

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: May 19, 2017

B. Campbell
J. Bradley
Ping Identity
H. Tschofenig
ARM
November 15, 2016

Resource Indicators for OAuth 2.0
draft-campbell-oauth-resource-indicators-02

Abstract

This straw-man specification defines an extension to The OAuth 2.0 Authorization Framework that enables the client and authorization server to more explicitly to communicate about the protected resource(s) to be accessed.

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 19, 2017.

Copyright Notice

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

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of

the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
1.1. Requirements Notation and Conventions	3
1.2. Terminology	3
2. Resource Parameter	3
3. IANA Considerations	5
3.1. OAuth Parameters Registration	5
3.1.1. Registry Contents	5
3.2. OAuth Extensions Error Registration	6
3.2.1. Registry Contents	6
4. Security Considerations	6
5. References	7
5.1. Normative References	7
5.2. Informative References	7
Appendix A. Acknowledgements	8
Appendix B. Document History	8
Authors' Addresses	8

1. Introduction

Several years of deployment and implementation experience with OAuth 2.0 [RFC6749] has uncovered a need, in some circumstances, for the client to explicitly signal to the authorization server where it intends to use the access token it is requesting.

Knowing which resource server will process the access token enables the authorization server to construct the token as necessary for that entity. Properly encrypting the token (or content within the token) to a particular resource server, for example, requires knowing which resource server will receive and decrypt the token. Furthermore, various resource servers oftentimes have different requirements with respect to the data contained in, or referenced by, the token and knowing the resource server where the client intends to use the token allows the the authorization server to mint the token accordingly.

Specific knowledge of the intended recipient(s) of the access token also helps facilitate improved security characteristics of the token itself. Bearer tokens, currently the only defined type of OAuth access token, allow any party in possession of a token to get access to the associated resources. To prevent misuse, two important security assumptions must hold: bearer tokens must be protected from disclosure in storage and in transit and the access token must only be valid for use at a specific resource server and for a specific

scope. When the authorization server is informed of the resource server that will process the access token, it can restrict the intended audience of that token such that it cannot be used at other resource servers. Section 5.2 of OAuth 2.0 Authorization Framework: Bearer Token Usage [RFC6750] prescribes including the token's intended recipients within the token to prevent token redirect.

Scope, from Section 3.3 of OAuth 2.0 [RFC6749], sometimes is overloaded to convey the location or identity of the resource server, however, doing so isn't always feasible or desirable. Scope is typically about what access is being requested rather than where that access will be redeemed (e.g. "email", "user:follow", "user_photos", and "channels:read" are a small sample of scope values in use).

A means for the client to signal to the authorization sever where it intends to use the access token it's requesting is important and useful. A number of implementations and deployments of OAuth 2.0 have already employed proprietary parameters toward that end. This specification aims to provide a standardized and interoperable alternative to the proprietary approaches going forward.

1.1. Requirements Notation and Conventions

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

1.2. Terminology

This specification uses the terms "access token", "refresh token", "authorization server", "resource server", "authorization endpoint", "authorization request", "authorization response", "token endpoint", "grant type", "access token request", "access token response", and "client" defined by The OAuth 2.0 Authorization Framework [RFC6749].

2. Resource Parameter

The client may indicate the resource server(s) for which it is requesting an access token by including the following parameter in the request.

resource

OPTIONAL. The value of the "resource" parameter indicates a resource server where the requested access token will be used. It MUST be an absolute URI, as specified by Section 4.3 of [RFC3986], and MUST NOT include a query or fragment component. If the authorization server fails to parse the provided value or does not

consider the resource server acceptable, it MUST reject the request and provide an error response with the error code "invalid_resource". Multiple "resource" parameters may be used to indicate that the issued token is intended to be used at multiple resource servers.

When an access token will be returned from the authorization endpoint, the "resource" parameter is used in the authorization request to the authorization endpoint as defined in Section 4.2.1 of OAuth 2.0 [RFC6749]. An example of an authorization request where the client tells the authorization server that it wants a token for use at "https://rs.example.com/" is shown in Figure 1 below.

```
GET /as/authorization.oauth2?response_type=token
&client_id=s6BhdRkqt3&state=laeb
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
&resource=https%3A%2F%2Frs.example.com%2F HTTP/1.1
Host: authorization-server.example.com
```

Figure 1: Protected Resource Request

When the access token is returned from the token endpoint, the request parameter is included in the token request to the token endpoint. Sections 4.1.1, 4.3.1, 4.4.2, 4.5 and 6 of OAuth 2.0 [RFC6749] define requests to the token endpoint with different grant types. An example of a token request, using a refresh token, where the client tells the authorization server that it wants a token for use at "https://rs.example.com/" is shown in Figure 2 below.

```
POST /as/token.oauth2 HTTP/1.1
Host: authorization-server.example.com
Authorization: Basic czZCaGRSa3F0MzpoY3FFelFsVW9lQUU5cHg0RlNyNHlJ
Content-Type: application/x-www-form-urlencoded

grant_type=refresh_token
&refresh_token=4LTC8lb0acc6Oy4esc1Nk9BWC0imAwH
&resource=https%3A%2F%2Frs.example.com%2F
```

Figure 2: Protected Resource Request

The "resource" parameter indicates the physical location of resource server, typically as an https URL, where the client intends to use the requested access token. This enables the authorization server to apply policy as appropriate for the resource, such as determining the type and content of the token to be issued, if and how the token is to be encrypted, and applying appropriate audience restrictions to the token.

The client SHOULD provide the most specific URI that it can for the set of resources or API it intends to access. In practice a client will know a base URI for the resource server application that it interacts with, which is appropriate to use as the value of the "resource" parameter. The client SHOULD use the base URI for the API unless specific knowledge of resource server dictates the client use a shorter path. For example, the value "https://rs.example.com/" would be used for a resource server that is the exclusive application on that host, however, if the resource server is one of many applications on that host, something like "https://rs.example.com/application/" would be used. Another example, for an API like SCIM [RFC7644] that has multiple endpoints such as "https://rs.example.com/scim/Users", "https://rs.example.com/scim/Groups", and "https://rs.example.com/scim/Schemas" The client should use "https://rs.example.com/scim/" as the resource so that the issued access token is valid for all the endpoints of the SCIM API.

The authorization server SHOULD audience restrict the access token to the resource server(s) indicated by the "resource" parameter. Audience restrictions can be communicated in JSON Web Tokens [RFC7519] with the "aud" claim and the top-level member of the same name provides the audience restriction information in a Token Introspection [RFC7662] response. The authorization server may use the exact "resource" value as the audience or it may map from that value to a more general URI or abstract identifier for the resource server.

The requested resource pertains to the access token that is the expected result of the request and not to the underlying access granted by the resource owner.

3. IANA Considerations

3.1. OAuth Parameters Registration

This specification registers the following value in the IANA "OAuth Parameters" registry [IANA.OAuth.Parameters] established by [RFC6749].

3.1.1. Registry Contents

- o Parameter name: resource
- o Parameter usage location: authorization request, token request
- o Change controller: IESG
- o Specification document(s): Section 2 of [[this specification]]

3.2. OAuth Extensions Error Registration

This specification registers the following error in the IANA "OAuth Extensions Error Registry" [IANA.OAuth.Parameters] established by [RFC6749].

3.2.1. Registry Contents

- o Error name: `invalid_resource`
- o Error usage location: implicit grant error response, token error response
- o Related protocol extension: resource parameter
- o Change controller: IESG
- o Specification document(s): Section 2 of [[this specification]]

4. Security Considerations

An access token that is audience restricted to a resource server, which obtains the token legitimately, cannot be used to access resources on behalf of the resource owner at other resource servers. The "resource" parameter enables a client to indicate the resource server where the requested access token will be used, which in turn enables the authorization server to apply the appropriate audience restrictions to the token.

Some Resource servers may host user content or be multi-tenant. In order to avoid attacks that might confuse a client into sending a AT to a user controlled resource it is important to use the a specific resource URI including path and not use just a host with no path. This will cause any AT issued for accessing the user controlled resource to have a invalid audience if replayed against the legitimate resource API.

Although multiple occurrences of the "resource" parameter may be included in a request, using only a single "resource" parameter is encouraged. A bearer token that has multiple intended recipients (audiences) can be used by any one of those recipients at any other. Thus, a high degree of trust between the involved parties is needed when using access tokens with multiple audiences. Furthermore an authorization server may be unwilling or unable to fulfill a token request with multiple resources.

[[TODO: I continue to question the value of allowing multiple resources vs the functional and security complexity that comes with doing so. Writing the preceding paragraph just underscores that concern. So just noting it here.]]

5. References

5.1. Normative References

- [IANA.OAuth.Parameters]
IANA, "OAuth Parameters",
<<http://www.iana.org/assignments/oauth-parameters>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997,
<<http://www.rfc-editor.org/info/rfc2119>>.
- [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005,
<<http://www.rfc-editor.org/info/rfc3986>>.
- [RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012,
<<http://www.rfc-editor.org/info/rfc6749>>.

5.2. Informative References

- [I-D.draft-tschofenig-oauth-audience]
Tschofenig, H., "OAuth 2.0: Audience Information", draft-tschofenig-oauth-audience (work in progress), February 2013.
- [RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012,
<<http://www.rfc-editor.org/info/rfc6750>>.
- [RFC7519] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015,
<<http://www.rfc-editor.org/info/rfc7519>>.
- [RFC7644] Hunt, P., Ed., Grizzle, K., Ansari, M., Wahlstroem, E., and C. Mortimore, "System for Cross-domain Identity Management: Protocol", RFC 7644, DOI 10.17487/RFC7644, September 2015, <<http://www.rfc-editor.org/info/rfc7644>>.
- [RFC7662] Richer, J., Ed., "OAuth 2.0 Token Introspection", RFC 7662, DOI 10.17487/RFC7662, October 2015,
<<http://www.rfc-editor.org/info/rfc7662>>.

Appendix A. Acknowledgements

The following individuals contributed to discussions relating to and giving rise to this draft specification:

George Fletcher, Hans Zandbelt, Justin Richer, Michael Jones, Nat Sakimura, Phil Hunt, Sergey Beryozkin, and Anthony "no go" Nadalin.

Appendix B. Document History

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

-01

- o Move Hannes Tschofenig, who wrote [I-D.draft-tschofenig-oauth-audience] in '13, from Acknowledgements to Authors.
- o Added IANA Considerations to register the "resource" parameter and "invalid_resource" error code.

-00

- o Initial draft to define a resource parameter for OAuth 2.0.

Authors' Addresses

Brian Campbell
Ping Identity

Email: brian.d.campbell@gmail.com

John Bradley
Ping Identity

Email: ve7jtb@ve7jtb.com

Hannes Tschofenig
ARM

Email: hannes.tschofenig@gmx.net

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: December 14, 2018

D. Hardt
Amazon
B. Campbell
Ping Identity
N. Sakimura
NRI
June 12, 2018

Distributed OAuth
draft-hardt-oauth-distributed-01

Abstract

The Distributed OAuth profile enables an OAuth client to discover what authorization server or servers may be used to obtain access tokens for a given resource, and what parameter values to provide in the access token request.

Status of This Memo

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

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

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

This Internet-Draft will expire on December 14, 2018.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of

the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

1. Introduction

In [RFC6749], there is a single resource server and authorization server. In more complex and distributed systems, a clients may access many different resource servers, which have different authorization servers managing access. For example, a client may be accessing two different resources that provides similar functionality, but each is in a different geopolitical region, which requires authorization from authorization servers located in each geopolitical region.

A priori knowledge by the client of the relationships between resource servers and authorizations servers is not practical as the number of resource servers and authorization servers scales up. The client needs to discover on-demand which authorization server to request authorization for a given resource, and what parameters to pass. Being able to discover how to access a protected resource also enables more flexible software development as changes to the scopes, realms and authorization servers can happen dynamically with no change to client code.

1.1. Notational Conventions

In this document, the key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" are to be interpreted as described in BCP 14, [RFC2119].

1.2. Terminology

Issuer: the party issuing the access token, also known as the authorization server.

All other terms are as defined in [RFC6749] and [RFC6750]

1.3. Protocol Overview

Figure 1 shows an abstract flow of distributed OAuth.

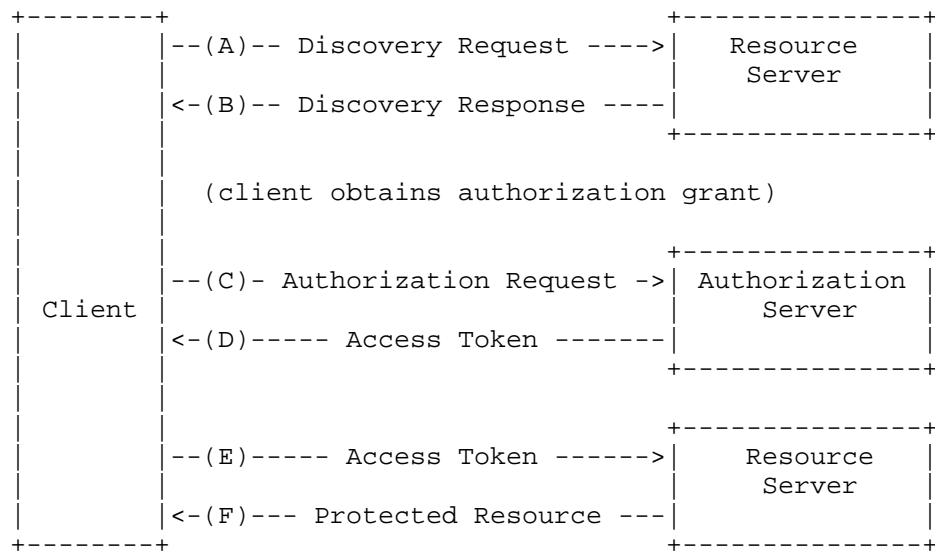


Figure 1: Abstract Protocol Flow

There are three steps where there are changes from the OAuth flow:

1) A discovery request (A) and discovery response (B) where the client discovers what is required to make an authenticated request. The client makes a request to the protected resource without supplying the Authorization header, or supplying an invalid access token. The resource server responds with a HTTP 401 response code and links of relation types "resource_uri" and the "oauth_server_metadata_uri". The client confirms the "host" value from the TLS connection is contained in the resource URI, and fetches each OAuth Server Metadata URI and per [OASM] discovers one or more authorization server end point URIs.

The client then obtains an authorization grant per one of the grant types in [RFC6749] section 4.

2) An authorization request (C) to an authorization server and includes the "resource_uri" link. The authorization servers provides an access token that is associated to the "resource_uri" value.

3) An authenticated request (E) to the resource server that confirms the "resource_uri" linked to the access token matches expected value.

2. Authorization Server Discovery

Figure 1, step (A)

To access a protected resource, the client needs to learn the authorization servers or issuers that can issue access tokens that are acceptable to the protected resource. There may be one or more issuers that can issue access tokens for the protected resource. To discover the issuers, the client attempts to make a call to the protected resource URI as defined in [RFC6750] section 2.1, except with an invalid access token or no HTTP "Authorization" request header field. The client notes the hostname of the protected resource that was confirmed by the TLS connection, and saves it as the "host" attribute.

Figure 1, step (B)

The resource server responds with the "WWW-Authenticate" HTTP header that includes the "error" attribute with a value of "invalid_token" and MAY also include the "scope" and "realm" attribute per [RFC6750] section 3, and a "Link" HTTP Header per [RFC8288] that MUST include one link of relation type "resource_uri" and one or more links of type "oauth_server_metadata_uri".

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

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="example_realm",
                  scope="example_scope",
                  error="invalid_token"
Link: <https://api.example.com/resource>; rel="resource_uri",
      <https://as.example.com/.well-known/oauth-authorization-server>; rel="
oauth_server_metadata_uri"
```

The client MUST confirm the host portion of the resource URI, as specified in the "resource_uri" link, contains the "host" attribute obtained from the TLS connection in step (A). The client MUST confirm the resource URI is contained in the protected resource URI where access was attempted. The client then retrieves one or more of the OAuth Server Metadata URIs to learn how to interact with the associated authorization server per [OASM] and create a list of one or more authorization server token endpoint URLs.

3. Authorization Grant

The client obtains an authorization grant per any of the mechanisms in [RFC6749] section 4.

4. Access Token Request

Figure 1, step (C)

The client makes an access token request to the authorization server token endpoint URL, or if more than URL is available, a randomly selected URL from the list. If the client is unable to connect to the URL, then the client MAY try to connect to another URL from the list.

The client SHOULD authenticate to the issuer using a proof of possession mechanism such as mutual TLS or a signed token containing the issuer as the audience.

Depending on the authorization grant mechanism used per [RFC6749] section 4, the client makes the access token request and MUST include "resource" as an additional parameter with the value of the resource URI. For example, if using the [RFC6749] section 4.4, Client Credentials Grant, the request would be (with extra spaces and line breaks for display purposes only):

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

grant_type=client_credentials
&scope=example_scope
&resource=https%3A%2F%2Fapi.example.com%2Fresource
```

Figure 1, step (D)

The authorization server MUST associate the resource URI with the issued access token in a way that can be accessed and verified by the protected resource. For JWT [RFC7519] formatted access tokens, the "aud" claim MUST be used to convey the resource URI. When Token Introspection [RFC7662] is used, the introspection response MUST contain the "aud" member with the resource URI as its value.

5. Accessing Protected Resource

Figure 1, step (E)

The client accesses the protected resource per [RFC6750] section 2.1. The Distributed OAuth Profile MUST only use the authorization request header field for passing the access token.

Figure 1, step (F)

The protected resource MUST verify the resource URI in or referenced by the access token is the protected resource's resource URI.

6. Security Considerations

Three new threats emerge when the client is dynamically discovering the authorization server and the request attributes: access token reuse, resource server impersonation, and malicious issuer.

6.1. Access Token Reuse

A malicious resource server impersonates the client and reuses the access token provided by the client to the malicious resource server with another resource server.

This is mitigated by constraining the access token to a specific audience, or to a specific client.

Audience restricting the access token is described in this document where the the resource URI is associated to the access token by inclusion or reference, so that only access tokens with the correct resource URI are accepted at a resource server.

Sender constraining the access token can be done through [MTLS], [OATB], or any other mechanism that the resource can use to associate the access token with the client.

6.2. Resource Server Impersonation

A malicious resource server tells a client to obtain an access token that can be used at a different resource server. When the client presents the access token, the malicious resource server uses the access token to access another resource server.

This is mitigated by the client obtaining the "host" value from the TLS certificate of the resource server, and the client verifying the "host" value is contained in the host portion of the resource URI, rather than the resource URI being any value declared by the resource server.

6.3. Malicious Issuer

A malicious resource server could redirect the client to a malicious issuer, or the issuer may be malicious. The malicious issuer may replay the client credentials with a valid issuer and obtain a valid access token for a protected resource.

This attack is mitigated by the client using a proof of possession authentication mechanism with the issuer such as [MTLS] or a signed token containing the issuer as the audience.

7. IANA Considerations

Pursuant to [RFC5988], the following link type registrations will be registered by mail to link-relations@ietf.org.

- o Relation Name: `oauth_server_metadata_uri`
- o Description: An OAuth 2.0 Server Metadata URI.
- o Reference: This specification
- o Relation Name: `resource_uri`
- o Description: An OAuth 2.0 Resource Endpoint specified in [RFC6750] section 3.2.
- o Reference: This specification

8. Acknowledgements

TBD.

9. Normative References

- [MTLS] Jones, M., Campbell, B., Bradley, J., and W. Denniss, "OAuth 2.0 Mutual TLS Client Authentication and Certificate Bound Access Tokens", June 2018, <https://datatracker.ietf.org/doc/draft-ietf-oauth-mtls/>.
- [OASM] Jones, M., Campbell, B., and J. Bradley, "OAuth 2.0 Authorization Server Metadata", June 2018, <https://datatracker.ietf.org/doc/draft-ietf-oauth-discovery/>.
- [OATB] Campbell, B., Bradley, J., Sakimora, N., and T. Lodderstedt, "OAuth 2.0 Token Binding", June 2018, <https://datatracker.ietf.org/doc/draft-ietf-oauth-token-binding/>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

- [RFC5988] Nottingham, M., "Web Linking", RFC 5988, DOI 10.17487/RFC5988, October 2010, <<https://www.rfc-editor.org/info/rfc5988>>.
- [RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <<https://www.rfc-editor.org/info/rfc6749>>.
- [RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, <<https://www.rfc-editor.org/info/rfc6750>>.
- [RFC7519] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <<https://www.rfc-editor.org/info/rfc7519>>.
- [RFC7662] Richer, J., Ed., "OAuth 2.0 Token Introspection", RFC 7662, DOI 10.17487/RFC7662, October 2015, <<https://www.rfc-editor.org/info/rfc7662>>.
- [RFC8288] Nottingham, M., "Web Linking", RFC 8288, DOI 10.17487/RFC8288, October 2017, <<https://www.rfc-editor.org/info/rfc8288>>.

Appendix A. Document History

A.1. draft-hardt-oauth-distributed-00

- o Initial version.

A.2. draft-hardt-oauth-distributed-01

- o resource identity expanded from just a hostname "host", to a URI that contains the hostname "resource URI"
- o use oauth discovery document to obtain token endpoint rather than explicitly returning token endpoint
- o use [RFC8288] to provide resource and discovery URIs
- o allow any authorization grant type be used to obtain an authorization grant
- o change attribute "host" to "resource"
- o require linking resource URI to access token

- o add client restriction to mitigate access token reuse
- o added Nat and Brian as authors

Authors' Addresses

Dick Hardt
Amazon

Email: dick.hardt@gmail.com

Brian Campbell
Ping Identity

Email: brian.d.campbell@gmail.com

Nat Sakimura
NRI

Email: n-sakimura@nri.co.jp

OAuth
Internet-Draft
Intended status: Standards Track
Expires: September 12, 2019

W. Denniss
Google
J. Bradley
Ping Identity
M. Jones
Microsoft
H. Tschofenig
ARM Limited
March 11, 2019

OAuth 2.0 Device Authorization Grant
draft-ietf-oauth-device-flow-15

Abstract

The OAuth 2.0 Device Authorization Grant is designed for internet-connected devices that either lack a browser to perform a user-agent based authorization, or are input-constrained to the extent that requiring the user to input text in order to authenticate during the authorization flow is impractical. It enables OAuth clients on such devices (like smart TVs, media consoles, digital picture frames, and printers) to obtain user authorization to access protected resources without using an on-device user-agent.

Status of This Memo

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

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

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

This Internet-Draft will expire on September 12, 2019.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction	3
2. Terminology	5
3. Protocol	5
3.1. Device Authorization Request	5
3.2. Device Authorization Response	7
3.3. User Interaction	8
3.3.1. Non-textual Verification URI Optimization	9
3.4. Device Access Token Request	10
3.5. Device Access Token Response	11
4. Discovery Metadata	12
5. Security Considerations	12
5.1. User Code Brute Forcing	13
5.2. Device Code Brute Forcing	13
5.3. Device Trustworthiness	14
5.4. Remote Phishing	14
5.5. Session Spying	15
5.6. Non-confidential Clients	15
5.7. Non-Visual Code Transmission	15
6. Usability Considerations	15
6.1. User Code Recommendations	16
6.2. Non-Browser User Interaction	17
7. IANA Considerations	17
7.1. OAuth Parameters Registration	17
7.1.1. Registry Contents	17
7.2. OAuth URI Registration	17
7.2.1. Registry Contents	17
7.3. OAuth Extensions Error Registration	17
7.3.1. Registry Contents	18
7.4. OAuth 2.0 Authorization Server Metadata	18
7.4.1. Registry Contents	18
8. Normative References	18
Appendix A. Acknowledgements	19
Appendix B. Document History	20
Authors' Addresses	22

1. Introduction

This OAuth 2.0 [RFC6749] protocol extension, sometimes referred to as "device flow", enables OAuth clients to request user authorization from applications on devices that have limited input capabilities or lack a suitable browser. Such devices include those smart TVs, media console, picture frames and printers which lack an easy input method or suitable browser required for traditional OAuth interactions. The authorization flow defined by this specification instructs the user to review the authorization request on a secondary device, such as a smartphone which does have the requisite input and browser capabilities to complete the user interaction.

The Device Authorization Grant is not intended to replace browser-based OAuth in native apps on capable devices like smartphones. Those apps should follow the practices specified in OAuth 2.0 for Native Apps [RFC8252].

The operating requirements to be able to use this authorization grant type are:

- (1) The device is already connected to the Internet.
- (2) The device is able to make outbound HTTPS requests.
- (3) The device is able to display or otherwise communicate a URI and code sequence to the user.
- (4) The user has a secondary device (e.g., personal computer or smartphone) from which they can process the request.

As the device authorization grant does not require two-way communication between the OAuth client and the user-agent (unlike other OAuth 2 grant types such as the Authorization Code and Implicit grant types), it supports several use cases that cannot be served by those other approaches.

Instead of interacting with the end user's user agent, the client instructs the end user to use another computer or device and connect to the authorization server to approve the access request. Since the protocol supports clients that can't receive incoming requests, clients poll the authorization server repeatedly until the end user completes the approval process.

The device typically chooses the set of authorization servers to support (i.e., its own authorization server, or those by providers it has relationships with). It is not uncommon for the device application to support only a single authorization server, such as

with a TV application for a specific media provider that supports only that media provider's authorization server. The user may not have an established relationship yet with that authorization provider, though one can potentially be set up during the authorization flow.

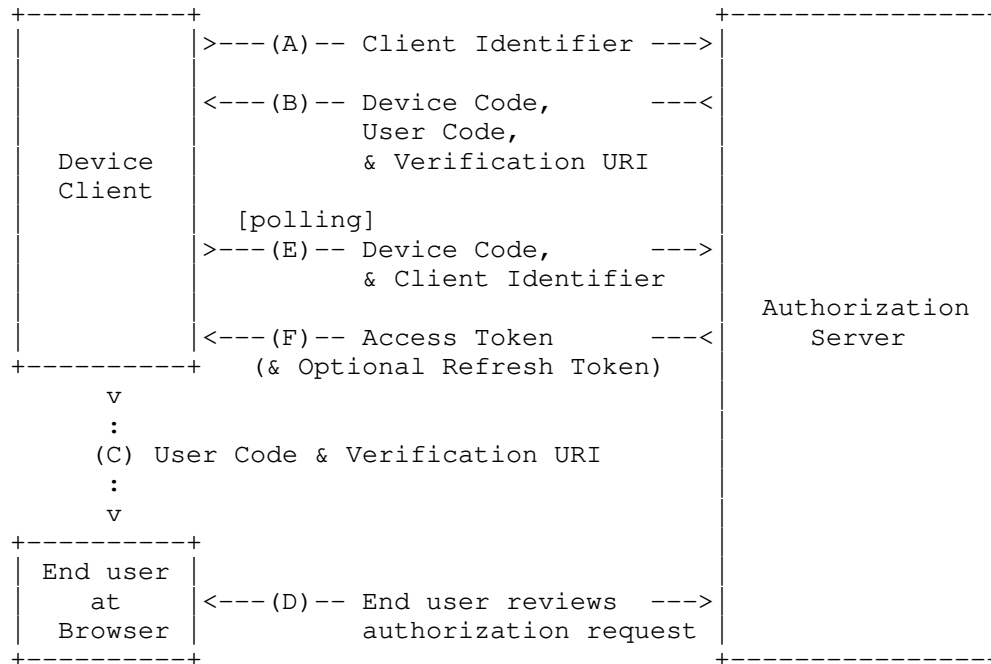


Figure 1: Device Authorization Flow

The device authorization flow illustrated in Figure 1 includes the following steps:

- (A) The client requests access from the authorization server and includes its client identifier in the request.
- (B) The authorization server issues a device code, an end-user code, and provides the end-user verification URI.
- (C) The client instructs the end user to use its user agent (on another device) and visit the provided end-user verification URI. The client provides the user with the end-user code to enter in order to review the authorization request.
- (D) The authorization server authenticates the end user (via the user agent) and prompts the user to grant the client's access

request. If the user agrees to the client's access request, the user enters the user code provided by the client. The authorization server validates the user code provided by the user.

(E) While the end user reviews the client's request (step D), the client repeatedly polls the authorization server to find out if the user completed the user authorization step. The client includes the verification code and its client identifier.

(F) The authorization server validates the verification code provided by the client and responds back with the access token if the user granted access, an error if they denied access, or indicates that the client should continue to poll.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

Device Authorization Endpoint:

The authorization server's endpoint capable of issuing device verification codes, user codes, and verification URLs.

Device Verification Code:

A short-lived token representing an authorization session.

End-User Verification Code:

A short-lived token which the device displays to the end user, is entered by the user on the authorization server, and is thus used to bind the device to the user.

3. Protocol

3.1. Device Authorization Request

This specification defines a new OAuth endpoint, the device authorization endpoint. This is separate from the OAuth authorization endpoint defined in [RFC6749] with which the user interacts with via a user-agent (i.e., a browser). By comparison, when using the device authorization endpoint, the OAuth client on the device interacts with the authorization server directly without presenting the request in a user-agent, and the end user authorizes the request on a separate device. This interaction is defined as follows.

The client initiates the authorization flow by requesting a set of verification codes from the authorization server by making an HTTP "POST" request to the device authorization endpoint.

The client constructs the request with the following parameters, sent as the body of the request, encoded with the "application/x-www-form-urlencoded" encoding algorithm defined by Section 4.10.22.6 of [HTML5]:

`client_id`

REQUIRED, if the client is not authenticating with the authorization server as described in Section 3.2.1. of [RFC6749]. The client identifier as described in Section 2.2 of [RFC6749].

`scope`

OPTIONAL. The scope of the access request as described by Section 3.3 of [RFC6749].

For example, the client makes the following HTTPS request:

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

client_id=459691054427
```

All requests from the device MUST use the Transport Layer Security (TLS) [RFC8446] protocol and implement the best practices of BCP 195 [RFC7525].

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.

The client authentication requirements of Section 3.2.1 of [RFC6749] apply to requests on this endpoint, which means that confidential clients (those that have established client credentials) authenticate in the same manner as when making requests to the token endpoint, and public clients provide the "client_id" parameter to identify themselves.

Due to the polling nature of this protocol (as specified in Section 3.4), care is needed to avoid overloading the capacity of the token endpoint. To avoid unneeded requests on the token endpoint, the client SHOULD only commence a device authorization request when prompted by the user, and not automatically, such as when the app starts or when the previous authorization session expires or fails.

3.2. Device Authorization Response

In response, the authorization server generates a unique device verification code and an end-user code that are valid for a limited time and includes them in the HTTP response body using the "application/json" format [RFC8259] with a 200 (OK) status code. The response contains the following parameters:

device_code

REQUIRED. The device verification code.

user_code

REQUIRED. The end-user verification code.

verification_uri

REQUIRED. The end-user verification URI on the authorization server. The URI should be short and easy to remember as end users will be asked to manually type it into their user-agent.

verification_uri_complete

OPTIONAL. A verification URI that includes the "user_code" (or other information with the same function as the "user_code"), designed for non-textual transmission.

expires_in

REQUIRED. The lifetime in seconds of the "device_code" and "user_code".

interval

OPTIONAL. The minimum amount of time in seconds that the client SHOULD wait between polling requests to the token endpoint. If no value is provided, clients MUST use 5 as the default.

For example:

HTTP/1.1 200 OK

Content-Type: application/json

Cache-Control: no-store

```
{
  "device_code": "GmRhmhcXhwAzkoEqiMEg_DnyEysNkuNhszIySk9eS",
  "user_code": "WDJB-MJHT",
  "verification_uri": "https://example.com/device",
  "verification_uri_complete":
    "https://example.com/device?user_code=WDJB-MJHT",
  "expires_in": 1800,
  "interval": 5
}
```


In the event of an error (such as an invalidly configured client), the authorization server responds in the same way as the token endpoint specified in Section 5.2 of [RFC6749].

3.3. User Interaction

After receiving a successful Authorization Response, the client displays or otherwise communicates the "user_code" and the "verification_uri" to the end user and instructs them to visit the URI in a user agent on a secondary device (for example, in a browser on their mobile phone), and enter the user code.

```
Using a browser on another device, visit:
https://example.com/device

And enter the code:
WDJB-MJHT
```

Figure 2: Example User Instruction

The authorizing user navigates to the "verification_uri" and authenticates with the authorization server in a secure TLS-protected ([RFC8446]) session. The authorization server prompts the end user to identify the device authorization session by entering the "user_code" provided by the client. The authorization server should then inform the user about the action they are undertaking and ask them to approve or deny the request. Once the user interaction is complete, the server MAY inform the user to return to their device.

During the user interaction, the device continuously polls the token endpoint with the "device_code", as detailed in Section 3.4, until the user completes the interaction, the code expires, or another error occurs. The "device_code" is not intended for the end user directly, and thus should not be displayed during the interaction to avoid confusing the end user.

Authorization servers supporting this specification MUST implement a user interaction sequence that starts with the user navigating to "verification_uri" and continues with them supplying the "user_code" at some stage during the interaction. Other than that, the exact sequence and implementation of the user interaction is up to the authorization server, for example, the authorization server may enable new users to sign up for an account during the authorization flow, or add additional security verification steps.

It is NOT RECOMMENDED for authorization servers to include the user code in the verification URI ("verification_uri"), as this increases the length and complexity of the URI that the user must type. While the user must still type the same number of characters with the "user_code" separated, once they successfully navigate to the "verification_uri", any errors in entering the code can be highlighted by the authorization server to improve the user experience. The next section documents user interaction with "verification_uri_complete", which is designed to carry both pieces of information.

3.3.1. Non-textual Verification URI Optimization

When "verification_uri_complete" is included in the Authorization Response (Section 3.2), clients MAY present this URI in a non-textual manner using any method that results in the browser being opened with the URI, such as with QR (Quick Response) codes or NFC (Near Field Communication), to save the user typing the URI.

For usability reasons, it is RECOMMENDED for clients to still display the textual verification URI ("verification_uri") for users not able to use such a shortcut. Clients MUST still display the "user_code", as the authorization server will require the user to confirm it to disambiguate devices, or as a remote phishing mitigation (See Section 5.4).

If the user starts the user interaction by browsing to "verification_uri_complete", then the user interaction described in Section 3.3 is still followed, but with the optimization that the user does not need to type the "user_code". The server SHOULD display the "user_code" to the user and ask them to verify that it matches the "user_code" being displayed on the device, to confirm they are authorizing the correct device. As before, in addition to taking steps to confirm the identity of the device, the user should also be afforded the choice to approve or deny the authorization request.

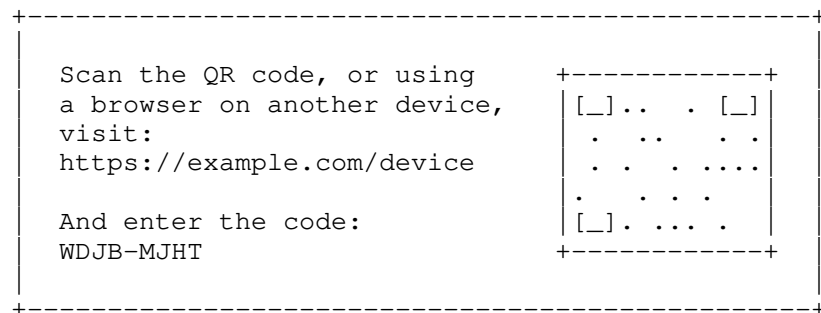


Figure 3: Example User Instruction with QR Code Representation of the Complete Verification URI

3.4. Device Access Token Request

After displaying instructions to the user, the client makes an Access Token Request to the token endpoint (as defined by Section 3.2 of [RFC6749]) with a "grant_type" of "urn:ietf:params:oauth:grant-type:device_code". This is an extension grant type (as defined by Section 4.5 of [RFC6749]) created by this specification, with the following parameters:

grant_type

REQUIRED. Value MUST be set to "urn:ietf:params:oauth:grant-type:device_code".

device_code

REQUIRED. The device verification code, "device_code" from the Device Authorization Response, defined in Section 3.2.

client_id

REQUIRED, if the client is not authenticating with the authorization server as described in Section 3.2.1. of [RFC6749]. The client identifier as described in Section 2.2 of [RFC6749].

For example, the client makes the following HTTPS request (line breaks are for display purposes only):

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

grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Adevice_code
&device_code=GmRhmcxhwAzkoEqiMEg_DnyEysNkuNhszIySk9eS
&client_id=459691054427
```

If 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 of [RFC6749]. Note that there are security implications of statically distributed client credentials, see Section 5.6.

The response to this request is defined in Section 3.5. Unlike other OAuth grant types, it is expected for the client to try the Access Token Request repeatedly in a polling fashion, based on the error code in the response.

3.5. Device Access Token Response

If the user has approved the grant, the token endpoint responds with a success response defined in Section 5.1 of [RFC6749]; otherwise it responds with an error, as defined in Section 5.2 of [RFC6749].

In addition to the error codes defined in Section 5.2 of [RFC6749], the following error codes are specified for use with the device authorization grant in token endpoint responses:

`authorization_pending`

The authorization request is still pending as the end user hasn't yet completed the user interaction steps (Section 3.3). The client SHOULD repeat the Access Token Request to the token endpoint (a process known as polling). Before each new request the client MUST wait at least the number of seconds specified by the "interval" parameter of the Device Authorization Response (see Section 3.2), or 5 seconds if none was provided, and respect any increase in the polling interval required by the "slow_down" error.

`slow_down`

A variant of "authorization_pending", the authorization request is still pending and polling should continue, but the interval MUST be increased by 5 seconds for this and all subsequent requests.

`access_denied`

The end user denied the authorization request.

expired_token

The "device_code" has expired and the device authorization session has concluded. The client MAY commence a new Device Authorization Request but SHOULD wait for user interaction before restarting to avoid unnecessary polling.

The "authorization_pending" and "slow_down" error codes define particularly unique behavior, as they indicate that the OAuth client should continue to poll the token endpoint by repeating the token request (implementing the precise behavior defined above). If the client receives an error response with any other error code, it MUST stop polling and SHOULD react accordingly, for example, by displaying an error to the user.

On encountering a connection timeout, clients MUST unilaterally reduce their polling frequency before retrying. The use of an exponential backoff algorithm to achieve this, such as by doubling the polling interval on each such connection timeout, is RECOMMENDED.

The assumption of this specification is that the separate device the user is authorizing the request on does not have a way to communicate back to device with the OAuth client. This protocol only requires a one-way channel in order to maximise the viability of the protocol in restricted environments, like an application running on a TV that is only capable of outbound requests. If a return channel were to exist for the chosen user interaction interface, then the device MAY wait until notified on that channel that the user has completed the action before initiating the token request (as an alternative to polling). Such behavior is, however, outside the scope of this specification.

4. Discovery Metadata

Support for this specification MAY be declared in the OAuth 2.0 Authorization Server Metadata [RFC8414] by including the value "urn:ietf:params:oauth:grant-type:device_code" in the "grant_types_supported" parameter, and by adding the following new parameter:

device_authorization_endpoint

OPTIONAL. URL of the authorization server's device authorization endpoint defined in Section 3.1.

5. Security Considerations

5.1. User Code Brute Forcing

Since the user code is typed by the user, shorter codes are more desirable for usability reasons. This means the entropy is typically less than would be used for the device code or other OAuth bearer token types where the code length does not impact usability. It is therefore recommended that the server rate-limit user code attempts.

The user code SHOULD have enough entropy that when combined with rate limiting and other mitigations makes a brute-force attack infeasible. For example, it's generally held that 128-bit symmetric keys for encryption are seen as good enough today because an attacker has to put in 2^{96} work to have a 2^{-32} chance of guessing correctly via brute force. The rate limiting and finite lifetime on the user code places an artificial limit on the amount of work an attacker can "do", so if, for instance, one uses a 8-character base-20 user code (with roughly 34.5 bits of entropy), the rate-limiting interval and validity period would need to only allow 5 attempts in order to get the same 2^{-32} probability of success by random guessing.

A successful brute forcing of the user code would enable the attacker to authenticate with their own credentials and make an authorization grant to the device. This is the opposite scenario to an OAuth bearer token being brute forced, whereby the attacker gains control of the victim's authorization grant. Such attacks may not always make economic sense, for example for a video app the device owner may then be able to purchase movies using the attacker's account, though a privacy risk would still remain and thus is important to protect against. Furthermore, some uses of the device flow give the granting account the ability to perform actions such as controlling the device, which needs to be protected.

The precise length of the user code and the entropy contained within is at the discretion of the authorization server, which needs to consider the sensitivity of their specific protected resources, the practicality of the code length from a usability standpoint, and any mitigations that are in place such as rate-limiting, when determining the user code format.

5.2. Device Code Brute Forcing

An attacker who guesses the device code would be able to potentially obtain the authorization code once the user completes the flow. As the device code is not displayed to the user and thus there are no usability considerations on the length, a very high entropy code SHOULD be used.

5.3. Device Trustworthiness

Unlike other native application OAuth 2.0 flows, the device requesting the authorization is not the same as the device that the user grants access from. Thus, signals from the approving user's session and device are not relevant to the trustworthiness of the client device.

