grant based on the profile used, e.g. providing lower scope where no client secret is provided from a native application. - 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 Provider.

A malicious client may exploit that feature and try to obtain such an authorization code instead of the legimate client.

Countermeasures:

- Service providers should not automatically process repeat authorizations where the client is not authenticated through a client secret or some other authentication mechanism such as signing with security certs (see Section 5.2.3.7) or validation of a pre-registered redirect uri (Section 5.2.3.5)

- Service Providers 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 redirect_uri parameter to abuse the authorization server as redirector.

Impact?

Countermeasure

- o don’t redirect to redirect_uri, if client identity or redirect_uri could not be verified

4.3. Token endpoint

4.3.1. Threat: Eavesdropping access tokens

The OAuth specification does not describe any mechanism for protecting Tokens from eavesdroppers when they are transmitted from the Service Provider to the Client.

Countermeasures:

- o Service Providers MUST ensure that these transmissions are protected using transport-layer mechanisms such as TLS or SSL (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 System security measures - Section 5.1.4.1.1

- o Store access token hashes only - Section 5.1.4.1.3

- o Standard SQL inj. Countermeasures - Section 5.1.4.1.2

4.3.3. Threat: Obtain client credentials over non secure transport

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 the possibility for phishing or immitation of a client service.

Countermeasures:

- Implement transport security through Confidentiality of Requests
- Alternative authentication means, which do not require to send plaintext credentials over the wire (Examples: Digest authentication)

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:

- Ensure proper handling of credentials as per Credential storage protection.

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:

- High entropy of secrets - Section 5.1.4.2.2
- Lock accounts - Section 5.1.4.2.3

4.3.6. DoS on dynamic client secret creation

If a Service Provider includes a nontrivial amount of entropy in client secrets and if the service provider automatically grants them, an attacker could exhaust the pool by repeatedly applying for them.

Countermeasures:

- The service provider should consider some verification step for new clients. The service provider should include a nontrivial amount of entropy in client secrets.

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 tokens endpoint, respectively.

4.4.1. Authorization Code

4.4.1.1. Threat: Malicious client obtains authorization

Attacker abuses valid client id

countermeasures

- client validation
- client authentication
- user consent

4.4.1.2. Threat: Eavesdropping authorization codes

The OAuth specification does not describe any mechanism for protecting authorization codes from eavesdroppers when they are transmitted from the Service Provider to the Client and where the Service Provider Grants an Access Token.

Note: A description of a similar attack on the SAML protocol can be found at http://www.oasis-open.org/committees/download.php/3405/oasis-sstc-saml-bindings-1.1.pdf (S.4.1.1.9.1).

Countermeasures:

- The authorization server SHOULD enforce a one time usage restriction (see Section 5.1.5.4).
- Authorization server as well as the client MUST ensure that these transmissions are protected using transport-layer mechanisms such as TLS or SSL (see Section 5.1.1).
- The authorization server shall 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).
- Limited duration of authorization codes - Section 5.1.5.3
- If an Authorization Server observes multiple attempts to redeem a authorization code, the Authorization Server may want to revoke all tokens granted based on the authorization code (see Section 5.2.1.1).
- In the absence of these countermeasures, reducing scope (Section 5.1.5.1) and expiry time (Section 5.1.5.3) for access

tokens can be used to reduce the damage in case of leaks.

4.4.1.3. 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:

- Credential storage protection can be employed - Section 5.1.4.1
- System security measures - Section 5.1.4.1.1
- Store access token hashes only - Section 5.1.4.1.3
- Standard SQL inj. Countermeasures - Section 5.1.4.1.2

4.4.1.4. 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 refresh token)

Countermeasures:

- For handle-based designs: Section 5.1.5.11
- For assertion-based designs: Section 5.1.5.9
- Binding of authorization code to client_id, adds another value the attacker has to guess - Section 5.2.4.4
- Binding of authorization code to redirect_uri, adds another value the attacker has to guess - Section 5.2.4.5
- Short expiration time - Section 5.1.5.3

4.4.1.5. Threat: Authorization code leaks when requesting access token

Authorization codes are passed via the browser which may unintentionally leak those codes to untrusted web sites and attackers by different ways:

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

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

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

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

Similar attacks on the SAML protocol are discussed in: http://www.thomasgross.net/publications/papers/GroPfi2006-SAML2_Analysis_Janus.WSSS_06.pdf and http://www.oasis-open.org/committees/download.php/11191/sstc-gross-sec-analysis-response-01.pdf.

Countermeasures:

- The authorization server shall 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).

- Authorization codes must be time-limited (see Section 5.1.5.3)

- Authorization codes should be single-use tokens (Section 5.1.5.4)

- If an Authorization Server observes multiple attempts to redeem a authorization code, the Authorization Server may want to revoke all tokens granted based on the authorization code (see Section 5.2.1.1)

- The resource server may reload the target page of the redirect_uri in order to automatically cleanup the browser cache.

4.4.1.6. Threat: Authorization code phishing

A hostile party could act as the client web server and get access to the authorization code. This could be achieved using DNS or ARP spoofing.

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

Countermeasures:

- The browser shall be utilized to authenticate the redirect_uri of the client using server authentication - Section 5.1.2
- The authorization server shall require the client to authenticate with a secret, so the binding of the authorization code to a certain client can be validated in a reliable way (see Section 5.2.4.4).

4.4.1.7. Threat: Session fixation

The session fixation attack leverages the 3-legged OAuth flow in an attempt to get another user to log-in and authorize access on behalf of the attacker. 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.

In this attack, the attacker is using a known client application (consumer site), and a target OAuth resource provider. The attack depends on the victim expecting the consumer site to request access to the resource provider.

The attacker utilizes the following flow:

The attacker initiates browser access to the consumer site, and initates access to data from the resource provider. The consumer site, initiates an authorization request and receives a redirect_uri back from the resource provider’s authorization server. Instead of following the link, the attacker stops the process and saves the redirect_uri. The attacker modifies the redirect_uri to allow control to be returned to the attacker site.

The attacker tricks another user (the victim) to open that redirect_uri and to authorize access (e.g. an email link, or blog link). The way the attacker achieve that goal is out of scope.

Having clicked, the link, the victim is requested to authenticate and authorize the consumer site to have access.

The authorization server redirects the user agent to the attackers web site instead of the original target web site.

The attacker obtains the authorization code from its web site, constructs a redirect_uri to the target web site (or app) based on the original authorization request's redirect_uri and the newly obtained authorization code and directs its user agent to this URL.

The web 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.

Countermeasures:

- The attacker must use another redirect_uri for its authorization process than the target web site because it needs to intercept the flow. So if the authorization server associates the authorization code with the redirect_uri of a particular end-user authorization, such a change (and with that such an attack) can be detected - see Section 5.2.4.4

- The authorization server may also enforce the usage and validation of pre-registered redirect Uris (see Section 5.2.3.5).

- For native apps, 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 a session fixation because the attacker does not have access the deployment-specific secret. Thus he will not be able to exchange the authorization code.

- The client may consider to use other flows, which are not vulnerable to session fixation attacks (see Section 4.4.2 or Section 4.4.3).

4.4.1.8. Threat: DoS, Exhaustion of resources attacks

If a Service Provider 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 by repeatedly directing user(s) browser to request code or access tokens. This is because more entropy means a larger number of tokens can be issued.

Countermeasures:

- The service provider should consider limiting the number of access tokens granted per user. The service provider should include a nontrivial amount of entropy in authorization codes.

4.4.2. Implicit Grant

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

Description: the access token is directly returned to the client as part of the redirect URL. This token might be eavesdropped by an attacker. The token is sent from server to client via a URI fragment of the redirect_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:

- Confidentiality of Requests - Section 5.1.1

- Bind token to client id - Section 5.1.5.8

4.4.2.2. Threat: Access token leak in browser history

An attacker could obtain the token from the browsers history.

Countermeasures:

- Shorten token duration (see Section 5.1.5.3) and reduced scope of the token may reduce the impact of that attack (see Section 5.1.5.1).

- Make these requests non-cachable

- Native apps can directly embedd a browser widget and therewith gain full control of the cache. So the app can cleanup browser history after authorization process.