Note that if an authorization server used with this flow is malicious, then it could man-in-the-middle the backchannel flow to another authorization server. In this scenario, the man-in-the-middle is not completely hidden from sight, as the end user would end up on the authorization page of the wrong service, giving them an opportunity to notice that the URL in the browser's address bar is wrong. For this to be possible, the device manufacturer must either directly be the attacker, shipping a device intended to perform the man-in-the-middle attack, or be using an authorization server that is controlled by an attacker, possibly because the attacker compromised the authorization server used by the device. In part, the person purchasing the device is counting on it and its business partners to be trustworthy.

5.4. Remote Phishing

It is possible for the device flow to be initiated on a device in an attacker's possession. For example, an attacker might send an email instructing the target user to visit the verification URL and enter the user code. To mitigate such an attack, it is RECOMMENDED to inform the user that they are authorizing a device during the user interaction step (see Section 3.3), and to confirm that the device is in their possession. The authorization server SHOULD display information about the device so that the person can notice if a software client was attempting to impersonating a hardware device.

For authorization servers that support the option specified in Section 3.3.1 for the client to append the user code to the authorization URI, it is particularly important to confirm that the device is in the user's possession, as the user no longer has to type the code manually. One possibility is to display the code during the authorization flow and asking the user to verify that the same code is being displayed on the device they are setting up.

The user code needs to have a long enough lifetime to be useable (allowing the user to retrieve their secondary device, navigate to the verification URI, login, etc.), but should be sufficiently short to limit the usability of a code obtained for phishing. This doesn't prevent a phisher presenting a fresh token, particularly in the case

they are interacting with the user in real time, but it does limit the viability of codes sent over email or SMS.

5.5. Session Spying

While the device is pending authorization, it may be possible for a malicious user to physically spy on the device user interface (by viewing the screen on which it's displayed, for example) and hijack the session by completing the authorization faster than the user that initiated it. Devices SHOULD take into account the operating environment when considering how to communicate the code to the user to reduce the chances it will be observed by a malicious user.

5.6. Non-confidential Clients

Device clients are generally incapable of maintaining the confidentiality of their credentials, as users in possession of the device can reverse engineer it and extract the credentials. Therefore, unless additional measures are taken, they should be treated as public clients (as defined by Section 2.1 of OAuth 2.0) susceptible to impersonation. The security considerations of Section 5.3.1 of [RFC6819] and Sections 8.5 and 8.6 of [RFC8252] apply to such clients.

The user may also be able to obtain the device_code and/or other OAuth bearer tokens issued to their client, which would allow them to use their own authorization grant directly by impersonating the client. Given that the user in possession of the client credentials can already impersonate the client and create a new authorization grant (with a new device_code), this doesn't represent a separate impersonation vector.

5.7. Non-Visual Code Transmission

There is no requirement that the user code be displayed by the device visually. Other methods of one-way communication can potentially be used, such as text-to-speech audio, or Bluetooth Low Energy. To mitigate an attack in which a malicious user can bootstrap their credentials on a device not in their control, it is RECOMMENDED that any chosen communication channel only be accessible by people in close proximity. E.g., users who can see, or hear the device.

6. Usability Considerations

This section is a non-normative discussion of usability considerations.

6.1. User Code Recommendations

For many users, their nearest Internet-connected device will be their mobile phone, and typically these devices offer input methods that are more time consuming than a computer keyboard to change the case or input numbers. To improve usability (improving entry speed, and reducing retries), these limitations should be taken into account when selecting the user-code character set.

One way to improve input speed is to restrict the character set to case-insensitive A-Z characters, with no digits. These characters can typically be entered on a mobile keyboard without using modifier keys. Further removing vowels to avoid randomly creating words results in the base-20 character set: "BCDFGHJKLMNPQRSTVWXZ". Dashes or other punctuation may be included for readability.

An example user code following this guideline containing 8 significant characters and dashes added for end-user readability, with a resulting entropy of 20^8 : "WDJB-MJHT".

Pure numeric codes are also a good choice for usability, especially for clients targeting locales where A-Z character keyboards are not used, though their length needs to be longer to maintain a high entropy.

An example numeric user code containing 9 significant digits and dashes added for end-user readability, with an entropy of 10^9 : "019-450-730".

When processing the inputted user code, the server should strip dashes and other punctuation it added for readability (making the inclusion of that punctuation by the user optional). For codes using only characters in the A-Z range as with the base-20 charset defined above, the user's input should be upper-cased before comparison to account for the fact that the user may input the equivalent lower-case characters. Further stripping of all characters outside the user_code charset is recommended to reduce instances where an errantly typed character (like a space character) invalidates otherwise valid input.

It is RECOMMENDED to avoid character sets that contain two or more characters that can easily be confused with each other like "0" and "O", or "1", "l" and "I". Furthermore, the extent practical, where a character set contains one character that may be confused with characters outside the character set the character outside the set MAY be substituted with the one in the character set that it is commonly confused with (for example, "O" for "0" when using a numerical 0-9 character set).

6.2. Non-Browser User Interaction

Devices and authorization servers MAY negotiate an alternative code transmission and user interaction method in addition to the one described in Section 3.3. Such an alternative user interaction flow could obviate the need for a browser and manual input of the code, for example, by using Bluetooth to transmit the code to the authorization server's companion app. Such interaction methods can utilize this protocol, as ultimately, the user just needs to identify the authorization session to the authorization server; however, user interaction other than via the verification URI is outside the scope of this specification.

7. IANA Considerations

7.1. OAuth Parameters Registration

This specification registers the following values in the IANA "OAuth Parameters" registry [IANA.OAuth.Parameters] established by [RFC6749].

7.1.1. Registry Contents

- o Parameter name: device_code
- o Parameter usage location: token request
- o Change controller: IESG
- o Specification Document: Section 3.1 of [[this specification]]

7.2. OAuth URI Registration

This specification registers the following values in the IANA "OAuth URI" registry [IANA.OAuth.Parameters] established by [RFC6755].

7.2.1. Registry Contents

- o URN: urn:ietf:params:oauth:grant-type:device_code
- o Common Name: Device flow grant type for OAuth 2.0
- o Change controller: IESG
- o Specification Document: Section 3.1 of [[this specification]]

7.3. OAuth Extensions Error Registration

This specification registers the following values in the IANA "OAuth Extensions Error Registry" registry [IANA.OAuth.Parameters] established by [RFC6749].

7.3.1. Registry Contents

- o Error name: `authorization_pending`
- o Error usage location: Token endpoint response
- o Related protocol extension: `[[this specification]]`
- o Change controller: IETF
- o Specification Document: Section 3.5 of `[[this specification]]`

- o Error name: `access_denied`
- o Error usage location: Token endpoint response
- o Related protocol extension: `[[this specification]]`
- o Change controller: IETF
- o Specification Document: Section 3.5 of `[[this specification]]`

- o Error name: `slow_down`
- o Error usage location: Token endpoint response
- o Related protocol extension: `[[this specification]]`
- o Change controller: IETF
- o Specification Document: Section 3.5 of `[[this specification]]`

- o Error name: `expired_token`
- o Error usage location: Token endpoint response
- o Related protocol extension: `[[this specification]]`
- o Change controller: IETF
- o Specification Document: Section 3.5 of `[[this specification]]`

7.4. OAuth 2.0 Authorization Server Metadata

This specification registers the following values in the IANA "OAuth 2.0 Authorization Server Metadata" registry [IANA.OAuth.Parameters] established by [RFC8414].

7.4.1. Registry Contents

- o Metadata name: `device_authorization_endpoint`
- o Metadata Description: The Device Authorization Endpoint.
- o Change controller: IESG
- o Specification Document: Section 4 of `[[this specification]]`

8. Normative References

- [HTML5] IANA, "HTML5",
<<https://www.w3.org/TR/2014/REC-html5-20141028/>>.
- [IANA.OAuth.Parameters] IANA, "OAuth Parameters",
<<http://www.iana.org/assignments/oauth-parameters>>.

- [RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <<https://www.rfc-editor.org/info/rfc6749>>.
- [RFC6755] Campbell, B. and H. Tschofenig, "An IETF URN Sub-Namespace for OAuth", RFC 6755, DOI 10.17487/RFC6755, October 2012, <<https://www.rfc-editor.org/info/rfc6755>>.
- [RFC6819] Lodderstedt, T., Ed., McGloin, M., and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations", RFC 6819, DOI 10.17487/RFC6819, January 2013, <<https://www.rfc-editor.org/info/rfc6819>>.
- [RFC7525] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 7525, DOI 10.17487/RFC7525, May 2015, <<https://www.rfc-editor.org/info/rfc7525>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC8252] Denniss, W. and J. Bradley, "OAuth 2.0 for Native Apps", BCP 212, RFC 8252, DOI 10.17487/RFC8252, October 2017, <<https://www.rfc-editor.org/info/rfc8252>>.
- [RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <<https://www.rfc-editor.org/info/rfc8259>>.
- [RFC8414] Jones, M., Sakimura, N., and J. Bradley, "OAuth 2.0 Authorization Server Metadata", RFC 8414, DOI 10.17487/RFC8414, June 2018, <<https://www.rfc-editor.org/info/rfc8414>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.

Appendix A. Acknowledgements

The starting point for this document was the Internet-Draft draft-recordon-oauth-v2-device, authored by David Recordon and Brent Goldman, which itself was based on content in draft versions of the OAuth 2.0 protocol specification removed prior to publication due to

a then lack of sufficient deployment expertise. Thank you to the OAuth working group members who contributed to those earlier drafts.

This document was produced in the OAuth working group under the chairpersonship of Rifaat Shekh-Yusef and Hannes Tschofenig with Benjamin Kaduk, Kathleen Moriarty, and Eric Rescorla serving as Security Area Directors.

The following individuals contributed ideas, feedback, and wording that shaped and formed the final specification:

Alissa Cooper, Ben Campbell, Brian Campbell, Roshni Chandrashekhar, Eric Fazendin, Benjamin Kaduk, Jamshid Khosravian, Torsten Lodderstedt, James Manger, Dan McNulty, Breno de Medeiros, Simon Moffatt, Stein Myrseth, Emond Papegaaïj, Justin Richer, Adam Roach, Nat Sakimura, Andrew Sciberras, Marius Scurtescu, Filip Skokan, Ken Wang, and Steven E. Wright.

Appendix B. Document History

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

-15

- o Renamed and dropped most usage of the term "flow"
- o Documented error responses on the authorization endpoint
- o Documented client authentication for the authorization endpoint

-14

- o Added more normative text on polling behavior.
- o Added discussion on risk of user retrieving their own device_code.
- o Editorial improvements.

-13

- o Added a longer discussion about entropy, proposed by Benjamin Kaduk.
- o Added device_code to OAuth IANA registry.
- o Expanded explanation of "case insensitive".
- o Added security section on Device Code Brute Forcing.
- o application/x-www-form-urlencoded normativly referenced.
- o Editorial improvements.

-12

- o Set a default polling interval to 5s explicitly.

- o Defined the `slow_down` behavior that it should increase the current interval by 5s.
- o `expires_in` now REQUIRED
- o Other changes in response to review feedback.

-11

- o Updated reference to OAuth 2.0 Authorization Server Metadata.

-10

- o Added a missing definition of `access_denied` for use on the token endpoint.
- o Corrected text documenting which error code should be returned for expired tokens (it's `"expired_token"`, not `"invalid_grant"`).
- o Corrected section reference to RFC 8252 (the section numbers had changed after the initial reference was made).
- o Fixed line length of one diagram (was causing `xml2rfc` warnings).
- o Added line breaks so the URN `grant_type` is presented on an unbroken line.
- o Typos fixed and other stylistic improvements.

-09

- o Addressed review comments by Security Area Director Eric Rescorla about the potential of a confused deputy attack.

-08

- o Expanded the User Code Brute Forcing section to include more detail on this attack.

-07

- o Replaced the `"user_code"` URI parameter optimization with `verification_uri_complete` following the IETF99 working group discussion.
- o Added security consideration about spying.
- o Required that `device_code` not be shown.
- o Added text regarding a minimum polling interval.

-06

- o Clarified usage of the `"user_code"` URI parameter optimization following the IETF98 working group discussion.

-05

- o response_type parameter removed from authorization request.
- o Added option for clients to include the user_code on the verification URI.
- o Clarified token expiry, and other nits.

-04

- o Security & Usability sections. OAuth Discovery Metadata.

-03

- o device_code is now a URN. Added IANA Considerations

-02

- o Added token request & response specification.

-01

- o Applied spelling and grammar corrections and added the Document History appendix.

-00

- o Initial working group draft based on draft-recordon-oauth-v2-device.

Authors' Addresses

William Denniss
Google
1600 Amphitheatre Pkwy
Mountain View, CA 94043
USA

Email: wdenniss@google.com
URI: <http://wdenniss.com/device-flow>

John Bradley
Ping Identity

Email: ve7jtb@ve7jtb.com
URI: <http://www.thread-safe.com/>

Michael B. Jones
Microsoft

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

Hannes Tschofenig
ARM Limited
Austria

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

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 5, 2018

M. Jones
Microsoft
N. Sakimura
NRI
J. Bradley
Ping Identity
March 4, 2018

OAuth 2.0 Authorization Server Metadata
draft-ietf-oauth-discovery-10

Abstract

This specification defines a metadata format that an OAuth 2.0 client can use to obtain the information needed to interact with an OAuth 2.0 authorization server, including its endpoint locations and authorization server capabilities.

Status of This Memo

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

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

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

This Internet-Draft will expire on September 5, 2018.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of

the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
1.1. Requirements Notation and Conventions	3
1.2. Terminology	3
2. Authorization Server Metadata	4
2.1. Signed Authorization Server Metadata	7
3. Obtaining Authorization Server Metadata	8
3.1. Authorization Server Metadata Request	9
3.2. Authorization Server Metadata Response	9
3.3. Authorization Server Metadata Validation	10
4. String Operations	11
5. Compatibility Notes	11
6. Security Considerations	12
6.1. TLS Requirements	12
6.2. Impersonation Attacks	12
6.3. Publishing Metadata in a Standard Format	13
6.4. Protected Resources	13
7. IANA Considerations	13
7.1. OAuth Authorization Server Metadata Registry	14
7.1.1. Registration Template	15
7.1.2. Initial Registry Contents	15
7.2. Updated Registration Instructions	18
7.3. Well-Known URI Registry	19
7.3.1. Registry Contents	19
8. References	19
8.1. Normative References	19
8.2. Informative References	21
Appendix A. Acknowledgements	22
Appendix B. Document History	22
Authors' Addresses	25

1. Introduction

This specification generalizes the metadata format defined by "OpenID Connect Discovery 1.0" [OpenID.Discovery] in a way that is compatible with OpenID Connect Discovery, while being applicable to a wider set of OAuth 2.0 use cases. This is intentionally parallel to the way that the "OAuth 2.0 Dynamic Client Registration Protocol" [RFC7591] specification generalized the dynamic client registration mechanisms defined by "OpenID Connect Dynamic Client Registration 1.0" [OpenID.Registration] in a way that was compatible with it.

The metadata for an authorization server is retrieved from a well-known location as a JSON [RFC7159] document, which declares its

endpoint locations and authorization server capabilities. This process is described in Section 3.

This metadata can either be communicated in a self-asserted fashion by the server origin via HTTPS or as a set of signed metadata values represented as claims in a JSON Web Token (JWT) [JWT]. In the JWT case, the issuer is vouching for the validity of the data about the authorization server. This is analogous to the role that the Software Statement plays in OAuth Dynamic Client Registration [RFC7591].

The means by which the client chooses an authorization server is out of scope. In some cases, its issuer identifier may be manually configured into the client. In other cases, it may be dynamically discovered, for instance, through the use of WebFinger [RFC7033], as described in Section 2 of "OpenID Connect Discovery 1.0" [OpenID.Discovery].

1.1. Requirements Notation and Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

All uses of JSON Web Signature (JWS) [JWS] and JSON Web Encryption (JWE) [JWE] data structures in this specification utilize the JWS Compact Serialization or the JWE Compact Serialization; the JWS JSON Serialization and the JWE JSON Serialization are not used.

1.2. Terminology

This specification uses the terms "Access Token", "Authorization Code", "Authorization Endpoint", "Authorization Grant", "Authorization Server", "Client", "Client Authentication", "Client Identifier", "Client Secret", "Grant Type", "Protected Resource", "Redirection URI", "Refresh Token", "Resource Owner", "Resource Server", "Response Type", and "Token Endpoint" defined by OAuth 2.0 [RFC6749], the terms "Claim Name", "Claim Value", and "JSON Web Token (JWT)" defined by JSON Web Token (JWT) [JWT], and the term "Response Mode" defined by OAuth 2.0 Multiple Response Type Encoding Practices [OAuth.Responses].

2. Authorization Server Metadata

Authorization servers can have metadata describing their configuration. The following authorization server metadata values are used by this specification and are registered in the IANA "OAuth Authorization Server Metadata" registry established in Section 7.1:

issuer

REQUIRED. The authorization server's issuer identifier, which is a URL that uses the "https" scheme and has no query or fragment components. Authorization server metadata is published at a ".well-known" RFC 5785 [RFC5785] location derived from this issuer identifier, as described in Section 3. The issuer identifier is used to prevent authorization server mix-up attacks, as described in "OAuth 2.0 Mix-Up Mitigation" [I-D.ietf-oauth-mix-up-mitigation].

authorization_endpoint

URL of the authorization server's authorization endpoint [RFC6749]. This is REQUIRED unless no grant types are supported that use the authorization endpoint.

token_endpoint

URL of the authorization server's token endpoint [RFC6749]. This is REQUIRED unless only the implicit grant type is supported.

jwks_uri

OPTIONAL. URL of the authorization server's JWK Set [JWK] document. The referenced document contains the signing key(s) the client uses to validate signatures from the authorization server. This URL MUST use the "https" scheme. The JWK Set MAY also contain the server's encryption key(s), which are used by clients to encrypt requests to the server. When both signing and encryption keys are made available, a "use" (public key use) parameter value is REQUIRED for all keys in the referenced JWK Set to indicate each key's intended usage.

registration_endpoint

OPTIONAL. URL of the authorization server's OAuth 2.0 Dynamic Client Registration endpoint [RFC7591].

scopes_supported

RECOMMENDED. JSON array containing a list of the OAuth 2.0 [RFC6749] "scope" values that this authorization server supports. Servers MAY choose not to advertise some supported scope values even when this parameter is used.

response_types_supported

REQUIRED. JSON array containing a list of the OAuth 2.0 "response_type" values that this authorization server supports. The array values used are the same as those used with the "response_types" parameter defined by "OAuth 2.0 Dynamic Client Registration Protocol" [RFC7591].

`response_modes_supported`

OPTIONAL. JSON array containing a list of the OAuth 2.0 "response_mode" values that this authorization server supports, as specified in OAuth 2.0 Multiple Response Type Encoding Practices [OAuth.Responses]. If omitted, the default is "["query", "fragment"]". The response mode value "form_post" is also defined in OAuth 2.0 Form Post Response Mode [OAuth.Post].

`grant_types_supported`

OPTIONAL. JSON array containing a list of the OAuth 2.0 grant type values that this authorization server supports. The array values used are the same as those used with the "grant_types" parameter defined by "OAuth 2.0 Dynamic Client Registration Protocol" [RFC7591]. If omitted, the default value is "["authorization_code", "implicit"]".

`token_endpoint_auth_methods_supported`

OPTIONAL. JSON array containing a list of client authentication methods supported by this token endpoint. Client authentication method values are used in the "token_endpoint_auth_method" parameter defined in Section 2 of [RFC7591]. If omitted, the default is "client_secret_basic" -- the HTTP Basic Authentication Scheme specified in Section 2.3.1 of OAuth 2.0 [RFC6749].

`token_endpoint_auth_signing_alg_values_supported`

OPTIONAL. JSON array containing a list of the JWS signing algorithms ("alg" values) supported by the token endpoint for the signature on the JWT [JWT] used to authenticate the client at the token endpoint for the "private_key_jwt" and "client_secret_jwt" authentication methods. This metadata entry MUST be present if either of these authentication methods are specified in the "token_endpoint_auth_methods_supported" entry. No default algorithms are implied if this entry is omitted. Servers SHOULD support "RS256". The value "none" MUST NOT be used.

`service_documentation`

OPTIONAL. URL of a page containing human-readable information that developers might want or need to know when using the authorization server. In particular, if the authorization server does not support Dynamic Client Registration, then information on how to register clients needs to be provided in this documentation.

ui_locales_supported

OPTIONAL. Languages and scripts supported for the user interface, represented as a JSON array of BCP47 [RFC5646] language tag values. If omitted, the set of supported languages and scripts is unspecified.

op_policy_uri

OPTIONAL. URL that the authorization server provides to the person registering the client to read about the authorization server's requirements on how the client can use the data provided by the authorization server. The registration process SHOULD display this URL to the person registering the client if it is given. As described in Section 5, despite the identifier "op_policy_uri", appearing to be OpenID-specific, its usage in this specification is actually referring to a general OAuth 2.0 feature that is not specific to OpenID Connect.

op_tos_uri

OPTIONAL. URL that the authorization server provides to the person registering the client to read about the authorization server's terms of service. The registration process SHOULD display this URL to the person registering the client if it is given. As described in Section 5, despite the identifier "op_tos_uri", appearing to be OpenID-specific, its usage in this specification is actually referring to a general OAuth 2.0 feature that is not specific to OpenID Connect.

revocation_endpoint

OPTIONAL. URL of the authorization server's OAuth 2.0 revocation endpoint [RFC7009].

revocation_endpoint_auth_methods_supported

OPTIONAL. JSON array containing a list of client authentication methods supported by this revocation endpoint. The valid client authentication method values are those registered in the IANA "OAuth Token Endpoint Authentication Methods" registry [IANA.OAuth.Parameters]. If omitted, the default is "client_secret_basic" -- the HTTP Basic Authentication Scheme specified in Section 2.3.1 of OAuth 2.0 [RFC6749].

revocation_endpoint_auth_signing_alg_values_supported

OPTIONAL. JSON array containing a list of the JWS signing algorithms ("alg" values) supported by the revocation endpoint for the signature on the JWT [JWT] used to authenticate the client at the revocation endpoint for the "private_key_jwt" and "client_secret_jwt" authentication methods. This metadata entry MUST be present if either of these authentication methods are specified in the "revocation_endpoint_auth_methods_supported"

entry. No default algorithms are implied if this entry is omitted. The value "none" MUST NOT be used.

`introspection_endpoint`

OPTIONAL. URL of the authorization server's OAuth 2.0 introspection endpoint [RFC7662].

`introspection_endpoint_auth_methods_supported`

OPTIONAL. JSON array containing a list of client authentication methods supported by this introspection endpoint. The valid client authentication method values are those registered in the IANA "OAuth Token Endpoint Authentication Methods" registry [IANA.OAuth.Parameters] or those registered in the IANA "OAuth Access Token Types" registry [IANA.OAuth.Parameters]. (These values are and will remain distinct, due to Section 7.2.) If omitted, the set of supported authentication methods MUST be determined by other means.

`introspection_endpoint_auth_signing_alg_values_supported`

OPTIONAL. JSON array containing a list of the JWS signing algorithms ("alg" values) supported by the introspection endpoint for the signature on the JWT [JWT] used to authenticate the client at the introspection endpoint for the "private_key_jwt" and "client_secret_jwt" authentication methods. This metadata entry MUST be present if either of these authentication methods are specified in the "introspection_endpoint_auth_methods_supported" entry. No default algorithms are implied if this entry is omitted. The value "none" MUST NOT be used.

`code_challenge_methods_supported`

OPTIONAL. JSON array containing a list of PKCE [RFC7636] code challenge methods supported by this authorization server. Code challenge method values are used in the "code_challenge_method" parameter defined in Section 4.3 of [RFC7636]. The valid code challenge method values are those registered in the IANA "PKCE Code Challenge Methods" registry [IANA.OAuth.Parameters]. If omitted, the authorization server does not support PKCE.

Additional authorization server metadata parameters MAY also be used. Some are defined by other specifications, such as OpenID Connect Discovery 1.0 [OpenID.Discovery].

2.1. Signed Authorization Server Metadata

In addition to JSON elements, metadata values MAY also be provided as a "signed_metadata" value, which is a JSON Web Token (JWT) [JWT] that asserts metadata values about the authorization server as a bundle. A set of claims that can be used in signed metadata are defined in

Section 2. The signed metadata MUST be digitally signed or MACed using JSON Web Signature (JWS) [JWS] and MUST contain an "iss" (issuer) claim denoting the party attesting to the claims in the signed metadata. Consumers of the metadata MAY ignore the signed metadata if they do not support this feature. If the consumer of the metadata supports signed metadata, metadata values conveyed in the signed metadata MUST take precedence over the corresponding values conveyed using plain JSON elements.

Signed metadata is included in the authorization server metadata JSON object using this OPTIONAL member:

`signed_metadata`

A JWT containing metadata values about the authorization server as claims. This is a string value consisting of the entire signed JWT. A "signed_metadata" metadata value SHOULD NOT appear as a claim in the JWT.

3. Obtaining Authorization Server Metadata

Authorization servers supporting metadata MUST make a JSON document containing metadata as specified in Section 2 available at a path formed by inserting a well-known URI string into the authorization server's issuer identifier between the host component and the path component, if any. By default, the well-known URI string used is `"/.well-known/oauth-authorization-server"`. This path MUST use the `"https"` scheme. The syntax and semantics of `".well-known"` are defined in RFC 5785 [RFC5785]. The well-known URI suffix used MUST be registered in the IANA "Well-Known URIs" registry [IANA.well-known].

Different applications utilizing OAuth authorization servers in application-specific ways may define and register different well-known URI suffixes used to publish authorization server metadata as used by those applications. For instance, if the Example application uses an OAuth authorization server in an Example-specific way, and there are Example-specific metadata values that it needs to publish, then it might register and use the `"example-configuration"` URI suffix and publish the metadata document at the path formed by inserting `"/.well-known/example-configuration"` between the host and path components of the authorization server's issuer identifier. Alternatively, many such applications will use the default well-known URI string `"/.well-known/oauth-authorization-server"`, which is the right choice for general-purpose OAuth authorization servers, and not register an application-specific one.

An OAuth 2.0 application using this specification MUST specify what well-known URI suffix it will use for this purpose. The same

authorization server MAY choose to publish its metadata at multiple well-known locations derived from its issuer identifier, for example, publishing metadata at both `"/.well-known/example-configuration"` and `"/.well-known/oauth-authorization-server"`.

Some OAuth applications will choose to use the well-known URI suffix `"openid-configuration"`. As described in Section 5, despite the identifier `"/.well-known/openid-configuration"`, appearing to be OpenID-specific, its usage in this specification is actually referring to a general OAuth 2.0 feature that is not specific to OpenID Connect.

3.1. Authorization Server Metadata Request

An authorization server metadata document MUST be queried using an HTTP `"GET"` request at the previously specified path.

The client would make the following request when the issuer identifier is `"https://example.com"` and the well-known URI suffix is `"oauth-authorization-server"` to obtain the metadata, since the issuer identifier contains no path component:

```
GET /.well-known/oauth-authorization-server HTTP/1.1
Host: example.com
```

If the issuer identifier value contains a path component, any terminating `"/` MUST be removed before inserting `"/.well-known/"` and the well-known URI suffix between the host component and the path component. The client would make the following request when the issuer identifier is `"https://example.com/issuer1"` and the well-known URI suffix is `"oauth-authorization-server"` to obtain the metadata, since the issuer identifier contains a path component:

```
GET /.well-known/oauth-authorization-server/issuer1 HTTP/1.1
Host: example.com
```

Using path components enables supporting multiple issuers per host. This is required in some multi-tenant hosting configurations. This use of `".well-known"` is for supporting multiple issuers per host; unlike its use in RFC 5785 [RFC5785], it does not provide general information about the host.

3.2. Authorization Server Metadata Response

The response is a set of claims about the authorization server's configuration, including all necessary endpoints and public key location information. A successful response MUST use the 200 OK HTTP status code and return a JSON object using the `"application/json"`

content type that contains a set of claims as its members that are a subset of the metadata values defined in Section 2. Other claims MAY also be returned.

Claims that return multiple values are represented as JSON arrays. Claims with zero elements MUST be omitted from the response.

An error response uses the applicable HTTP status code value.

The following is a non-normative example response:

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

{
  "issuer":
    "https://server.example.com",
  "authorization_endpoint":
    "https://server.example.com/authorize",
  "token_endpoint":
    "https://server.example.com/token",
  "token_endpoint_auth_methods_supported":
    ["client_secret_basic", "private_key_jwt"],
  "token_endpoint_auth_signing_alg_values_supported":
    ["RS256", "ES256"],
  "userinfo_endpoint":
    "https://server.example.com/userinfo",
  "jwks_uri":
    "https://server.example.com/jwks.json",
  "registration_endpoint":
    "https://server.example.com/register",
  "scopes_supported":
    ["openid", "profile", "email", "address",
     "phone", "offline_access"],
  "response_types_supported":
    ["code", "code token"],
  "service_documentation":
    "http://server.example.com/service_documentation.html",
  "ui_locales_supported":
    ["en-US", "en-GB", "en-CA", "fr-FR", "fr-CA"]
}
```

3.3. Authorization Server Metadata Validation

The "issuer" value returned MUST be identical to the authorization server's issuer identifier value into which the well-known URI string was inserted to create the URL used to retrieve the metadata. If

these values are not identical, the data contained in the response MUST NOT be used.

4. String Operations

Processing some OAuth 2.0 messages requires comparing values in the messages to known values. For example, the member names in the metadata response might be compared to specific member names such as "issuer". Comparing Unicode [UNICODE] strings, however, has significant security implications.

Therefore, 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.

Note that this is the same equality comparison procedure described in Section 8.3 of [RFC7159].

5. Compatibility Notes

The identifiers `"/.well-known/openid-configuration"`, `"op_policy_uri"`, and `"op_tos_uri"` contain strings referring to the OpenID Connect [OpenID.Core] family of specifications that were originally defined by "OpenID Connect Discovery 1.0" [OpenID.Discovery]. Despite the reuse of these identifiers that appear to be OpenID-specific, their usage in this specification is actually referring to general OAuth 2.0 features that are not specific to OpenID Connect.

The algorithm for transforming the issuer identifier to an authorization server metadata location defined in Section 3 is equivalent to the corresponding transformation defined in Section 4 of "OpenID Connect Discovery 1.0" [OpenID.Discovery], provided that the issuer identifier contains no path component. However, they are different when there is a path component, because OpenID Connect Discovery 1.0 specifies that the well-known URI string is appended to the issuer identifier (e.g., `"https://example.com/issuer1/.well-known/openid-configuration"`), whereas this specification specifies that the well-known URI string is inserted before the path component

of the issuer identifier (e.g., "https://example.com/.well-known/openid-configuration/issuer1").

Going forward, OAuth authorization server metadata locations should use the transformation defined in this specification. However, when deployed in legacy environments in which the OpenID Connect Discovery 1.0 transformation is already used, it may be necessary during a transition period to publish metadata for issuer identifiers containing a path component at both locations. During this transition period, applications should first apply the transformation defined in this specification and attempt to retrieve the authorization server metadata from the resulting location; only if the retrieval from that location fails should they fall back to attempting to retrieve it from the alternate location obtained using the transformation defined by OpenID Connect Discovery 1.0. This backwards-compatibility behavior should only be necessary when the well-known URI suffix employed by the application is "openid-configuration".

6. Security Considerations

6.1. TLS Requirements

Implementations **MUST** support 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. The authorization server **MUST** support TLS version 1.2 [RFC5246] and **MAY** support additional transport-layer security mechanisms meeting its security requirements. When using TLS, the client **MUST** perform a TLS/SSL server certificate check, per RFC 6125 [RFC6125]. Implementation security considerations can be found in Recommendations for Secure Use of TLS and DTLS [BCP195].

To protect against information disclosure and tampering, confidentiality protection **MUST** be applied using TLS with a ciphersuite that provides confidentiality and integrity protection.

6.2. Impersonation Attacks

TLS certificate checking **MUST** be performed by the client, as described in Section 6.1, when making an authorization server metadata request. Checking that the server certificate is valid for the issuer identifier URL prevents man-in-middle and DNS-based attacks. These attacks could cause a client to be tricked into using an attacker's keys and endpoints, which would enable impersonation of the legitimate authorization server. If an attacker can accomplish this, they can access the resources that the affected client has access to using the authorization server that they are impersonating.

An attacker may also attempt to impersonate an authorization server by publishing a metadata document that contains an "issuer" claim using the issuer identifier URL of the authorization server being impersonated, but with its own endpoints and signing keys. This would enable it to impersonate that authorization server, if accepted by the client. To prevent this, the client MUST ensure that the issuer identifier URL it is using as the prefix for the metadata request exactly matches the value of the "issuer" metadata value in the authorization server metadata document received by the client.

6.3. Publishing Metadata in a Standard Format

Publishing information about the authorization server in a standard format makes it easier for both legitimate clients and attackers to use the authorization server. Whether an authorization server publishes its metadata in an ad-hoc manner or in the standard format defined by this specification, the same defenses against attacks that might be mounted that use this information should be applied.

6.4. Protected Resources

Secure determination of appropriate protected resources to use with an authorization server for all use cases is out of scope of this specification. This specification assumes that the client has a means of determining appropriate protected resources to use with an authorization server and that the client is using the correct metadata for each authorization server. Implementers need to be aware that if an inappropriate protected resource is used by the client, that an attacker may be able to act as a man-in-the-middle proxy to a valid protected resource without it being detected by the authorization server or the client.

The ways to determine the appropriate protected resources to use with an authorization server are in general, application-dependent. For instance, some authorization servers are used with a fixed protected resource or set of protected resources, the locations of which may be well known, or which could be published as metadata values by the authorization server. In other cases, the set of resources that can be used with an authorization server can be dynamically changed by administrative actions. Many other means of determining appropriate associations between authorization servers and protected resources are also possible.

7. IANA Considerations

The following registration procedure is used for the registry established by this specification.

Values are registered on a Specification Required [RFC8126] basis after a two-week review period on the `oauth-ext-review@ietf.org` mailing list, on the advice of one or more Designated Experts. However, to allow for the allocation of values prior to publication, the Designated Experts may approve registration once they are satisfied that such a specification will be published.

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

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

Criteria that should be applied by the Designated Experts includes determining whether the proposed registration duplicates existing functionality, determining whether it is likely to be of general applicability or whether it is useful only for a single application, and whether the registration makes sense.

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

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

7.1. OAuth Authorization Server Metadata Registry

This specification establishes the IANA "OAuth Authorization Server Metadata" registry for OAuth 2.0 authorization server metadata names. The registry records the authorization server metadata member and a reference to the specification that defines it.

The Designated Experts must either:

(a) require that metadata names and values being registered use only printable ASCII characters excluding double quote (`'\"'`) and backslash

('\'') (the Unicode characters with code points U+0021, U+0023 through U+005B, and U+005D through U+007E), or

(b) if new metadata members or values are defined that use other code points, require that their definitions specify the exact Unicode code point sequences used to represent them. Furthermore, proposed registrations that use Unicode code points that can only be represented in JSON strings as escaped characters must not be accepted.

7.1.1. Registration Template

Metadata Name:

The name requested (e.g., "issuer"). This name is case-sensitive. Names may not match other registered names in a case-insensitive manner (one that would cause a match if the Unicode toLowerCase() operation were applied to both strings) unless the Designated Experts state that there is a compelling reason to allow an exception.

Metadata Description:

Brief description of the metadata (e.g., "Issuer identifier URL").

Change Controller:

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

Specification Document(s):

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

7.1.2. Initial Registry Contents

- o Metadata Name: "issuer"
- o Metadata Description: Authorization server's issuer identifier URL
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "authorization_endpoint"
- o Metadata Description: URL of the authorization server's authorization endpoint
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "token_endpoint"

- o Metadata Description: URL of the authorization server's token endpoint
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "jwks_uri"
- o Metadata Description: URL of the authorization server's JWK Set document
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "registration_endpoint"
- o Metadata Description: URL of the authorization server's OAuth 2.0 Dynamic Client Registration Endpoint
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "scopes_supported"
- o Metadata Description: JSON array containing a list of the OAuth 2.0 "scope" values that this authorization server supports
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "response_types_supported"
- o Metadata Description: JSON array containing a list of the OAuth 2.0 "response_type" values that this authorization server supports
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "response_modes_supported"
- o Metadata Description: JSON array containing a list of the OAuth 2.0 "response_mode" values that this authorization server supports
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "grant_types_supported"
- o Metadata Description: JSON array containing a list of the OAuth 2.0 grant type values that this authorization server supports
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "token_endpoint_auth_methods_supported"
- o Metadata Description: JSON array containing a list of client authentication methods supported by this token endpoint
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "token_endpoint_auth_signing_alg_values_supported"

- o Metadata Description: JSON array containing a list of the JWS signing algorithms supported by the token endpoint for the signature on the JWT used to authenticate the client at the token endpoint
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "service_documentation"
- o Metadata Description: URL of a page containing human-readable information that developers might want or need to know when using the authorization server
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "ui_locales_supported"
- o Metadata Description: Languages and scripts supported for the user interface, represented as a JSON array of BCP47 language tag values
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "op_policy_uri"
- o Metadata Description: URL that the authorization server provides to the person registering the client to read about the authorization server's requirements on how the client can use the data provided by the authorization server
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "op_tos_uri"
- o Metadata Description: URL that the authorization server provides to the person registering the client to read about the authorization server's terms of service
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "revocation_endpoint"
- o Metadata Description: URL of the authorization server's OAuth 2.0 revocation endpoint
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "revocation_endpoint_auth_methods_supported"
- o Metadata Description: JSON array containing a list of client authentication methods supported by this revocation endpoint
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name:
"revocation_endpoint_auth_signing_alg_values_supported"
- o Metadata Description: JSON array containing a list of the JWS signing algorithms supported by the revocation endpoint for the signature on the JWT used to authenticate the client at the revocation endpoint
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "introspection_endpoint"
- o Metadata Description: URL of the authorization server's OAuth 2.0 introspection endpoint
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "introspection_endpoint_auth_methods_supported"
- o Metadata Description: JSON array containing a list of client authentication methods supported by this introspection endpoint
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name:
"introspection_endpoint_auth_signing_alg_values_supported"
- o Metadata Description: JSON array containing a list of the JWS signing algorithms supported by the introspection endpoint for the signature on the JWT used to authenticate the client at the introspection endpoint
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

- o Metadata Name: "code_challenge_methods_supported"
- o Metadata Description: PKCE code challenge methods supported by this authorization server
- o Change Controller: IESG
- o Specification Document(s): Section 2 of [[this specification]]

7.2. Updated Registration Instructions

This specification adds to the instructions for the Designated Experts of the following IANA registries, both of which are in the "OAuth Parameters" registry [IANA.OAuth.Parameters]:

- o OAuth Access Token Types
- o OAuth Token Endpoint Authentication Methods

IANA has added a link to this specification in the Reference sections of these registries. [[RFC Editor: The above sentence is written in the past tense as it would appear in the final specification, even

though these links won't actually be created until after the IESG has requested publication of the specification. Please delete this note after the links are in place.]]

For these registries, the designated experts must reject registration requests in one registry for values already occurring in the other registry. This is necessary because the "introspection_endpoint_auth_methods_supported" parameter allows for the use of values from either registry. That way, because the values in the two registries will continue to be mutually exclusive, no ambiguities will arise.

7.3. Well-Known URI Registry

This specification registers the well-known URI defined in Section 3 in the IANA "Well-Known URIs" registry [IANA.well-known] established by RFC 5785 [RFC5785].

7.3.1. Registry Contents

- o URI suffix: "oauth-authorization-server"
- o Change controller: IESG
- o Specification document: Section 3 of [[this specification]]
- o Related information: (none)

8. References

8.1. Normative References

- [BCP195] Sheffer, Y., Holz, R., and P. Saint-Andre,
"Recommendations for Secure Use of Transport Layer
Security (TLS) and Datagram Transport Layer Security
(DTLS)", BCP 195, RFC 7525, DOI 10.17487/RFC7525, May
2015, <<http://www.rfc-editor.org/info/bcp195>>.
- [IANA.OAuth.Parameters]
IANA, "OAuth Parameters",
<<http://www.iana.org/assignments/oauth-parameters>>.
- [JWE] Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)",
RFC 7516, DOI 10.17487/RFC7516, May 2015,
<<http://tools.ietf.org/html/rfc7516>>.
- [JWK] Jones, M., "JSON Web Key (JWK)", RFC 7517,
DOI 10.17487/RFC7517, May 2015,
<<http://tools.ietf.org/html/rfc7517>>.

- [JWS] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <<http://tools.ietf.org/html/rfc7515>>.
- [JWT] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <<http://tools.ietf.org/html/rfc7519>>.
- [OAuth.Post] Jones, M. and B. Campbell, "OAuth 2.0 Form Post Response Mode", April 2015, <http://openid.net/specs/oauth-v2-form-post-response-mode-1_0.html>.
- [OAuth.Responses] de Medeiros, B., Ed., Scurtescu, M., Tarjan, P., and M. Jones, "OAuth 2.0 Multiple Response Type Encoding Practices", February 2014, <http://openid.net/specs/oauth-v2-multiple-response-types-1_0.html>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <<https://www.rfc-editor.org/info/rfc5246>>.
- [RFC5646] Phillips, A., Ed. and M. Davis, Ed., "Tags for Identifying Languages", BCP 47, RFC 5646, DOI 10.17487/RFC5646, September 2009, <<https://www.rfc-editor.org/info/rfc5646>>.
- [RFC5785] Nottingham, M. and E. Hammer-Lahav, "Defining Well-Known Uniform Resource Identifiers (URIs)", RFC 5785, DOI 10.17487/RFC5785, April 2010, <<https://www.rfc-editor.org/info/rfc5785>>.
- [RFC6125] Saint-Andre, P. and J. Hodges, "Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)", RFC 6125, DOI 10.17487/RFC6125, March 2011, <<https://www.rfc-editor.org/info/rfc6125>>.
- [RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <<https://www.rfc-editor.org/info/rfc6749>>.

- [RFC7009] Lodderstedt, T., Ed., Dronia, S., and M. Scurtescu, "OAuth 2.0 Token Revocation", RFC 7009, DOI 10.17487/RFC7009, August 2013, <<https://www.rfc-editor.org/info/rfc7009>>.
- [RFC7033] Jones, P., Salgueiro, G., Jones, M., and J. Smarr, "WebFinger", RFC 7033, DOI 10.17487/RFC7033, September 2013, <<https://www.rfc-editor.org/info/rfc7033>>.
- [RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, <<https://www.rfc-editor.org/info/rfc7159>>.
- [RFC7591] Richer, J., Ed., Jones, M., Bradley, J., Machulak, M., and P. Hunt, "OAuth 2.0 Dynamic Client Registration Protocol", RFC 7591, DOI 10.17487/RFC7591, July 2015, <<https://www.rfc-editor.org/info/rfc7591>>.
- [RFC7636] Sakimura, N., Ed., Bradley, J., and N. Agarwal, "Proof Key for Code Exchange by OAuth Public Clients", RFC 7636, DOI 10.17487/RFC7636, September 2015, <<https://www.rfc-editor.org/info/rfc7636>>.
- [RFC7662] Richer, J., Ed., "OAuth 2.0 Token Introspection", RFC 7662, DOI 10.17487/RFC7662, October 2015, <<https://www.rfc-editor.org/info/rfc7662>>.
- [RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <<https://www.rfc-editor.org/info/rfc8126>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [UNICODE] The Unicode Consortium, "The Unicode Standard", <<http://www.unicode.org/versions/latest/>>.
- [USA15] Davis, M. and K. Whistler, "Unicode Normalization Forms", Unicode Standard Annex 15, June 2015, <<http://www.unicode.org/reports/tr15/>>.

8.2. Informative References

- [I-D.ietf-oauth-mix-up-mitigation]
Jones, M., Bradley, J., and N. Sakimura, "OAuth 2.0 Mix-Up Mitigation", draft-ietf-oauth-mix-up-mitigation-01 (work in progress), July 2016.

[IANA.well-known]

IANA, "Well-Known URIs",
<<http://www.iana.org/assignments/well-known-uris>>.

[OpenID.Core]

Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and
C. Mortimore, "OpenID Connect Core 1.0", November 2014,
<http://openid.net/specs/openid-connect-core-1_0.html>.

[OpenID.Discovery]

Sakimura, N., Bradley, J., Jones, M., and E. Jay, "OpenID
Connect Discovery 1.0", November 2014,
<[http://openid.net/specs/
openid-connect-discovery-1_0.html](http://openid.net/specs/openid-connect-discovery-1_0.html)>.

[OpenID.Registration]

Sakimura, N., Bradley, J., and M. Jones, "OpenID Connect
Dynamic Client Registration 1.0", November 2014,
<[http://openid.net/specs/
openid-connect-registration-1_0.html](http://openid.net/specs/openid-connect-registration-1_0.html)>.

Appendix A. Acknowledgements

This specification is based on the OpenID Connect Discovery 1.0 specification, which was produced by the OpenID Connect working group of the OpenID Foundation. This specification standardizes the de facto usage of the metadata format defined by OpenID Connect Discovery to publish OAuth authorization server metadata.

The authors would like to thank the following people for their reviews of this specification: Shwetha Bhandari, Ben Campbell, Brian Campbell, Brian Carpenter, William Denniss, Vladimir Dzhuvinov, Donald Eastlake, Samuel Erdtman, George Fletcher, Dick Hardt, Phil Hunt, Alexey Melnikov, Tony Nadalin, Mark Nottingham, Eric Rescorla, Justin Richer, Adam Roach, Hannes Tschofenig, and Hans Zandbelt.

Appendix B. Document History

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

-10

- o Clarified the meaning of "case-insensitive", as suggested by Alexey Melnikov.

-09

- o Revised the transformation between the issuer identifier and the authorization server metadata location to conform to BCP 190, as suggested by Adam Roach.
- o Defined the characters allowed in registered metadata names and values, as suggested by Alexey Melnikov.
- o Changed to using the RFC 8174 boilerplate instead of the RFC 2119 boilerplate, as suggested by Ben Campbell.
- o Acknowledged additional reviewers.

-08

- o Changed the "authorization_endpoint" to be REQUIRED only when grant types are supported that use the authorization endpoint.
- o Added the statement, to provide historical context, that this specification standardizes the de facto usage of the metadata format defined by OpenID Connect Discovery to publish OAuth authorization server metadata.
- o Applied clarifications suggested by Mark Nottingham about when application-specific well-known suffixes are and are not appropriate.
- o Acknowledged additional reviewers.

-07

- o Applied clarifications suggested by EKR.

-06

- o Incorporated resolutions to working group last call comments.

-05

- o Removed the "protected_resources" element and the reference to draft-jones-oauth-resource-metadata.

-04

- o Added the ability to list protected resources with the "protected_resources" element.
- o Added ability to provide signed metadata with the "signed_metadata" element.

- o Removed "Discovery" from the name, since this is now just about authorization server metadata.

-03

- o Changed term "issuer URL" to "issuer identifier" for terminology consistency, paralleling the same terminology consistency change in the mix-up mitigation spec.

-02

- o Changed the title to OAuth 2.0 Authorization Server Discovery Metadata.
- o Made "jwks_uri" and "registration_endpoint" OPTIONAL.
- o Defined the well-known URI string "/.well-known/oauth-authorization-server".
- o Added security considerations about publishing authorization server discovery metadata in a standard format.
- o Added security considerations about protected resources.
- o Added more information to the "grant_types_supported" and "response_types_supported" definitions.
- o Referenced the working group Mix-Up Mitigation draft.
- o Changed some example metadata values.
- o Acknowledged individuals for their contributions to the specification.

-01

- o Removed WebFinger discovery.
- o Clarified the relationship between the issuer identifier URL and the well-known URI path relative to it at which the discovery metadata document is located.

-00

- o Created the initial working group version based on draft-jones-oauth-discovery-01, with no normative changes.

Authors' Addresses

Michael B. Jones
Microsoft

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

Nat Sakimura
Nomura Research Institute, Ltd.

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

John Bradley
Ping Identity

Email: ve7jtb@ve7jtb.com
URI: <http://www.thread-safe.com/>

OAuth Working Group
Internet-Draft
Updates: RFC 7519 (if approved)
Intended status: Best Current Practice
Expires: April 15, 2020

Y. Sheffer
Intuit
D. Hardt

M. Jones
Microsoft
October 13, 2019

JSON Web Token Best Current Practices
draft-ietf-oauth-jwt-bcp-07

Abstract

JSON Web Tokens, also known as JWTs, are URL-safe JSON-based security tokens that contain a set of claims that can be signed and/or encrypted. JWTs are being widely used and deployed as a simple security token format in numerous protocols and applications, both in the area of digital identity, and in other application areas. The goal of this Best Current Practices document is to provide actionable guidance leading to secure implementation and deployment of JWTs.

Status of This Memo

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

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

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

This Internet-Draft will expire on April 15, 2020.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

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

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

Table of Contents

1. Introduction	3
1.1. Target Audience	4
1.2. Conventions used in this document	4
2. Threats and Vulnerabilities	4
2.1. Weak Signatures and Insufficient Signature Validation . .	4
2.2. Weak Symmetric Keys	5
2.3. Incorrect Composition of Encryption and Signature	5
2.4. Plaintext Leakage through Analysis of Ciphertext Length .	5
2.5. Insecure Use of Elliptic Curve Encryption	5
2.6. Multiplicity of JSON Encodings	6
2.7. Substitution Attacks	6
2.8. Cross-JWT Confusion	6
2.9. Indirect Attacks on the Server	6
3. Best Practices	7
3.1. Perform Algorithm Verification	7
3.2. Use Appropriate Algorithms	7
3.3. Validate All Cryptographic Operations	8
3.4. Validate Cryptographic Inputs	8
3.5. Ensure Cryptographic Keys have Sufficient Entropy	8
3.6. Avoid Length-Dependent Encryption Inputs	9
3.7. Use UTF-8	9
3.8. Validate Issuer and Subject	9
3.9. Use and Validate Audience	9
3.10. Do Not Trust Received Claims	10
3.11. Use Explicit Typing	10
3.12. Use Mutually Exclusive Validation Rules for Different Kinds of JWTs	11
4. Security Considerations	11
5. IANA Considerations	12
6. Acknowledgements	12
7. References	12
7.1. Normative References	12
7.2. Informative References	13
Appendix A. Document History	15
A.1. draft-ietf-oauth-jwt-bcp-07	15
A.2. draft-ietf-oauth-jwt-bcp-06	15
A.3. draft-ietf-oauth-jwt-bcp-05	15
A.4. draft-ietf-oauth-jwt-bcp-04	15
A.5. draft-ietf-oauth-jwt-bcp-03	15
A.6. draft-ietf-oauth-jwt-bcp-02	15

A.7. draft-ietf-oauth-jwt-bcp-01	15
A.8. draft-ietf-oauth-jwt-bcp-00	15
A.9. draft-sheffer-oauth-jwt-bcp-01	15
A.10. draft-sheffer-oauth-jwt-bcp-00	15
Authors' Addresses	16

1. Introduction

JSON Web Tokens, also known as JWTs [RFC7519], are URL-safe JSON-based security tokens that contain a set of claims that can be signed and/or encrypted. The JWT specification has seen rapid adoption because it encapsulates security-relevant information in one easy-to-protect location, and because it is easy to implement using widely-available tools. One application area in which JWTs are commonly used is representing digital identity information, such as OpenID Connect ID Tokens [OpenID.Core] and OAuth 2.0 [RFC6749] access tokens and refresh tokens, the details of which are deployment-specific.

Since the JWT specification was published, there have been several widely published attacks on implementations and deployments. Such attacks are the result of under-specified security mechanisms, as well as incomplete implementations and incorrect usage by applications.

The goal of this document is to facilitate secure implementation and deployment of JWTs. Many of the recommendations in this document are about implementation and use of the cryptographic mechanisms underlying JWTs that are defined by JSON Web Signature (JWS) [RFC7515], JSON Web Encryption (JWE) [RFC7516], and JSON Web Algorithms (JWA) [RFC7518]. Others are about use of the JWT claims themselves.

These are intended to be minimum recommendations for the use of JWTs in the vast majority of implementation and deployment scenarios. Other specifications that reference this document can have stricter requirements related to one or more aspects of the format, based on their particular circumstances; when that is the case, implementers are advised to adhere to those stricter requirements. Furthermore, this document provides a floor, not a ceiling, so stronger options are always allowed (e.g., depending on differing evaluations of the importance of cryptographic strength vs. computational load).

Community knowledge about the strength of various algorithms and feasible attacks can change quickly, and experience shows that a Best Current Practice (BCP) document about security is a point-in-time statement. Readers are advised to seek out any errata or updates that apply to this document.

1.1. Target Audience

The intended audience of this document is:

- Implementers of JWT libraries (and the JWS and JWE libraries used by those libraries),
- Implementers of code that uses such libraries (to the extent that some mechanisms may not be provided by libraries, or until they are), and
- Developers of specifications that rely on JWTs, both inside and outside the IETF.

1.2. Conventions used in this document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. Threats and Vulnerabilities

This section lists some known and possible problems with JWT implementations and deployments. Each problem description is followed by references to one or more mitigations to those problems.

2.1. Weak Signatures and Insufficient Signature Validation

Signed JSON Web Tokens carry an explicit indication of the signing algorithm, in the form of the "alg" header parameter, to facilitate cryptographic agility. This, in conjunction with design flaws in some libraries and applications, have led to several attacks:

- The algorithm can be changed to "none" by an attacker, and some libraries would trust this value and "validate" the JWT without checking any signature.
- An "RS256" (RSA, 2048 bit) parameter value can be changed into "HS256" (HMAC, SHA-256), and some libraries would try to validate the signature using HMAC-SHA256 and using the RSA public key as the HMAC shared secret (see [McLean] and CVE-2015-9235).

For mitigations, see Section 3.1 and Section 3.2.

2.2. Weak Symmetric Keys

In addition, some applications use a keyed MAC algorithm such as "HS256" to sign tokens, but supply a weak symmetric key with insufficient entropy (such as a human memorable password). Such keys are vulnerable to offline brute-force or dictionary attacks once an attacker gets hold of such a token [Langkemper].

For mitigations, see Section 3.5.

2.3. Incorrect Composition of Encryption and Signature

Some libraries that decrypt a JWE-encrypted JWT to obtain a JWS-signed object do not always validate the internal signature.

For mitigations, see Section 3.3.

2.4. Plaintext Leakage through Analysis of Ciphertext Length

Many encryption algorithms leak information about the length of the plaintext, with a varying amount of leakage depending on the algorithm and mode of operation. This problem is exacerbated when the plaintext is initially compressed, because the length of the compressed plaintext and, thus, the ciphertext depend not only on the length of the original plaintext but also on its content. Compression attacks are particularly powerful when there is attacker-controlled data in the same compression space as secret data, as is the case for some attacks on HTTPS.

See [Kelsey] for general background on compression and encryption, and [Alawatugoda] for a specific example of attacks on HTTP cookies.

For mitigations, see Section 3.6.

2.5. Insecure Use of Elliptic Curve Encryption

Per [Sanso], several JOSE libraries fail to validate their inputs correctly when performing elliptic curve key agreement (the "ECDH-ES" algorithm). An attacker that is able to send JWEs of its choosing that use invalid curve points and observe the cleartext outputs resulting from decryption with the invalid curve points can use this vulnerability to recover the recipient's private key.

For mitigations, see Section 3.4.

2.6. Multiplicity of JSON Encodings

Previous versions of the JSON format such as the obsoleted [RFC7159] allowed several different character encodings: UTF-8, UTF-16 and UTF-32. This is not the case anymore, with the latest standard [RFC8259] only allowing UTF-8 except for internal use within a "closed ecosystem". This ambiguity where older implementations and those used within closed environments may generate non-standard encodings, may result in the JWT being misinterpreted by its recipient. This in turn could be used by a malicious sender to bypass the recipient's validation checks.

For mitigations, see Section 3.7.

2.7. Substitution Attacks

There are attacks in which one recipient will be given a JWT that was intended for it, and will attempt to use it at a different recipient for which that JWT was not intended. For instance, if an OAuth 2.0 [RFC6749] access token is legitimately presented to an OAuth 2.0 protected resource for which it is intended, that protected resource might then present that same access token to a different protected resource for which the access token is not intended, in an attempt to gain access. If such situations are not caught, this can result in the attacker gaining access to resources that it is not entitled to access.

For mitigations, see Section 3.8 and Section 3.9.

2.8. Cross-JWT Confusion

As JWTs are being used by more different protocols in diverse application areas, it becomes increasingly important to prevent cases of JWT tokens that have been issued for one purpose being subverted and used for another. Note that this is a specific type of substitution attack. If the JWT could be used in an application context in which it could be confused with other kinds of JWTs, then mitigations MUST be employed to prevent these substitution attacks.

For mitigations, see Section 3.8, Section 3.9, Section 3.11, and Section 3.12.

2.9. Indirect Attacks on the Server

Various JWT claims are used by the recipient to perform lookup operations, such as database and LDAP searches. Others include URLs that are similarly looked up by the server. Any of these claims can

be used by an attacker as vectors for injection attacks or server-side request forgery (SSRF) attacks.

For mitigations, see Section 3.10.

3. Best Practices

The best practices listed below should be applied by practitioners to mitigate the threats listed in the preceding section.

3.1. Perform Algorithm Verification

Libraries MUST enable the caller to specify a supported set of algorithms and MUST NOT use any other algorithms when performing cryptographic operations. The library MUST ensure that the "alg" or "enc" header specifies the same algorithm that is used for the cryptographic operation. Moreover, each key MUST be used with exactly one algorithm, and this MUST be checked when the cryptographic operation is performed.

3.2. Use Appropriate Algorithms

As Section 5.2 of [RFC7515] says, "it is an application decision which algorithms may be used in a given context. Even if a JWS can be successfully validated, unless the algorithm(s) used in the JWS are acceptable to the application, it SHOULD consider the JWS to be invalid."

Therefore, applications MUST only allow the use of cryptographically current algorithms that meet the security requirements of the application. This set will vary over time as new algorithms are introduced and existing algorithms are deprecated due to discovered cryptographic weaknesses. Applications MUST therefore be designed to enable cryptographic agility.

That said, if a JWT is cryptographically protected end-to-end by a transport layer, such as TLS using cryptographically current algorithms, there may be no need to apply another layer of cryptographic protections to the JWT. In such cases, the use of the "none" algorithm can be perfectly acceptable. The "none" algorithm should only be used when the JWT is cryptographically protected by other means. JWTs using "none" are often used in application contexts in which the content is optionally signed; then the URL-safe claims representation and processing can be the same in both the signed and unsigned cases. JWT libraries SHOULD NOT generate JWTs using "none" unless explicitly requested to do by the caller. Similarly, JWT libraries SHOULD NOT consume JWTs using "none" unless explicitly requested by the caller.

Applications SHOULD follow these algorithm-specific recommendations:

- Avoid all RSA-PKCS1 v1.5 encryption algorithms ([RFC8017], Sec. 7.2}, preferring RSA-OAEP ([RFC8017], Sec. 7.1).
- ECDSA signatures [ANSI-X962-2005] require a unique random value for every message that is signed. If even just a few bits of the random value are predictable across multiple messages then the security of the signature scheme may be compromised. In the worst case, the private key may be recoverable by an attacker. To counter these attacks, JWT libraries SHOULD implement ECDSA using the deterministic approach defined in [RFC6979]. This approach is completely compatible with existing ECDSA verifiers and so can be implemented without new algorithm identifiers being required.

3.3. Validate All Cryptographic Operations

All cryptographic operations used in the JWT MUST be validated and the entire JWT MUST be rejected if any of them fail to validate. This is true not only of JWTs with a single set of Header Parameters but also for Nested JWTs, in which both outer and inner operations MUST be validated using the keys and algorithms supplied by the application.

3.4. Validate Cryptographic Inputs

Some cryptographic operations, such as Elliptic Curve Diffie-Hellman key agreement ("ECDH-ES") take inputs that may contain invalid values, such as points not on the specified elliptic curve or other invalid points (see, e.g. [Valenta], Sec. 7.1). The JWS/JWE library itself must validate these inputs before using them or it must use underlying cryptographic libraries that do so (or both!).

ECDH-ES ephemeral public key (epk) inputs should be validated according to the recipient's chosen elliptic curve. For the NIST prime-order curves P-256, P-384 and P-521, validation MUST be performed according to Section 5.6.2.3.4 "ECC Partial Public-Key Validation Routine" of NIST Special Publication 800-56A revision 3 [nist-sp-800-56a-r3]. Likewise, if the "X25519" or "X448" [RFC8037] algorithms are used, then the security considerations in [RFC8037] apply.

3.5. Ensure Cryptographic Keys have Sufficient Entropy

The Key Entropy and Random Values advice in Section 10.1 of [RFC7515] and the Password Considerations in Section 8.8 of [RFC7518] MUST be followed. In particular, human-memorizable passwords MUST NOT be directly used as the key to a keyed-MAC algorithm such as "HS256".

In particular, passwords should only be used to perform key encryption, rather than content encryption, as described in Section 4.8 of [RFC7518]. Note that even when used for key encryption, password-based encryption is still subject to brute-force attacks.

3.6. Avoid Length-Dependent Encryption Inputs

Compression of data SHOULD NOT be done before encryption, because such compressed data often reveals information about the plaintext.

3.7. Use UTF-8

[RFC7515], [RFC7516], and [RFC7519] all specify that UTF-8 be used for encoding and decoding JSON used in Header Parameters and JWT Claims Sets. This is also in line with the latest JSON specification [RFC8259]. Implementations and applications MUST do this, and not use or admit the use of other Unicode encodings for these purposes.

3.8. Validate Issuer and Subject

When a JWT contains an "iss" (issuer) claim, the application MUST validate that the cryptographic keys used for the cryptographic operations in the JWT belong to the issuer. If they do not, the application MUST reject the JWT.

The means of determining the keys owned by an issuer is application-specific. As one example, OpenID Connect [OpenID.Core] issuer values are "https" URLs that reference a JSON metadata document that contains a "jwks_uri" value that is an "https" URL from which the issuer's keys are retrieved as a JWK Set [RFC7517]. This same mechanism is used by [RFC8414]. Other applications may use different means of binding keys to issuers.

Similarly, when the JWT contains a "sub" (subject) claim, the application MUST validate that the subject value corresponds to a valid subject and/or issuer/subject pair at the application. This may include confirming that the issuer is trusted by the application. If the issuer, subject, or the pair are invalid, the application MUST reject the JWT.

3.9. Use and Validate Audience

If the same issuer can issue JWTs that are intended for use by more than one relying party or application, the JWT MUST contain an "aud" (audience) claim that can be used to determine whether the JWT is being used by an intended party or was substituted by an attacker at an unintended party.

In such cases, the relying party or application MUST validate the audience value and if the audience value is not present or not associated with the recipient, it MUST reject the JWT.

3.10. Do Not Trust Received Claims

The "kid" (key ID) header is used by the relying application to perform key lookup. Applications should ensure that this does not create SQL or LDAP injection vulnerabilities, by validating and/or sanitizing the received value.

Similarly, blindly following a "jku" (JWK set URL) or "x5u" (X.509 URL) header, which may contain an arbitrary URL, could result in server-side request forgery (SSRF) attacks. Applications SHOULD protect against such attacks, e.g., by matching the URL to a whitelist of allowed locations, and ensuring no cookies are sent in the GET request.

3.11. Use Explicit Typing

Sometimes, one kind of JWT can be confused for another. If a particular kind of JWT is subject to such confusion, that JWT can include an explicit JWT type value, and the validation rules can specify checking the type. This mechanism can prevent such confusion. Explicit JWT typing is accomplished by using the "typ" header parameter. For instance, the [RFC8417] specification uses the "application/secevent+jwt" media type to perform explicit typing of Security Event Tokens (SETs).

Per the definition of "typ" in Section 4.1.9 of [RFC7515], it is RECOMMENDED that the "application/" prefix be omitted from the "typ" value. Therefore, for example, the "typ" value used to explicitly include a type for a SET SHOULD be "secevent+jwt". When explicit typing is employed for a JWT, it is RECOMMENDED that a media type name of the format "application/example+jwt" be used, where "example" is replaced by the identifier for the specific kind of JWT.

When applying explicit typing to a Nested JWT, the "typ" header parameter containing the explicit type value MUST be present in the inner JWT of the Nested JWT (the JWT whose payload is the JWT Claims Set). In some cases the same "typ" header parameter value will be present in the outer JWT as well, to explicitly type the entire Nested JWT.

Note that the use of explicit typing may not achieve disambiguation from existing kinds of JWTs, as the validation rules for existing kinds of JWTs often do not use the "typ" header parameter value. Explicit typing is RECOMMENDED for new uses of JWTs.

3.12. Use Mutually Exclusive Validation Rules for Different Kinds of JWTs

Each application of JWTs defines a profile specifying the required and optional JWT claims and the validation rules associated with them. If more than one kind of JWT can be issued by the same issuer, the validation rules for those JWTs MUST be written such that they are mutually exclusive, rejecting JWTs of the wrong kind. To prevent substitution of JWTs from one context into another, application developers may employ a number of strategies:

- Use explicit typing for different kinds of JWTs. Then the distinct "typ" values can be used to differentiate between the different kinds of JWTs.
- Use different sets of required claims or different required claim values. Then the validation rules for one kind of JWT will reject those with different claims or values.
- Use different sets of required header parameters or different required header parameter values. Then the validation rules for one kind of JWT will reject those with different header parameters or values.
- Use different keys for different kinds of JWTs. Then the keys used to validate one kind of JWT will fail to validate other kinds of JWTs.
- Use different "aud" values for different uses of JWTs from the same issuer. Then audience validation will reject JWTs substituted into inappropriate contexts.
- Use different issuers for different kinds of JWTs. Then the distinct "iss" values can be used to segregate the different kinds of JWTs.

Given the broad diversity of JWT usage and applications, the best combination of types, required claims, values, header parameters, key usages, and issuers to differentiate among different kinds of JWTs will, in general, be application specific. As discussed in Section 3.11, for new JWT applications, the use of explicit typing is RECOMMENDED.

4. Security Considerations

This entire document is about security considerations when implementing and deploying JSON Web Tokens.

5. IANA Considerations

This document requires no IANA actions.

6. Acknowledgements

Thanks to Antonio Sanso for bringing the "ECDH-ES" invalid point attack to the attention of JWE and JWT implementers. Tim McLean [McLean] published the RSA/HMAC confusion attack. Thanks to Nat Sakimura for advocating the use of explicit typing. Thanks to Neil Madden for his numerous comments, and to Carsten Bormann, Brian Campbell, Brian Carpenter, Alissa Cooper, Roman Danyliw, Ben Kaduk, Mirja Kuehlewind, Barry Leiba, Eric Rescorla, Adam Roach, Martin Vigoureux, and Eric Vyncke for their reviews.

7. References

7.1. Normative References

- [nist-sp-800-56a-r3] Barker, E., Chen, L., Keller, S., Roginsky, A., Vassilev, A., and R. Davis, "Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography, Draft NIST Special Publication 800-56A Revision 3", April 2018, <<https://doi.org/10.6028/NIST.SP.800-56Ar3>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC6979] Pornin, T., "Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)", RFC 6979, DOI 10.17487/RFC6979, August 2013, <<https://www.rfc-editor.org/info/rfc6979>>.
- [RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <<https://www.rfc-editor.org/info/rfc7515>>.
- [RFC7516] Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)", RFC 7516, DOI 10.17487/RFC7516, May 2015, <<https://www.rfc-editor.org/info/rfc7516>>.
- [RFC7518] Jones, M., "JSON Web Algorithms (JWA)", RFC 7518, DOI 10.17487/RFC7518, May 2015, <<https://www.rfc-editor.org/info/rfc7518>>.

- [RFC7519] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <<https://www.rfc-editor.org/info/rfc7519>>.
- [RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <<https://www.rfc-editor.org/info/rfc8017>>.
- [RFC8037] Liusvaara, I., "CFRG Elliptic Curve Diffie-Hellman (ECDH) and Signatures in JSON Object Signing and Encryption (JOSE)", RFC 8037, DOI 10.17487/RFC8037, January 2017, <<https://www.rfc-editor.org/info/rfc8037>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <<https://www.rfc-editor.org/info/rfc8259>>.

7.2. Informative References

- [Alawatugoda] Alawatugoda, J., Stebila, D., and C. Boyd, "Protecting Encrypted Cookies from Compression Side-Channel Attacks", Financial Cryptography and Data Security pp. 86-106, DOI 10.1007/978-3-662-47854-7_6, 2015.
- [ANSI-X962-2005] "American National Standard X9.62: The Elliptic Curve Digital Signature Algorithm (ECDSA)", November 2005.
- [Kelsey] Kelsey, J., "Compression and Information Leakage of Plaintext", Fast Software Encryption pp. 263-276, DOI 10.1007/3-540-45661-9_21, 2002.
- [Langkemper] Langkemper, S., "Attacking JWT Authentication", September 2016, <<https://www.sjoerdlangkemper.nl/2016/09/28/attacking-jwt-authentication/>>.
- [McLean] McLean, T., "Critical vulnerabilities in JSON Web Token libraries", March 2015, <<https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/>>.

- [OpenID.Core] Sakimura, N., Bradley, J., Jones, M., Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0", November 2014, <http://openid.net/specs/openid-connect-core-1_0.html>.
- [RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <<https://www.rfc-editor.org/info/rfc6749>>.
- [RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, <<https://www.rfc-editor.org/info/rfc7159>>.
- [RFC7517] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, <<https://www.rfc-editor.org/info/rfc7517>>.
- [RFC8414] Jones, M., Sakimura, N., and J. Bradley, "OAuth 2.0 Authorization Server Metadata", RFC 8414, DOI 10.17487/RFC8414, June 2018, <<https://www.rfc-editor.org/info/rfc8414>>.
- [RFC8417] Hunt, P., Ed., Jones, M., Denniss, W., and M. Ansari, "Security Event Token (SET)", RFC 8417, DOI 10.17487/RFC8417, July 2018, <<https://www.rfc-editor.org/info/rfc8417>>.
- [Sanso] Sanso, A., "Critical Vulnerability Uncovered in JSON Encryption", March 2017, <<https://blogs.adobe.com/security/2017/03/critical-vulnerability-uncovered-in-json-encryption.html>>.
- [Valenta] Valenta, L., Sullivan, N., Sanso, A., and N. Heninger, "In search of CurveSwap: Measuring elliptic curve implementations in the wild", March 2018, <<https://ia.cr/2018/298>>.

Appendix A. Document History

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

A.1. draft-ietf-oauth-jwt-bcp-07

- IESG review comments.

A.2. draft-ietf-oauth-jwt-bcp-06

- Second AD review.
- Removed unworkable recommendation to pad encrypted passwords.

A.3. draft-ietf-oauth-jwt-bcp-05

- Genart review comments.

A.4. draft-ietf-oauth-jwt-bcp-04

- AD review comments.

A.5. draft-ietf-oauth-jwt-bcp-03

- Acknowledgements.

A.6. draft-ietf-oauth-jwt-bcp-02

- Implemented WGLC feedback.

A.7. draft-ietf-oauth-jwt-bcp-01

- Feedback from Brian Campbell.

A.8. draft-ietf-oauth-jwt-bcp-00

- Initial WG draft. No change from the latest individual version.

A.9. draft-sheffer-oauth-jwt-bcp-01

- Added explicit typing.

A.10. draft-sheffer-oauth-jwt-bcp-00

- Initial version.

Authors' Addresses

Yaron Sheffer
Intuit

EMail: yaronf.ietf@gmail.com

Dick Hardt

EMail: dick.hardt@gmail.com

Michael B. Jones
Microsoft

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

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: February 23, 2020

B. Campbell
Ping Identity
J. Bradley
Yubico
N. Sakimura
Nomura Research Institute
T. Lodderstedt
YES.com AG
August 22, 2019

OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound
Access Tokens
draft-ietf-oauth-mtls-17

Abstract

This document describes OAuth client authentication and certificate-bound access and refresh tokens using mutual Transport Layer Security (TLS) authentication with X.509 certificates. OAuth clients are provided a mechanism for authentication to the authorization server using mutual TLS, based on either self-signed certificates or public key infrastructure (PKI). OAuth authorization servers are provided a mechanism for binding access tokens to a client's mutual-TLS certificate, and OAuth protected resources are provided a method for ensuring that such an access token presented to it was issued to the client presenting the token.

Status of This Memo

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

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

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

This Internet-Draft will expire on February 23, 2020.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction	3
1.1. Requirements Notation and Conventions	5
1.2. Terminology	5
2. Mutual TLS for OAuth Client Authentication	5
2.1. PKI Mutual-TLS Method	6
2.1.1. PKI Method Metadata Value	7
2.1.2. Client Registration Metadata	7
2.2. Self-Signed Certificate Mutual-TLS Method	8
2.2.1. Self-Signed Method Metadata Value	8
2.2.2. Client Registration Metadata	8
3. Mutual-TLS Client Certificate-Bound Access Tokens	9
3.1. JWT Certificate Thumbprint Confirmation Method	10
3.2. Confirmation Method for Token Introspection	11
3.3. Authorization Server Metadata	12
3.4. Client Registration Metadata	12
4. Public Clients and Certificate-Bound Tokens	13
5. Metadata for Mutual-TLS Endpoint Aliases	13
6. Implementation Considerations	15
6.1. Authorization Server	15
6.2. Resource Server	16
6.3. Certificate Expiration and Bound Access Tokens	16
6.4. Implicit Grant Unsupported	16
6.5. TLS Termination	17
7. Security Considerations	17
7.1. Certificate-Bound Refresh Tokens	17
7.2. Certificate Thumbprint Binding	17
7.3. TLS Versions and Best Practices	18
7.4. X.509 Certificate Spoofing	18
7.5. X.509 Certificate Parsing and Validation Complexity	18
8. Privacy Considerations	19
9. IANA Considerations	19

9.1.	JWT Confirmation Methods Registration	19
9.2.	Authorization Server Metadata Registration	19
9.3.	Token Endpoint Authentication Method Registration	20
9.4.	Token Introspection Response Registration	20
9.5.	Dynamic Client Registration Metadata Registration	21
10.	References	22
10.1.	Normative References	22
10.2.	Informative References	24
Appendix A.	Example "cnf" Claim, Certificate and JWK	25
Appendix B.	Relationship to Token Binding	26
Appendix C.	Acknowledgements	26
Appendix D.	Document(s) History	27
Authors' Addresses	31

1. Introduction

The OAuth 2.0 Authorization Framework [RFC6749] enables third-party client applications to obtain delegated access to protected resources. In the prototypical abstract OAuth flow, illustrated in Figure 1, the client obtains an access token from an entity known as an authorization server and then uses that token when accessing protected resources, such as HTTPS APIs.

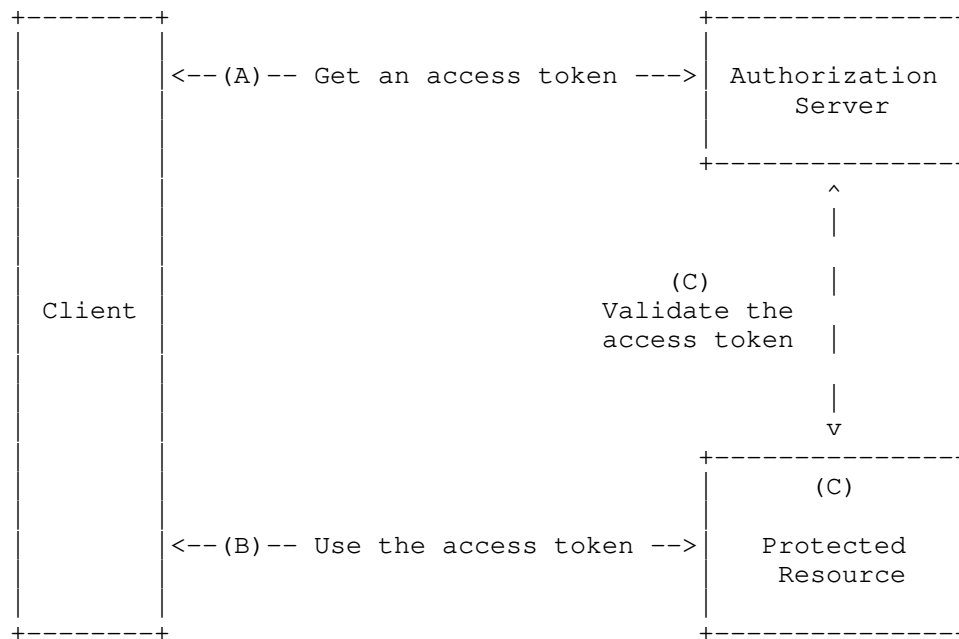


Figure 1: Abstract OAuth 2.0 Protocol Flow

The flow illustrated in Figure 1 includes the following steps:

- (A) The client makes an HTTPS "POST" request to the authorization server and presents a credential representing the authorization grant. For certain types of clients (those that have been issued or otherwise established a set of client credentials) the request must be authenticated. In the response, the authorization server issues an access token to the client.
- (B) The client includes the access token when making a request to access a protected resource.
- (C) The protected resource validates the access token in order to authorize the request. In some cases, such as when the token is self-contained and cryptographically secured, the validation can be done locally by the protected resource. Other cases require that the protected resource call out to the authorization server to determine the state of the token and obtain meta-information about it.

Layering on the abstract flow above, this document standardizes enhanced security options for OAuth 2.0 utilizing client-certificate-based mutual TLS. Section 2 provides options for authenticating the request in step (A). Step (C) is supported with semantics to express the binding of the token to the client certificate for both local and remote processing in Section 3.1 and Section 3.2 respectively. This ensures that, as described in Section 3, protected resource access in step (B) is only possible by the legitimate client using a certificate-bound token and holding the private key corresponding to the certificate.

OAuth 2.0 defines a shared-secret method of client authentication but also allows for definition and use of additional client authentication mechanisms when interacting directly with the authorization server. This document describes an additional mechanism of client authentication utilizing mutual-TLS certificate-based authentication, which provides better security characteristics than shared secrets. While [RFC6749] documents client authentication for requests to the token endpoint, extensions to OAuth 2.0 (such as Introspection [RFC7662], Revocation [RFC7009], and the Backchannel Authentication Endpoint in [OpenID.CIBA]) define endpoints that also utilize client authentication and the mutual TLS methods defined herein are applicable to those endpoints as well.

Mutual-TLS certificate-bound access tokens ensure that only the party in possession of the private key corresponding to the certificate can utilize the token to access the associated resources. Such a constraint is sometimes referred to as key confirmation, proof-of-

possession, or holder-of-key and is unlike the case of the bearer token described in [RFC6750], where any party in possession of the access token can use it to access the associated resources. Binding an access token to the client's certificate prevents the use of stolen access tokens or replay of access tokens by unauthorized parties.

Mutual-TLS certificate-bound access tokens and mutual-TLS client authentication are distinct mechanisms, which are complementary but don't necessarily need to be deployed or used together.

Additional client metadata parameters are introduced by this document in support of certificate-bound access tokens and mutual-TLS client authentication. The authorization server can obtain client metadata via the Dynamic Client Registration Protocol [RFC7591], which defines mechanisms for dynamically registering OAuth 2.0 client metadata with authorization servers. Also the metadata defined by RFC7591, and registered extensions to it, imply a general data model for clients that is useful for authorization server implementations even when the Dynamic Client Registration Protocol isn't in play. Such implementations will typically have some sort of user interface available for managing client configuration.

1.1. Requirements Notation and Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

1.2. Terminology

Throughout this document the term "mutual TLS" refers to the process whereby, in addition to the normal TLS server authentication with a certificate, a client presents its X.509 certificate and proves possession of the corresponding private key to a server when negotiating a TLS session. In contemporary versions of TLS [RFC8446] [RFC5246] this requires that the client send the Certificate and CertificateVerify messages during the handshake and for the server to verify the CertificateVerify and Finished messages.

2. Mutual TLS for OAuth Client Authentication

This section defines, as an extension of OAuth 2.0, Section 2.3 [RFC6749], two distinct methods of using mutual-TLS X.509 client certificates as client credentials. The requirement of mutual TLS for client authentication is determined by the authorization server

based on policy or configuration for the given client (regardless of whether the client was dynamically registered, statically configured, or otherwise established).

In order to utilize TLS for OAuth client authentication, the TLS connection between the client and the authorization server MUST have been established or reestablished with mutual-TLS X.509 certificate authentication (i.e. the Client Certificate and Certificate Verify messages are sent during the TLS Handshake).

For all requests to the authorization server utilizing mutual-TLS client authentication, the client MUST include the "client_id" parameter, described in OAuth 2.0, Section 2.2 [RFC6749]. The presence of the "client_id" parameter enables the authorization server to easily identify the client independently from the content of the certificate. The authorization server can locate the client configuration using the client identifier and check the certificate presented in the TLS Handshake against the expected credentials for that client. The authorization server MUST enforce the binding between client and certificate as described in either Section 2.1 or Section 2.2 below. If no certificate is presented or that which is presented doesn't match that which is expected for the given "client_id", the authorization server returns a normal OAuth 2.0 error response per Section 5.2 of RFC6749 [RFC6749] with the "invalid_client" error code to indicate failed client authentication.

2.1. PKI Mutual-TLS Method

The PKI (public key infrastructure) method of mutual-TLS OAuth client authentication adheres to the way in which X.509 certificates are traditionally used for authentication. It relies on a validated certificate chain [RFC5280] and a single subject distinguished name (DN) or a single subject alternative name (SAN) to authenticate the client. Only one subject name value of any type is used for each client. The TLS handshake is utilized to validate the client's possession of the private key corresponding to the public key in the certificate and to validate the corresponding certificate chain. The client is successfully authenticated if the subject information in the certificate matches the single expected subject configured or registered for that particular client (note that a predictable treatment of DN values, such as the distinguishedNameMatch rule from [RFC4517], is needed in comparing the certificate's subject DN to the client's registered DN). Revocation checking is possible with the PKI method but if and how to check a certificate's revocation status is a deployment decision at the discretion of the authorization server. Clients can rotate their X.509 certificates without the need to modify the respective authentication data at the authorization

server by obtaining a new certificate with the same subject from a trusted certificate authority (CA).

2.1.1. PKI Method Metadata Value

For the PKI method of mutual-TLS client authentication, this specification defines and registers the following authentication method metadata value into the "OAuth Token Endpoint Authentication Methods" registry [IANA.OAuth.Parameters].

`tls_client_auth`

Indicates that client authentication to the authorization server will occur with mutual TLS utilizing the PKI method of associating a certificate to a client.

2.1.2. Client Registration Metadata

In order to convey the expected subject of the certificate, the following metadata parameters are introduced for the OAuth 2.0 Dynamic Client Registration Protocol [RFC7591] in support of the PKI method of mutual-TLS client authentication. A client using the "tls_client_auth" authentication method MUST use exactly one of the below metadata parameters to indicate the certificate subject value that the authorization server is to expect when authenticating the respective client.

`tls_client_auth_subject_dn`

An [RFC4514] string representation of the expected subject distinguished name of the certificate, which the OAuth client will use in mutual-TLS authentication.

`tls_client_auth_san_dns`

A string containing the value of an expected `dNSName` SAN entry in the certificate, which the OAuth client will use in mutual-TLS authentication.

`tls_client_auth_san_uri`

A string containing the value of an expected `uniformResourceIdentifier` SAN entry in the certificate, which the OAuth client will use in mutual-TLS authentication.

`tls_client_auth_san_ip`

A string representation of an IP address in either dotted decimal notation (for IPv4) or colon-delimited hexadecimal (for IPv6, as defined in [RFC5952]) that is expected to be present as an `iPAddress` SAN entry in the certificate, which the OAuth client will use in mutual-TLS authentication. Per section 8 of [RFC5952]

the IP address comparison of the value in this parameter and the SAN entry in the certificate is to be done in binary format.

`tls_client_auth_san_email`

A string containing the value of an expected rfc822Name SAN entry in the certificate, which the OAuth client will use in mutual-TLS authentication.

2.2. Self-Signed Certificate Mutual-TLS Method

This method of mutual-TLS OAuth client authentication is intended to support client authentication using self-signed certificates. As a prerequisite, the client registers its X.509 certificates (using "jwks" defined in [RFC7591]) or a reference to a trusted source for its X.509 certificates (using "jwks_uri" from [RFC7591]) with the authorization server. During authentication, TLS is utilized to validate the client's possession of the private key corresponding to the public key presented within the certificate in the respective TLS handshake. In contrast to the PKI method, the client's certificate chain is not validated by the server in this case. The client is successfully authenticated if the certificate that it presented during the handshake matches one of the certificates configured or registered for that particular client. The Self-Signed Certificate method allows the use of mutual TLS to authenticate clients without the need to maintain a PKI. When used in conjunction with a "jwks_uri" for the client, it also allows the client to rotate its X.509 certificates without the need to change its respective authentication data directly with the authorization server.

2.2.1. Self-Signed Method Metadata Value

For the Self-Signed Certificate method of mutual-TLS client authentication, this specification defines and registers the following authentication method metadata value into the "OAuth Token Endpoint Authentication Methods" registry [IANA.OAuth.Parameters].

`self_signed_tls_client_auth`

Indicates that client authentication to the authorization server will occur using mutual TLS with the client utilizing a self-signed certificate.

2.2.2. Client Registration Metadata

For the Self-Signed Certificate method of binding a certificate with a client using mutual TLS client authentication, the existing "jwks_uri" or "jwks" metadata parameters from [RFC7591] are used to convey the client's certificates via JSON Web Key (JWK) in a JWK Set (JWKS) [RFC7517]. The "jwks" metadata parameter is a JWK Set

containing the client's public keys as an array of JWKs while the "jwks_uri" parameter is a URL that references a client's JWK Set. A certificate is represented with the "x5c" parameter of an individual JWK within the set. Note that the members of the JWK representing the public key (e.g. "n" and "e" for RSA, "x" and "y" for EC) are required parameters per [RFC7518] so will be present even though they are not utilized in this context. Also note that that Section 4.7 of [RFC7517] requires that the key in the first certificate of the "x5c" parameter match the public key represented by those other members of the JWK.

3. Mutual-TLS Client Certificate-Bound Access Tokens

When mutual TLS is used by the client on the connection to the token endpoint, the authorization server is able to bind the issued access token to the client certificate. Such a binding is accomplished by associating the certificate with the token in a way that can be accessed by the protected resource, such as embedding the certificate hash in the issued access token directly, using the syntax described in Section 3.1, or through token introspection as described in Section 3.2. Binding the access token to the client certificate in that fashion has the benefit of decoupling that binding from the client's authentication with the authorization server, which enables mutual TLS during protected resource access to serve purely as a proof-of-possession mechanism. Other methods of associating a certificate with an access token are possible, per agreement by the authorization server and the protected resource, but are beyond the scope of this specification.

In order for a resource server to use certificate-bound access tokens, it must have advance knowledge that mutual TLS is to be used for some or all resource accesses. In particular, the access token itself cannot be used as input to the decision of whether or not to request mutual TLS, since from the TLS perspective those are "Application Data", only exchanged after the TLS handshake has been completed, and the initial CertificateRequest occurs during the handshake, before the Application Data is available. Although subsequent opportunities for a TLS client to present a certificate may be available, e.g., via TLS 1.2 renegotiation [RFC5246] or TLS 1.3 post-handshake authentication [RFC8446], this document makes no provision for their usage. It is expected to be common that a mutual-TLS-using resource server will require mutual TLS for all resources hosted thereupon, or will serve mutual-TLS-protected and regular resources on separate hostname+port combinations, though other workflows are possible. How resource server policy is synchronized with the AS is out of scope for this document.

Within the scope of an mutual-TLS-protected resource-access flow, the client makes protected resource requests as described in [RFC6750], however, those requests MUST be made over a mutually authenticated TLS connection using the same certificate that was used for mutual TLS at the token endpoint.

The protected resource MUST obtain, from its TLS implementation layer, the client certificate used for mutual TLS and MUST verify that the certificate matches the certificate associated with the access token. If they do not match, the resource access attempt MUST be rejected with an error per [RFC6750] using an HTTP 401 status code and the "invalid_token" error code.

Metadata to convey server and client capabilities for mutual-TLS client certificate-bound access tokens is defined in Section 3.3 and Section 3.4 respectively.

3.1. JWT Certificate Thumbprint Confirmation Method

When access tokens are represented as JSON Web Tokens (JWT) [RFC7519], the certificate hash information SHOULD be represented using the "x5t#S256" confirmation method member defined herein.

To represent the hash of a certificate in a JWT, this specification defines the new JWT Confirmation Method [RFC7800] member "x5t#S256" for the X.509 Certificate SHA-256 Thumbprint. The value of the "x5t#S256" member is a base64url-encoded [RFC4648] SHA-256 [SHS] hash (a.k.a. thumbprint, fingerprint or digest) of the DER encoding [X690] of the X.509 certificate [RFC5280]. The base64url-encoded value MUST omit all trailing pad '=' characters and MUST NOT include any line breaks, whitespace, or other additional characters.

The following is an example of a JWT payload containing an "x5t#S256" certificate thumbprint confirmation method. The new JWT content introduced by this specification is the "cnf" confirmation method claim at the bottom of the example that has the "x5t#S256" confirmation method member containing the value that is the hash of the client certificate to which the access token is bound.