4.4.2.3. Threat: Malicious client obtains authorization

An malicious client could attempt to obtain a token by fraud. Client secrets are not an effective countermeasure in this case.

The following countermeasures are advisable:

- Always require user consent and let end-user validate client identity - Section 5.2.4.3

- No automatic processing of repeated authorizations - Section 5.2.4.1

4.4.3. Resource Owner Password Credentials

The "password" grant type (see OAuth Core 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. The "password" 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.

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:

- Except for migration reasons, minimize use of this grant type

- The authorization server must validate the client id associated with the particular refresh token with every refresh request - Section 5.2.2.2

- Service Providers MUST ensure that these transmissions are protected using transport-layer mechanisms such as TLS or SSL (see Section 5.1.1).

4.4.3.1. Threat: Accidental exposure of passwords at client site

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

Countermeasures:

- Use other flows, which do not rely on the client’s cooperation for secure resource owner credential handling

- Use digest authentication instead of plaintext credential processing
- Obfuscation of passwords in logs

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

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

Countermeasures:

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

4.4.3.3. Threat: Client obtains refresh token through automatic authorization

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

Countermeasures:

- Use other flows, which do not rely on the client’s cooperation for resource owner interaction
- The authorization server may generally refuse to issue refresh tokens in this flow (see Section 5.2.2.1). If the particular client is trustworthy and can be authenticated in a reliable way

(cf. client authentication), the authorization server could relax that restriction. Resource owners may allow or deny (e.g. in their preferences) to issue refresh tokens using this flow as well.

- The authorization server could notify the resource owner by an appropriate media, e.g. e-Mail, of the refresh token issued (see Section 5.1.3).

4.4.3.4. Threat: Obtain user passwords on transport

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

Impact: disclosure of a single end-users password.

Countermeasures:

- Confidentiality of Requests - Section 5.1.1
- alternative authentication means, which do not require to send plaintext credentials over the wire (Examples: Digest authentication)

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 service providers domain since many users tend to use the same credentials on different services.

Countermeasures:

- Credential storage protection can be employed - 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:

- Password policy - Section 5.1.4.2.1
- Lock accounts - Section 5.1.4.2.3
- Tar pit - Section 5.1.4.2.4
- CAPTCHA - Section 5.1.4.2.5
- Abandon on grant type "password"
- Client authentication (see Section 5.2.3) will provide another authentication factor and thus hinder the attack.

4.4.4. Client Credentials

[TBD]

4.5. Refreshing an Access Token

4.5.1. Threat: Eavesdropping refresh tokens from authorization server

The OAuth specification does not describe any mechanism for protecting Tokens from eavesdroppers when they are transmitted from the Service Provider to the Client.

Countermeasures:

- Service Providers MUST ensure that these transmissions are protected using transport-layer mechanisms such as TLS or SSL (see Section 5.1.1).
- If end-to-end confidentiality cannot be guaranteed, reducing scope (see Section 5.1.5.1) and expiry time (see Section 5.1.5.3) for issued access tokens can be used to reduce the damage in case of leaks.

4.5.2. Threat: Obtaining refresh token from authorization server database

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

Impact: disclosure of all refresh tokens

Countermeasures:

- Credential storage protection - Section 5.1.4.1
- Bind token to client id, if the attacker cannot obtain the required id and secret - Section 5.1.5.8

4.5.3. Threat: Obtain refresh token by online guessing

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

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

Countermeasures:

- For handle-based designs - Section 5.1.5.11
- For assertion-based designs - Section 5.1.5.9
- Bind token to client id, because the attacker would guess the matching client id, too - Section 5.1.5.8

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 suceed.

Countermeasures:

- 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 MUST by treated with the same care as other credentials (e.g. end-user passwords).

Countermeasures:

- Access tokens sent as bearer tokens, SHOULD NOT be sent in the clear over an insecure channel. Instead transport protection means shall be utilized to prevent eavesdropping by an attacker (see Section 5.1.1).

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

- The access token can be bound to a client's identity and require the client to authenticate with the resource server (see Section 5.4.2). Client authentication MUST be performed without exposing the required secret to the transport channel.

4.6.2. Threat: Replay authorized resource server requests

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

Countermeasures:

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

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

4.6.3. Threat: Guessing access tokens

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

Impact: Access to a single user's data.

Countermeasures:

- Handle Tokens should have a reasonable entropy (see Section 5.1.5.11) in order to make guessing a valid token value difficult.

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

- Security can be further strengthened by using a short access token duration (see Section 5.1.5.2 and Section 5.1.5.3).

4.6.4. Threat: Access token phishing by counterfeit resource server

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

Countermeasures:

- Clients SHOULD not make authenticated requests with an access token to unfamiliar resource servers, regardless of the presence of a secure channel. If the resource server address is well-known to the client, it may authenticate the resource servers (see Section 5.1.2).

- Associate the endpoint address 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 address 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 address in the authorization process.

- Associate an access token with a client and authenticate the client with resource server requests (typically via signature in order to not disclose secret to a potential attacker). This prevents the attack because the counterfeit server is assumed to miss the capabilities to correctly authenticate on behalf of the legitimate client to the resource server (Section 5.4.2).

- Restrict the token scope (see Section 5.1.5.1) and or limit the token to a certain resource server (Section 5.1.5.5).

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

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

Countermeasures:

- Tokens should be restricted to particular resource servers (see Section 5.1.5.5).

4.6.6. Threat: Leak of confidential data in HTTP-Proxies

The HTTP Authorization scheme (OAuth HTTP Authorization Scheme) is optional. However, [RFC2616](Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1," .) 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:

- Service Providers not using the HTTP Authorization scheme (OAuth HTTP Authorization Scheme - see Section 5.4.1) should take care to use other mechanisms, such as the Cache-Control header, to ensure that authenticated content is protected.

- 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 HTTP referrers.

Countermeasures:

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

- Set logging configuration appropriately

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

- HTTP referrers can be prevented by reloading the target page again without URI parameters

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

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

5. Security Considerations

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

5.1. General

5.1.1. 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 attacks through using content of request, e.g. secrets or tokens, or mount replay attacks.

Attacks can be mitigated by using transport-layer mechanisms such as TLS or SSL. VPN may considered as well.

This is a countermeasure against the following threats:

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

5.1.2. 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 DNS name of the server to the public key presented by the server during connection establishment.

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

This is a countermeasure against the following threats:

- Spoofing
- Proxying
- Phishing by conterfeit 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 shall always be in control of the authorization processes and get the necessary information to meet informed decisions. Moreover, user involvement is a further security countermeasure. The user can probably recognize certain kinds of attacks better than the authorization server. Information can be presented/exchanged during the authorization process, after the authorization process, and every time the user wishes to get informed by using techniques such as:

- User consent forms
- Notification messages (e-Mail, SMS, ...)
- Activity/Event logs
- User self-care apps or portals

5.1.4. Credentials

This sections describes countermeasures used to protect all kind 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. Credential storage protection

5.1.4.1.1. 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. Standard SQL inj. Countermeasures

[TBD]

5.1.4.1.3. No cleartext storage of credentials

The authorization server may consider to not store credential in clear text. Typical approaches are to store hashes instead. If the credential lacks a reasonable entropy level (because it is a user password) an additional salt will harden the storage to prevent offline dictionary attacks. 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.

5.1.4.1.4. Encryption of credentials

[TBD]

5.1.4.1.5. Use of asymmetric cryptography

Usage of asymmetric cryptography will free the authorization server of the obligation to manage credentials. Nevertheless, it MUST ensure the integrity of the respective public keys.

5.1.4.2. Online attacks on secrets

5.1.4.2.1. Password policy

The authorization server may decide to enforce a complex user password policy in order to increase the user passwords' entropy. This will hinder online password attacks.

5.1.4.2.2. High entropy of secrets

When creating token handles or other secrets not intended for usage by human users, the authorization server MUST include a reasonable level of entropy in order to mitigate the risk of guessing attacks.

The token value MUST be constructed from a cryptographically strong random or pseudo-random number sequence [RFC1750] generated by the Authorization Server. The probability of any two Authorization Code values being identical MUST be less than or equal to 2^(-128) and SHOULD be less than or equal to 2^(-160).