```
{
  "iss": "https://server.example.com",
  "sub": "ty.webb@example.com",
  "exp": 1493726400,
  "nbf": 1493722800,
  "cnf": {
    "x5t#S256": "bwcK0esc3ACC3DB2Y5_lESsXE8o9ltc05O89jdN-dg2"
  }
}
```

Figure 2: Example JWT Claims Set with an X.509 Certificate Thumbprint Confirmation Method

3.2. Confirmation Method for Token Introspection

OAuth 2.0 Token Introspection [RFC7662] defines a method for a protected resource to query an authorization server about the active state of an access token as well as to determine meta-information about the token.

For a mutual-TLS client certificate-bound access token, the hash of the certificate to which the token is bound is conveyed to the protected resource as meta-information in a token introspection response. The hash is conveyed using the same "cnf" with "x5t#S256" member structure as the certificate SHA-256 thumbprint confirmation method, described in Section 3.1, as a top-level member of the introspection response JSON. The protected resource compares that certificate hash to a hash of the client certificate used for mutual-TLS authentication and rejects the request, if they do not match.

The following is an example of an introspection response for an active token with an "x5t#S256" certificate thumbprint confirmation method. The new introspection response content introduced by this specification is the "cnf" confirmation method at the bottom of the example that has the "x5t#S256" confirmation method member containing the value that is the hash of the client certificate to which the access token is bound.

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

{
  "active": true,
  "iss": "https://server.example.com",
  "sub": "ty.webb@example.com",
  "exp": 1493726400,
  "nbf": 1493722800,
  "cnf": {
    "x5t#S256": "bwcK0esc3ACC3DB2Y5_lESsXE8o9ltc05089jdN-dg2"
  }
}
```

Figure 3: Example Introspection Response for a Certificate-Bound Access Token

3.3. Authorization Server Metadata

This document introduces the following new authorization server metadata [RFC8414] parameter to signal the server's capability to issue certificate bound access tokens:

`tls_client_certificate_bound_access_tokens`
OPTIONAL. Boolean value indicating server support for mutual-TLS client certificate-bound access tokens. If omitted, the default value is "false".

3.4. Client Registration Metadata

The following new client metadata parameter is introduced to convey the client's intention to use certificate bound access tokens:

`tls_client_certificate_bound_access_tokens`
OPTIONAL. Boolean value used to indicate the client's intention to use mutual-TLS client certificate-bound access tokens. If omitted, the default value is "false".

Note that, if a client that has indicated the intention to use mutual-TLS client certificate-bound tokens makes a request to the token endpoint over a non-mutual-TLS connection, it is at the authorization server's discretion as to whether to return an error or issue an unbound token.

4. Public Clients and Certificate-Bound Tokens

Mutual-TLS OAuth client authentication and certificate-bound access tokens can be used independently of each other. Use of certificate-bound access tokens without mutual-TLS OAuth client authentication, for example, is possible in support of binding access tokens to a TLS client certificate for public clients (those without authentication credentials associated with the "client_id"). The authorization server would configure the TLS stack in the same manner as for the Self-Signed Certificate method such that it does not verify that the certificate presented by the client during the handshake is signed by a trusted CA. Individual instances of a client would create a self-signed certificate for mutual TLS with both the authorization server and resource server. The authorization server would not use the mutual-TLS certificate to authenticate the client at the OAuth layer but would bind the issued access token to that certificate, for which the client has proven possession of the corresponding private key. The access token is then bound to the certificate and can only be used by the client possessing the certificate and corresponding private key and utilizing them to negotiate mutual TLS on connections to the resource server. When the authorization server issues a refresh token to such a client, it SHOULD also bind the refresh token to the respective certificate. And check the binding when the refresh token is presented to get new access tokens. The implementation details of the binding the refresh token are at the discretion of the authorization server.

5. Metadata for Mutual-TLS Endpoint Aliases

The process of negotiating client certificate-based mutual TLS involves a TLS server requesting a certificate from the TLS client (the client does not provide one unsolicited). Although a server can be configured such that client certificates are optional, meaning that the connection is allowed to continue when the client does not provide a certificate, the act of a server requesting a certificate can result in undesirable behavior from some clients. This is particularly true of web browsers as TLS clients, which will typically present the end-user with an intrusive certificate selection interface when the server requests a certificate.

Authorization servers supporting both clients using mutual TLS and conventional clients MAY chose to isolate the server side mutual-TLS behavior to only clients intending to do mutual TLS, thus avoiding any undesirable effects it might have on conventional clients. The following authorization server metadata parameter is introduced to facilitate such separation:

mtls_endpoint_aliases

OPTIONAL. A JSON object containing alternative authorization server endpoints that, when present, an OAuth client intending to do mutual TLS uses in preference to the conventional endpoints. The parameter value itself consists of one or more endpoint parameters, such as "token_endpoint", "revocation_endpoint", "introspection_endpoint", etc., conventionally defined for the top-level of authorization server metadata. An OAuth client intending to do mutual TLS (for OAuth client authentication and/or to acquire or use certificate-bound tokens) when making a request directly to the authorization server MUST use the alias URL of the endpoint within the "mtls_endpoint_aliases", when present, in preference to the endpoint URL of the same name at top-level of metadata. When an endpoint is not present in "mtls_endpoint_aliases", then the client uses the conventional endpoint URL defined at the top-level of the authorization server metadata. Metadata parameters within "mtls_endpoint_aliases" that do not define endpoints to which an OAuth client makes a direct request have no meaning and SHOULD be ignored.

Below is an example of an authorization server metadata document with the "mtls_endpoint_aliases" parameter, which indicates aliases for the token, revocation, and introspection endpoints that an OAuth client intending to do mutual TLS would in preference to the conventional token, revocation, and introspection endpoints. Note that the endpoints in "mtls_endpoint_aliases" use a different host than their conventional counterparts, which allows the authorization server (via TLS "server_name" extension [RFC6066] or actual distinct hosts) to differentiate its TLS behavior as appropriate.

```
{
  "issuer": "https://server.example.com",
  "authorization_endpoint": "https://server.example.com/authz",
  "token_endpoint": "https://server.example.com/token",
  "introspection_endpoint": "https://server.example.com/introspect",
  "revocation_endpoint": "https://server.example.com/revo",
  "jwks_uri": "https://server.example.com/jwks",
  "response_types_supported": ["code"],
  "response_modes_supported": ["fragment", "query", "form_post"],
  "grant_types_supported": ["authorization_code", "refresh_token"],
  "token_endpoint_auth_methods_supported":
    ["tls_client_auth", "client_secret_basic", "none"],
  "tls_client_certificate_bound_access_tokens": true
  "mtls_endpoint_aliases": {
    "token_endpoint": "https://mtls.example.com/token",
    "revocation_endpoint": "https://mtls.example.com/revo",
    "introspection_endpoint": "https://mtls.example.com/introspect"
  }
}
```

Figure 4: Example Authorization Server Metadata with Mutual-TLS
Endpoint Aliases

6. Implementation Considerations

6.1. Authorization Server

The authorization server needs to set up its TLS configuration appropriately for the OAuth client authentication methods it supports.

An authorization server that supports mutual-TLS client authentication and other client authentication methods or public clients in parallel would make mutual TLS optional (i.e. allowing a handshake to continue after the server requests a client certificate but the client does not send one).

In order to support the Self-Signed Certificate method alone, the authorization server would configure the TLS stack in such a way that it does not verify whether the certificate presented by the client during the handshake is signed by a trusted CA certificate.

As described in Section 3, the authorization server binds the issued access token to the TLS client certificate, which means that it will only issue certificate-bound tokens for a certificate which the client has proven possession of the corresponding private key.

The authorization server may also consider hosting the token endpoint, and other endpoints requiring client authentication, on a separate host name or port in order to prevent unintended impact on the TLS behavior of its other endpoints, e.g. the authorization endpoint. As described in Section 5, it may further isolate any potential impact of the server requesting client certificates by offering a distinct set of endpoints on a separate host or port, which are aliases for the originals that a client intending to do mutual TLS will use in preference to the conventional endpoints.

6.2. Resource Server

OAuth divides the roles and responsibilities such that the resource server relies on the authorization server to perform client authentication and obtain resource owner (end-user) authorization. The resource server makes authorization decisions based on the access token presented by the client but does not directly authenticate the client per se. The manner in which an access token is bound to the client certificate and how a protected resource verifies the proof-of-possession decouples that from the specific method that the client used to authenticate with the authorization server. Mutual TLS during protected resource access can therefore serve purely as a proof-of-possession mechanism. As such, it is not necessary for the resource server to validate the trust chain of the client's certificate in any of the methods defined in this document. The resource server would therefore configure the TLS stack in a way that it does not verify whether the certificate presented by the client during the handshake is signed by a trusted CA certificate.

6.3. Certificate Expiration and Bound Access Tokens

As described in Section 3, an access token is bound to a specific client certificate, which means that the same certificate must be used for mutual TLS on protected resource access. It also implies that access tokens are invalidated when a client updates the certificate, which can be handled similar to expired access tokens where the client requests a new access token (typically with a refresh token) and retries the protected resource request.

6.4. Implicit Grant Unsupported

This document describes binding an access token to the client certificate presented on the TLS connection from the client to the authorization server's token endpoint, however, such binding of access tokens issued directly from the authorization endpoint via the implicit grant flow is explicitly out of scope. End users interact directly with the authorization endpoint using a web browser and the use of client certificates in user's browsers bring operational and

usability issues, which make it undesirable to support certificate-bound access tokens issued in the implicit grant flow. Implementations wanting to employ certificate-bound access tokens should utilize grant types that involve the client making an access token request directly to the token endpoint (e.g. the authorization code and refresh token grant types).

6.5. TLS Termination

An authorization server or resource server MAY choose to terminate TLS connections at a load balancer, reverse proxy, or other network intermediary. How the client certificate metadata is securely communicated between the intermediary and the application server in this case is out of scope of this specification.

7. Security Considerations

7.1. Certificate-Bound Refresh Tokens

The OAuth 2.0 Authorization Framework [RFC6749] requires that an authorization server bind refresh tokens to the client to which they were issued and that confidential clients (those having established authentication credentials with the authorization server) authenticate to the AS when presenting a refresh token. As a result, refresh tokens are indirectly certificate-bound by way of the client ID and the associated requirement for (certificate-based) authentication to the authorization server when issued to clients utilizing the "tls_client_auth" or "self_signed_tls_client_auth" methods of client authentication. Section 4 describes certificate-bound refresh tokens issued to public clients (those without authentication credentials associated with the "client_id").

7.2. Certificate Thumbprint Binding

The binding between the certificate and access token specified in Section 3.1 uses a cryptographic hash of the certificate. It relies on the hash function having sufficient second-preimage resistance so as to make it computationally infeasible to find or create another certificate that produces to the same hash output value. The SHA-256 hash function was used because it meets the aforementioned requirement while being widely available. If, in the future, certificate thumbprints need to be computed using hash function(s) other than SHA-256, it is suggested that additional related JWT confirmation methods members be defined for that purpose and registered in the IANA "JWT Confirmation Methods" registry [IANA.JWT.Claims] for JWT "cnf" member values.

Community knowledge about the strength of various algorithms and feasible attacks can change suddenly, and experience shows that a document about security is a point-in-time statement. Readers are advised to seek out any errata or updates that apply to this document.

7.3. TLS Versions and Best Practices

In the abstract this document is applicable with any TLS version supporting certificate-based client authentication. Both TLS 1.3 [RFC8446] and TLS 1.2 [RFC5246] are cited herein because, at the time of writing, 1.3 is the newest version while 1.2 is the most widely deployed. General implementation and security considerations for TLS, including version recommendations, can be found in [BCP195].

TLS certificate validation (for both client and server certificates) requires a local database of trusted certificate authorities (CAs). Decisions about what CAs to trust and how to make such a determination of trust are out of scope for this document.

7.4. X.509 Certificate Spoofing

If the PKI method of client authentication is used, an attacker could try to impersonate a client using a certificate with the same subject (DN or SAN) but issued by a different CA, which the authorization server trusts. To cope with that threat, the authorization server SHOULD only accept as trust anchors a limited number of CAs whose certificate issuance policy meets its security requirements. There is an assumption then that the client and server agree out of band on the set of trust anchors that the server uses to create and validate the certificate chain. Without this assumption the use of a subject to identify the client certificate would open the server up to certificate spoofing attacks.

7.5. X.509 Certificate Parsing and Validation Complexity

Parsing and validation of X.509 certificates and certificate chains is complex and implementation mistakes have previously exposed security vulnerabilities. Complexities of validation include (but are not limited to) [CX5P] [DCW] [RFC5280]:

- o checking of Basic Constraints, basic and extended Key Usage constraints, validity periods, and critical extensions;
- o handling of embedded NUL bytes in ASN.1 counted-length strings, and non-canonical or non-normalized string representations in subject names;

- o handling of wildcard patterns in subject names;
- o recursive verification of certificate chains and checking certificate revocation.

For these reasons, implementors SHOULD use an established and well-tested X.509 library (such as one used by an established TLS library) for validation of X.509 certificate chains and SHOULD NOT attempt to write their own X.509 certificate validation procedures.

8. Privacy Considerations

In TLS versions prior to 1.3, the client's certificate is sent unencrypted in the initial handshake and can potentially be used by third parties to monitor, track, and correlate client activity. This is likely of little concern for clients that act on behalf of a significant number of end-users because individual user activity will not be discernible amidst the client activity as a whole. However, clients that act on behalf of a single end-user, such as a native application on a mobile device, should use TLS version 1.3 whenever possible or consider the potential privacy implications of using mutual TLS on earlier versions.

9. IANA Considerations

9.1. JWT Confirmation Methods Registration

This specification requests registration of the following value in the IANA "JWT Confirmation Methods" registry [IANA.JWT.Claims] for JWT "cnf" member values established by [RFC7800].

- o Confirmation Method Value: "x5t#S256"
- o Confirmation Method Description: X.509 Certificate SHA-256 Thumbprint
- o Change Controller: IESG
- o Specification Document(s): Section 3.1 of [[this specification]]

9.2. Authorization Server Metadata Registration

This specification requests registration of the following values in the IANA "OAuth Authorization Server Metadata" registry [IANA.OAuth.Parameters] established by [RFC8414].

- o Metadata Name: "tls_client_certificate_bound_access_tokens"
- o Metadata Description: Indicates authorization server support for mutual-TLS client certificate-bound access tokens.
- o Change Controller: IESG
- o Specification Document(s): Section 3.3 of [[this specification]]

- o Metadata Name: "mtls_endpoint_aliases"
- o Metadata Description: JSON object containing alternative authorization server endpoints, which a client intending to do mutual TLS will use in preference to the conventional endpoints.
- o Change Controller: IESG
- o Specification Document(s): Section 5 of [[this specification]]

9.3. Token Endpoint Authentication Method Registration

This specification requests registration of the following values in the IANA "OAuth Token Endpoint Authentication Methods" registry [IANA.OAuth.Parameters] established by [RFC7591].

- o Token Endpoint Authentication Method Name: "tls_client_auth"
- o Change Controller: IESG
- o Specification Document(s): Section 2.1.1 of [[this specification]]
- o Token Endpoint Authentication Method Name: "self_signed_tls_client_auth"
- o Change Controller: IESG
- o Specification Document(s): Section 2.2.1 of [[this specification]]

9.4. Token Introspection Response Registration

Proof-of-Possession Key Semantics for JSON Web Tokens [RFC7800] defined the "cnf" (confirmation) claim, which enables confirmation key information to be carried in a JWT. However, the same proof-of-possession semantics are also useful for introspected access tokens whereby the protected resource obtains the confirmation key data as meta-information of a token introspection response and uses that information in verifying proof-of-possession. Therefore this specification defines and registers proof-of-possession semantics for OAuth 2.0 Token Introspection [RFC7662] using the "cnf" structure. When included as a top-level member of an OAuth token introspection response, "cnf" has the same semantics and format as the claim of the same name defined in [RFC7800]. While this specification only explicitly uses the "x5t#S256" confirmation method member (see Section 3.2), it needs to define and register the higher level "cnf" structure as an introspection response member in order to define and use the more specific certificate thumbprint confirmation method.

As such, this specification requests registration of the following value in the IANA "OAuth Token Introspection Response" registry [IANA.OAuth.Parameters] established by [RFC7662].

- o Claim Name: "cnf"

- o Claim Description: Confirmation
- o Change Controller: IESG
- o Specification Document(s): [RFC7800] and [[this specification]]

9.5. Dynamic Client Registration Metadata Registration

This specification requests registration of the following client metadata definitions in the IANA "OAuth Dynamic Client Registration Metadata" registry [IANA.OAuth.Parameters] established by [RFC7591]:

- o Client Metadata Name: "tls_client_certificate_bound_access_tokens"
- o Client Metadata Description: Indicates the client's intention to use mutual-TLS client certificate-bound access tokens.
- o Change Controller: IESG
- o Specification Document(s): Section 3.4 of [[this specification]]
- o Client Metadata Name: "tls_client_auth_subject_dn"
- o Client Metadata Description: String value specifying the expected subject DN of the client certificate.
- o Change Controller: IESG
- o Specification Document(s): Section 2.1.2 of [[this specification]]
- o Client Metadata Name: "tls_client_auth_san_dns"
- o Client Metadata Description: String value specifying the expected dNSName SAN entry in the client certificate.
- o Change Controller: IESG
- o Specification Document(s): Section 2.1.2 of [[this specification]]
- o Client Metadata Name: "tls_client_auth_san_uri"
- o Client Metadata Description: String value specifying the expected uniformResourceIdentifier SAN entry in the client certificate.
- o Change Controller: IESG
- o Specification Document(s): Section 2.1.2 of [[this specification]]
- o Client Metadata Name: "tls_client_auth_san_ip"
- o Client Metadata Description: String value specifying the expected ipAddress SAN entry in the client certificate.
- o Change Controller: IESG
- o Specification Document(s): Section 2.1.2 of [[this specification]]
- o Client Metadata Name: "tls_client_auth_san_email"
- o Client Metadata Description: String value specifying the expected rfc822Name SAN entry in the client certificate.
- o Change Controller: IESG

- o Specification Document(s): Section 2.1.2 of [[this specification]]

10. References

10.1. Normative References

- [BCP195] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 7525, DOI 10.17487/RFC7525, May 2015, <<http://www.rfc-editor.org/info/bcp195>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC4514] Zeilenga, K., Ed., "Lightweight Directory Access Protocol (LDAP): String Representation of Distinguished Names", RFC 4514, DOI 10.17487/RFC4514, June 2006, <<https://www.rfc-editor.org/info/rfc4514>>.
- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <<https://www.rfc-editor.org/info/rfc4648>>.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <<https://www.rfc-editor.org/info/rfc5246>>.
- [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, DOI 10.17487/RFC5280, May 2008, <<https://www.rfc-editor.org/info/rfc5280>>.
- [RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <<https://www.rfc-editor.org/info/rfc6749>>.
- [RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, <<https://www.rfc-editor.org/info/rfc6750>>.

- [RFC7517] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, <<https://www.rfc-editor.org/info/rfc7517>>.
- [RFC7519] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <<https://www.rfc-editor.org/info/rfc7519>>.
- [RFC7591] Richer, J., Ed., Jones, M., Bradley, J., Machulak, M., and P. Hunt, "OAuth 2.0 Dynamic Client Registration Protocol", RFC 7591, DOI 10.17487/RFC7591, July 2015, <<https://www.rfc-editor.org/info/rfc7591>>.
- [RFC7662] Richer, J., Ed., "OAuth 2.0 Token Introspection", RFC 7662, DOI 10.17487/RFC7662, October 2015, <<https://www.rfc-editor.org/info/rfc7662>>.
- [RFC7800] Jones, M., Bradley, J., and H. Tschofenig, "Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs)", RFC 7800, DOI 10.17487/RFC7800, April 2016, <<https://www.rfc-editor.org/info/rfc7800>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC8414] Jones, M., Sakimura, N., and J. Bradley, "OAuth 2.0 Authorization Server Metadata", RFC 8414, DOI 10.17487/RFC8414, June 2018, <<https://www.rfc-editor.org/info/rfc8414>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.
- [SHS] National Institute of Standards and Technology, "Secure Hash Standard (SHS)", FIPS PUB 180-4, March 2012, <<http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf>>.
- [X690] International Telephone and Telegraph Consultative Committee, "ASN.1 encoding rules: Specification of basic encoding Rules (BER), Canonical encoding rules (CER) and Distinguished encoding rules (DER)", CCITT Recommendation X.690, July 2015.

10.2. Informative References

- [CX5P] Wong, D., "Common x509 certificate validation/creation pitfalls", September 2016, <<https://www.cryptologie.net/article/374/common-x509-certificate-validationcreation-pitfalls>>.
- [DCW] Georgiev, M., Iyengar, S., Jana, S., Anubhai, R., Boneh, D., and V. Shmatikov, "The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software", <http://www.cs.utexas.edu/~shmat/shmat_ccs12.pdf>.
- [I-D.ietf-oauth-token-binding] Jones, M., Campbell, B., Bradley, J., and W. Denniss, "OAuth 2.0 Token Binding", draft-ietf-oauth-token-binding-06 (work in progress), March 2018.
- [IANA.JWT.Claims] IANA, "JSON Web Token Claims", <<http://www.iana.org/assignments/jwt>>.
- [IANA.OAuth.Parameters] IANA, "OAuth Parameters", <<http://www.iana.org/assignments/oauth-parameters>>.
- [OpenID.CIBA] Fernandez, G., Walter, F., Nennker, A., Tonge, D., and B. Campbell, "OpenID Connect Client Initiated Backchannel Authentication Flow - Core 1.0", January 2019, <https://openid.net/specs/openid-client-initiated-backchannel-authentication-core-1_0.html>.
- [RFC4517] Legg, S., Ed., "Lightweight Directory Access Protocol (LDAP): Syntaxes and Matching Rules", RFC 4517, DOI 10.17487/RFC4517, June 2006, <<https://www.rfc-editor.org/info/rfc4517>>.
- [RFC5952] Kawamura, S. and M. Kawashima, "A Recommendation for IPv6 Address Text Representation", RFC 5952, DOI 10.17487/RFC5952, August 2010, <<https://www.rfc-editor.org/info/rfc5952>>.
- [RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <<https://www.rfc-editor.org/info/rfc6066>>.

- [RFC7009] Lodderstedt, T., Ed., Dronia, S., and M. Scurtescu, "OAuth 2.0 Token Revocation", RFC 7009, DOI 10.17487/RFC7009, August 2013, <<https://www.rfc-editor.org/info/rfc7009>>.
- [RFC7518] Jones, M., "JSON Web Algorithms (JWA)", RFC 7518, DOI 10.17487/RFC7518, May 2015, <<https://www.rfc-editor.org/info/rfc7518>>.

Appendix A. Example "cnf" Claim, Certificate and JWK

For reference, an "x5t#S256" value and the X.509 Certificate from which it was calculated are provided in the following examples, Figure 5 and Figure 6 respectively. A JWK representation of the certificate's public key along with the "x5c" member is also provided in Figure 7.

```
"cnf": {"x5t#S256": "A4DtL2JmUMhAsvJj5tKyn64SqzmuXbMrJa0n761y5v0"}
```

Figure 5: x5t#S256 Confirmation Claim

```
-----BEGIN CERTIFICATE-----
MIIBBjCBRAIBAJAKBggqhkJOPQQDAjAPMQ0wCwYDVQQDDARTdGxzMB4XDTE4MTAx
ODEyMzcwOVoxDTIyMDUwMjEyMzcwOVowDzENMAsgAlUEAwEebXRsczBZMBMGByqG
SM49AgEGCCqGSM49AwEHA0IABNcnxwqV6hY8QnhxxzFQ03C7HKW9OylMbnQZjjJ
/Au08/coZwxS7LFA4vOLS9WuneIXhbGGWvsDSb0tH6IxLm8wCgYIKoZIZj0EAWID
SQAwwRgIhAP0RC1E+vwJD/D1AGHGzuri+hlV/PpQEKTWUveORWz83AiEA5x2eXZOV
bUlJSGQgjd5vaUaK1LR50Q2DmFfQj1L+SY=
-----END CERTIFICATE-----
```

Figure 6: PEM Encoded Self-Signed Certificate

```
{
  "kty": "EC",
  "x": "1yfLHCpXqFjxCeHHMVDtCLscpb07KUxudBmOMn8C7Q",
  "y": "8_coZwxS7LFA4vOLS9WuneIXhbGGWvsDSb0tH6IxLm8",
  "crv": "P-256",
  "x5c": [
    "MIIBBjCBRAIBAJAKBggqhkJOPQQDAjAPMQ0wCwYDVQQDDARTdGxzMB4XDTE4MTAx
    xODEyMzcwOVoxDTIyMDUwMjEyMzcwOVowDzENMAsgAlUEAwEebXRsczBZMBMGBy
    qGSM49AgEGCCqGSM49AwEHA0IABNcnxwqV6hY8QnhxxzFQ03C7HKW9OylMbnQZ
    jjJ/Au08/coZwxS7LFA4vOLS9WuneIXhbGGWvsDSb0tH6IxLm8wCgYIKoZIZj0E
    AWIDSQAwwRgIhAP0RC1E+vwJD/D1AGHGzuri+hlV/PpQEKTWUveORWz83AiEA5x2
    eXZOVbUlJSGQgjd5vaUaK1LR50Q2DmFfQj1L+SY="
  ]
}
```

Figure 7: JSON Web Key

Appendix B. Relationship to Token Binding

OAuth 2.0 Token Binding [I-D.ietf-oauth-token-binding] enables the application of Token Binding to the various artifacts and tokens employed throughout OAuth. That includes binding of an access token to a Token Binding key, which bears some similarities in motivation and design to the mutual-TLS client certificate-bound access tokens defined in this document. Both documents define what is often called a proof-of-possession security mechanism for access tokens, whereby a client must demonstrate possession of cryptographic keying material when accessing a protected resource. The details differ somewhat between the two documents but both have the authorization server bind the access token that it issues to an asymmetric key pair held by the client. The client then proves possession of the private key from that pair with respect to the TLS connection over which the protected resource is accessed.

Token Binding uses bare keys that are generated on the client, which avoids many of the difficulties of creating, distributing, and managing certificates used in this specification. However, at the time of writing, Token Binding is fairly new and there is relatively little support for it in available application development platforms and tooling. Until better support for the underlying core Token Binding specifications exists, practical implementations of OAuth 2.0 Token Binding are infeasible. Mutual TLS, on the other hand, has been around for some time and enjoys widespread support in web servers and development platforms. As a consequence, OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens can be built and deployed now using existing platforms and tools. In the future, the two specifications are likely to be deployed in parallel for solving similar problems in different environments. Authorization servers may even support both specifications simultaneously using different proof-of-possession mechanisms for tokens issued to different clients.

Appendix C. Acknowledgements

Scott "not Tomlinson" Tomilson and Matt Peterson were involved in design and development work on a mutual-TLS OAuth client authentication implementation, which predates this document. Experience and learning from that work informed some of the content of this document.

This specification was developed within the OAuth Working Group under the chairmanship of Hannes Tschofenig and Rifaat Shekh-Yusef with Eric Rescorla, Benjamin Kaduk, and Roman Danyliw serving as Security Area Directors. Additionally, the following individuals contributed ideas, feedback, and wording that helped shape this specification:

Vittorio Bertocci, Sergey Beryozkin, Ralph Bragg, Sophie Bremer, Roman Danyliw, Vladimir Dzhuvinov, Samuel Erdtman, Evan Gilman, Leif Johansson, Michael Jones, Phil Hunt, Benjamin Kaduk, Takahiko Kawasaki, Sean Leonard, Kepeng Li, Neil Madden, James Manger, Jim Manico, Nov Matake, Sascha Preibisch, Eric Rescorla, Justin Richer, Vincent Roca, Filip Skokan, Dave Tonge, and Hannes Tschofenig.

Appendix D. Document(s) History

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

draft-ietf-oauth-mtls-17

- o Updates from IESG ballot position comments.

draft-ietf-oauth-mtls-16

- o Editorial updates from last call review.

draft-ietf-oauth-mtls-15

- o Editorial updates from second AD review.

draft-ietf-oauth-mtls-14

- o Editorial clarifications around there being only a single subject registered/configured per client for the `tls_client_auth` method.
- o Add a brief explanation about how, with `tls_client_auth` and `self_signed_tls_client_auth`, refresh tokens are certificate-bound indirectly via the client authentication.
- o Add mention of refresh tokens in the abstract.

draft-ietf-oauth-mtls-13

- o Add an abstract protocol flow and diagram to serve as an overview of OAuth in general and baseline to describe the various ways in which the mechanisms defined herein are intended to be used.
- o A little bit less of that German influence.
- o Rework the TLS references a bit and, in the Terminology section, clean up the description of what messages are sent and verified in the handshake to do 'mutual TLS'.
- o Move the explanation about "cnf" introspection registration into the IANA Considerations.
- o Add CIBA as an informational reference and additional example of an OAuth extension that defines an endpoint that utilizes client authentication.
- o Shorten a few of the section titles.

- o Add new client metadata values to allow for the use of a SAN in the PKI MTLS client authentication method.
- o Add privacy considerations attempting to discuss the implications of the client cert being sent in the clear in TLS 1.2.
- o Changed the 'Certificate Bound Access Tokens Without Client Authentication' section to 'Public Clients and Certificate-Bound Tokens' and moved it up to be a top level section while adding discussion of binding refresh tokens for public clients.
- o Reword/restructure the main PKI method section somewhat to (hopefully) improve readability.
- o Reword/restructure the Self-Signed method section a bit to (hopefully) make it more comprehensible.
- o Reword the AS and RS Implementation Considerations somewhat to (hopefully) improve readability.
- o Clarify that the protected resource obtains the client certificate used for mutual TLS from its TLS implementation layer.
- o Add Security Considerations section about the certificate thumbprint binding that includes the hash algorithm agility recommendation.
- o Add an "mtls_endpoint_aliases" AS metadata parameter that is a JSON object containing alternative authorization server endpoints, which a client intending to do mutual TLS will use in preference to the conventional endpoints.
- o Minor editorial updates.

draft-ietf-oauth-mtls-12

- o Add an example certificate, JWK, and confirmation method claim.
- o Minor editorial updates based on implementer feedback.
- o Additional Acknowledgements.

draft-ietf-oauth-mtls-11

- o Editorial updates.
- o Mention/reference TLS 1.3 RFC8446 in the TLS Versions and Best Practices section.

draft-ietf-oauth-mtls-10

- o Update draft-ietf-oauth-discovery reference to RFC8414

draft-ietf-oauth-mtls-09

- o Change "single certificates" to "self-signed certificates" in the Abstract

draft-ietf-oauth-mtls-08

- o Incorporate clarifications and editorial improvements from Justin Richer's WGLC review
- o Drop the use of the "sender constrained" terminology per WGLC feedback from Neil Madden (including changing the metadata parameters from `mutual_tls_sender_constrained_access_tokens` to `tls_client_certificate_bound_access_tokens`)
- o Add a new security considerations section on X.509 parsing and validation per WGLC feedback from Neil Madden and Benjamin Kaduk
- o Note that a server can terminate TLS at a load balancer, reverse proxy, etc. but how the client certificate metadata is securely communicated to the backend is out of scope per WGLC feedback
- o Note that revocation checking is at the discretion of the AS per WGLC feedback
- o Editorial updates and clarifications
- o Update draft-ietf-oauth-discovery reference to -10 and draft-ietf-oauth-token-binding to -06
- o Add folks involved in WGLC feedback to the acknowledgements list

draft-ietf-oauth-mtls-07

- o Update to use the boilerplate from RFC 8174

draft-ietf-oauth-mtls-06

- o Add an appendix section describing the relationship of this document to OAuth Token Binding as requested during the Singapore meeting <https://datatracker.ietf.org/doc/minutes-100-oauth/>
- o Add an explicit note that the implicit flow is not supported for obtaining certificate bound access tokens as discussed at the Singapore meeting <https://datatracker.ietf.org/doc/minutes-100-oauth/>
- o Add/incorporate text to the Security Considerations on Certificate Spoofing as suggested https://mailarchive.ietf.org/arch/msg/oauth/V26070X-60tbVSeUz_7W2k94vCo
- o Changed the title to be more descriptive
- o Move the Security Considerations section to before the IANA Considerations
- o Elaborated on certificate-bound access tokens a bit more in the Abstract
- o Update draft-ietf-oauth-discovery reference to -08

draft-ietf-oauth-mtls-05

- o Editorial fixes

draft-ietf-oauth-mtls-04

- o Change the name of the 'Public Key method' to the more accurate 'Self-Signed Certificate method' and also change the associated authentication method metadata value to "self_signed_tls_client_auth".
- o Removed the "tls_client_auth_root_dn" client metadata field as discussed in <https://mailarchive.ietf.org/arch/msg/oauth/swDV2y0be6o8czGKQileJV-g8qc>
- o Update draft-ietf-oauth-discovery reference to -07
- o Clarify that MTLS client authentication isn't exclusive to the token endpoint and can be used with other endpoints, e.g. RFC 7009 revocation and 7662 introspection, that utilize client authentication as discussed in <https://mailarchive.ietf.org/arch/msg/oauth/bZ6mft0G7D3ccebhOxnEYUv4puI>
- o Reorganize the document somewhat in an attempt to more clearly make a distinction between mTLS client authentication and certificate-bound access tokens as well as a more clear delineation between the two (PKI/Public key) methods for client authentication
- o Editorial fixes and clarifications

draft-ietf-oauth-mtls-03

- o Introduced metadata and client registration parameter to publish and request support for mutual TLS sender constrained access tokens
- o Added description of two methods of binding the cert and client, PKI and Public Key.
- o Indicated that the "tls_client_auth" authentication method is for the PKI method and introduced "pub_key_tls_client_auth" for the Public Key method
- o Added implementation considerations, mainly regarding TLS stack configuration and trust chain validation, as well as how to do binding of access tokens to a TLS client certificate for public clients, and considerations around certificate-bound access tokens
- o Added new section to security considerations on cert spoofing
- o Add text suggesting that a new cnf member be defined in the future, if hash function(s) other than SHA-256 need to be used for certificate thumbprints

draft-ietf-oauth-mtls-02

- o Fixed editorial issue <https://mailarchive.ietf.org/arch/msg/oauth/U46UMeh8XIOQnvXY9pHFq1MKPns>
- o Changed the title (hopefully "Mutual TLS Profile for OAuth 2.0" is better than "Mutual TLS Profiles for OAuth Clients").

draft-ietf-oauth-mtls-01

- o Added more explicit details of using RFC 7662 token introspection with mutual TLS sender constrained access tokens.
- o Added an IANA OAuth Token Introspection Response Registration request for "cnf".
- o Specify that `tls_client_auth_subject_dn` and `tls_client_auth_root_dn` are RFC 4514 String Representation of Distinguished Names.
- o Changed `tls_client_auth_issuer_dn` to `tls_client_auth_root_dn`.
- o Changed the text in the Section 3 to not be specific about using a hash of the cert.
- o Changed the abbreviated title to 'OAuth Mutual TLS' (previously was the acronym MTLSPOC).

draft-ietf-oauth-mtls-00

- o Created the initial working group version from draft-campbell-oauth-mtls

draft-campbell-oauth-mtls-01

- o Fix some typos.
- o Add to the acknowledgements list.

draft-campbell-oauth-mtls-00

- o Add a Mutual TLS sender constrained protected resource access method and a `x5t#S256 cnf` method for JWT access tokens (concepts taken in part from draft-sakimura-oauth-jpop-04).
- o Fixed `"token_endpoint_auth_methods_supported"` to `"token_endpoint_auth_method"` for client metadata.
- o Add `"tls_client_auth_subject_dn"` and `"tls_client_auth_issuer_dn"` client metadata parameters and mention using `"jwks_uri"` or `"jwks"`.
- o Say that the authentication method is determined by client policy regardless of whether the client was dynamically registered or statically configured.
- o Expand acknowledgements to those that participated in discussions around draft-campbell-oauth-tls-client-auth-00
- o Add Nat Sakimura and Torsten Lodderstedt to the author list.

draft-campbell-oauth-tls-client-auth-00

- o Initial draft.