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. 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 attackes 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. Usage of CAPTCHAs

The idea is to prevent programms from automatically checkinga 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. Basis of this decision is out of scope, examples are:

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

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

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

5.1.5.2. Expiration time

Tokens should generally expire after a reasonable duration. This complements and strengthens other security measures (such as signatures) and reduces the impact of all kinds of token leaks.

5.1.5.3. Short expiration time

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

- replay
- reduce impact of token leak
- reduce likelyhood of sucessful online guessing

Note: Short token duration requires preciser clock synchronisation between authorization server and resource server. Furthermore, shorter duration may require more token refreshments (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 request, which can be performed with a certain token. This mechanism can be used to mitigate the following threats:

- replay of tokens
- reduce likelyhood of sucessful online guessing

Additionally, If an Authorization Server observes multiple attempts to redeem a authorization code, the Authorization Server may want to revoke all tokens granted based on the authorization code.

5.1.5.5. Bind tokens to a particular resource server (Audience)

Authorization servers in multi-service environments may consider to issue tokens with different content to different resource servers and to explicitely indicate in the token the target server a token is intended to be sent to (cf. Audience in SAML Assertions). This countermeasure can be used in the following situations:

- It reduce the impact of a successful replay attempt, since the token is applicable to a single resource server, only.

- It prevents abuse of a token by a rough resource server or client, since the token can only be used on that server. It is rejected by other servers.

- It reduce the impact of a leakage of a valid token to a conterfeit resource server.

5.1.5.6. Use endpoint address as token audience

This may be used to indicate to a resource server, which endpoint address 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 address of that server.

5.1.5.7. Audience and Token scopes

Deployments may consider to use only tokens with explicitely 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 identity. This identity match must be validated for every request with that token. This means can be used, to

- detect token leakage and

- prevent token abuse.

Note: Validating the client identity may require the target server to authenticate the client’s identity. 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 shall be signed in order to detect any attempt to modify or produce faked tokens.

5.1.5.10. Encryption of token content

Self-contained may be encrypted for privacy reasons or to protect system internal data.

5.1.5.11. Random token value with high entropy

When creating token handles, the authorization server MUST include a reasonable level of entropy in order to mitigate the risk of guessing attacks. The token value MUST be constructed from a cryptographically strong random or pseudo-random number sequence [RFC1750] generated by the Authorization Server. The probability of any two Authorization Code values being identical MUST be less than or equal to 2^(-128) and SHOULD be less than or equal to 2^(-160).

5.1.6. Access tokens

- keep them in transient memory (accessible by the client app only)
- exposure to 3rd parties (malicious app)
- limit number of access tokens granted to a user

5.2. Authorization Server

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 a authorization code, the Authorization Server may want to revoke all tokens granted based on the authorization code.

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 areo 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 MUST bind every refresh token to the id of the client such a token was originally issued to and validate this binding for every request to refresh that token. This measure is a countermeasure against refresh token theft or leakage.

Note: This binding MUST be protected from unauthorized modifications.

5.2.2.3. Refresh Token Replacement

Refresh token replacement 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. Refresh Token Revocation

The authorization server may allow clients or end-users to explicitely request the invalidation of refresh tokens.

This is a countermeasure againts:

- device theft
- ...

5.2.2.5. Combine refresh token requests with user-provided secret

The exchange of a refresh token can be bound to the presence of a certain user-provided secret, such as a PIN, a password or a SIM card. This is a kind of multi-factor authentication on the tokens endpoint, since an attacker must possess both factors in order to be able to obtain an access token.

5.2.2.6. Device identification

The authorization server may require to bind authentication credentials to a device identifier or token assigned to that device. As the IMEI can be spoofed, that is not suitable, For mobile phones, a registration process can be used to assign a unique token to the device using an sms message. That token or identifer can then be

validated with when authenticating user credentials.

This is a countermeasure against the following threats:

- phishing 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

- Collate sub-sequent requests to the same client,
- Indicate the trustworthiness of a particular client to the end-user,
- Authorize access of clients to certain features on the authorization or resource server, and
- Log a client identity to log files for analysis or statics.

Due to the different capababilities and characterictics of the different client types, there are different ways to support achieve objectives, which will be described in this section. Generally spoken, 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 extrem, a service provider could activate every client installation by hand of 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 clients with inappropriate security policy

Authorization servers should not issue secrets to clients, if these cannot sufficiently protect it. This prevents the server from overestimating the value of a sucessful authentication of the client.

For example, it is of limited benefit to create a single client id and secret which is shared by all installations of a native app. First of all, this secret must be somehow transmitted from the developer via the respective distribution channel, e.g. an app market, to all installations of the app on end-user devices. So the

secret is typically burned into the source code of the app or a associated resource bundle, which cannot be entirely protected from reverse engineering. Second, effectively such secrets cannot be revoked since this would immediatly put all installations out of work. Moreover, since the authorization server cannot really trust on the client's identity, 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. Client_id only in combination with forced user consent

The authorization may issue a client id, but only accept authorization request, which are approved by the end-user. This measure precludes automatic authorization processes. This is a countermeasure for clients without secret against the following threats:

- ...
- ...

5.2.3.3. Client_id only in combination with redirect_uri

The authorization may issue a client id, but bind this client_id to a certain pre-configured redirect_uri. So any authorization request with another redirect_uri is refused automatically. Alternatively, the authorization server may not accept any dynamic redirect_uri for such a client_id and instead always redirect to the well-known pre-configured redirect_uri. This is a countermeasure for clients of LOA 2 against the following threats:

- ...
- ...

5.2.3.4. Deployment-specific client secrets

A authorization server may issue separate client ids and corresponding secrets to the different deployments of a client.

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 redirect_uri, address, and whatever proofs useful. The

web site provider has to ensure the security of the client secret on the site. As a result, such client could reach LOA 7.

For native applications, things are more complicated because every installation of the app on any device is another deployment. Deployment specific secrets will require

1. Either to obtain a client_id and client_secret during download process from the app 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 LOA 7, whereas the second option does not allow to validate properties of the client thus can achieve at most LOA 6. But this would at least help to prevent several replay attacks. Moreover, deployment-specific client_id and secret allow to selectively revoke all refresh tokens of a specific deployment at once. This is a countermeasure against the following threats:

o ...

o ...

5.2.3.5. Validation of pre-registered redirect_uri

An authorization server may require clients to register their redirect_uri or a pattern (TBD: make definition more precise) thereof. The way this registration is performed is out of scope of this document. Every actual redirect_uri sent with the respective client_id to the end-user authorization endpoint must comply with that pattern. Otherwise the authorization server must assume the inbound GET request has been sent by an attacker and refuse it.

Note: the authorization server MUST NOT redirect the user agent back to the redirect_uri of the authorization request.

o Session fixation: allows to detect session fixation attempts already after first redirect to end-user authorization endpoint

o For clients of LOA 2/5/7, this measure also helps to detect malicious apps early in the end-user authorization process. This reduces the need for a interactive validation by the user.

The underlying assumption of this measure is that an attacker must

use another redirect_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. This is a countermeasure against the following threats:

- session fixation
- malicious apps (for deployment-specific clients with secret)

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, it only works for clients bound to certain deployments at development/ configuration time. As soon as dynamic resource server discovery gets involved, that's no longer feasable.

5.2.3.6. Client secret revocation

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 for client identifiers and secrets used across multiple deployments of a particular native or web application.

This a countermeasure against:

- ...
- ...

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

Assumption: prevents an attacker from obtaining a client secret because this secret is kept in some hardware security module?

5.2.4. End-user authorization

5.2.4.1. Automatic processing of repeated authorizations requires client validation

Service providers should not automatically process repeat authorizations where the client is not authenticated through a client secret or some other authentication mechanism such as signing with security certs (5.7.2.7. Use strong client authentication (e.g.

client_assertion / client_token)) or validation of a pre-registered redirect uri (5.7.2.5. Validation of pre-registered redirect_uri).

5.2.4.2. Informed decisions based on transparency

The authorization server shall intelligible explain to the end-user what happens in the authorization process and what the consequences are. For example, the user shall understand what access he is about to grant to which client for what duration. It shall also be obvious to the user, whether the server is able to reliably certify certain client properties (web site address, 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-users can be involed 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 app the end-user is using. This measure is especially helpful in all situation where the authorization server is unable to authenticate the client. It is a countermeasure against:

- Malicious app
- ...