Authors' Addresses

Brian Campbell
Ping Identity

Email: brian.d.campbell@gmail.com

John Bradley
Yubico

Email: ve7jtb@ve7jtb.com
URI: <http://www.thread-safe.com/>

Nat Sakimura
Nomura Research Institute

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

Torsten Lodderstedt
YES.com AG

Email: torsten@lodderstedt.net

Web Authorization Protocol
Internet-Draft
Intended status: Best Current Practice
Expires: 19 June 2022

T. Lodderstedt
yes.com
J. Bradley
Yubico
A. Labunets
Independent Researcher
D. Fett
yes.com
16 December 2021

OAuth 2.0 Security Best Current Practice
draft-ietf-oauth-security-topics-19

Abstract

This document describes best current security practice for OAuth 2.0. It updates and extends the OAuth 2.0 Security Threat Model to incorporate practical experiences gathered since OAuth 2.0 was published and covers new threats relevant due to the broader application of OAuth 2.0.

Status of This Memo

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

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

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

This Internet-Draft will expire on 19 June 2022.

Copyright Notice

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

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

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

Table of Contents

1. Introduction	3
1.1. Structure	4
1.2. Conventions and Terminology	4
2. Recommendations	5
2.1. Protecting Redirect-Based Flows	5
2.1.1. Authorization Code Grant	6
2.1.2. Implicit Grant	7
2.2. Token Replay Prevention	7
2.2.1. Access Tokens	7
2.2.2. Refresh Tokens	7
2.3. Access Token Privilege Restriction	8
2.4. Resource Owner Password Credentials Grant	8
2.5. Client Authentication	9
2.6. Other Recommendations	9
3. The Updated OAuth 2.0 Attacker Model	10
4. Attacks and Mitigations	12
4.1. Insufficient Redirect URI Validation	12
4.1.1. Redirect URI Validation Attacks on Authorization Code Grant	12
4.1.2. Redirect URI Validation Attacks on Implicit Grant	14
4.1.3. Countermeasures	15
4.2. Credential Leakage via Referer Headers	16
4.2.1. Leakage from the OAuth Client	16
4.2.2. Leakage from the Authorization Server	17
4.2.3. Consequences	17
4.2.4. Countermeasures	17
4.3. Credential Leakage via Browser History	18
4.3.1. Authorization Code in Browser History	18
4.3.2. Access Token in Browser History	18
4.4. Mix-Up Attacks	19
4.4.1. Attack Description	19
4.4.2. Countermeasures	21
4.5. Authorization Code Injection	23
4.5.1. Attack Description	23
4.5.2. Discussion	24
4.5.3. Countermeasures	25
4.5.4. Limitations	26
4.6. Access Token Injection	27
4.6.1. Countermeasures	27
4.7. Cross Site Request Forgery	27
4.7.1. Countermeasures	27

4.8.	PKCE Downgrade Attack	28
4.8.1.	Attack Description	28
4.8.2.	Countermeasures	29
4.9.	Access Token Leakage at the Resource Server	30
4.9.1.	Access Token Phishing by Counterfeit Resource Server	30
4.9.2.	Compromised Resource Server	35
4.10.	Open Redirection	36
4.10.1.	Client as Open Redirector	36
4.10.2.	Authorization Server as Open Redirector	36
4.11.	307 Redirect	37
4.12.	TLS Terminating Reverse Proxies	38
4.13.	Refresh Token Protection	38
4.13.1.	Discussion	39
4.13.2.	Recommendations	39
4.14.	Client Impersonating Resource Owner	41
4.14.1.	Countermeasures	41
4.15.	Clickjacking	41
5.	Acknowledgements	42
6.	IANA Considerations	42
7.	Security Considerations	42
8.	Normative References	42
9.	Informative References	44
Appendix A.	Document History	48
Authors' Addresses	52

1. Introduction

Since its publication in [RFC6749] and [RFC6750], OAuth 2.0 ("OAuth" in the following) has gotten massive traction in the market and became the standard for API protection and the basis for federated login using OpenID Connect [OpenID]. While OAuth is used in a variety of scenarios and different kinds of deployments, the following challenges can be observed:

- * OAuth implementations are being attacked through known implementation weaknesses and anti-patterns. Although most of these threats are discussed in the OAuth 2.0 Threat Model and Security Considerations [RFC6819], continued exploitation demonstrates a need for more specific recommendations, easier to implement mitigations, and more defense in depth.
- * OAuth is being used in environments with higher security requirements than considered initially, such as Open Banking, eHealth, eGovernment, and Electronic Signatures. Those use cases call for stricter guidelines and additional protection.

- * OAuth is being used in much more dynamic setups than originally anticipated, creating new challenges with respect to security. Those challenges go beyond the original scope of [RFC6749], [RFC6750], and [RFC6819].

OAuth initially assumed a static relationship between client, authorization server and resource servers. The URLs of AS and RS were known to the client at deployment time and built an anchor for the trust relationship among those parties. The validation whether the client talks to a legitimate server was based on TLS server authentication (see [RFC6819], Section 4.5.4). With the increasing adoption of OAuth, this simple model dissolved and, in several scenarios, was replaced by a dynamic establishment of the relationship between clients on one side and the authorization and resource servers of a particular deployment on the other side. This way, the same client could be used to access services of different providers (in case of standard APIs, such as e-mail or OpenID Connect) or serve as a frontend to a particular tenant in a multi-tenancy environment. Extensions of OAuth, such as the OAuth 2.0 Dynamic Client Registration Protocol [RFC7591] and OAuth 2.0 Authorization Server Metadata [RFC8414] were developed in order to support the usage of OAuth in dynamic scenarios.

- * Technology has changed. For example, the way browsers treat fragments when redirecting requests has changed, and with it, the implicit grant's underlying security model.

This document provides updated security recommendations to address these challenges. It does not supplant the security advice given in [RFC6749], [RFC6750], and [RFC6819], but complements those documents.

1.1. Structure

The remainder of this document is organized as follows: The next section summarizes the most important recommendations of the OAuth working group for every OAuth implementor. Afterwards, the updated the OAuth attacker model is presented. Subsequently, a detailed analysis of the threats and implementation issues that can be found in the wild today is given along with a discussion of potential countermeasures.

1.2. Conventions and Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

This specification uses the terms "access token", "authorization endpoint", "authorization grant", "authorization server", "client", "client identifier" (client ID), "protected resource", "refresh token", "resource owner", "resource server", and "token endpoint" defined by OAuth 2.0 [RFC6749].

2. Recommendations

This section describes the set of security mechanisms the OAuth working group recommends to OAuth implementers.

2.1. Protecting Redirect-Based Flows

When comparing client redirect URIs against pre-registered URIs, authorization servers MUST utilize exact string matching except for port numbers in localhost redirection URIs of native apps, see Section 4.1.3. This measure contributes to the prevention of leakage of authorization codes and access tokens (see Section 4.1). It can also help to detect mix-up attacks (see Section 4.4).

Clients and AS MUST NOT expose URLs that forward the user's browser to arbitrary URIs obtained from a query parameter ("open redirector"). Open redirectors can enable exfiltration of authorization codes and access tokens, see Section 4.10.1.

Clients MUST prevent Cross-Site Request Forgery (CSRF). In this context, CSRF refers to requests to the redirection endpoint that do not originate at the authorization server, but a malicious third party (see Section 4.4.1.8. of [RFC6819] for details). Clients that have ensured that the authorization server supports PKCE [RFC7636] MAY rely the CSRF protection provided by PKCE. In OpenID Connect flows, the nonce parameter provides CSRF protection. Otherwise, one-time use CSRF tokens carried in the state parameter that are securely bound to the user agent MUST be used for CSRF protection (see Section 4.7.1).

When an OAuth client can interact with more than one authorization server, a defense against mix-up attacks (see Section 4.4) is REQUIRED. To this end, clients SHOULD

- * use the iss parameter as a countermeasure according to [I-D.ietf-oauth-iss-auth-resp], or
- * use an alternative countermeasure based on an iss value in the authorization response (such as the iss Claim in the ID Token in [OpenID] or in [JARM] responses), processing it as described in [I-D.ietf-oauth-iss-auth-resp].

In the absence of these options, clients MAY instead use distinct redirect URIs to identify authorization endpoints and token endpoints, as described in Section 4.4.2.

An AS that redirects a request potentially containing user credentials MUST avoid forwarding these user credentials accidentally (see Section 4.11 for details).

2.1.1. Authorization Code Grant

Clients MUST prevent injection (replay) of authorization codes into the authorization response by attackers. Public clients MUST use PKCE [RFC7636] to this end. For confidential clients, the use of PKCE [RFC7636] is RECOMMENDED. With additional precautions, described in Section 4.5.3.2, confidential clients MAY use the OpenID Connect nonce parameter and the respective Claim in the ID Token [OpenID] instead. In any case, the PKCE challenge or OpenID Connect nonce MUST be transaction-specific and securely bound to the client and the user agent in which the transaction was started.

Note: Although PKCE was designed as a mechanism to protect native apps, this advice applies to all kinds of OAuth clients, including web applications.

When using PKCE, clients SHOULD use PKCE code challenge methods that do not expose the PKCE verifier in the authorization request. Otherwise, attackers that can read the authorization request (cf. Attacker A4 in Section 3) can break the security provided by PKCE. Currently, S256 is the only such method.

Authorization servers MUST support PKCE [RFC7636].

Authorization servers MUST provide a way to detect their support for PKCE. It is RECOMMENDED for AS to publish the element `code_challenge_methods_supported` in their AS metadata ([RFC8414]) containing the supported PKCE challenge methods (which can be used by the client to detect PKCE support). AS MAY instead provide a deployment-specific way to ensure or determine PKCE support by the AS.

Authorization servers MUST mitigate PKCE Downgrade Attacks by ensuring that a token request containing a `code_verifier` parameter is accepted only if a `code_challenge` parameter was present in the authorization request, see Section 4.8.2 for details.

2.1.2. Implicit Grant

The implicit grant (response type "token") and other response types causing the authorization server to issue access tokens in the authorization response are vulnerable to access token leakage and access token replay as described in Section 4.1, Section 4.2, Section 4.3, and Section 4.6.

Moreover, no viable mechanism exists to cryptographically bind access tokens issued in the authorization response to a certain client as it is recommended in Section 2.2. This makes replay detection for such access tokens at resource servers impossible.

In order to avoid these issues, clients SHOULD NOT use the implicit grant (response type "token") or other response types issuing access tokens in the authorization response, unless access token injection in the authorization response is prevented and the aforementioned token leakage vectors are mitigated.

Clients SHOULD instead use the response type "code" (aka authorization code grant type) as specified in Section 2.1.1 or any other response type that causes the authorization server to issue access tokens in the token response, such as the "code id_token" response type. This allows the authorization server to detect replay attempts by attackers and generally reduces the attack surface since access tokens are not exposed in URLs. It also allows the authorization server to sender-constrain the issued tokens (see next section).

2.2. Token Replay Prevention

2.2.1. Access Tokens

A sender-constrained access token scopes the applicability of an access token to a certain sender. This sender is obliged to demonstrate knowledge of a certain secret as prerequisite for the acceptance of that token at the recipient (e.g., a resource server).

Authorization and resource servers SHOULD use mechanisms for sender-constraining access tokens to prevent token replay, such as Mutual TLS for OAuth 2.0 [RFC8705] (see Section 4.9.1.1.2).

2.2.2. Refresh Tokens

Refresh tokens for public clients MUST be sender-constrained or use refresh token rotation as described in Section 4.13. [RFC6749] already mandates that refresh tokens for confidential clients can only be used by the client for which they were issued.

2.3. Access Token Privilege Restriction

The privileges associated with an access token SHOULD be restricted to the minimum required for the particular application or use case. This prevents clients from exceeding the privileges authorized by the resource owner. It also prevents users from exceeding their privileges authorized by the respective security policy. Privilege restrictions also help to reduce the impact of access token leakage.

In particular, access tokens SHOULD be restricted to certain resource servers (audience restriction), preferably to a single resource server. To put this into effect, the authorization server associates the access token with certain resource servers and every resource server is obliged to verify, for every request, whether the access token sent with that request was meant to be used for that particular resource server. If not, the resource server MUST refuse to serve the respective request. Clients and authorization servers MAY utilize the parameters scope or resource as specified in [RFC6749] and [I-D.ietf-oauth-resource-indicators], respectively, to determine the resource server they want to access.

Additionally, access tokens SHOULD be restricted to certain resources and actions on resource servers or resources. To put this into effect, the authorization server associates the access token with the respective resource and actions and every resource server is obliged to verify, for every request, whether the access token sent with that request was meant to be used for that particular action on the particular resource. If not, the resource server must refuse to serve the respective request. Clients and authorization servers MAY utilize the parameter scope as specified in [RFC6749] and authorization_details as specified in [I-D.ietf-oauth-rar] to determine those resources and/or actions.

2.4. Resource Owner Password Credentials Grant

The resource owner password credentials grant MUST NOT be used. This grant type insecurely exposes the credentials of the resource owner to the client. Even if the client is benign, this results in an increased attack surface (credentials can leak in more places than just the AS) and users are trained to enter their credentials in places other than the AS.

Furthermore, adapting the resource owner password credentials grant to two-factor authentication, authentication with cryptographic credentials (cf. WebCrypto [webcrypto], WebAuthn [webauthn]), and authentication processes that require multiple steps can be hard or impossible.

2.5. Client Authentication

Authorization servers SHOULD use client authentication if possible.

It is RECOMMENDED to use asymmetric (public-key based) methods for client authentication such as mTLS [RFC8705] or private_key_jwt [OpenID]. When asymmetric methods for client authentication are used, authorization servers do not need to store sensitive symmetric keys, making these methods more robust against a number of attacks.

2.6. Other Recommendations

The use of OAuth Metadata [RFC8414] can help to improve the security of OAuth deployments:

- * It ensures that security features and other new OAuth features can be enabled automatically by compliant software libraries.
- * It reduces chances for misconfigurations, for example misconfigured endpoint URLs (that might belong to an attacker) or misconfigured security features.
- * It can help to facilitate rotation of cryptographic keys and to ensure cryptographic agility.

It is therefore RECOMMENDED that AS publish OAuth metadata according to [RFC8414] and that clients make use of this metadata to configure themselves when available.

Authorization servers SHOULD NOT allow clients to influence their client_id or sub value or any other Claim if that can cause confusion with a genuine resource owner (see Section 4.14).

It is RECOMMENDED to use end-to-end TLS. If TLS traffic needs to be terminated at an intermediary, refer to Section 4.12 for further security advice.

Authorization responses MUST NOT be transmitted over unencrypted network connections. To this end, AS MUST NOT allow redirect URIs that use the http scheme except for native clients that use Loopback Interface Redirection as described in [RFC8252], Section 7.3.

3. The Updated OAuth 2.0 Attacker Model

In [RFC6819], an attacker model is laid out that describes the capabilities of attackers against which OAuth deployments must be protected. In the following, this attacker model is updated to account for the potentially dynamic relationships involving multiple parties (as described in Section 1), to include new types of attackers and to define the attacker model more clearly.

OAuth MUST ensure that the authorization of the resource owner (RO) (with a user agent) at the authorization server (AS) and the subsequent usage of the access token at the resource server (RS) is protected at least against the following attackers:

- * (A1) Web Attackers that can set up and operate an arbitrary number of network endpoints including browsers and servers (except for the concrete RO, AS, and RS). Web attackers may set up web sites that are visited by the RO, operate their own user agents, and participate in the protocol.

Web attackers may, in particular, operate OAuth clients that are registered at AS, and operate their own authorization and resource servers that can be used (in parallel) by the RO and other resource owners.

It must also be assumed that web attackers can lure the user to open arbitrary attacker-chosen URIs at any time. In practice, this can be achieved in many ways, for example, by injecting malicious advertisements into advertisement networks, or by sending legit-looking emails.

Web attackers can use their own user credentials to create new messages as well as any secrets they learned previously. For example, if a web attacker learns an authorization code of a user through a misconfigured redirect URI, the web attacker can then try to redeem that code for an access token.

They cannot, however, read or manipulate messages that are not targeted towards them (e.g., sent to a URL controlled by a non-attacker controlled AS).

- * (A2) Network Attackers that additionally have full control over the network over which protocol participants communicate. They can eavesdrop on, manipulate, and spoof messages, except when these are properly protected by cryptographic methods (e.g., TLS). Network attackers can also block arbitrary messages.

While an example for a web attacker would be a customer of an internet service provider, network attackers could be the internet service provider itself, an attacker in a public (wifi) network using ARP spoofing, or a state-sponsored attacker with access to internet exchange points, for instance.

These attackers conform to the attacker model that was used in formal analysis efforts for OAuth [arXiv.1601.01229]. This is a minimal attacker model. Implementers MUST take into account all possible types of attackers in the environment in which their OAuth implementations are expected to run. Previous attacks on OAuth have shown that OAuth deployments SHOULD in particular consider the following, stronger attackers in addition to those listed above:

- * (A3) Attackers that can read, but not modify, the contents of the authorization response (i.e., the authorization response can leak to an attacker).

Examples for such attacks include open redirector attacks, problems existing on mobile operating systems (where different apps can register themselves on the same URI), mix-up attacks (see Section 4.4), where the client is tricked into sending credentials to a attacker-controlled AS, and the fact that URLs are often stored/logged by browsers (history), proxy servers, and operating systems.

- * (A4) Attackers that can read, but not modify, the contents of the authorization request (i.e., the authorization request can leak, in the same manner as above, to an attacker).
- * (A5) Attackers that can acquire an access token issued by AS. For example, a resource server can be compromised by an attacker, an access token may be sent to an attacker-controlled resource server due to a misconfiguration, or an RO is social-engineered into using a attacker-controlled RS. See also Section 4.9.2.

(A3), (A4) and (A5) typically occur together with either (A1) or (A2). Attackers can collaborate to reach a common goal.

Note that in this attacker model, an attacker (see A1) can be a RO or act as one. For example, an attacker can use his own browser to replay tokens or authorization codes obtained by any of the attacks described above at the client or RS.

This document focusses on threats resulting from these attackers. Attacks in an even stronger attacker model are discussed, for example, in [arXiv.1901.11520].

4. Attacks and Mitigations

This section gives a detailed description of attacks on OAuth implementations, along with potential countermeasures. Attacks and mitigations already covered in [RFC6819] are not listed here, except where new recommendations are made.

4.1. Insufficient Redirect URI Validation

Some authorization servers allow clients to register redirect URI patterns instead of complete redirect URIs. The authorization servers then match the redirect URI parameter value at the authorization endpoint against the registered patterns at runtime. This approach allows clients to encode transaction state into additional redirect URI parameters or to register a single pattern for multiple redirect URIs.

This approach turned out to be more complex to implement and more error prone to manage than exact redirect URI matching. Several successful attacks exploiting flaws in the pattern matching implementation or concrete configurations have been observed in the wild. Insufficient validation of the redirect URI effectively breaks client identification or authentication (depending on grant and client type) and allows the attacker to obtain an authorization code or access token, either

- * by directly sending the user agent to a URI under the attackers control, or
- * by exposing the OAuth credentials to an attacker by utilizing an open redirector at the client in conjunction with the way user agents handle URL fragments.

These attacks are shown in detail in the following subsections.

4.1.1. Redirect URI Validation Attacks on Authorization Code Grant

For a client using the grant type code, an attack may work as follows:

Assume the redirect URL pattern `https://*.somesite.example/*` is registered for the client with the client ID `s6BhdRkqt3`. The intention is to allow any subdomain of `somesite.example` to be a valid redirect URI for the client, for example `https://appl.somesite.example/redirect`. A naive implementation on the authorization server, however, might interpret the wildcard `*` as "any character" and not "any character valid for a domain name". The authorization server, therefore, might permit

`https://attacker.example/.somesite.example` as a redirect URI, although `attacker.example` is a different domain potentially controlled by a malicious party.

The attack can then be conducted as follows:

First, the attacker needs to trick the user into opening a tampered URL in his browser that launches a page under the attacker's control, say `https://www.evil.example` (see Attacker A1.)

This URL initiates the following authorization request with the client ID of a legitimate client to the authorization endpoint (line breaks for display only):

```
GET /authorize?response_type=code&client_id=s6BhdRkqt3&state=9ad67f13
    &redirect_uri=https%3A%2F%2Fattacker.example%2F.somesite.example
    HTTP/1.1
Host: server.somesite.example
```

The authorization server validates the redirect URI and compares it to the registered redirect URL patterns for the client `s6BhdRkqt3`. The authorization request is processed and presented to the user.

If the user does not see the redirect URI or does not recognize the attack, the code is issued and immediately sent to the attacker's domain. If an automatic approval of the authorization is enabled (which is not recommended for public clients according to [RFC6749]), the attack can be performed even without user interaction.

If the attacker impersonated a public client, the attacker can exchange the code for tokens at the respective token endpoint.

This attack will not work as easily for confidential clients, since the code exchange requires authentication with the legitimate client's secret. The attacker can, however, use the legitimate confidential client to redeem the code by performing an authorization code injection attack, see Section 4.5.

Note: Vulnerabilities of this kind can also exist if the authorization server handles wildcards properly. For example, assume that the client registers the redirect URL pattern `https://*.somesite.example/*` and the authorization server interprets this as "allow redirect URIs pointing to any host residing in the domain `somesite.example`". If an attacker manages to establish a host or subdomain in `somesite.example`, he can impersonate the legitimate client. This could be caused, for example, by a subdomain takeover attack [subdomaintakeover], where an outdated CNAME record (say, `external-service.somesite.example`) points to an external DNS name

that does no longer exist (say, customer-abc.service.example) and can be taken over by an attacker (e.g., by registering as customer-abc with the external service).

4.1.2. Redirect URI Validation Attacks on Implicit Grant

The attack described above works for the implicit grant as well. If the attacker is able to send the authorization response to a URI under his control, he will directly get access to the fragment carrying the access token.

Additionally, implicit clients can be subject to a further kind of attack. It utilizes the fact that user agents re-attach fragments to the destination URL of a redirect if the location header does not contain a fragment (see [RFC7231], Section 9.5). The attack described here combines this behavior with the client as an open redirector (see Section 4.10.1) in order to get access to access tokens. This allows circumvention even of very narrow redirect URI patterns, but not strict URL matching.

Assume the registered URL pattern for client s6BhdRkqt3 is `https://client.somesite.example/cb?*`, i.e., any parameter is allowed for redirects to `https://client.somesite.example/cb`. Unfortunately, the client exposes an open redirector. This endpoint supports a parameter `redirect_to` which takes a target URL and will send the browser to this URL using an HTTP Location header `redirect 303`.

The attack can now be conducted as follows:

First, and as above, the attacker needs to trick the user into opening a tampered URL in his browser that launches a page under the attacker's control, say `https://www.evil.example`.

Afterwards, the website initiates an authorization request that is very similar to the one in the attack on the code flow. Different to above, it utilizes the open redirector by encoding `redirect_to=https://attacker.example` into the parameters of the redirect URI and it uses the response type "token" (line breaks for display only):

```
GET /authorize?response_type=token&state=9ad67f13
    &client_id=s6BhdRkqt3
    &redirect_uri=https%3A%2F%2Fclient.somesite.example
    %2Fcb%26redirect_to%253Dhttps%253A%252F
    %252Fattacker.example%252F HTTP/1.1
Host: server.somesite.example
```

Now, since the redirect URI matches the registered pattern, the authorization server permits the request and sends the resulting access token in a 303 redirect (some response parameters omitted for readability):

HTTP/1.1 303 See Other

Location: `https://client.somesite.example/cb?
redirect_to%3Dhttps%3A%2F%2Fattacker.example%2Fcb
#access_token=2YotnFZFEjrlzCsicMWpAA&...`

At `example.com`, the request arrives at the open redirector. The endpoint will read the redirect parameter and will issue an HTTP 303 Location header redirect to the URL `https://attacker.example/`.

HTTP/1.1 303 See Other

Location: `https://attacker.example/`

Since the redirector at `client.somesite.example` does not include a fragment in the Location header, the user agent will re-attach the original fragment `#access_token=2YotnFZFEjrlzCsicMWpAA&... to the URL and will navigate to the following URL:`

`https://attacker.example/#access_token=2YotnFZFEjrlz...`

The attacker's page at `attacker.example` can now access the fragment and obtain the access token.

4.1.3. Countermeasures

The complexity of implementing and managing pattern matching correctly obviously causes security issues. This document therefore advises to simplify the required logic and configuration by using exact redirect URI matching. This means the authorization server MUST compare the two URIs using simple string comparison as defined in [RFC3986], Section 6.2.1. The only exception are native apps using a localhost URI: In this case, the AS MUST allow variable port numbers as described in [RFC8252], Section 7.3.

Additional recommendations:

- * Servers on which callbacks are hosted MUST NOT expose open redirectors (see Section 4.10).

- * Browsers reattach URL fragments to Location redirection URLs only if the URL in the Location header does not already contain a fragment. Therefore, servers MAY prevent browsers from reattaching fragments to redirection URLs by attaching an arbitrary fragment identifier, for example #_, to URLs in Location headers.
- * Clients SHOULD use the authorization code response type instead of response types causing access token issuance at the authorization endpoint. This offers countermeasures against reuse of leaked credentials through the exchange process with the authorization server and token replay through sender-constraining of the access tokens.

If the origin and integrity of the authorization request containing the redirect URI can be verified, for example when using [I-D.ietf-oauth-jwsreq] or [I-D.ietf-oauth-par] with client authentication, the authorization server MAY trust the redirect URI without further checks.

4.2. Credential Leakage via Referer Headers

The contents of the authorization request URI or the authorization response URI can unintentionally be disclosed to attackers through the Referer HTTP header (see [RFC7231], Section 5.5.2), by leaking either from the AS's or the client's web site, respectively. Most importantly, authorization codes or state values can be disclosed in this way. Although specified otherwise in [RFC7231], Section 5.5.2, the same may happen to access tokens conveyed in URI fragments due to browser implementation issues as illustrated by Chromium Issue 168213 [bug.chromium].

4.2.1. Leakage from the OAuth Client

Leakage from the OAuth client requires that the client, as a result of a successful authorization request, renders a page that

- * contains links to other pages under the attacker's control and a user clicks on such a link, or
- * includes third-party content (advertisements in iframes, images, etc.), for example if the page contains user-generated content (blog).

As soon as the browser navigates to the attacker's page or loads the third-party content, the attacker receives the authorization response URL and can extract code or state (and potentially access token).

4.2.2. Leakage from the Authorization Server

In a similar way, an attacker can learn state from the authorization request if the authorization endpoint at the authorization server contains links or third-party content as above.

4.2.3. Consequences

An attacker that learns a valid code or access token through a Referer header can perform the attacks as described in Section 4.1.1, Section 4.5, and Section 4.6. If the attacker learns state, the CSRF protection achieved by using state is lost, resulting in CSRF attacks as described in [RFC6819], Section 4.4.1.8.

4.2.4. Countermeasures

The page rendered as a result of the OAuth authorization response and the authorization endpoint SHOULD NOT include third-party resources or links to external sites.

The following measures further reduce the chances of a successful attack:

- * Suppress the Referer header by applying an appropriate Referrer Policy [webappsec-referrer-policy] to the document (either as part of the "referrer" meta attribute or by setting a Referrer-Policy header). For example, the header Referrer-Policy: no-referrer in the response completely suppresses the Referer header in all requests originating from the resulting document.
- * Use authorization code instead of response types causing access token issuance from the authorization endpoint.
- * Bind authorization code to a confidential client or PKCE challenge. In this case, the attacker lacks the secret to request the code exchange.
- * As described in [RFC6749], Section 4.1.2, authorization codes MUST be invalidated by the AS after their first use at the token endpoint. For example, if an AS invalidated the code after the legitimate client redeemed it, the attacker would fail exchanging this code later.

This does not mitigate the attack if the attacker manages to exchange the code for a token before the legitimate client does so. Therefore, [RFC6749] further recommends that, when an attempt is made to redeem a code twice, the AS SHOULD revoke all tokens issued previously based on that code.

- * The state value SHOULD be invalidated by the client after its first use at the redirection endpoint. If this is implemented, and an attacker receives a token through the Referer header from the client's web site, the state was already used, invalidated by the client and cannot be used again by the attacker. (This does not help if the state leaks from the AS's web site, since then the state has not been used at the redirection endpoint at the client yet.)
- * Use the form post response mode instead of a redirect for the authorization response (see [oauth-v2-form-post-response-mode]).

4.3. Credential Leakage via Browser History

Authorization codes and access tokens can end up in the browser's history of visited URLs, enabling the attacks described in the following.

4.3.1. Authorization Code in Browser History

When a browser navigates to `client.example/redirection_endpoint?code=abcd` as a result of a redirect from a provider's authorization endpoint, the URL including the authorization code may end up in the browser's history. An attacker with access to the device could obtain the code and try to replay it.

Countermeasures:

- * Authorization code replay prevention as described in [RFC6819], Section 4.4.1.1, and Section 4.5.
- * Use form post response mode instead of redirect for the authorization response (see [oauth-v2-form-post-response-mode]).

4.3.2. Access Token in Browser History

An access token may end up in the browser history if a client or a web site that already has a token deliberately navigates to a page like `provider.com/get_user_profile?access_token=abcdef`. [RFC6750] discourages this practice and advises to transfer tokens via a header, but in practice web sites often pass access tokens in query parameters.

In case of the implicit grant, a URL like `client.example/redirection_endpoint#access_token=abcdef` may also end up in the browser history as a result of a redirect from a provider's authorization endpoint.

Countermeasures:

- * Clients MUST NOT pass access tokens in a URI query parameter in the way described in Section 2.3 of [RFC6750]. The authorization code grant or alternative OAuth response modes like the form post response mode [oauth-v2-form-post-response-mode] can be used to this end.

4.4. Mix-Up Attacks

Mix-up is an attack on scenarios where an OAuth client interacts with two or more authorization servers and at least one authorization server is under the control of the attacker. This can be the case, for example, if the attacker uses dynamic registration to register the client at his own authorization server or if an authorization server becomes compromised.

The goal of the attack is to obtain an authorization code or an access token for an uncompromised authorization server. This is achieved by tricking the client into sending those credentials to the compromised authorization server (the attacker) instead of using them at the respective endpoint of the uncompromised authorization/resource server.

4.4.1. Attack Description

The description here follows [arXiv.1601.01229], with variants of the attack outlined below.

Preconditions: For this variant of the attack to work, we assume that

- * the implicit or authorization code grant are used with multiple AS of which one is considered "honest" (H-AS) and one is operated by the attacker (A-AS), and
- * the client stores the AS chosen by the user in a session bound to the user's browser and uses the same redirection endpoint URI for each AS.

In the following, we assume that the client is registered with H-AS (URI: <https://honest.as.example>, client ID: 7ZGZldHQ) and with A-AS (URI: <https://attacker.example>, client ID: 666RVZJTA). URLs shown in the following example are shorted for presentation to only include parameters relevant for the attack.

Attack on the authorization code grant:

1. The user selects to start the grant using A-AS (e.g., by clicking on a button at the client's website).
2. The client stores in the user's session that the user selected "A-AS" and redirects the user to A-AS's authorization endpoint with a Location header containing the URL
`https://attacker.example/
authorize?response_type=code&client_id=666RVZJTA.`
3. When the user's browser navigates to the attacker's authorization endpoint, the attacker immediately redirects the browser to the authorization endpoint of H-AS. In the authorization request, the attacker replaces the client ID of the client at A-AS with the client's ID at H-AS. Therefore, the browser receives a redirection (303 See Other) with a Location header pointing to
`https://honest.as.example/
authorize?response_type=code&client_id=7ZGZldHQ`
4. The user authorizes the client to access her resources at H-AS. (Note that a vigilant user might at this point detect that she intended to use A-AS instead of H-AS. The first attack variant listed below avoids this.) H-AS issues a code and sends it (via the browser) back to the client.
5. Since the client still assumes that the code was issued by A-AS, it will try to redeem the code at A-AS's token endpoint.
6. The attacker therefore obtains code and can either exchange the code for an access token (for public clients) or perform an authorization code injection attack as described in Section 4.5.

Variants:

- * ***Mix-Up With Interception***: This variant works only if the attacker can intercept and manipulate the first request/response pair from a user's browser to the client (in which the user selects a certain AS and is then redirected by the client to that AS), as in Attacker A2. This capability can, for example, be the result of a man-in-the-middle attack on the user's connection to the client. In the attack, the user starts the flow with H-AS. The attacker intercepts this request and changes the user's selection to A-AS. The rest of the attack proceeds as in Steps 2 and following above.
- * ***Implicit Grant***: In the implicit grant, the attacker receives an access token instead of the code; the rest of the attack works as above.

- * ***Per-AS Redirect URIs***: If clients use different redirect URIs for different ASs, do not store the selected AS in the user's session, and ASs do not check the redirect URIs properly, attackers can mount an attack called "Cross-Social Network Request Forgery". These attacks have been observed in practice. Refer to [oauth_security_jcs_14] for details.
- * ***OpenID Connect***: There are variants that can be used to attack OpenID Connect. In these attacks, the attacker misuses features of the OpenID Connect Discovery [OpenIDDisc] mechanism or replays access tokens or ID Tokens to conduct a mix-up attack. The attacks are described in detail in [arXiv.1704.08539], Appendix A, and [arXiv.1508.04324v2], Section 6 ("Malicious Endpoints Attacks").

4.4.2. Countermeasures

When an OAuth client can only interact with one authorization server, a mix-up defense is not required. In scenarios where an OAuth client interacts with two or more authorization servers, however, clients **MUST** prevent mix-up attacks. Two different methods are discussed in the following.

For both defenses, clients **MUST** store, for each authorization request, the issuer they sent the authorization request to and bind this information to the user agent. The issuer serves, via the associated metadata, as an abstract identifier for the combination of the authorization endpoint and token endpoint that are to be used in the flow. If an issuer identifier is not available, for example, if neither OAuth metadata [RFC8414] nor OpenID Connect Discovery [OpenIDDisc] are used, a different unique identifier for this tuple or the tuple itself can be used instead. For brevity of presentation, such a deployment-specific identifier will be subsumed under the issuer (or issuer identifier) in the following.

Note: Just storing the authorization server URL is not sufficient to identify mix-up attacks. An attacker might declare an uncompromised AS's authorization endpoint URL as "his" AS URL, but declare a token endpoint under his own control.

4.4.2.1. Mix-Up Defense via Issuer Identification

This defense requires that the authorization server sends his issuer identifier in the authorization response to the client. When receiving the authorization response, the client **MUST** compare the received issuer identifier to the stored issuer identifier. If there is a mismatch, the client **MUST** abort the interaction.

There are different ways this issuer identifier can be transported to the client:

- * The issuer information can be transported, for example, via a separate response parameter `iss`, defined in [I-D.ietf-oauth-iss-auth-resp].
- * When OpenID Connect is used and an ID Token is returned in the authorization response, the client can evaluate the `iss` Claim in the ID Token.

In both cases, the `iss` value MUST be evaluated according to [I-D.ietf-oauth-iss-auth-resp].

While this defense may require deploying new OAuth features to transport the issuer information, it is a robust and relatively simple defense against mix-up.

4.4.2.2. Mix-Up Defense via Distinct Redirect URIs

For this defense, clients MUST use a distinct redirect URI for each issuer they interact with.

Clients MUST check that the authorization response was received from the correct issuer by comparing the distinct redirect URI for the issuer to the URI where the authorization response was received on. If there is a mismatch, the client MUST abort the flow.

While this defense builds upon existing OAuth functionality, it cannot be used in scenarios where clients only register once for the use of many different issuers (as in some open banking schemes) and due to the tight integration with the client registration, it is harder to deploy automatically.

Furthermore, an attacker might be able to circumvent the protection offered by this defense by registering a new client with the "honest" AS using the redirect URI that the client assigned to the attacker's AS. The attacker could then run the attack as described above, replacing the client ID with the client ID of his newly created client.

This defense SHOULD therefore only be used if other options are not available.

4.5. Authorization Code Injection

In an authorization code injection attack, the attacker attempts to inject a stolen authorization code into the attacker's own session with the client. The aim is to associate the attacker's session at the client with the victim's resources or identity.

This attack is useful if the attacker cannot exchange the authorization code for an access token himself. Examples include:

- * The code is bound to a particular confidential client and the attacker is unable to obtain the required client credentials to redeem the code himself.
- * The attacker wants to access certain functions in this particular client. As an example, the attacker wants to impersonate his victim in a certain app or on a certain web site.
- * The authorization or resource servers are limited to certain networks that the attacker is unable to access directly.

In the following attack description and discussion, we assume the presence of a web (A1) or network attacker (A2).

4.5.1. Attack Description

The attack works as follows:

1. The attacker obtains an authorization code by performing any of the attacks described above.
2. He starts a regular OAuth authorization process with the legitimate client from his device.
3. The attacker injects the stolen authorization code in the response of the authorization server to the legitimate client. Since this response is passing through the attacker's device, the attacker can use any tool that can intercept and manipulate the authorization response to this end. The attacker does not need to control the network.
4. The legitimate client sends the code to the authorization server's token endpoint, along with the client's client ID, client secret and actual redirect_uri.
5. The authorization server checks the client secret, whether the code was issued to the particular client, and whether the actual redirect URI matches the redirect_uri parameter (see [RFC6749]).

6. All checks succeed and the authorization server issues access and other tokens to the client. The attacker has now associated his session with the legitimate client with the victim's resources and/or identity.

4.5.2. Discussion

Obviously, the check in step (5.) will fail if the code was issued to another client ID, e.g., a client set up by the attacker. The check will also fail if the authorization code was already redeemed by the legitimate user and was one-time use only.

An attempt to inject a code obtained via a manipulated redirect URI should also be detected if the authorization server stored the complete redirect URI used in the authorization request and compares it with the `redirect_uri` parameter.

[RFC6749], Section 4.1.3, requires the AS to "... 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 that their values are identical.". In the attack scenario described above, the legitimate client would use the correct redirect URI it always uses for authorization requests. But this URI would not match the tampered redirect URI used by the attacker (otherwise, the redirect would not land at the attackers page). So the authorization server would detect the attack and refuse to exchange the code.