5.2.4.4. Binding of authorization code to client_id

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

- Session fixation since an attacker cannot use another client_id to exchange an authorization code into a token
- Online guessing of authorization codes

Note: This binding MUST be protected from unauthorized modifications.

5.2.4.5. Binding of authorization code to redirect_uri

The authorization server MAY bind every authorization code to the redirect_uri used as redirect target of the client in the end-user authorization process. This binding MUST be validated when the client attempts to exchange the respective authorization code for an access token. This measure is a countermeasure against session fixation since an attacker cannot use another redirect_uri to

exchange an authorization code into a token.

5.3. Client App Security

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

[Anything more to say ? :-)]

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

5.3.3. Store secrets in a secure storage

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

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

Another option is to swap refresh token storage to a trusted backend server. This mean in turn requires a resilient authentication mechanisms between client and backend server. Note: Applications must ensure that confidential data are kept confidential even after readin from secure storage, which typically means to keep this data in the local memory of the app.

5.3.4. Utilize device lock to prevent unauthorized device access

5.3.5. Platform security measures

- Validation process
- software package signatures
- Remote removal
-

5.4. 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 identitiy and encourage resource servers to validate that binding. This will require the resource server to authenticate the originator of a request as the legitimate owner of a particular token. There are a couple of options to implement this countermeasure:

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

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

- Alternatively, the authorization server may issue a token-bound secret, which the client uses to sign the request. 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

This mechanisms is 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 must be uniquly identifiable and must not be processed twice by the resource server. This countermeasure helps to mitigate:

- o modifications of the message and
- o replay attempts

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 Francisco Corella for his feedback.

Appendix A. Document History

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

-1

- 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 redirect_uri of the client"

8. References

8.1. Normative References

[I-D.ietf-oauth-v2] Hammer-Lahav, E., Recordon, D., and D. Hardt, "The OAuth 2.0 Authorization Protocol", draft-ietf-oauth-v2-13 (work in progress), February 2011.

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

8.2. Informative References

[I-D.hammer-oauth-v2-mac-token] Hammer-Lahav, E., "HTTP Authentication: MAC

Authentication", draft-hammer-oauth-v2-mac-token-02 (work in progress), January 2011.

[I-D.ietf-oauth-v2-bearer]
Jones, M., Hardt, D., and D. Recordon, "The OAuth 2.0 Protocol: Bearer Tokens", draft-ietf-oauth-v2-bearer-03 (work in progress), February 2011.

[I-D.lodderstedt-oauth-revocation]
Lodderstedt, T. and S. Dronia, "Token Revocation", draft-lodderstedt-oauth-revocation-01 (work in progress), January 2011.

[portable-contacts]
Smarr, J., "Portable Contacts 1.0 Draft C", August 2008.

Authors' Addresses

Dr.-Ing. Torsten Lodderstedt (editor)
Deutsche Telekom AG

Email: torsten@lodderstedt.net

Mark McGloin
IBM

Email: mark.mcgloin@ie.ibm.com

Phil Hunt
Oracle Corporation

Email: phil.hunt@yahoo.com

This Internet-Draft, draft-zeltsan-oauth-use-cases-02.txt, has expired, and has been deleted from the Internet-Drafts directory. An Internet-Draft expires 185 days from the date that it is posted unless it is replaced by an updated version, or the Secretariat has been notified that the document is under official review by the IESG or has been passed to the RFC Editor for review and/or publication as an RFC. This Internet-Draft was not published as an RFC.

Internet-Drafts are not archival documents, and copies of Internet-Drafts that have been deleted from the directory are not available. The Secretariat does not have any information regarding the future plans of the authors or working group, if applicable, with respect to this deleted Internet-Draft. For more information, or to request a copy of the document, please contact the authors directly.

Draft Authors:
George Fletcher<gffletch@aol.com>
Torsten Lodderstedt<torsten@lodderstedt.net>
Zachary Zeltsan<Zachary.Zeltsan@alcatel-lucent.com>