Note: This check could also detect attempts to inject an authorization code which had been obtained from another instance of the same client on another device, if certain conditions are fulfilled:

- * the redirect URI itself needs to contain a nonce or another kind of one-time use, secret data and
- * the client has bound this data to this particular instance of the client.

But this approach conflicts with the idea to enforce exact redirect URI matching at the authorization endpoint. Moreover, it has been observed that providers very often ignore the `redirect_uri` check requirement at this stage, maybe because it doesn't seem to be security-critical from reading the specification.

Other providers just pattern match the `redirect_uri` parameter against the registered redirect URI pattern. This saves the authorization server from storing the link between the actual redirect URI and the

respective authorization code for every transaction. But this kind of check obviously does not fulfill the intent of the specification, since the tampered redirect URI is not considered. So any attempt to inject an authorization code obtained using the `client_id` of a legitimate client or by utilizing the legitimate client on another device will not be detected in the respective deployments.

It is also assumed that the requirements defined in [RFC6749], Section 4.1.3, increase client implementation complexity as clients need to store or re-construct the correct redirect URI for the call to the token endpoint.

This document therefore recommends to instead bind every authorization code to a certain client instance on a certain device (or in a certain user agent) in the context of a certain transaction using one of the mechanisms described next.

4.5.3. Countermeasures

There are two good technical solutions to achieve this goal, outlined in the following.

4.5.3.1. PKCE

The PKCE parameter `code_challenge` along with the corresponding `code_verifier` as specified in [RFC7636] can be used as a countermeasure. When the attacker attempts to inject an authorization code, the verifier check fails: the client uses its correct verifier, but the code is associated with a challenge that does not match this verifier. PKCE is a deployed OAuth feature, although its originally intended use was solely focused on securing native apps, not the broader use recommended by this document.

4.5.3.2. Nonce

OpenID Connect's existing nonce parameter can be used for the same purpose. The nonce value is one-time use and created by the client. The client is supposed to bind it to the user agent session and sends it with the initial request to the OpenID Provider (OP). The OP binds nonce to the authorization code and attests this binding in the ID Token, which is issued as part of the code exchange at the token endpoint. If an attacker injected an authorization code in the authorization response, the nonce value in the client session and the nonce value in the ID token will not match and the attack is detected. The assumption is that an attacker cannot get hold of the user agent state on the victim's device, where he has stolen the respective authorization code.

It is important to note that this countermeasure only works if the client properly checks the nonce parameter in the ID Token and does not use any issued token until this check has succeeded. More precisely, a client protecting itself against code injection using the nonce parameter,

1. MUST validate the nonce in the ID Token obtained from the token endpoint, even if another ID Token was obtained from the authorization response (e.g., `response_type=code+id_token`), and
2. MUST ensure that, unless and until that check succeeds, all tokens (ID Tokens and the access token) are disregarded and not used for any other purpose.

4.5.3.3. Other Solutions

Other solutions, like binding state to the code, using token binding for the code, or per-instance client credentials are conceivable, but lack support and bring new security requirements.

PKCE is the most obvious solution for OAuth clients as it is available today (originally intended for OAuth native apps) whereas nonce is appropriate for OpenID Connect clients.

4.5.4. Limitations

An attacker can circumvent the countermeasures described above if he can modify the nonce or code_challenge values that are used in the victim's authorization request. The attacker can modify these values to be the same ones as those chosen by the client in his own session in Step 2 of the attack above. (This requires that the victim's session with the client begins after the attacker started his session with the client.) If the attacker is then able to capture the authorization code from the victim, the attacker will be able to inject the stolen code in Step 3 even if PKCE or nonce are used.

This attack is complex and requires a close interaction between the attacker and the victim's session. Nonetheless, measures to prevent attackers from reading the contents of the authorization response still need to be taken, as described in Section 4.1, Section 4.2, Section 4.3, Section 4.4, and Section 4.10.

4.6. Access Token Injection

In an access token injection attack, the attacker attempts to inject a stolen access token into a legitimate client (that is not under the attacker's control). This will typically happen if the attacker wants to utilize a leaked access token to impersonate a user in a certain client.

To conduct the attack, the attacker starts an OAuth flow with the client using the implicit grant and modifies the authorization response by replacing the access token issued by the authorization server or directly makes up an authorization server response including the leaked access token. Since the response includes the state value generated by the client for this particular transaction, the client does not treat the response as a CSRF attack and uses the access token injected by the attacker.

4.6.1. Countermeasures

There is no way to detect such an injection attack on the OAuth protocol level, since the token is issued without any binding to the transaction or the particular user agent.

The recommendation is therefore to use the authorization code grant type instead of relying on response types issuing access tokens at the authorization endpoint. Authorization code injection can be detected using one of the countermeasures discussed in Section 4.5.

4.7. Cross Site Request Forgery

An attacker might attempt to inject a request to the redirect URI of the legitimate client on the victim's device, e.g., to cause the client to access resources under the attacker's control. This is a variant of an attack known as Cross-Site Request Forgery (CSRF).

4.7.1. Countermeasures

The traditional countermeasures are CSRF tokens that are bound to the user agent and passed in the state parameter to the authorization server as described in [RFC6819]. The same protection is provided by PKCE or the OpenID Connect nonce value.

When using PKCE instead of state or nonce for CSRF protection, it is important to note that:

- * Clients MUST ensure that the AS supports PKCE before using PKCE for CSRF protection. If an authorization server does not support PKCE, state or nonce MUST be used for CSRF protection.

- * If state is used for carrying application state, and integrity of its contents is a concern, clients MUST protect state against tampering and swapping. This can be achieved by binding the contents of state to the browser session and/or signed/encrypted state values [I-D.bradley-oauth-jwt-encoded-state].

AS therefore MUST provide a way to detect their support for PKCE. Using AS metadata according to [RFC8414] is RECOMMENDED, but AS MAY instead provide a deployment-specific way to ensure or determine PKCE support.

4.8. PKCE Downgrade Attack

An authorization server that supports PKCE but does not make its use mandatory for all flows can be susceptible to a PKCE downgrade attack.

The first prerequisite for this attack is that there is an attacker-controllable flag in the authorization request that enables or disables PKCE for the particular flow. The presence or absence of the code_challenge parameter lends itself for this purpose, i.e., the AS enables and enforces PKCE if this parameter is present in the authorization request, but does not enforce PKCE if the parameter is missing.

The second prerequisite for this attack is that the client is not using state at all (e.g., because the client relies on PKCE for CSRF prevention) or that the client is not checking state correctly.

Roughly speaking, this attack is a variant of a CSRF attack. The attacker achieves the same goal as in the attack described in Section 4.7: He injects an authorization code (and with that, an access token) that is bound to his resources into a session between his victim and the client.

4.8.1. Attack Description

1. The user has started an OAuth session using some client at an AS. In the authorization request, the client has set the parameter code_challenge=sha256(abc) as the PKCE code challenge. The client is now waiting to receive the authorization response from the user's browser.
2. To conduct the attack, the attacker uses his own device to start an authorization flow with the targeted client. The client now uses another PKCE code challenge, say code_challenge=sha256(xyz), in the authorization request. The attacker intercepts the request and removes the entire code_challenge parameter from the

request. Since this step is performed on the attacker's device, the attacker has full access to the request contents, for example using browser debug tools.

3. If the authorization server allows for flows without PKCE, it will create a code that is not bound to any PKCE code challenge.
4. The attacker now redirects the user's browser to an authorization response URL which contains the code for the attacker's session with the AS.
5. The user's browser sends the authorization code to the client, which will now try to redeem the code for an access token at the AS. The client will send `code_verifier=abc` as the PKCE code verifier in the token request.
6. Since the authorization server sees that this code is not bound to any PKCE code challenge, it will not check the presence or contents of the `code_verifier` parameter. It will issue an access token that belongs to the attacker's resource to the client under the user's control.

4.8.2. Countermeasures

Using state properly would prevent this attack. However, practice has shown that many OAuth clients do not use or check state properly.

Therefore, AS MUST take precautions against this threat.

Note that from the view of the AS, in the attack described above, a `code_verifier` parameter is received at the token endpoint although no `code_challenge` parameter was present in the authorization request for the OAuth flow in which the authorization code was issued.

This fact can be used to mitigate this attack. [RFC7636] already mandates that

- * an AS that supports PKCE MUST check whether a code challenge is contained in the authorization request and bind this information to the code that is issued; and
- * when a code arrives at the token endpoint, and there was a `code_challenge` in the authorization request for which this code was issued, there must be a valid `code_verifier` in the token request.

Beyond this, to prevent PKCE downgrade attacks, the AS MUST ensure that if there was no code_challenge in the authorization request, a request to the token endpoint containing a code_verifier is rejected.

Note: AS that mandate the use of PKCE in general or for particular clients implicitly implement this security measure.

4.9. Access Token Leakage at the Resource Server

Access tokens can leak from a resource server under certain circumstances.

4.9.1. Access Token Phishing by Counterfeit Resource Server

An attacker may setup his own resource server and trick a client into sending access tokens to it that are valid for other resource servers (see Attackers A1 and A5). If the client sends a valid access token to this counterfeit resource server, the attacker in turn may use that token to access other services on behalf of the resource owner.

This attack assumes the client is not bound to one specific resource server (and its URL) at development time, but client instances are provided with the resource server URL at runtime. This kind of late binding is typical in situations where the client uses a service implementing a standardized API (e.g., for e-Mail, calendar, health, or banking) and where the client is configured by a user or administrator for a service which this user or company uses.

4.9.1.1. Countermeasures

There are several potential mitigation strategies, which will be discussed in the following sections.

4.9.1.1.1. Metadata

An authorization server could provide the client with additional information about the location where it is safe to use its access tokens.

In the simplest form, this would require the AS to publish a list of its known resource servers, illustrated in the following example using a non-standard metadata parameter resource_servers:

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

```
{
  "issuer": "https://server.somesite.example",
  "authorization_endpoint":
    "https://server.somesite.example/authorize",
  "resource_servers": [
    "email.somesite.example",
    "storage.somesite.example",
    "video.somesite.example"
  ]
  ...
}
```

The AS could also return the URL(s) an access token is good for in the token response, illustrated by the example and non-standard return parameter `access_token_resource_server`:

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

{
  "access_token": "2YotnFZFEjrlzCsicMWpAA",
  "access_token_resource_server":
    "https://hostedresource.somesite.example/path1",
  ...
}
```

This mitigation strategy would rely on the client to enforce the security policy and to only send access tokens to legitimate destinations. Results of OAuth related security research (see for example [oauth_security_ubic] and [oauth_security_cmu]) indicate a large portion of client implementations do not or fail to properly implement security controls, like state checks. So relying on clients to prevent access token phishing is likely to fail as well. Moreover given the ratio of clients to authorization and resource servers, it is considered the more viable approach to move as much as possible security-related logic to those entities. Clearly, the client has to contribute to the overall security. But there are alternative countermeasures, as described in the next sections, which provide a better balance between the involved parties.

4.9.1.1.2. Sender-Constrained Access Tokens

As the name suggests, sender-constrained access token scope the applicability of an access token to a certain sender. This sender is obliged to demonstrate knowledge of a certain secret as prerequisite for the acceptance of that token at a resource server.

A typical flow looks like this:

1. The authorization server associates data with the access token that binds this particular token to a certain client. The binding can utilize the client identity, but in most cases the AS utilizes key material (or data derived from the key material) known to the client.
2. This key material must be distributed somehow. Either the key material already exists before the AS creates the binding or the AS creates ephemeral keys. The way pre-existing key material is distributed varies among the different approaches. For example, X.509 Certificates can be used in which case the distribution happens explicitly during the enrollment process. Or the key material is created and distributed at the TLS layer, in which case it might automatically happen during the setup of a TLS connection.
3. The RS must implement the actual proof of possession check. This is typically done on the application level, often tied to specific material provided by transport layer (e.g., TLS). The RS must also ensure that replay of the proof of possession is not possible.

There exist several proposals to demonstrate the proof of possession in the scope of the OAuth working group:

- * *OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens* ([RFC8705]): The approach as specified in this document allows the use of mutual TLS (mTLS) for both client authentication and sender-constrained access tokens. For the purpose of sender-constrained access tokens, the client is identified towards the resource server by the fingerprint of its public key. During processing of an access token request, the authorization server obtains the client's public key from the TLS stack and associates its fingerprint with the respective access tokens. The resource server in the same way obtains the public key from the TLS stack and compares its fingerprint with the fingerprint associated with the access token.

- * ***DPoP*** ([I-D.ietf-oauth-dpop]): DPoP (Demonstration of Proof-of-Possession at the Application Layer) outlines an application-level sender-constraining for access and refresh tokens that can be used in cases where neither mTLS nor OAuth Token Binding (see below) are available. It uses proof-of-possession based on a public/private key pair and application-level signing. DPoP can be used with public clients and, in case of confidential clients, can be combined with any client authentication method.
- * ***OAuth Token Binding*** ([I-D.ietf-oauth-token-binding]): In this approach, an access token is, via the token binding ID, bound to key material representing a long term association between a client and a certain TLS host. Negotiation of the key material and proof of possession in the context of a TLS handshake is taken care of by the TLS stack. The client needs to determine the token binding ID of the target resource server and pass this data to the access token request. The authorization server then associates the access token with this ID. The resource server checks on every invocation that the token binding ID of the active TLS connection and the token binding ID of associated with the access token match. Since all crypto-related functions are covered by the TLS stack, this approach is very client developer friendly. As a prerequisite, token binding as described in [RFC8473] (including federated token bindings) must be supported on all ends (client, authorization server, resource server).
- * ***Signed HTTP Requests*** ([I-D.ietf-oauth-signed-http-request]): This approach utilizes [I-D.ietf-oauth-pop-key-distribution] and represents the elements of the signature in a JSON object. The signature is built using JWS. The mechanism has built-in support for signing of HTTP method, query parameters and headers. It also incorporates a timestamp as basis for replay prevention.
- * ***JWT Pop Tokens*** ([I-D.sakimura-oauth-jpop]): This draft describes different ways to constrain access token usage, namely TLS or request signing. Note: Since the authors of this draft contributed the TLS-related proposal to [RFC8705], this document only considers the request signing part. For request signing, the draft utilizes [I-D.ietf-oauth-pop-key-distribution] and [RFC7800]. The signature data is represented in a JWT and JWS is used for signing. Replay prevention is provided by building the signature over a server-provided nonce, client-provided nonce and a nonce counter.

At the time of writing, OAuth Mutual TLS is the most widely implemented and the only standardized sender-constraining method. The use of OAuth Mutual TLS therefore is RECOMMENDED.

Note that the security of sender-constrained tokens is undermined when an attacker gets access to the token and the key material. This is in particular the case for corrupted client software and cross-site scripting attacks (when the client is running in the browser). If the key material is protected in a hardware or software security module or only indirectly accessible (like in a TLS stack), sender-constrained tokens at least protect against a use of the token when the client is offline, i.e., when the security module or interface is not available to the attacker. This applies to access tokens as well as to refresh tokens (see Section 4.13).

4.9.1.1.3. Audience Restricted Access Tokens

Audience restriction essentially restricts access tokens to a particular resource server. The authorization server associates the access token with the particular resource server and the resource server SHOULD verify the intended audience. If the access token fails the intended audience validation, the resource server must refuse to serve the respective request.

In general, audience restrictions limit the impact of token leakage. In the case of a counterfeit resource server, it may (as described below) also prevent abuse of the phished access token at the legitimate resource server.

The audience can be expressed using logical names or physical addresses (like URLs). In order to prevent phishing, it is necessary to use the actual URL the client will send requests to. In the phishing case, this URL will point to the counterfeit resource server. If the attacker tries to use the access token at the legitimate resource server (which has a different URL), the resource server will detect the mismatch (wrong audience) and refuse to serve the request.

In deployments where the authorization server knows the URLs of all resource servers, the authorization server may just refuse to issue access tokens for unknown resource server URLs.

The client SHOULD tell the authorization server the intended resource server. The proposed mechanism [I-D.ietf-oauth-resource-indicators] could be used or by encoding the information in the scope value.

Instead of the URL, it is also possible to utilize the fingerprint of the resource server's X.509 certificate as audience value. This variant would also allow to detect an attempt to spoof the legitimate resource server's URL by using a valid TLS certificate obtained from a different CA. It might also be considered a privacy benefit to hide the resource server URL from the authorization server.

Audience restriction may seem easier to use since it does not require any crypto on the client-side. Still, since every access token is bound to a specific resource server, the client also needs to obtain a single RS-specific access token when accessing several resource servers. (Resource indicators, as specified in [I-D.ietf-oauth-resource-indicators], can help to achieve this.) [I-D.ietf-oauth-token-binding] has the same property since different token binding ids must be associated with the access token. Using [RFC8705], on the other hand, allows a client to use the access token at multiple resource servers.

It shall be noted that audience restrictions, or generally speaking an indication by the client to the authorization server where it wants to use the access token, has additional benefits beyond the scope of token leakage prevention. It allows the authorization server to create different access token whose format and content is specifically minted for the respective server. This has huge functional and privacy advantages in deployments using structured access tokens.

4.9.2. Compromised Resource Server

An attacker may compromise a resource server to gain access to the resources of the respective deployment. Such a compromise may range from partial access to the system, e.g., its log files, to full control of the respective server.

If the attacker were able to gain full control, including shell access, all controls can be circumvented and all resources be accessed. The attacker would also be able to obtain other access tokens held on the compromised system that would potentially be valid to access other resource servers.

Preventing server breaches by hardening and monitoring server systems is considered a standard operational procedure and, therefore, out of the scope of this document. This section focuses on the impact of OAuth-related breaches and the replaying of captured access tokens.

The following measures should be taken into account by implementers in order to cope with access token replay by malicious actors:

- * Sender-constrained access tokens as described in Section 4.9.1.1.2 SHOULD be used to prevent the attacker from replaying the access tokens on other resource servers. Depending on the severity of the penetration, sender-constrained access tokens will also prevent replay on the compromised system.

- * Audience restriction as described in Section 4.9.1.1.3 SHOULD be used to prevent replay of captured access tokens on other resource servers.
- * The resource server MUST treat access tokens like any other credentials. It is considered good practice to not log them and not store them in plain text.

The first and second recommendation also apply to other scenarios where access tokens leak (see Attacker A5).

4.10. Open Redirection

The following attacks can occur when an AS or client has an open redirector. An open redirector is an endpoint that forwards a user's browser to an arbitrary URI obtained from a query parameter.

4.10.1. Client as Open Redirector

Clients MUST NOT expose open redirectors. Attackers may use open redirectors to produce URLs pointing to the client and utilize them to exfiltrate authorization codes and access tokens, as described in Section 4.1.2. Another abuse case is to produce URLs that appear to point to the client. This might trick users into trusting the URL and follow it in their browser. This can be abused for phishing.

In order to prevent open redirection, clients should only redirect if the target URLs are whitelisted or if the origin and integrity of a request can be authenticated. Countermeasures against open redirection are described by OWASP [owasp_redir].

4.10.2. Authorization Server as Open Redirector

Just as with clients, attackers could try to utilize a user's trust in the authorization server (and its URL in particular) for performing phishing attacks. OAuth authorization servers regularly redirect users to other web sites (the clients), but must do so in a safe way.

[RFC6749], Section 4.1.2.1, already prevents open redirects by stating that the AS MUST NOT automatically redirect the user agent in case of an invalid combination of `client_id` and `redirect_uri`.

However, an attacker could also utilize a correctly registered redirect URI to perform phishing attacks. The attacker could, for example, register a client via dynamic client registration [RFC7591] and intentionally send an erroneous authorization request, e.g., by using an invalid scope value, thus instructing the AS to redirect the user agent to its phishing site.

The AS **MUST** take precautions to prevent this threat. Based on its risk assessment, the AS needs to decide whether it can trust the redirect URI and **SHOULD** only automatically redirect the user agent if it trusts the redirect URI. If the URI is not trusted, the AS **MAY** inform the user and rely on the user to make the correct decision.

4.11. 307 Redirect

At the authorization endpoint, a typical protocol flow is that the AS prompts the user to enter her credentials in a form that is then submitted (using the HTTP POST method) back to the authorization server. The AS checks the credentials and, if successful, redirects the user agent to the client's redirection endpoint.

In [RFC6749], the HTTP status code 302 is used for this purpose, but "any other method available via the user-agent to accomplish this redirection is allowed". When the status code 307 is used for redirection instead, the user agent will send the user credentials via HTTP POST to the client.

This discloses the sensitive credentials to the client. If the relying party is malicious, it can use the credentials to impersonate the user at the AS.

The behavior might be unexpected for developers, but is defined in [RFC7231], Section 6.4.7. This status code does not require the user agent to rewrite the POST request to a GET request and thereby drop the form data in the POST request body.

In the HTTP standard [RFC7231], only the status code 303 unambiguously enforces rewriting the HTTP POST request to an HTTP GET request. For all other status codes, including the popular 302, user agents can opt not to rewrite POST to GET requests and therefore to reveal the user credentials to the client. (In practice, however, most user agents will only show this behaviour for 307 redirects.)

AS which redirect a request that potentially contains user credentials therefore **MUST NOT** use the HTTP 307 status code for redirection. If an HTTP redirection (and not, for example, JavaScript) is used for such a request, AS **SHOULD** use HTTP status code 303 "See Other".

4.12. TLS Terminating Reverse Proxies

A common deployment architecture for HTTP applications is to hide the application server behind a reverse proxy that terminates the TLS connection and dispatches the incoming requests to the respective application server nodes.

This section highlights some attack angles of this deployment architecture with relevance to OAuth and gives recommendations for security controls.

In some situations, the reverse proxy needs to pass security-related data to the upstream application servers for further processing. Examples include the IP address of the request originator, token binding ids, and authenticated TLS client certificates. This data is usually passed in custom HTTP headers added to the upstream request.

If the reverse proxy would pass through any header sent from the outside, an attacker could try to directly send the faked header values through the proxy to the application server in order to circumvent security controls that way. For example, it is standard practice of reverse proxies to accept X-Forwarded-For headers and just add the origin of the inbound request (making it a list). Depending on the logic performed in the application server, the attacker could simply add a whitelisted IP address to the header and render a IP whitelist useless.

A reverse proxy must therefore sanitize any inbound requests to ensure the authenticity and integrity of all header values relevant for the security of the application servers.

If an attacker was able to get access to the internal network between proxy and application server, the attacker could also try to circumvent security controls in place. It is, therefore, essential to ensure the authenticity of the communicating entities. Furthermore, the communication link between reverse proxy and application server must be protected against eavesdropping, injection, and replay of messages.

4.13. Refresh Token Protection

Refresh tokens are a convenient and user-friendly way to obtain new access tokens after the expiration of access tokens. Refresh tokens also add to the security of OAuth since they allow the authorization server to issue access tokens with a short lifetime and reduced scope thus reducing the potential impact of access token leakage.

4.13.1. Discussion

Refresh tokens are an attractive target for attackers since they represent the overall grant a resource owner delegated to a certain client. If an attacker is able to exfiltrate and successfully replay a refresh token, the attacker will be able to mint access tokens and use them to access resource servers on behalf of the resource owner.

[RFC6749] already provides a robust baseline protection by requiring

- * confidentiality of the refresh tokens in transit and storage,
- * the transmission of refresh tokens over TLS-protected connections between authorization server and client,
- * the authorization server to maintain and check the binding of a refresh token to a certain client and authentication of this client during token refresh, if possible, and
- * that refresh tokens cannot be generated, modified, or guessed.

[RFC6749] also lays the foundation for further (implementation specific) security measures, such as refresh token expiration and revocation as well as refresh token rotation by defining respective error codes and response behavior.

This specification gives recommendations beyond the scope of [RFC6749] and clarifications.

4.13.2. Recommendations

Authorization servers SHOULD determine, based on a risk assessment, whether to issue refresh tokens to a certain client. If the authorization server decides not to issue refresh tokens, the client MAY refresh access tokens by utilizing other grant types, such as the authorization code grant type. In such a case, the authorization server may utilize cookies and persistent grants to optimize the user experience.

If refresh tokens are issued, those refresh tokens MUST be bound to the scope and resource servers as consented by the resource owner. This is to prevent privilege escalation by the legitimate client and reduce the impact of refresh token leakage.

For confidential clients, [RFC6749] already requires that refresh tokens can only be used by the client for which they were issued.

Authorization server MUST utilize one of these methods to detect refresh token replay by malicious actors for public clients:

- * *Sender-constrained refresh tokens:* the authorization server cryptographically binds the refresh token to a certain client instance by utilizing [RFC8705] or [I-D.ietf-oauth-token-binding].
- * *Refresh token rotation:* the authorization server issues a new refresh token with every access token refresh response. The previous refresh token is invalidated but information about the relationship is 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 cannot determine which party submitted the invalid refresh token, but it will revoke the active refresh token. This stops the attack at the cost of forcing the legitimate client to obtain a fresh authorization grant.

Implementation note: the grant to which a refresh token belongs may be encoded into the refresh token itself. This can enable an authorization server to efficiently determine the grant to which a refresh token belongs, and by extension, all refresh tokens that need to be revoked. Authorization servers MUST ensure the integrity of the refresh token value in this case, for example, using signatures.

Authorization servers MAY revoke refresh tokens automatically in case of a security event, such as:

- * password change
- * logout at the authorization server

Refresh tokens SHOULD expire if the client has been inactive for some time, i.e., the refresh token has not been used to obtain fresh access tokens for some time. The expiration time is at the discretion of the authorization server. It might be a global value or determined based on the client policy or the grant associated with the refresh token (and its sensitivity).

4.14. Client Impersonating Resource Owner

Resource servers may make access control decisions based on the identity of the resource owner as communicated in the sub Claim returned by the authorization server in a token introspection response [RFC7662] or other mechanisms. If a client is able to choose its own client_id during registration with the authorization server, then there is a risk that it can register with the same sub value as a privileged user. A subsequent access token obtained under the client credentials grant may be mistaken for an access token authorized by the privileged user if the resource server does not perform additional checks.

4.14.1. Countermeasures

Authorization servers SHOULD NOT allow clients to influence their client_id or sub value or any other Claim if that can cause confusion with a genuine resource owner. Where this cannot be avoided, authorization servers MUST provide other means for the resource server to distinguish between access tokens authorized by a resource owner from access tokens authorized by the client itself.

4.15. Clickjacking

As described in Section 4.4.1.9 of [RFC6819], the authorization request is susceptible to clickjacking. An attacker can use this vector to obtain the user's authentication credentials, change the scope of access granted to the client, and potentially access the user's resources.

Authorization servers MUST prevent clickjacking attacks. Multiple countermeasures are described in [RFC6819], including the use of the X-Frame-Options HTTP response header field and frame-busting JavaScript. In addition to those, authorization servers SHOULD also use Content Security Policy (CSP) level 2 [CSP-2] or greater.

To be effective, CSP must be used on the authorization endpoint and, if applicable, other endpoints used to authenticate the user and authorize the client (e.g., the device authorization endpoint, login pages, error pages, etc.). This prevents framing by unauthorized origins in user agents that support CSP. The client MAY permit being framed by some other origin than the one used in its redirection endpoint. For this reason, authorization servers SHOULD allow administrators to configure allowed origins for particular clients and/or for clients to register these dynamically.

Using CSP allows authorization servers to specify multiple origins in a single response header field and to constrain these using flexible patterns (see [CSP-2] for details). Level 2 of this standard provides a robust mechanism for protecting against clickjacking by using policies that restrict the origin of frames (using frame-ancestors) together with those that restrict the sources of scripts allowed to execute on an HTML page (by using script-src). A non-normative example of such a policy is shown in the following listing:

```
HTTP/1.1 200 OK
Content-Security-Policy: frame-ancestors https://ext.example.org:8000
Content-Security-Policy: script-src 'self'
X-Frame-Options: ALLOW-FROM https://ext.example.org:8000
...
```

Because some user agents do not support [CSP-2], this technique SHOULD be combined with others, including those described in [RFC6819], unless such legacy user agents are explicitly unsupported by the authorization server. Even in such cases, additional countermeasures SHOULD still be employed.

5. Acknowledgements

We would like to thank Jim Manico, Phil Hunt, Nat Sakimura, Christian Mainka, Doug McDorman, Johan Peeters, Joseph Heenan, Brock Allen, Vittorio Bertocci, David Waite, Nov Mataka, Tomek Stojekci, Dominick Baier, Neil Madden, William Dennis, Dick Hardt, Petteri Stenius, Annabelle Richard Backman, Aaron Parecki, George Fletscher, Brian Campbell, Konstantin Lapine, Tim Würtele, Guido Schmitz, Hans Zandbelt, Jared Jennings, Michael Peck, Pedram Hosseyni, Michael B. Jones, Travis Spencer, and Karsten Meyer zu Selhausen for their valuable feedback.

6. IANA Considerations

This draft includes no request to IANA.

7. Security Considerations

All relevant security considerations have been given in the functional specification.

8. Normative References

- [RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <<https://www.rfc-editor.org/info/rfc6749>>.

- [oauth-v2-form-post-response-mode]
Jones, M. and B. Campbell, "OAuth 2.0 Form Post Response Mode", 27 April 2015, <http://openid.net/specs/oauth-v2-form-post-response-mode-1_0.html>.
- [RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <<https://www.rfc-editor.org/info/rfc7231>>.
- [OpenIDDisc]
Sakimura, N., Bradley, J., Jones, M., and E. Jay, "OpenID Connect Discovery 1.0 incorporating errata set 1", 8 November 2014, <https://openid.net/specs/openid-connect-discovery-1_0.html>.
- [RFC7662] Richer, J., Ed., "OAuth 2.0 Token Introspection", RFC 7662, DOI 10.17487/RFC7662, October 2015, <<https://www.rfc-editor.org/info/rfc7662>>.
- [RFC8252] Denniss, W. and J. Bradley, "OAuth 2.0 for Native Apps", BCP 212, RFC 8252, DOI 10.17487/RFC8252, October 2017, <<https://www.rfc-editor.org/info/rfc8252>>.
- [RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, <<https://www.rfc-editor.org/info/rfc6750>>.
- [RFC6819] Lodderstedt, T., Ed., McGloin, M., and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations", RFC 6819, DOI 10.17487/RFC6819, January 2013, <<https://www.rfc-editor.org/info/rfc6819>>.
- [RFC8705] Campbell, B., Bradley, J., Sakimura, N., and T. Lodderstedt, "OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens", February 2020, <<https://www.rfc-editor.org/info/rfc8705>>.
- [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <<https://www.rfc-editor.org/info/rfc3986>>.
- [OpenID] Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0 incorporating errata set 1", 8 November 2014, <https://openid.net/specs/openid-connect-core-1_0.html>.

- [RFC8414] Jones, M., Sakimura, N., and J. Bradley, "OAuth 2.0 Authorization Server Metadata", RFC 8414, DOI 10.17487/RFC8414, June 2018, <<https://www.rfc-editor.org/info/rfc8414>>.

9. Informative References

- [arXiv.1601.01229]
Fett, D., Küsters, R., and G. Schmitz, "A Comprehensive Formal Security Analysis of OAuth 2.0", 6 January 2016, <<http://arxiv.org/abs/1601.01229/>>.
- [I-D.ietf-oauth-dpop]
Fett, D., Campbell, B., Bradley, J., Lodderstedt, T., Jones, M., and D. Waite, "OAuth 2.0 Demonstrating Proof-of-Possession at the Application Layer (DPoP)", Work in Progress, Internet-Draft, draft-ietf-oauth-dpop-04, 4 October 2021, <<https://datatracker.ietf.org/doc/html/draft-ietf-oauth-dpop-04>>.
- [CSP-2] West, M., Barth, A., and D. Veditz, "Content Security Policy Level 2", July 2015, <<https://www.w3.org/TR/CSP2>>.
- [I-D.ietf-oauth-rar]
Lodderstedt, T., Richer, J., and B. Campbell, "OAuth 2.0 Rich Authorization Requests", Work in Progress, Internet-Draft, draft-ietf-oauth-rar-08, 18 October 2021, <<https://datatracker.ietf.org/doc/html/draft-ietf-oauth-rar-08>>.
- [I-D.ietf-oauth-signed-http-request]
Richer, J., Bradley, J., and H. Tschofenig, "A Method for Signing HTTP Requests for OAuth", Work in Progress, Internet-Draft, draft-ietf-oauth-signed-http-request-03, 8 August 2016, <<https://datatracker.ietf.org/doc/html/draft-ietf-oauth-signed-http-request-03>>.
- [bug.chromium]
"Referer header includes URL fragment when opening link using New Tab", <<https://bugs.chromium.org/p/chromium/issues/detail?id=168213>>.
- [webappsec-referrer-policy]
Eisinger, J. and E. Stark, "Referrer Policy", 20 April 2017, <<https://w3c.github.io/webappsec-referrer-policy>>.

- [I-D.bradley-oauth-jwt-encoded-state]
Bradley, J., Lodderstedt, D. T., and H. Zandbelt,
"Encoding claims in the OAuth 2 state parameter using a
JWT", Work in Progress, Internet-Draft, draft-bradley-
oauth-jwt-encoded-state-09, 4 November 2018,
<[https://datatracker.ietf.org/doc/html/draft-bradley-
oauth-jwt-encoded-state-09](https://datatracker.ietf.org/doc/html/draft-bradley-oauth-jwt-encoded-state-09)>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
Requirement Levels", BCP 14, RFC 2119,
DOI 10.17487/RFC2119, March 1997,
<<https://www.rfc-editor.org/info/rfc2119>>.
- [webcrypto]
Watson, M., "Web Cryptography API", 26 January 2017,
<<https://www.w3.org/TR/2017/REC-WebCryptoAPI-20170126/>>.
- [oauth_security_jcs_14]
Bansal, C., Bhargavan, K., Delignat-Lavaud, A., and S.
Maffeis, "Discovering concrete attacks on website
authorization by formal analysis", 23 April 2014,
<<https://www.doc.ic.ac.uk/~maffeis/papers/jcs14.pdf>>.
- [owasp_redir]
"OWASP Cheat Sheet Series - Unvalidated Redirects and
Forwards",
<[https://cheatsheetseries.owasp.org/cheatsheets/
Unvalidated_Redirects_and_Forwards_Cheat_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Unvalidated_Redirects_and_Forwards_Cheat_Sheet.html)>.
- [RFC7591] Richer, J., Ed., Jones, M., Bradley, J., Machulak, M., and
P. Hunt, "OAuth 2.0 Dynamic Client Registration Protocol",
RFC 7591, DOI 10.17487/RFC7591, July 2015,
<<https://www.rfc-editor.org/info/rfc7591>>.
- [arXiv.1901.11520]
Fett, D., Hosseini, P., and R. Küsters, "An Extensive
Formal Security Analysis of the OpenID Financial-grade
API", 31 January 2019, <<http://arxiv.org/abs/1901.11520>>.
- [I-D.ietf-oauth-par]
Lodderstedt, T., Campbell, B., Sakimura, N., Tonge, D.,
and F. Skokan, "OAuth 2.0 Pushed Authorization Requests",
Work in Progress, Internet-Draft, draft-ietf-oauth-par-10,
29 July 2021, <[https://datatracker.ietf.org/doc/html/
draft-ietf-oauth-par-10](https://datatracker.ietf.org/doc/html/draft-ietf-oauth-par-10)>.

- [arXiv.1704.08539]
Fett, D., Küsters, R., and G. Schmitz, "The Web SSO Standard OpenID Connect: In-Depth Formal Security Analysis and Security Guidelines", 27 April 2017, <<http://arxiv.org/abs/1704.08539/>>.
- [arXiv.1508.04324v2]
Mladenov, V., Mainka, C., and J. Schwenk, "On the security of modern Single Sign-On Protocols: Second-Order Vulnerabilities in OpenID Connect", 7 January 2016, <<http://arxiv.org/abs/1508.04324v2/>>.
- [webauthn] Balfanz, D., Czeskis, A., Hodges, J., Jones, J.C., Jones, M.B., Kumar, A., Liao, A., Lindemann, R., and E. Lundberg, "Web Authentication: An API for accessing Public Key Credentials Level 1", 4 March 2019, <<https://www.w3.org/TR/2019/REC-webauthn-1-20190304/>>.
- [RFC8473] Popov, A., Nystroem, M., Balfanz, D., Ed., Harper, N., and J. Hodges, "Token Binding over HTTP", RFC 8473, DOI 10.17487/RFC8473, October 2018, <<https://www.rfc-editor.org/info/rfc8473>>.
- [I-D.ietf-oauth-pop-key-distribution]
Bradley, J., Hunt, P., Jones, M. B., Tschofenig, H., and M. Meszaros, "OAuth 2.0 Proof-of-Possession: Authorization Server to Client Key Distribution", Work in Progress, Internet-Draft, draft-ietf-oauth-pop-key-distribution-07, 27 March 2019, <<https://datatracker.ietf.org/doc/html/draft-ietf-oauth-pop-key-distribution-07>>.
- [I-D.ietf-oauth-jwsreq]
Sakimura, N., Bradley, J., and M. B. Jones, "The OAuth 2.0 Authorization Framework: JWT-Secured Authorization Request (JAR)", Work in Progress, Internet-Draft, draft-ietf-oauth-jwsreq-34, 8 April 2021, <<https://datatracker.ietf.org/doc/html/draft-ietf-oauth-jwsreq-34>>.
- [oauth_security_ubic]
Sun, S.-T. and K. Beznosov, "The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems", October 2012, <<http://passwordresearch.com/papers/paper267.html>>.
- [oauth_security_cmu]
Chen, E., Pei, Y., Chen, S., Tian, Y., Kotcher, R., and P. Tague, "OAuth Demystified for Mobile Application

Developers", November 2014,
<<http://css.csail.mit.edu/6.858/2012/readings/oauth-sso.pdf>>.

[RFC7636] Sakimura, N., Ed., Bradley, J., and N. Agarwal, "Proof Key for Code Exchange by OAuth Public Clients", RFC 7636, DOI 10.17487/RFC7636, September 2015, <<https://www.rfc-editor.org/info/rfc7636>>.

[I-D.ietf-oauth-resource-indicators]
Campbell, B., Bradley, J., and H. Tschofenig, "Resource Indicators for OAuth 2.0", Work in Progress, Internet-Draft, draft-ietf-oauth-resource-indicators-08, 11 September 2019, <<https://datatracker.ietf.org/doc/html/draft-ietf-oauth-resource-indicators-08>>.

[I-D.ietf-oauth-token-binding]
Jones, M. B., Campbell, B., Bradley, J., and W. Denniss, "OAuth 2.0 Token Binding", Work in Progress, Internet-Draft, draft-ietf-oauth-token-binding-08, 19 October 2018, <<https://datatracker.ietf.org/doc/html/draft-ietf-oauth-token-binding-08>>.

[I-D.sakimura-oauth-jpop]
Sakimura, N., Li, K., and J. Bradley, "The OAuth 2.0 Authorization Framework: JWT Pop Token Usage", Work in Progress, Internet-Draft, draft-sakimura-oauth-jpop-05, 22 July 2019, <<https://datatracker.ietf.org/doc/html/draft-sakimura-oauth-jpop-05>>.

[subdomaintakeover]
Liu, D., Hao, S., and H. Wang, "All Your DNS Records Point to Us: Understanding the Security Threats of Dangling DNS Records", 24 October 2016, <<https://www.eecis.udel.edu/~hnw/paper/ccs16a.pdf>>.

[RFC7800] Jones, M., Bradley, J., and H. Tschofenig, "Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs)", RFC 7800, DOI 10.17487/RFC7800, April 2016, <<https://www.rfc-editor.org/info/rfc7800>>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.

[I-D.ietf-oauth-iss-auth-resp]
Selhausen, K. M. Z. and D. Fett, "OAuth 2.0 Authorization Server Issuer Identification", Work in Progress, Internet-

Draft, draft-ietf-oauth-iss-auth-resp-04, 2 December 2021,
<<https://datatracker.ietf.org/doc/html/draft-ietf-oauth-iss-auth-resp-04>>.

[JARM] Lodderstedt, T. and B. Campbell, "Financial-grade API: JWT Secured Authorization Response Mode for OAuth 2.0 (JARM)", 17 October 2018,
<<https://openid.net/specs/openid-financial-api-jarm.html>>.

Appendix A. Document History

[[To be removed from the final specification]]

-19

- * Changed affiliation of Andrey Labunets
- * Editorial change to clarify the new recommendations for refresh tokens

-18

- * Fix editorial and spelling issues.
- * Change wording for disallowing HTTP redirect URIs.

-17

- * Make the use of metadata RECOMMENDED for both servers and clients
- * Make announcing PKCE support in metadata the RECOMMENDED way (before: either metadata or deployment-specific way)
- * AS also MUST NOT expose open redirectors.
- * Mention that attackers can collaborate.
- * Update recommendations regarding mix-up defense, building upon [I-D.ietf-oauth-iss-auth-resp].
- * Improve description of mix-up attack.
- * Make HTTPS mandatory for most redirect URIs.

-16

- * Make MTLS a suggestion, not RECOMMENDED.

- * Add important requirements when using nonce for code injection protection.
- * Highlight requirements for refresh token sender-constraining.
- * Make PKCE a MUST for public clients.
- * Describe PKCE Downgrade Attacks and countermeasures.
- * Allow variable port numbers in localhost redirect URIs as in RFC8252, Section 7.3.

-15

- * Update reference to DPoP
- * Fix reference to RFC8414
- * Move to xml2rfcv3

-14

- * Added info about using CSP to prevent clickjacking
- * Changes from WGLC feedback
- * Editorial changes
- * AS MUST announce PKCE support either in metadata or using deployment-specific ways (before: SHOULD)

-13

- * Discourage use of Resource Owner Password Credentials Grant
- * Added text on client impersonating resource owner
- * Recommend asymmetric methods for client authentication
- * Encourage use of PKCE mode "S256"
- * PKCE may replace state for CSRF protection
- * AS SHOULD publish PKCE support
- * Cleaned up discussion on auth code injection
- * AS MUST support PKCE

-12

- * Added updated attacker model

-11

- * Adapted section 2.1.2 to outcome of consensus call
- * more text on refresh token inactivity and implementation note on refresh token replay detection via refresh token rotation

-10

- * incorporated feedback by Joseph Heenan
- * changed occurrences of SHALL to MUST
- * added text on lack of token/cert binding support tokens issued in the authorization response as justification to not recommend issuing tokens there at all
- * added requirement to authenticate clients during code exchange (PKCE or client credential) to 2.1.1.
- * added section on refresh tokens
- * editorial enhancements to 2.1.2 based on feedback

-09

- * changed text to recommend not to use implicit but code
- * added section on access token injection
- * reworked sections 3.1 through 3.3 to be more specific on implicit grant issues

-08

- * added recommendations re implicit and token injection
- * uppercased key words in Section 2 according to RFC 2119

-07

- * incorporated findings of Doug McDorman
- * added section on HTTP status codes for redirects

- * added new section on access token privilege restriction based on comments from Johan Peeters

-06

- * reworked section 3.8.1
- * incorporated Phil Hunt's feedback
- * reworked section on mix-up
- * extended section on code leakage via referrer header to also cover state leakage
- * added Daniel Fett as author
- * replaced text intended to inform WG discussion by recommendations to implementors
- * modified example URLs to conform to RFC 2606

-05

- * Completed sections on code leakage via referrer header, attacks in browser, mix-up, and CSRF
- * Reworked Code Injection Section
- * Added reference to OpenID Connect spec
- * removed refresh token leakage as respective considerations have been given in section 10.4 of RFC 6749
- * first version on open redirection
- * incorporated Christian Mainka's review feedback

-04

- * Restructured document for better readability
- * Added best practices on Token Leakage prevention

-03

- * Added section on Access Token Leakage at Resource Server
- * incorporated Brian Campbell's findings

-02

- * Folded Mix up and Access Token leakage through a bad AS into new section for dynamic OAuth threats
- * reworked dynamic OAuth section

-01

- * Added references to mitigation methods for token leakage
- * Added reference to Token Binding for Authorization Code
- * incorporated feedback of Phil Hunt
- * fixed numbering issue in attack descriptions in section 2

-00 (WG document)

- * turned the ID into a WG document and a BCP
- * Added federated app login as topic in Other Topics

Authors' Addresses

Torsten Lodderstedt
yes.com

Email: torsten@lodderstedt.net

John Bradley
Yubico

Email: ve7jtb@ve7jtb.com

Andrey Labunets
Independent Researcher

Email: isciurus@gmail.com

Daniel Fett
yes.com

Email: mail@danielfett.de

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 22, 2019

M. Jones
Microsoft
B. Campbell
Ping Identity
J. Bradley
Yubico
W. Denniss
Google
October 19, 2018

OAuth 2.0 Token Binding
draft-ietf-oauth-token-binding-08

Abstract

This specification enables OAuth 2.0 implementations to apply Token Binding to Access Tokens, Authorization Codes, Refresh Tokens, JWT Authorization Grants, and JWT Client Authentication. This cryptographically binds these tokens to a client's Token Binding key pair, possession of which is proven on the TLS connections over which the tokens are intended to be used. This use of Token Binding protects these tokens from man-in-the-middle and token export and replay attacks.

Status of This Memo

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

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

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

This Internet-Draft will expire on April 22, 2019.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction	3
1.1. Requirements Notation and Conventions	3
1.2. Terminology	3
2. Token Binding for Refresh Tokens	4
2.1. Example Token Binding for Refresh Tokens	4
3. Token Binding for Access Tokens	6
3.1. Access Tokens Issued from the Authorization Endpoint . .	7
3.1.1. Example Access Token Issued from the Authorization Endpoint	8
3.2. Access Tokens Issued from the Token Endpoint	9
3.2.1. Example Access Token Issued from the Token Endpoint .	9
3.3. Protected Resource Token Binding Validation	11
3.3.1. Example Protected Resource Request	11
3.4. Representing Token Binding in JWT Access Tokens	11
3.5. Representing Token Binding in Introspection Responses . .	12
4. Token Binding Metadata	13
4.1. Token Binding Client Metadata	13
4.2. Token Binding Authorization Server Metadata	13
5. Token Binding for Authorization Codes	14
5.1. Native Application Clients	14
5.1.1. Code Challenge	14
5.1.1.1. Example Code Challenge	15
5.1.2. Code Verifier	15
5.1.2.1. Example Code Verifier	16
5.2. Web Server Clients	16
5.2.1. Code Challenge	17
5.2.1.1. Example Code Challenge	17
5.2.2. Code Verifier	18
5.2.2.1. Example Code Verifier	18
6. Token Binding JWT Authorization Grants and Client Authentication	19
6.1. JWT Format and Processing Requirements	19
6.2. Token Bound JWTs for Client Authentication	20
6.3. Token Bound JWTs for as Authorization Grants	20
7. Security Considerations	21
7.1. Phasing in Token Binding	21

7.2. Binding of Refresh Tokens	21
8. IANA Considerations	22
8.1. OAuth Dynamic Client Registration Metadata Registration	22
8.1.1. Registry Contents	22
8.2. OAuth Authorization Server Metadata Registration	23
8.2.1. Registry Contents	23
8.3. PKCE Code Challenge Method Registration	23
8.3.1. Registry Contents	23
9. Token Endpoint Authentication Method Registration	23
9.1. Registry Contents	24
10. Sub-Namespace Registrations	24
10.1. Registry Contents	24
11. References	24
11.1. Normative References	24
11.2. Informative References	26
Appendix A. Acknowledgements	27
Appendix B. Document History	27
Authors' Addresses	29

1. Introduction

This specification enables OAuth 2.0 [RFC6749] implementations to apply Token Binding (TLS Extension for Token Binding Protocol Negotiation [RFC8472], The Token Binding Protocol Version 1.0 [RFC8471] and Token Binding over HTTP [RFC8473]) to Access Tokens, Authorization Codes, Refresh Tokens, JWT Authorization Grants, and JWT Client Authentication. This cryptographically binds these tokens to a client's Token Binding key pair, possession of which is proven on the TLS connections over which the tokens are intended to be used. This use of Token Binding protects these tokens from man-in-the-middle and token export and replay attacks.

1.1. Requirements Notation and Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

1.2. Terminology

This specification uses the terms "Access Token", "Authorization Code", "Authorization Endpoint", "Authorization Server", "Client", "Protected Resource", "Refresh Token", and "Token Endpoint" defined by OAuth 2.0 [RFC6749], the terms "Claim" and "JSON Web Token (JWT)" defined by JSON Web Token (JWT) [JWT], the term "User Agent" defined by RFC 7230 [RFC7230], and the terms "Provided", "Referred", "Token

Binding" and "Token Binding ID" defined by Token Binding over HTTP [RFC8473].

2. Token Binding for Refresh Tokens

Token Binding of refresh tokens is a straightforward first-party scenario, applying term "first-party" as used in Token Binding over HTTP [RFC8473]. It cryptographically binds the refresh token to the client's Token Binding key pair, possession of which is proven on the TLS connections between the client and the token endpoint. This case is straightforward because the refresh token is both retrieved by the client from the token endpoint and sent by the client to the token endpoint. Unlike the federation use cases described in Token Binding over HTTP [RFC8473], Section 4, and the access token case described in the next section, only a single TLS connection is involved in the refresh token case.

Token Binding a refresh token requires that the authorization server do two things. First, when refresh token is sent to the client, the authorization server needs to remember the Provided Token Binding ID and remember its association with the issued refresh token. Second, when a token request containing a refresh token is received at the token endpoint, the authorization server needs to verify that the Provided Token Binding ID for the request matches the remembered Token Binding ID associated with the refresh token. If the Token Binding IDs do not match, the authorization server should return an error in response to the request.

How the authorization server remembers the association between the refresh token and the Token Binding ID is an implementation detail that beyond the scope of this specification. Some authorization servers will choose to store the Token Binding ID (or a cryptographic hash of it, such a SHA-256 hash [SHS]) in the refresh token itself, provided it is integrity-protected, thus reducing the amount of state to be kept by the server. Other authorization servers will add the Token Binding ID value (or a hash of it) to an internal data structure also containing other information about the refresh token, such as grant type information. These choices make no difference to the client, since the refresh token is opaque to it.

2.1. Example Token Binding for Refresh Tokens

This section provides an example of what the interactions around a Token Bound refresh token might look like, along with some details of the involved processing. Token Binding of refresh tokens is most useful for native application clients so the example has protocol elements typical of a native client flow. Extra line breaks in all examples are for display purposes only.

A native application client makes the following access token request with an authorization code using a TLS connection where Token Binding has been negotiated. A PKCE "code_verifier" is included because use of PKCE is considered best practice for native application clients [BCP212]. The base64url-encoded representation of the exported keying material (EKM) from that TLS connection is "p6ZuSwfl6pIe8es5KyeV76T4swZmQp0_awd27jHfrbo", which is needed to validate the Token Binding Message.

```
POST /as/token.oauth2 HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded
Sec-Token-Binding: AIkAAgBBQGto7hHRR0Y5nkOWqc9KNfwW95dEFmSI_tCZ_Cbl
7LWlt6Xjp3DbjiDjavGfiKP2HV_2JSE42VzmKOVVV8m7eqAAQOKiDKl0i0z6v4X5B
P7uc0pFestVZ42TTOdJmoHpji06Qq3jsCiCRSJx9ck2fWJYx8tLVXRZPATB3x6c24
ay0ZEAAA

grant_type=authorization_code&code=4bwcZesc7Xacc330ltc66Wxk8EAfp9j2
&code_verifier=2x6_ylS390-8V7jaT9wj.8qP9nKmYcf.V-rD9O4r_1
&client_id=example-native-client-id
```

Figure 1: Initial Request with Code

A refresh token is issued in response to the prior request. Although it looks like a typical response to the client, the authorization server has bound the refresh token to the Provided Token Binding ID from the encoded Token Binding message in the "Sec-Token-Binding" header of the request. In this example, that binding is done by saving the Token Binding ID alongside other information about the refresh token in some server side persistent storage. The base64url-encoded representation of that Token Binding ID is "AgBBQGto7hHRR0Y5nkOWqc9KNfwW95dEFmSI_tCZ_Cbl7LWlt6Xjp3DbjiDjavGfiKP2HV_2JSE42VzmKOVVV8m7eqA".

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

{
  "access_token": "EdRs7qMrLb167Z9fV2dcwoLTC",
  "refresh_token": "ACClZEIQTjW9arT9GOJGGd7QNwqOMmUYfsJTiv8his4",
  "token_type": "Bearer",
  "expires_in": 3600
}
```

Figure 2: Successful Response

When the access token expires, the client requests a new one with a refresh request to the token endpoint. In this example, the request is made on a new TLS connection so the EKM (base64url-encoded: "va-84Ukw4Zqfd7uWotFrAJda96WwgbdaPDX2knoOiAE") and signature in the Token Binding Message are different than in the initial request.

```
POST /as/token.oauth2 HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded
Sec-Token-Binding: AikAAgBBQGto7hHRR0Y5nkOWqc9KNfwW95dEFmSI_tCZ_Cbl
  7LWlt6Xjp3DbjiDJavGFikP2HV_2JSE42VzmKOVVV8m7eqAAQCpGbaG_YRf27qOra
  L0UT4fsKKjL6PukuOT00qzamoAXxOq7m_id7O3mLpnbsM7kwSxLi7iNHzzDgCAkP
  t3lHwAAA

refresh_token=ACClZEIQTjW9arT9GOJGGd7QNwqOMmUYfsJTiv8his4
&grant_type=refresh_token&client_id=example-native-client-id
```

Figure 3: Refresh Request

However, because the Token Binding ID is long-lived and may span multiple TLS sessions and connections, it is the same as in the initial request. That Token Binding ID is what the refresh token is bound to, so the authorization server is able to verify it and issue a new access token.

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

{
  "access_token": "bwcESCwC4yOCQ8iPsgcn117k7",
  "token_type": "Bearer",
  "expires_in": 3600
}
```

Figure 4: Successful Response

3. Token Binding for Access Tokens

Token Binding for access tokens cryptographically binds the access token to the client's Token Binding key pair, possession of which is proven on the TLS connections between the client and the protected resource. Token Binding is applied to access tokens in a similar manner to that described in Token Binding over HTTP [RFC8473], Section 4 (Federation Use Cases). It also builds upon the mechanisms for Token Binding of ID Tokens defined in OpenID Connect Token Bound Authentication 1.0 [OpenID.TokenBinding].

In the OpenID Connect [OpenID.Core] use case, HTTP redirects are used to pass information between the identity provider and the relying party; this HTTP redirect makes the Token Binding ID of the relying party available to the identity provider as the Referred Token Binding ID, information about which is then added to the ID Token. No such redirect occurs between the authorization server and the protected resource in the access token case; therefore, information about the Token Binding ID for the TLS connection between the client and the protected resource needs to be explicitly communicated by the client to the authorization server to achieve Token Binding of the access token.

This information is passed to the authorization server using the Referred Token Binding ID, just as in the ID Token case. The only difference is that the client needs to explicitly communicate the Token Binding ID of the TLS connection between the client and the protected resource to the Token Binding implementation so that it is sent as the Referred Token Binding ID in the request to the authorization server. This functionality provided by Token Binding implementations is described in Implementation Considerations of Token Binding over HTTP [RFC8473], Section 6.

Note that to obtain this Token Binding ID, the client may need to establish a TLS connection between itself and the protected resource prior to making the request to the authorization server so that the Provided Token Binding ID for the TLS connection to the protected resource can be obtained. How the client retrieves this Token Binding ID from the underlying Token Binding API is implementation and operating system specific. An alternative, if supported, is for the client to generate a Token Binding key to use for the protected resource, use the Token Binding ID for that key, and then later use that key when the TLS connection to the protected resource is established.

3.1. Access Tokens Issued from the Authorization Endpoint

For access tokens returned directly from the authorization endpoint, such as with the implicit grant defined in OAuth 2.0 [RFC6749], Section 4.2, the Token Binding ID of the client's TLS channel to the protected resource is sent with the authorization request as the Referred Token Binding ID in the "Sec-Token-Binding" header, and is used to Token Bind the access token.

Upon receiving the Referred Token Binding ID in an authorization request, the authorization server associates (Token Binds) the ID with the access token in a way that can be accessed by the protected resource. Such methods include embedding the Referred Token Binding ID (or a cryptographic hash of it) in the issued access token itself,

possibly using the syntax described in Section 3.4, or through token introspection as described in Section 3.5. The method for associating the referred token binding ID with the access token is determined by the authorization server and the protected resource, and is beyond the scope for this specification.

3.1.1. Example Access Token Issued from the Authorization Endpoint

This section provides an example of what the interactions around a Token Bound access token issued from the authorization endpoint might look like, along with some details of the involved processing. Extra line breaks in all examples are for display purposes only.

The client directs the user-agent to make the following HTTP request to the authorization endpoint. It is a typical authorization request that, because Token Binding was negotiated on the underlying TLS connection and the user-agent was signaled to reveal the Referred Token Binding, also includes the "Sec-Token-Binding" header with a Token Binding Message that contains both a Provided and Referred Token Binding. The base64url-encoded EKM from the TLS connection over which the request was made is "jI5UAYjs5XCPISUGQIwgcSrOiVIWq4fhLVIFTQ4nLxc".

```
GET /as/authorization.oauth2?response_type=token
    &client_id=example-client-id&state=rM8pZxG1c3gKy6rEbsD8s
    &redirect_uri=https%3A%2F%2Fclient%2Eexample%2Eorg%2Fcb HTTP/1.1
Host: server.example.com
Sec-Token-Binding: ARIAAGBBQIEE8mSMtDy2dj9EEBdXaQT9W3Rq1NS-jW8ebPoF
6FyL0jIfATVE55zlircgOTZmEglxeIrc3DsGegwjs4bhw14AQGKDLAXFFMyQkZegC
wlbTlqX3F9HTt-1JxFU_pil6ezka7qVRCpSF0BQLfSqlsxMbYfSSCJX1BDtrIL7PX
j__fUAAAECAEFAlBNUnP3te5WrwlEwiejEz0OpesmC5PElWc7kZ5nlLSqQTj1ciIp
5vQ30LLUCyM_a2BYTUPKtd5EdS-PalT4t6ABADgeizRa5NkTMuX4zOdC-R4cLNWVV
08lLu2Psko-UJLR_XAH4Q0H7-m0_nQR1zBN78nYMKPvHsz8L3zWKRvYXEgAA
```

Figure 5: Authorization Request

The authorization server issues an access token and delivers it to the client by redirecting the user-agent with the following HTTP response:

```
HTTP/1.1 302 Found
Location: https://client.example.org/cb#state=rM8pZxG1c3gKy6rEbsD8s
    &expires_in=3600&token_type=Bearer
    &access_token=eyJhbGciOiJIUzI1IiwiaXNjaWkiOiJ1IiwiaWF0IjoiMTYxMjM0MjM0In08xy5W5sQ
```

Figure 6: Authorization Response

The access token is bound to the Referred Token Binding ID from the authorization request, which when represented as a JWT, as described in Section 3.4, contains the SHA-256 hash of the Token Binding ID as the value of the "tbh" (token binding hash) member of the "cnf" (confirmation) claim. The confirmation claim portion of the JWT Claims Set is shown in the following figure.

```
{
  ...other claims omitted for brevity...
  "cnf":{
    "tbh": "vowQESa_MgbGJwIXaFm_BTN2QDPwh8PhuBm-EtUAqxc"
  }
}
```

Figure 7: Confirmation Claim

3.2. Access Tokens Issued from the Token Endpoint

For access tokens returned from the token endpoint, the Token Binding ID of the client's TLS channel to the protected resource is sent as the Referred Token Binding ID in the "Sec-Token-Binding" header, and is used to Token Bind the access token. This applies to all the grant types from OAuth 2.0 [RFC6749] using the token endpoint, including, but not limited to the refresh and authorization code token requests, as well as some extension grants, such as JWT assertion authorization grants [RFC7523].

Upon receiving the Referred Token Binding ID in a token request, the authorization server associates (Token Binds) the ID with the access token in a way that can be accessed by the protected resource. Such methods include embedding the Referred Token Binding ID (or a cryptographic hash of it) in the issued access token itself, possibly using the syntax described in Section 3.4, or through token introspection as described in Section 3.5. The method for associating the referred token binding ID with the access token is determined by the authorization server and the protected resource, and is beyond the scope for this specification.

Note that if the request results in a new refresh token being generated, it can be Token bound using the Provided Token Binding ID, per Section 2.

3.2.1. Example Access Token Issued from the Token Endpoint

This section provides an example of what the interactions around a Token Bound access token issued from the token endpoint might look like, along with some details of the involved processing. Extra line breaks in all examples are for display purposes only.

The client makes an access token request to the token endpoint and includes the "Sec-Token-Binding" header with a Token Binding Message that contains both Provided and Referred Token Binding IDs. The Provided Token Binding ID is used to validate the token binding of the refresh token in the request (and to Token Bind a new refresh token, if one is issued), and the Referred Token Binding ID is used to Token Bind the access token that is generated. The base64url-encoded EKM from the TLS connection over which the access token request was made is "4jTc5e1QpocqPTZ5l6jsb6pRP18IFKdwwPvasYjnl-E".

```
POST /as/token.oauth2 HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded
Sec-Token-Binding: ARIAAGBBQJFXJir2w4gbJ7grBx9uTYWIrs9V50-PW4ZijegQ
0LUM-_bGnGT6DizzUK-m5n3dQUIkeH7ybn6wb1C5dGyV_IIAQDDFToFrHt41Zppq7
u_SEMF_E-KimAB-HewWl2MvZzAQ9QKoWiJCLFicKjgtrlRrA2-jaJvoB8o51DTGXQ
ydWYkAAAECAEFaUc1GlyU83rqTGHEauloqvNwyOfDsdXzIyT_4xlFcldsMxjFkJac
IBJFGuYcccvnCaK_duFi3QKFENuwxql-H9ABAMCu7IjJOUA4IyE6YoEcfz9BMPQqw
M5M6hw4RZNQd58fsTCCslQE_NmNC19JXy4NkdkeZBxqvZGPr0y8QZ_bmAwAA

refresh_token=gZR_ZI8EAhLgWR-gWxBimbgZRzi_8EAhLgWRgWxBimbf
&grant_type=refresh_token&client_id=example-client-id
```

Figure 8: Access Token Request

The authorization server issues an access token bound to the Referred Token Binding ID and delivers it in a response the client.

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

{
  "access_token": "eyJhbGciOiJIUzI1NiIsImtp[...omitted...]cs29j5c3",
  "token_type": "Bearer",
  "expires_in": 3600
}
```

Figure 9: Response

The access token is bound to the Referred Token Binding ID of the access token request, which when represented as a JWT, as described in Section 3.4, contains the SHA-256 hash of the Token Binding ID as the value of the "tbh" (token binding hash) member of the "cnf" (confirmation) claim. The confirmation claim portion of the JWT Claims Set of the access token is shown in the following figure.

```

{
  ...other claims omitted for brevity...
  "cnf":{
    "tbh": "7NRBu9iDdJlYCTOqyeYuLxXv0blEA-yTpmGirAwKAws"
  }
}

```

Figure 10: Confirmation Claim

3.3. Protected Resource Token Binding Validation

Upon receiving a token bound access token, the protected resource validates the binding by comparing the Provided Token Binding ID to the Token Binding ID for the access token. Alternatively, cryptographic hashes of these Token Binding ID values can be compared. If the values do not match, the resource access attempt MUST be rejected with an error.

3.3.1. Example Protected Resource Request

For example, a protected resource request using the access token from Section 3.2.1 would look something like the following. The base64url-encoded EKM from the TLS connection over which the request was made is "7LsNP3BTlaHHdXdk6meEWjtSkipVLb7YS6iHp-JXmuE". The protected resource validates the binding by comparing the Provided Token Binding ID from the "Sec-Token-Binding" header to the token binding hash confirmation of the access token. Extra line breaks in the example are for display purposes only.

```

GET /api/stuff HTTP/1.1
Host: resource.example.org
Authorization: Bearer eyJhbGciOiJIJFZlIiwiaXNjaWkiOiJlcs29j5c3
Sec-Token-Binding: AIkAAgBBQLgtRpWFPN66kxhxGrtaKrzCmTHw7HV8yMk_-MdR
  XJXbDMYxZCWnCASRRrmHHHL5wmpP3bhYt0ChRDBsMapfh_QAQN1He3Ftj4Wa_S_fz
  ZVns4saLfj6aBoMSQW6rLsl9IivHze7LrGjKyCfPTKXjajebxp-TLPFZCc0JTqTY5
  _OMBAAAA

```

Figure 11: Protected Resource Request

3.4. Representing Token Binding in JWT Access Tokens

If the access token is represented as a JWT, the token binding information SHOULD be represented in the same way that it is in token bound OpenID Connect ID Tokens [OpenID.TokenBinding]. That specification defines the new JWT Confirmation Method RFC 7800 [RFC7800] member "tbh" (token binding hash) to represent the SHA-256 hash of a Token Binding ID in an ID Token. The value of the "tbh" member is the base64url encoding of the SHA-256 hash of the Token

Binding ID. All trailing pad '=' characters are omitted from the encoded value and no line breaks, whitespace, or other additional characters are included.

The following example demonstrates the JWT Claims Set of an access token containing the base64url encoding of the SHA-256 hash of a Token Binding ID as the value of the "tbh" (token binding hash) element in the "cnf" (confirmation) claim:

```
{
  "iss": "https://server.example.com",
  "aud": "https://resource.example.org",
  "sub": "brian@example.com",
  "iat": 1467324320,
  "exp": 1467324920,
  "cnf": {
    "tbh": "7NRBu9iDdJlYCTOqyeYuLxXv0bleA-yTpmGirAwKAws"
  }
}
```

Figure 12: JWT with Token Binding Hash Confirmation Claim

3.5. Representing Token Binding in Introspection Responses

OAuth 2.0 Token Introspection [RFC7662] defines a method for a protected resource to query an authorization server about the active state of an access token as well as to determine meta-information about the token.

For a token bound access token, the hash of the Token Binding ID to which the token is bound is conveyed to the protected resource as meta-information in a token introspection response. The hash is conveyed using same structure as the token binding hash confirmation method, described in Section 3.4, as a top-level member of the introspection response JSON. The protected resource compares that token binding hash to a hash of the provided Token Binding ID and rejects the request, if they do not match.

The following is an example of an introspection response for an active token bound access token with a "tbh" token binding hash confirmation method.

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

{
  "active": true,
  "iss": "https://server.example.com",
  "aud": "https://resource.example.org",
  "sub": "brian@example.com",
  "iat": 1467324320,
  "exp": 1467324920,
  "cnf": {
    "tbh": "7NRBu9iDdJlYCTOqyeYuLxXv0blEA-yTpmGirAwKAws"
  }
}
```

Figure 13: Example Introspection Response for a Token Bound Access Token

4. Token Binding Metadata

4.1. Token Binding Client Metadata

Clients supporting Token Binding that also support the OAuth 2.0 Dynamic Client Registration Protocol [RFC7591] use these metadata values to declare their support for Token Binding of access tokens and refresh tokens:

`client_access_token_token_binding_supported`
OPTIONAL. Boolean value specifying whether the client supports Token Binding of access tokens. If omitted, the default value is "false".

`client_refresh_token_token_binding_supported`
OPTIONAL. Boolean value specifying whether the client supports Token Binding of refresh tokens. If omitted, the default value is "false". Authorization servers MUST NOT Token Bind refresh tokens issued to a client that does not support Token Binding of refresh tokens, but MAY reject requests completely from such clients if token binding is required by authorization server policy by returning an OAuth error response.

4.2. Token Binding Authorization Server Metadata

Authorization servers supporting Token Binding that also support OAuth 2.0 Authorization Server Metadata [RFC8414] use these metadata values to declare their support for Token Binding of access tokens and refresh tokens:

`as_access_token_token_binding_supported`

OPTIONAL. Boolean value specifying whether the authorization server supports Token Binding of access tokens. If omitted, the default value is "false".

`as_refresh_token_token_binding_supported`

OPTIONAL. Boolean value specifying whether the authorization server supports Token Binding of refresh tokens. If omitted, the default value is "false".

5. Token Binding for Authorization Codes

There are two variations for Token Binding of an authorization code. One is appropriate for native application clients and the other for web server clients. The nature of where the various components reside for the different client types demands different methods of Token Binding the authorization code so that it is bound to a Token Binding key on the end user's device. This ensures that a lost or stolen authorization code cannot be successfully utilized from a different device. For native application clients, the code is bound to a Token Binding key pair that the native client itself possesses. For web server clients, the code is bound to a Token Binding key pair on the end user's browser. Both variations utilize the extensible framework of Proof Key for Code Exchange (PKCE) [RFC7636], which enables the client to show possession of a certain key when exchanging the authorization code for tokens. The following subsections individually describe each of the two PKCE methods respectively.

5.1. Native Application Clients

This section describes a PKCE method suitable for native application clients that cryptographically binds the authorization code to a Token Binding key pair on the client, which the client proves possession of on the TLS connection during the access token request containing the authorization code. The authorization code is bound to the Token Binding ID that the native application client uses to resolve the authorization code at the token endpoint. This binding ensures that the client that made the authorization request is the same client that is presenting the authorization code.

5.1.1. Code Challenge

As defined in Proof Key for Code Exchange [RFC7636], the client sends the code challenge as part of the OAuth 2.0 authorization request with the two additional parameters: "code_challenge" and "code_challenge_method".

For this Token Binding method of PKCE, "TB-S256" is used as the value of the "code_challenge_method" parameter.

The value of the "code_challenge" parameter is the base64url encoding (per Section 5 of [RFC4648] with all trailing padding ('=') characters omitted and without the inclusion of any line breaks or whitespace) of the SHA-256 hash of the Provided Token Binding ID that the client will use when calling the authorization server's token endpoint. Note that, prior to making the authorization request, the client may need to establish a TLS connection between itself and the authorization server's token endpoint in order to establish the appropriate Token Binding ID.

When the authorization server issues the authorization code in the authorization response, it associates the code challenge and method values with the authorization code so they can be verified later when the authorization code is presented in the access token request.

5.1.1.1. Example Code Challenge

For example, a native application client sends an authorization request by sending the user's browser to the authorization endpoint. The resulting HTTP request looks something like the following (with extra line breaks for display purposes only).

```
GET /as/authorization.oauth2?response_type=code
  &client_id=example-native-client-id&state=oUC2jyYtzRCrMyWrVnGj
  &code_challenge=rBlgOyMY4teiuJMDgOwkrpsAjPyI07D2WseM-dnq6eE
  &code_challenge_method=TB-S256 HTTP/1.1
Host: server.example.com
```

Figure 14: Authorization Request with PKCE Challenge

5.1.1.2. Code Verifier

Upon receipt of the authorization code, the client sends the access token request to the token endpoint. The Token Binding Protocol [RFC8471] is negotiated on the TLS connection between the client and the authorization server and the "Sec-Token-Binding" header, as defined in Token Binding over HTTP [RFC8473], is included in the access token request. The authorization server extracts the Provided Token Binding ID from the header value, hashes it with SHA-256, and compares it to the "code_challenge" value previously associated with the authorization code. If the values match, the token endpoint continues processing as normal (as defined by OAuth 2.0 [RFC6749]). If the values do not match, an error response indicating "invalid_grant" MUST be returned.

The "Sec-Token-Binding" header contains sufficient information for verification of the authorization code and its association to the original authorization request. However, PKCE [RFC7636] requires that a "code_verifier" parameter be sent with the access token request, so the static value "provided_tb" is used to meet that requirement and indicate that the Provided Token Binding ID is used for the verification.

5.1.2.1. Example Code Verifier

An example access token request, correlating to the authorization request in the previous example, to the token endpoint over a TLS connection for which Token Binding has been negotiated would look like the following (with extra line breaks for display purposes only). The base64url-encoded EKM from the TLS connection over which the request was made is

```
"pNVKtPuQFvylNYn000QowWrQKoeMkeX9H32hVuU71Bs".
```

```
POST /as/token.oauth2 HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded
Sec-Token-Binding: AIkAAgBBQEO09GRFP-LM0hoWw6-2i318BsuuUum5AL8bt1sz
  lr1EFfp5DMXMNW3O8WjcIXr2DKJnI4xnuGse6GywQd9RbD0AQJDb3xyo9PBxj8M6Y
  jLt-6OaxgDkyoBoTkrynNbLc8tJQ0JtXomKzBbj5qPtHDduXc6xz_lzvNpxSPxi42
  8m7wkAAA
```

```
grant_type=authorization_code&code=mJAReTWKX7zI3oHUNd4o3PeNqNqxKGp6
  &code_verifier=provided_tb&client_id=example-native-client-id
```

Figure 15: Token Request with PKCE Verifier

5.2. Web Server Clients

This section describes a PKCE method suitable for web server clients, which cryptographically binds the authorization code to a Token Binding key pair on the browser. The authorization code is bound to the Token Binding ID that the browser uses to deliver the authorization code to a web server client, which is sent to the authorization server as the Referred Token Binding ID during the authorization request. The web server client conveys the Token Binding ID to the authorization server when making the access token request containing the authorization code. This binding ensures that the authorization code cannot successfully be played or replayed to the web server client from a different browser than the one that made the authorization request.

5.2.1. Code Challenge

As defined in Proof Key for Code Exchange [RFC7636], the client sends the code challenge as part of the OAuth 2.0 Authorization Request with the two additional parameters: "code_challenge" and "code_challenge_method".

The client must send the authorization request through the browser such that the Token Binding ID established between the browser and itself is revealed to the authorization server's authorization endpoint as the Referred Token Binding ID. Typically, this is done with an HTTP redirection response and the "Include-Referred-Token-Binding-ID" header, as defined in Token Binding over HTTP [RFC8473], Section 5.3.

For this Token Binding method of PKCE, "referred_tb" is used for the value of the "code_challenge_method" parameter.

The value of the "code_challenge" parameter is "referred_tb". The static value for the required PKCE parameter indicates that the authorization code is to be bound to the Referred Token Binding ID from the Token Binding Message sent in the "Sec-Token-Binding" header of the authorization request.

When the authorization server issues the authorization code in the authorization response, it associates the Token Binding ID (or hash thereof) and code challenge method with the authorization code so they can be verified later when the authorization code is presented in the access token request.

5.2.1.1. Example Code Challenge

For example, the web server client sends the authorization request by redirecting the browser to the authorization endpoint. That HTTP redirection response looks like the following (with extra line breaks for display purposes only).

```
HTTP/1.1 302 Found
Location: https://server.example.com?response_type=code
        &client_id=example-web-client-id&state=P4FUFqYzslj3ffsYCP34d3
        &redirect_uri=https%3A%2F%2Fclient%2Eexample%2Eorg%2Fcb
        &code_challenge=referred_tb&code_challenge_method=referred_tb
Include-Referred-Token-Binding-ID: true
```

Figure 16: Redirect the Browser

The redirect includes the "Include-Referred-Token-Binding-ID" response header field that signals to the user-agent that it should

reveal, to the authorization server, the Token Binding ID used on the connection to the web server client. The resulting HTTP request to the authorization server looks something like the following (with extra line breaks for display purposes only). The base64url-encoded EKM from the TLS connection over which the request was made is "7gOdRzMhPeO-1YwZGmnVHyReN5vd2CxcSRBN69Ue4cI".

```
GET /as/authorization.oauth2?response_type=code
  &client_id=example-web-client-id&state=dryo8YFpWacBUPjhBf4Nvt51
  &redirect_uri=https%3A%2F%2Fclient%2Eexample%2Eorg%2Fcb
  &code_challenge=referred_tb
  &code_challenge_method=referred_tb HTTP/1.1
Host: server.example.com
Sec-Token-Binding: ARIAAGBBQB-XOPf5ePlf7ikATiAFEGOS503lPmRfkyymzdWw
  HCxl0njxC3D0E_OVfBNqrIQxzIfkF7tWby2ZfyaE6XpwTsAQBYqhFX78vMOgDX_F
  d_b2dlHyHlMmkIz8iMVB_YreM98OUaJFz5IB7PG9nZ11j58LoG5QhmQoI9NXYktKZ
  RXxrYAAAECAEFAdUFTnfQADknluDbQnvJEK6oQs38L92gv-KO-qlYadLoDIKe2h53
  hSiKwIP98iRj_unedkNkAMyg9e2mY4Gp7WwBAeDUOwaSXNz1e6gKohwN4SAZ5eNyx
  45Mh8VI4woLlBipLoqrJRok6dxFkWGHRMuBRocLGUj5PiOoxybQH_Tom3gAA
```

Figure 17: Authorization Request

5.2.2. Code Verifier

The web server client receives the authorization code from the browser and extracts the Provided Token Binding ID from the "Sec-Token-Binding" header of the request. The client sends the base64url-encoded (per Section 5 of [RFC4648] with all trailing padding ('=') characters omitted and without the inclusion of any line breaks or whitespace) Provided Token Binding ID as the value of the "code_verifier" parameter in the access token request to the authorization server's token endpoint. The authorization server compares the value of the "code_verifier" parameter to the Token Binding ID value previously associated with the authorization code. If the values match, the token endpoint continues processing as normal (as defined by OAuth 2.0 [RFC6749]). If the values do not match, an error response indicating "invalid_grant" MUST be returned.

5.2.2.1. Example Code Verifier

Continuing the example from the previous section, the authorization server sends the code to the web server client by redirecting the browser to the client's "redirect_uri", which results in the browser making a request like the following (with extra line breaks for display purposes only) to the web server client over a TLS channel for which Token Binding has been established. The base64url-encoded EKM from the TLS connection over which the request was made is "EzW60vyINbsb_tajt8ij3tV6cwY2KH-i8BdEMYXcNn0".

```
GET /cb?state=dryo8YFpWacbUPjhBf4Nvt5l&code=jwD3oOa5cQvvLc81bwc4CMw
Host: client.example.org
Sec-Token-Binding: AikAAgBBQHVBu530AA5J9bg20J7yRJOqELN_C_doL_ijvqpW
  GnS6AyCnloed4UoisCD_fIkY_7p3nZDZADMoPXtpmOBqelsAQEwgC9Zpg7QFCDBib
  6GlZki3MhH32KNfLefLJclvRlxE8l7OMfPLZHP2Woxh6rEtmgBcAABubEbTz7muNl
  Ln8uoAAA
```

Figure 18: Authorization Response to Web Server Client

The web server client takes the Provided Token Binding ID from the above request from the browser and sends it, base64url encoded, to the authorization server in the "code_verifier" parameter of the authorization code grant type request. Extra line breaks in the example request are for display purposes only.

```
POST /as/token.oauth2 HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded
Authorization: Basic b3JnLmV4YWlwbGUuY2xpZW50OmlldGY5OGNoaWNhZ28=

grant_type=authorization_code&code=jwD3oOa5cQvvLc81bwc4CMw
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Eorg%2Fcb
&client_id=example-web-client-id
&code_verifier=AgBBQHVBu530AA5J9bg20J7yRJOqELN_C_doL_ijv
qpWGnS6AyCnloed4UoisCD_fIkY_7p3nZDZADMoPXtpmOBqels
```

Figure 19: Exchange Authorization Code

6. Token Binding JWT Authorization Grants and Client Authentication

The JWT Profile for OAuth 2.0 Client Authentication and Authorization Grants [RFC7523] defines the use of bearer JWTs as a means for requesting an OAuth 2.0 access token as well as for client authentication. This section describes extensions to that specification enabling the application of Token Binding to JWT client authentication and JWT authorization grants.

6.1. JWT Format and Processing Requirements

In addition the requirements set forth in Section 3 of RFC 7523 [RFC7523], the following criteria must also be met for token bound JWTs used as authorization grants or for client authentication.

- o The JWT MUST contain a "cnf" (confirmation) claim with a "tbh" (token binding hash) member identifying the Token Binding ID of the Provided Token Binding used by the client on the TLS connection to the authorization server. The authorization server MUST reject any JWT that has a token binding hash confirmation

that does not match the corresponding hash of the Provided Token Binding ID from the "Sec-Token-Binding" header of the request.

6.2. Token Bound JWTs for Client Authentication

To use a token bound JWT for client authentication, the client uses the parameter values and encodings from Section 2.2 of RFC 7523 [RFC7523] with one exception: the value of the "client_assertion_type" is "urn:ietf:params:oauth:client-assertion-type:jwt-token-bound".

The "OAuth Token Endpoint Authentication Methods" registry [IANA.OAuth.Parameters] contains values, each of which specify a method of authenticating a client to the authorization server. The values are used to indicate supported and utilized client authentication methods in authorization server metadata, such as [OpenID.Discovery] and [RFC8414], and in OAuth 2.0 Dynamic Client Registration Protocol [RFC7591]. The values "private_key_jwt" and "client_secret_jwt" are designated by OpenID Connect [OpenID.Core] as authentication method values for bearer JWT client authentication using asymmetric and symmetric JWS [RFC7515] algorithms respectively. For Token Bound JWT for client authentication, this specification defines and registers the following authentication method values.

private_key_token_bound_jwt

Indicates that client authentication to the authorization server will occur with a Token Bound JWT, which is signed with a client's private key.

client_secret_token_bound_jwt

Indicates that client authentication to the authorization server will occur with a Token Bound JWT, which is integrity protected with a MAC using the octets of the UTF-8 representation of the client secret as the shared key.

Note that just as with the "private_key_jwt" and "client_secret_jwt" authentication methods, the "token_endpoint_auth_signing_alg" client registration parameter may be used to indicate the JWS algorithm used for signing the client authentication JWT for the authentication methods defined above.

6.3. Token Bound JWTs for as Authorization Grants

To use a token bound JWT for an authorization grant, the client uses the parameter values and encodings from Section 2.1 of RFC 7523 [RFC7523] with one exception: the value of the "grant_type" is "urn:ietf:params:oauth:grant-type:jwt-token-bound".

7. Security Considerations

7.1. Phasing in Token Binding

Many OAuth implementations will be deployed in situations in which not all participants support Token Binding. Any combination of the client, the authorization server, the protected resource, and the user agent may not yet support Token Binding, in which case it will not work end-to-end.

It is a context-dependent deployment choice whether to allow interactions to proceed in which Token Binding is not supported or whether to treat the omission of Token Binding at any step as a fatal error. Particularly in dynamic deployment environments in which End Users have choices of clients, authorization servers, protected resources, and/or user agents, it is recommended that, for some reasonable period of time during which Token Binding technology is being adopted, authorizations using one or more components that do not implement Token Binding be allowed to successfully proceed. This enables different components to be upgraded to supporting Token Binding at different times, providing a smooth transition path for phasing in Token Binding. However, when Token Binding has been performed, any Token Binding key mismatches **MUST** be treated as fatal errors.

In more controlled deployment environments where the participants in an authorization interaction are known or expected to support Token Binding and yet one or more of them does not use it, the authorization **SHOULD** be aborted with an error. For instance, an authorization server should reject a token request that does not include the "Sec-Token-Binding" header, if the request is from a client known to support Token Binding (via configuration or the "client_access_token_token_binding_supported" metadata parameter).

7.2. Binding of Refresh Tokens

Section 6 of RFC 6749 [RFC6749] requires that a refresh token be bound to the client to which it was issued and that, 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 when presenting the refresh token. As a result, for non-public clients, refresh tokens are indirectly bound to the client's credentials and cannot be used without the associated client authentication. Non-public clients then are afforded protections (equivalent to the strength of their authentication credentials) against unauthorized replay of refresh tokens and it is reasonable to not Token Bind refresh tokens for such clients while still Token Binding the issued access tokens. Refresh

tokens issued to public clients, however, do not have the benefit of such protections and authorization servers MAY elect to disallow public clients from registering or establishing configuration that would allow Token Bound access tokens but unbound refresh tokens.

Some web-based confidential clients implemented as distributed nodes may be perfectly capable of implementing access token binding (if the access token remains on the node it was bound to, the token binding keys would be locally available for that node to prove possession), but may struggle with refresh token binding due to an inability to share token binding key material between nodes. As confidential clients already have credentials which are required to use the refresh token, and those credentials should only ever be sent over TLS server-to-server between the client and the Token Endpoint, there is still value in token binding access tokens without token binding refresh tokens. Authorization servers SHOULD consider supporting access token binding without refresh token binding for confidential web clients as there are still security benefits to do so.

Clients MUST declare through dynamic (Section 4.1) or static registration information what types of token bound tokens they support to enable the server to bind tokens accordingly, taking into account any phase-in policies. Authorization servers MAY reject requests from any client who does not support token binding (by returning an OAuth error response) per their own security policies.

8. IANA Considerations

8.1. OAuth Dynamic Client Registration Metadata Registration

This specification registers the following client metadata definitions in the IANA "OAuth Dynamic Client Registration Metadata" registry [IANA.OAuth.Parameters] established by [RFC7591]:

8.1.1. Registry Contents

- o Client Metadata Name:
"client_access_token_token_binding_supported"
- o Client Metadata Description: Boolean value specifying whether the client supports Token Binding of access tokens
- o Change Controller: IESG
- o Specification Document(s): Section 4.1 of [[this specification]]

- o Client Metadata Name:
"client_refresh_token_token_binding_supported"
- o Client Metadata Description: Boolean value specifying whether the client supports Token Binding of refresh tokens
- o Change Controller: IESG

- o Specification Document(s): Section 4.1 of [[this specification]]

8.2. OAuth Authorization Server Metadata Registration

This specification registers the following metadata definitions in the IANA "OAuth Authorization Server Metadata" registry [IANA.OAuth.Parameters] established by [RFC8414]:

8.2.1. Registry Contents

- o Metadata Name: "as_access_token_token_binding_supported"
- o Metadata Description: Boolean value specifying whether the authorization server supports Token Binding of access tokens
- o Change Controller: IESG
- o Specification Document(s): Section 4.2 of [[this specification]]
- o Metadata Name: "as_refresh_token_token_binding_supported"
- o Metadata Description: Boolean value specifying whether the authorization server supports Token Binding of refresh tokens
- o Change Controller: IESG
- o Specification Document(s): Section 4.2 of [[this specification]]

8.3. PKCE Code Challenge Method Registration

This specification requests registration of the following Code Challenge Method Parameter Names in the IANA "PKCE Code Challenge Methods" registry [IANA.OAuth.Parameters] established by [RFC7636].

8.3.1. Registry Contents

- o Code Challenge Method Parameter Name: TB-S256
- o Change controller: IESG
- o Specification document(s): Section 5.1.1 of [[this specification]]
- o Code Challenge Method Parameter Name: referred_tb
- o Change controller: IESG
- o Specification document(s): Section 5.2.1 of [[this specification]]

9. Token Endpoint Authentication Method Registration

This specification requests registration of the following values in the IANA "OAuth Token Endpoint Authentication Methods" registry [IANA.OAuth.Parameters] established by [RFC7591].

9.1. Registry Contents

- o Token Endpoint Authentication Method Name:
"client_secret_token_bound_jwt"
- o Change Controller: IESG
- o Specification Document(s): Section 6 of [[this specification]]

- o Token Endpoint Authentication Method Name:
"private_key_token_bound_jwt"
- o Change Controller: IESG
- o Specification Document(s): Section 6 of [[this specification]]

10. Sub-Namespace Registrations

This specification requests registration of the following values in the IANA "OAuth URI" registry [IANA.OAuth.Parameters] established in An IETF URN Sub-Namespace for OAuth [RFC6755].

10.1. Registry Contents

- o URN: urn:ietf:params:oauth:grant-type:jwt-token-bound
- o Common Name: Token Bound JWT Grant Type for OAuth 2.0
- o Change controller: IESG
- o Specification Document: Section 6 of [[this specification]]

- o URN: urn:ietf:params:oauth:client-assertion-type:jwt-token-bound
- o Common Name: Token Bound JWT for OAuth 2.0 Client Authentication
- o Change controller: IESG
- o Specification Document: Section 6 of [[this specification]]

11. References

11.1. Normative References

- [IANA.OAuth.Parameters]
IANA, "OAuth Parameters",
<<http://www.iana.org/assignments/oauth-parameters>>.
- [JWT] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015,
<<http://tools.ietf.org/html/rfc7519>>.
- [OpenID.TokenBinding]
Jones, M., Bradley, J., and B. Campbell, "OpenID Connect Token Bound Authentication 1.0", October 2017,
<http://openid.net/specs/openid-connect-token-bound-authentication-1_0-03.html>.

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <<https://www.rfc-editor.org/info/rfc4648>>.
- [RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <<https://www.rfc-editor.org/info/rfc6749>>.
- [RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <<https://www.rfc-editor.org/info/rfc7230>>.
- [RFC7523] Jones, M., Campbell, B., and C. Mortimore, "JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants", RFC 7523, DOI 10.17487/RFC7523, May 2015, <<https://www.rfc-editor.org/info/rfc7523>>.
- [RFC7591] Richer, J., Ed., Jones, M., Bradley, J., Machulak, M., and P. Hunt, "OAuth 2.0 Dynamic Client Registration Protocol", RFC 7591, DOI 10.17487/RFC7591, July 2015, <<https://www.rfc-editor.org/info/rfc7591>>.
- [RFC7636] Sakimura, N., Ed., Bradley, J., and N. Agarwal, "Proof Key for Code Exchange by OAuth Public Clients", RFC 7636, DOI 10.17487/RFC7636, September 2015, <<https://www.rfc-editor.org/info/rfc7636>>.
- [RFC7662] Richer, J., Ed., "OAuth 2.0 Token Introspection", RFC 7662, DOI 10.17487/RFC7662, October 2015, <<https://www.rfc-editor.org/info/rfc7662>>.
- [RFC7800] Jones, M., Bradley, J., and H. Tschofenig, "Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs)", RFC 7800, DOI 10.17487/RFC7800, April 2016, <<https://www.rfc-editor.org/info/rfc7800>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.

- [RFC8414] Jones, M., Sakimura, N., and J. Bradley, "OAuth 2.0 Authorization Server Metadata", RFC 8414, DOI 10.17487/RFC8414, June 2018, <<https://www.rfc-editor.org/info/rfc8414>>.
- [RFC8471] Popov, A., Ed., Nystroem, M., Balfanz, D., and J. Hodges, "The Token Binding Protocol Version 1.0", RFC 8471, DOI 10.17487/RFC8471, October 2018, <<https://www.rfc-editor.org/info/rfc8471>>.
- [RFC8472] Popov, A., Ed., Nystroem, M., and D. Balfanz, "Transport Layer Security (TLS) Extension for Token Binding Protocol Negotiation", RFC 8472, DOI 10.17487/RFC8472, October 2018, <<https://www.rfc-editor.org/info/rfc8472>>.
- [RFC8473] Popov, A., Nystroem, M., Balfanz, D., Ed., Harper, N., and J. Hodges, "Token Binding over HTTP", RFC 8473, DOI 10.17487/RFC8473, October 2018, <<https://www.rfc-editor.org/info/rfc8473>>.
- [SHS] National Institute of Standards and Technology, "Secure Hash Standard (SHS)", FIPS PUB 180-4, March 2012, <<http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf>>.

11.2. Informative References

- [BCP212] Denniss, W. and J. Bradley, "OAuth 2.0 for Native Apps", BCP 212, RFC 8252, DOI 10.17487/RFC8252, October 2017, <<https://www.rfc-editor.org/info/rfc8252>>.
- [OpenID.Core] Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0", August 2015, <http://openid.net/specs/openid-connect-core-1_0.html>.
- [OpenID.Discovery] Sakimura, N., Bradley, J., Jones, M., and E. Jay, "OpenID Connect Discovery 1.0", August 2015, <http://openid.net/specs/openid-connect-discovery-1_0.html>.
- [RFC6755] Campbell, B. and H. Tschofenig, "An IETF URN Sub-Namespace for OAuth", RFC 6755, DOI 10.17487/RFC6755, October 2012, <<https://www.rfc-editor.org/info/rfc6755>>.

[RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <<https://www.rfc-editor.org/info/rfc7515>>.

Appendix A. Acknowledgements

This specification was developed within the OAuth Working Group under the chairmanship of Hannes Tschofenig and Rifaat Shekh-Yusef with Kathleen Moriarty, Eric Rescorla, and Benjamin Kaduk serving as Security Area Directors. Additionally, the following individuals contributed ideas, feedback, and wording that helped shape this specification: Dirk Balfanz, Andrei Popov, Justin Richer, and Nat Sakimura.

Appendix B. Document History

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

-08

- o Update reference to -03 of openid-connect-token-bound-authentication.
- o Update the references to the core token binding specs, which are now RFCs 8471, 8472, and 8473.
- o Update reference to AS metadata, which is now RFC 8414.
- o Add chairs and ADs to the Acknowledgements.

-07

- o Explicitly state that the base64url encoding of the tbh value doesn't include any trailing pad characters, line breaks, whitespace, etc.
- o Update to latest references for tokbind drafts and draft-ietf-oauth-discovery.
- o Update reference to Implementation Considerations in draft-ietf-tokbind-https, which is section 6 rather than 5.
- o Try to tweak text that references specific sections in other documents so that the HTML generated by the ietf tools doesn't link to the current document (based on old suggestion from Barry <https://www.ietf.org/mail-archive/web/jose/current/msg04571.html>).

-06

- o Use the boilerplate from RFC 8174.
- o Update reference for draft-ietf-tokbind-https to -12 and draft-ietf-oauth-discovery to -09.
- o Minor editorial fixes.

-05

- o State that authorization servers should not token bind refresh tokens issued to a client that doesn't support bound refresh tokens, which can be indicated by the "client_refresh_token_token_binding_supported" client metadata parameter.
- o Add Token Binding for JWT Authorization Grants and JWT Client Authentication.
- o Adjust the language around aborting authorizations in Phasing in Token Binding to be somewhat more general and not only about downgrades.
- o Remove reference to, and usage of, 'OAuth 2.0 Protected Resource Metadata', which is no longer a going concern.
- o Moved "Token Binding Metadata" section before "Token Binding for Authorization Codes" to be closer to the "Token Binding for Access Tokens" and "Token Binding for Refresh Tokens", to which it is more closely related.
- o Update references for draft-ietf-tokbind- negotiation(-10), protocol(-16), and https(-10), as well as draft-ietf-oauth-discovery(-07), and BCP212/RFC8252 OAuth 2.0 for Native Apps.

-04

- o Define how to convey token binding information of an access token via RFC 7662 OAuth 2.0 Token Introspection (note that the Introspection Response Registration request for cnf/Confirmation is in <https://tools.ietf.org/html/draft-ietf-oauth-mtls-02#section-4.3> which will likely be published and registered prior to this document).
- o Minor editorial fixes.
- o Added an open issue about needing to allow for web server clients to opt-out of having refresh tokens bound while still allowing for binding of access tokens (following from mention of the problem on

slide 16 of the presentation from Chicago
<https://www.ietf.org/proceedings/98/slides/slides-98-oauth-sessb-token-binding-00.pdf>).

-03

- o Fix a few mistakes in and around the examples that were noticed preparing the slides for IETF 98 Chicago.

-02

- o Added a section on Token Binding for authorization codes with one variation for native clients and one for web server clients.
- o Updated language to reflect that the binding is to the token binding key pair and that proof-of-possession of that key is done on the TLS connection.
- o Added a bunch of examples.
- o Added a few Open Issues so they are tracked in the document.
- o Updated the Token Binding and OAuth Metadata references.
- o Added William Denniss as an author.

-01

- o Changed Token Binding for access tokens to use the Referred Token Binding ID, now that the Implementation Considerations in the Token Binding HTTPS specification make it clear that implementations will enable using the Referred Token Binding ID.
- o Defined Protected Resource Metadata value.
- o Changed to use the more specific term "protected resource" instead of "resource server".

-00

- o Created the initial working group version from draft-jones-oauth-token-binding-00.

Authors' Addresses

Michael B. Jones
Microsoft

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

Brian Campbell
Ping Identity

Email: brian.d.campbell@gmail.com

John Bradley
Yubico

Email: ve7jtb@ve7jtb.com
URI: <http://www.thread-safe.com/>

William Denniss
Google
1600 Amphitheatre Pkwy
Mountain View, CA 94043
USA

Email: wdenniss@google.com
URI: <http://wdenniss.com/>

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: May 19, 2018

N. Sakimura
Nomura Research Institute
N. Matake
GREE, Inc.
S. Preibisch
CA Technologies
November 15, 2017

OAuth Response Metadata
draft-sakimura-oauth-meta-08

Abstract

This specification defines an extensible metadata framework that may be inserted into the OAuth 2.0 responses to assist the clients to process those responses. It is expressed either as a link header, or query parameters. It will allow the client to learn the metadata about the particular response. For example, the client can learn where the members in the response could be used, what is the characteristics of the payload is, how it should be processed, and so on. Since they are just additional response header/query parameters, any client that does not understand this extension should not break and work normally while supporting clients can utilize the metadata to take the advantage of the extension.

Status of This Memo

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

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

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

This Internet-Draft will expire on May 19, 2018.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction	2
1.1. Requirements Notation and Conventions	3
1.2. terminology	3
2. Resource Endpoint Resonse	3
3. Token Endpoint Response	4
4. Authorization Endpoint HEAD response	5
5. Authorization Response	6
6. IANA Considerations	6
6.1. Link Type Registration	6
7. Security Considerations	7
7.1. Authorization Response Query Parameter Tampering by a Bad User	7
8. Acknowledgements	7
9. Document History	7
10. References	9
10.1. Normative References	9
10.2. Informational References	9
Authors' Addresses	10

1. Introduction

Although OAuth 2.0 [RFC6749] has been known for its REST friendliness, OAuth itself is not RESTful, as it heavily relies on out-of-band information to drive the interactions. This situation can be eased by hypertext-enabling the endpoint responses through the introduction of data structure that represents such hypertext and other metadata.

Hyper-text enabling the OAuth responses has many advantages. For example,

- o The protected resource can tell which authorization servers it supports.
- o Permissioned resource discovery: It is possible to tell the client which resource endpoint it should use. This has a privacy

advantage. The location of the resource by itself may be a sensitive information as its location may reveal information about the resource owner. Therefore, it may be sensible to tell the location only after the user consent.

- o It is possible to give a hint on the processing of the payload.
- o It will be resitant to IdP Mix-up attack.
- o It will be resitant to Code Phishing Attack.

This specification defines methods to represent such metadata in the authorization and token responses.

1.1. Requirements Notation and Conventions

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

1.2. terminology

This specification uses the terms "Access Token", "Authorization Code", "Authorization Endpoint", "Authorization Grant", "Authorization Server", "Client", "Client Authentication", "Protected Resource", "Refresh Token", and "Token Endpoint" defined by OAuth 2.0 [RFC6749].

2. Resource Endpoint Resonse

Resource Endpoints that implement this specification returns the following link relation (rel) and the corresonding URI value as defined in [RFC5988] in the response header. The response header can be returned in response to HEAD, GET, or POST request to the endpoint.

duri The URI of the corresponding authorization server's discovery document, from which the client can learn the server capabilities and endpoints.

auri The URI of the corresponding Authorization Endpoint URI.

A typical example of the use of these header values are in the case of the Client accessing the protected resource without a propoer credential. For example, in the case of an [RFC6750] protected resource, the unauthorized access may result in a response header that includes both WWW-Authenticate header as well as the Web Linking

header indicating either the Authorization Endpoint URI or the discovery document URI.

There is no cardinality restriction on relations put in place by [RFC5988]. Therefore, the resource can respond with multiple Authorization Endpoint URI or Discovery Document URIs from which the Client may choose the appropriate one.

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="example"
Link: <https://example.com/.well-known/openid-configuration>; rel="duri",
      <https://example.net/.well-known/openid-configuration>; rel="duri",
      <https://example.com/authz/>; rel="auri",
      <https://example.net/az/>; rel="auri",
      <https://example.com/payment-upon-trial-expiry>; rel="payments"
```

3. Token Endpoint Response

Token Endpoints that implements this specification returns the following link relation (rel) and the corresponding URI value as defined in [RFC5988] in the Access Token Response defined in [RFC6749].

ruri Resource URI. The value of this parameter is the URI of the Resource Endpoint that the Access Token is supposed to be used at. If this value is present, the client MUST NOT send the Access Token to any other URI.

turi Token Endpoint URI. The value of this parameter is the URI of the Token Endpoint that the Refresh Token can be sent to obtain a new Access Token. If this value is present, then the client MUST NOT send the refresh token to any other places.

Any other rels that are registered in Link Relation Type Registry defined in [RFC5988] registry can be used.

There is no cardinality restriction on relations put in place by [RFC5988]. Therefore, the Token Endpoing can respond with multiple Resource Endpoint URI or Discovery Document URIs from which the Client may choose the appropriate one.

Following is an example of an HTTPS response.

```
HTTP/1.1 200 OK
Link: <https://example.com/userinfo>; rel="ruri",
      <https://example.net/photostream/>; rel="ruri",
      <https://example.com/payment-upon-trial-expiry>; rel="payments"
Content-Type: application/JSON; charset=utf-8

{
    "access_token": "aCeSsToKen"
}
```

4. Authorization Endpoint HEAD response

Authorization Endpoints that implements this specification returns the following link relation (rel) and the corresponding URI value as defined in [RFC5988] in the response to the HEAD request.

auri The canonical URI of the Authorization Endpoint URI.

turi Token Endpoint URI. The value of this parameter is the URI of the Token Endpoint that the Authorization Code can be sent to obtain the Access Token.

duri The URI of discovery document, from which the client can learn the server capabilities and endpoints.

ruri Resource URI. The value of this parameter is the URI of the Resource Endpoint that the Access Token can be used at. If this parameter is specified, the client MUST NOT send the Access Token to any other URIs than the value of this parameter.

There is no cardinality restriction on relations put in place by [RFC5988]. Therefore, the Authorization Endpoint can respond with multiple Endpoint URIs with a same relation type from which the Client may choose the appropriate one.

Following is an example of an HTTPS response.

```
HTTP/1.1 200 OK
Link: <https://example.com/.well-known/openid-configuration>; rel="duri",
      <https://example.net/.well-known/openid-configuration>; rel="duri",
      <https://example.com/payment-upon-trial-expiry>; rel="payments"
Content-Type: application/JSON; charset=utf-8
```


5. Authorization Response

While [RFC5988] defines a useful way of conveying link relations, it cannot be utilized for a redirect based communication such as the authorization response of OAuth 2.0. This section defines a way to return a limited set of those link relations as query parameters so that it can be conveyed over the redirection.

The authorization response of the implementation of this specification may return the following query parameter in the redirect URI.

turi Token Endpoint URI. The value of this parameter is the URI of the Token Endpoint that the Authorization Code can be sent to obtain the Access Token. If this parameter is specified, the client MUST check that the value of **turi** matches exactly with the pre-registered token endpoint URI of the Authorization Server that the session recovered from the state variable points to. The client MUST NOT send the code to any other URIs than the value of this parameter.

ruri Resource URI. The value of this parameter is the URI of the Resource Endpoint that the Access Token can be used at. If this parameter is specified, the client MUST NOT send the Access Token to any other URIs than the value of this parameter.

As long as the link relation type string does not collide with the underlying protocol parameters, they can also be specified as a query parameter. The value MUST be encoded in application/x-www-form-urlencoded.

The following is an example of such response. Line breaks are for display purposes only.

```
HTTP/1.1 302 Found
Location: https://client.example.com/cb?code=Splxl0BeZQQYbYS6WxSbIA
        &turi=https%3A%2F%2Fexample.com%2Ftoken
        &state=xyz
```

6. IANA Considerations

6.1. Link Type Registration

Pursuant to [RFC5988], the following link type registrations [[will be]] registered by mail to link-relations@ietf.org.

- o Relation Name: **turi**

- o Description: An OAuth 2.0 Token Endpoint specified in section 3.2 of [RFC6749].
- o Reference: This specification
- o Relation Name: ruri
- o Description: An OAuth 2.0 Resource Endpoint specified in section 3.2 of [RFC6750].
- o Reference: This specification

7. Security Considerations

7.1. Authorization Response Query Parameter Tampering by a Bad User

The query response parameters may be tampered by the man-in-the-browser. It can also be tampered by a malicious user. In general, anything that comes via the browser/user-agent can be tainted and untrusted.

This specification mandates the turi check so that tampering of turi by the malicious user will be detected. It does not mandate ruri check as the user can get the Access Token and send it to anywhere he wants anyways when it is returned to the browser.

However, other parameters are not protected. The Client MUST treat them tainted and implement its own check rules for each parameters.

To solve this "Tampering by bad user", either HMAC(concat(params)) need to be sent with them or have all of them inside the JWS.

8. Acknowledgements

Members of OAuth WG helped to form this specification. Notably: Hannes tschofenig, John Bradley, Justin Richer, Kaoru Maeda, Masashi Kurabayashi, Michael B. Jones, Phil Hunt, William Dennis, James Manger, (add yourselves).

9. Document History

-07

- o Added note that there is no cardinality requirements so that multiple endpoints can be returned by repeating a same rel.
- o Added resource endpoint response.

-06

- o Removed duri description from token response as it is not needed.
- o Made the processing instruction more precise.
- o Added RFC5988 defined link relation type in the example.
- o Swaped the order of the authorization response and token response. Now, token response gets explained first so that the reader will grasp the basic concept according to RFC5988 and regards the authorization response extension as a mapping of RFC5988 into query parameter form.

-05

- o Factored out JSON Meta and now using query param and Web Linking.

-04

- o Date refresh.

-03

- o Date refresh.

-02

- o Added Mike Kelly as an author.
- o xref fix.
- o Introduced "operations" as in draft-ietf-scim-api-00#section-3.5.
- o Updated the informative reference to HAL.
- o Added description to OAuth Token Endpoint hrefs.
- o Added content-type to the example.
- o Added Area and Working Group.

-01

- o Some format changes, reference fix, and typo fixes.
- o Changed 'items' to 'elements' to match the JSON terminology.

-00

o Initial Draft

10. References

10.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [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, DOI 10.17487/RFC2616, June 1999, <<https://www.rfc-editor.org/info/rfc2616>>.
- [RFC5988] Nottingham, M., "Web Linking", RFC 5988, DOI 10.17487/RFC5988, October 2010, <<https://www.rfc-editor.org/info/rfc5988>>.
- [RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <<https://www.rfc-editor.org/info/rfc6749>>.
- [RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, <<https://www.rfc-editor.org/info/rfc6750>>.

10.2. Informational References

- [HAL] Kelly, M., "JSON Hypermedia API Language", February 2013.
- [oauth-lrdd] Mills, W., "Link Type Registrations for OAuth 2", October 2012.
- [RFC4627] Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)", RFC 4627, DOI 10.17487/RFC4627, July 2006, <<https://www.rfc-editor.org/info/rfc4627>>.

[RFC6570] Gregorio, J., Fielding, R., Hadley, M., Nottingham, M.,
and D. Orchard, "URI Template", RFC 6570,
DOI 10.17487/RFC6570, March 2012,
<<https://www.rfc-editor.org/info/rfc6570>>.

Authors' Addresses

Nat Sakimura
Nomura Research Institute

Email: sakimura@gmail.com

Nov Matake
GREE, Inc.

Email: nov@matake.jp
URI: <http://matake.jp>

Sascha Preibisch
CA Technologies

Email: Sascha.Preibisch@gmail.com

OAuth
Internet-Draft
Updates: 6750 (if approved)
Intended status: Standards Track
Expires: August 22, 2013

H. Tschofenig, Ed.
Nokia Siemens Networks
February 18, 2013

OAuth 2.0: Audience Information
draft-tschofenig-oauth-audience-00.txt

Abstract

The OAuth 2.0 Bearer Token specification allows any party in possession of a bearer token to get access to the associated resources (without demonstrating possession of a cryptographic key). To prevent misuse, two important security assumptions must hold: bearer tokens must be protected from disclosure in storage and in transport and the access token must only be valid for use with a specific resource server (the audience) and with a specific scope.

This document defines a new header that is used by the client to indicate what resource server, as the intended recipient, it wants to access. This information is subsequently also communicated by the authorization server securely to the resource server, for example within the audience field of the 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 August 22, 2013.

Copyright Notice

Copyright (c) 2013 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction	3
2. Terminology	4
3. Audience Parameter	5
4. Processing Instructions	6
5. Security Considerations	8
6. IANA Considerations	9
7. Acknowledgments	10
8. References	11
8.1. Normative References	11
8.2. Informative References	11
Author's Address	12

1. Introduction

The OAuth 2.0 Bearer Token specification [1] allows any party in possession of a bearer token to get access to the associated resources (without demonstrating possession of a cryptographic key). To prevent misuse, two important security assumptions must hold: bearer tokens must be protected from disclosure in storage and in transport and the access token must only be valid for use with a specific resource server with a specific scope.

[1] describes this requirement in the following way:

"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."

In general, if there is an authorization restriction then the respective parties must be aware of this restriction. In our case, the respective parties are authorization server (who has a trust relationship with the resource owner to accept for reject requests for data sharing and creates the access token), the client (who initiates the access to the protected resource), and the resource server (who protects the access to the resource and grants only access to those clients who have been approved by the authorization server).

Unfortunately, at the time of writing of [1] the access token format was still in early stages of the design and more details about how to communicate the audience information between the different parties was left unspecified. This document defines a new field for usage with OAuth 2.0. Note that it is not only useful for OAuth 2.0 bearer tokens but also for MAC tokens [5]: the authorization server needs to be told which resource server has to obtain the session key securely in order for the security properties to hold.

Restricting the usage of access tokens is important for several reasons: First, a stolen access token cannot be used with resource servers it has not been created for. Second, if the scope is included it cannot be used for requesting access to resources that exceed the indicated permissions. A resource server, who obtains an access token legitimately, cannot access resources on behalf of the resource owner at other resource servers.

2. Terminology

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 [2].

3. Audience Parameter

When the client interacts with the resource server it constructs the access token request to the token endpoint by adding the audience parameter using the "application/x-www-form-urlencoded" format with a character encoding of UTF-8 in the HTTP request entity-body.

The audience URI MUST be an absolute URI as defined by Section 4.3 of [3]. It MAY include an "application/x-www-form-urlencoded" formatted query component (Section 3.4 of [3]). The URI MUST NOT include a fragment component.

The ABNF syntax is defined as follows where by the "URI-reference" definition is taken from [3]:

audience = URI-reference

[QUESTION: Is it OK to just assume a URI here as the audience identifier?]

4. Processing Instructions

Step (0): As an initial step the client typically determines the resource server it wants to interact with, for example, as part of a discovery procedure.

[QUESTION: Should we talk about WebFinger or SWD to be more specific?]

Step (1): The client starts the OAuth 2.0 protocol interaction based on the selected grant type.

Step (2): When the client interacts with the token endpoint to obtain an access token it MUST populate the newly defined 'audience' parameter with the information obtained in step (0).

Step (2): The resource server who obtains the request needs to parse it to determine whether the provided audience value matches any of the authorized resource servers it has a relationship with. If the authorization server fails to parse the provided value it MUST reject the request using an error response with the error code "invalid_request". If the authorization server does not consider the resource server acceptable then it MUST return an error response with the error code "access_denied". In both cases additional error information may be provided via the error_description, and the error_uri parameters. If the request has, however, been verified successfully then the authorization server MUST include the audience claim into the access token with the value copied from the audience field provided by the client. In case the access token is encoded using the JSON Web Token format [6] the "aud" claim MUST be used. The access token MUST be protected against modification by protecting it with either a digital signature or a keyed message digest. The authorization server returns the access token to the client, as specified in [4].

[QUESTION: Should we just focus on a JSON-based encoding of the access token since it is the only specified format?]

Step (3): The client follows the OAuth 2.0 specification [4] and the specification relevant for the selected token type (e.g., the bearer token specification) to interact with the resource server to make a request to the protected resource with the attached access token.

Step (4): When the resource server receives the access token it verifies it according to chosen access token encoding. For example, in case the JSON Web Token format is used then it must

adhere to the guidance in [6]. In any case, the resource server MUST verify whether the URI contained in the "aud" claim matches it's own. If the comparison fails the resource server MUST return an error to the client.

[NOTE: More guidance is required in [6] regarding the matching procedure.]

5. Security Considerations

The sole purpose of this document is to extend the OAuth 2.0 protocol to improve security.

6. IANA Considerations

This document requires IANA to add a new value to the OAuth parameters registry:

- o Parameter name: audience
- o Parameter usage location: token request
- o Change controller: IETF
- o Specification document(s): [[This document.]

7. Acknowledgments

The author would like to thank Leif Johansson, and Eve Maler for their feedback.

8. References

8.1. Normative References

- [1] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, October 2012.
- [2] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [3] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.
- [4] Hardt, D., "The OAuth 2.0 Authorization Framework", RFC 6749, October 2012.

8.2. Informative References

- [5] Richer, J., Mills, W., and H. Tschofenig, "OAuth 2.0 Message Authentication Code (MAC) Tokens", draft-ietf-oauth-v2-http-mac-02 (work in progress), November 2012.
- [6] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", draft-ietf-oauth-json-web-token-06 (work in progress), December 2012.

Author's Address

Hannes Tschofenig (editor)
Nokia Siemens Networks
Linnoitustie 6
Espoo 02600
Finland

Phone: +358 (50) 4871445
Email: Hannes.Tschofenig@gmx.net
URI: <http://www.tschofenig.priv.at>

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: May 17, 2018

W. Denniss
Google
KM. McGuinness
Okta
J. Bradley
Yubico
November 13, 2017

OAuth 2.0 Device Posture Signals
draft-wdenniss-oauth-device-posture-01

Abstract

Enterprise and security focused OAuth providers typically want additional signals to confirm user presence when users return to previously authorized apps. Rather than requiring a full reauthentication, or require enrollment in a mobile device management solution, some authorization servers may be willing to accept device posture signals from the app, like the fact that device has a lock screen, as confirmation of user presence. This document details how OAuth native app clients can communicate device posture signals to OAuth providers.

Status of This Memo

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

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

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

This Internet-Draft will expire on May 17, 2018.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

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

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

Table of Contents

1. Introduction	2
2. Notational Conventions	3
3. Terminology	3
4. Device Posture Signal Dictionary	3
5. Authorization Request Device Posture Hint	5
6. Token Endpoint Device Posture Enforcement	5
7. Security Considerations	6
7.1. Device Posture Scope	6
7.2. Spoofed Devices	6
7.3. App Trustworthiness	6
8. IANA Considerations	7
8.1. OAuth Parameters Registration	7
8.1.1. Registry Contents	7
8.2. OAuth Extensions Error Registration	7
8.2.1. Registry Contents	7
8.3. Device Posture Keys Registry	7
8.3.1. Registration Template	8
8.3.2. Initial Registry Contents	9
9. References	10
9.1. Normative References	10
9.2. Informative References	11
Appendix A. Acknowledgements	11
Authors' Addresses	11

1. Introduction

Users who follow strong security practices on their devices - such as configuring screen locks, and not enabling admin privileges (commonly known as "rooting" or "jailbreaking") - shouldn't need to reauthenticate frequently to the individual apps on their device.

This specification details how apps can send device posture signals to the OAuth Token Endpoint, enabling it to enforce device policy compliance, and avoid the need for reauthentication in some cases.

It is designed to provide a mechanism for honest apps to communicate device posture. By itself it doesn't protect against malicious

users, dishonest apps, or compromised devices, but the signal format described could carry signals that do.

2. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in Key words for use in RFCs to Indicate Requirement Levels [RFC2119]. If these words are used without being spelled in uppercase then they are to be interpreted with their normal natural language meanings.

3. Terminology

In addition to the terms defined in referenced specifications, this document uses the following terms:

MDM

Mobile Device Management.

EMM

Enterprise Mobility Management.

4. Device Posture Signal Dictionary

The device posture is a dictionary of signals asserted by the app about the device. The structure is sent as an added parameter in several places during the OAuth flow, as documented in the subsequent sections.

All device posture keys are OPTIONAL and MUST only be set when the attribute can be obtained by the app. The standard attribute keys are as follows:

screen_lock

Boolean. True if the device has a screen lock, such as a pin, pattern biometric, etc.

root_privileges

Boolean. True if user apps can access root device privileges. For mobile operating systems, known as "jailbreaking" on iOS and "rooting" on Android.

full_disk_encryption

Boolean. True if data stored on the device is fully encrypted at rest.

device_id

String. A unique identifier for the device.

device_os

String. The name of the operating system running on the device such as "iOS" or "Android".

device_os_version

String. The current version of the operating system.

device_vendor

String. The vendor of the device such as "Apple" or "Google".

device_model

String. The model of the device such as "iPhone X" or "Pixel 2".

device_attestation

Dictionary. An attestation from the operating system, containing a signed-statement about the device and/or the app. The format is a dictionary, the specifics of which depends on the operating system.

app_id

String. The platform-specific identifier (URI) for the application. For Android, the format of the URI is android:apk-key-hash:<hash-of-apk-signing-cert>. For iOS, the format of URI is ios:bundle-id:<ios-bundle-id-of-app>.

app_managed

Boolean. True if the app is managed by a MDM/EMM system.

An example device posture dictionary:

```
{
  "screen_lock": true,
  "root_privileges": false,
  "full_disk_encryption": true,
  "device_id": "6bddele8-0667-40f9-9993-16aa52ee6b38",
  "device_os": "iOS",
  "device_os_version": "11.1",
  "device_vendor": "Apple",
  "device_model": "iPhone X",
  "app_id": "ios:bundle-id:com.example.myapp",
  "app_managed": false
}
```

5. Authorization Request Device Posture Hint

Clients MAY send the device posture signal dictionary to the authorization server in the authorization request. These signals, except for those that are signed and bound to the device are susceptible to client-side modification by end-users. While untrusted, such signals can still be used as hints by the authorization server to present a better user experience, like informing the user they need a lock screen.

Error encountered during authorization can be displayed to the user in the browser making this a more user friendly way to instruct the user on how to move their device into conformance. The token endpoint (on which errors are less user-friendly as there's no user agent), can then enforce the restrictions per Section 6.

The following parameters are added to the OAuth 2.0 Authorization Request or the OpenID Connect Authorization Request:

`device_posture_hint`

JSON String. URL-encoded JSON dictionary, contains the Device Posture Signals defined in Section 4.

A AS receiving `device_posture_hint` may pass the value on to upstream OpenID Connect IDP using the same parameter to enable the Authentication Server to make policy decisions.

6. Token Endpoint Device Posture Enforcement

Clients that follow this specification MUST send the device posture signals on every request to the token endpoint.

Token Endpoints SHOULD verify that the posture conforms to their requirements and act accordingly.

The following parameters are added to all requests to the Token Endpoint:

`device_posture`

JSON String. URL-encoded JSON dictionary, contains the Device Posture Signals defined in Section 4.

The app MUST obtain fresh device posture information before every request to the Token Endpoint, and MUST NOT include stale information (rather, it should drop any signals it cannot freshly obtain).

For token refresh requests, where the device posture has been previously communicated, if an attribute is missing, the Token

Endpoint may choose to use the previous value, based on it's own policy and freshness requirements.

If the policy does not meet requirements, the Token Endpoint SHOULD return the following error code:

`device_posture_invalid`

Error indicating that the device posture does not meet requirements. The error description SHOULD contain details on why this is the case.

7. Security Considerations

7.1. Device Posture Scope

This specification is designed to help authorization servers enforce security policy (like requiring a lock screen) on end-users. The intent is to enforce restrictions on honest users, to force them to follow security practices set out by the authorization server. By itself, it offers no protection against malicious users, dishonest apps, or compromised devices.

Combined with other technologies like device-based attestations and token binding may enable such protection, and this specification could be used to transmit secure signals, but that topic is out of scope for this specification.

7.2. Spoofed Devices

It is possible to at a device level completely spoof the device posture. Even statements signed by the operating system are vulnerable to spoofing, as it's possible a statement from the real device can be replayed on a spoofed device, unless such statements include a binding to the device itself. Per Section 7.1, this topic is out of scope for this specification.

7.3. App Trustworthiness

This specification is designed to allow trusted apps to report device posture to the authorization server to help the server enforce security policy on end-users. It does not by itself force apps to be honest, or genuine. Genuine apps (i.e. apps not lying about their client ID) might be dishonest about the device posture, and apps that are normally honest, could be spoofed, unless anti-spoofing countermeasures that are out of scope of this specification are employed.

8. IANA Considerations

8.1. OAuth Parameters Registration

This specification registers the following value in the IANA "OAuth Parameters" registry [IANA.OAuth.Parameters] established by [RFC6749].

8.1.1. Registry Contents

- o Parameter name: device_posture_hint
- o Parameter usage location: authorization request
- o Change controller: IESG
- o Specification document(s): Section 5 of [[this specification]]

- o Parameter name: device_posture
- o Parameter usage location: token request
- o Change controller: IESG
- o Specification document(s): Section 6 of [[this specification]]

8.2. OAuth Extensions Error Registration

This specification registers the following error in the IANA "OAuth Extensions Error Registry" [IANA.OAuth.Parameters] established by [RFC6749].

8.2.1. Registry Contents

- o Error name: device_posture_invalid
- o Error usage location: authorization response, token error response
- o Related protocol extension: resource parameter
- o Change controller: IESG
- o Specification document(s): Section 6 of [[this specification]]

8.3. Device Posture Keys Registry

This specification establishes the IANA "Device Posture Keys" registry for Device Posture Dictionary keys. The registry records the Device Posture key and a reference to the specification that defines it. This specification registers the Device Posture keys defined in Section 4.

Keys are registered on an Expert Review [RFC5226] basis after a three-week review period on the oauth-reg-review@ietf.org mailing list, on the advice of one or more Designated Experts.

Registration requests sent to the mailing list for review should use an appropriate subject (e.g., "Request to register Device Posture Key: screen_lock").

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

Criteria that should be applied by the Designated Experts includes determining whether the proposed registration duplicates existing functionality, whether it is likely to be of general applicability or whether it is useful only for a single application, whether the value is actually being used, and whether the registration description is clear.

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

It is suggested that the same Designated Experts evaluate these registration requests as those who evaluate registration requests for the IANA "OAuth Parameters" registry [IANA.OAuth.Parameters].

8.3.1. Registration Template

Device Posture Signal Key:

The key name requested (e.g., "screen_lock"). Names may not match other registered names in a case-insensitive manner unless the Designated Experts state that there is a compelling reason to allow an exception.

Device Posture Signal Key Description:

Brief description of the device posture signal (e.g., "Screen lock active").

Change Controller:

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

Specification Document(s):

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

8.3.2. Initial Registry Contents

- o Device Posture Signal Key: "screen_lock"
- o Device Posture Signal Key Description: Boolean. 'true' when the device has a screen lock enabled.
- o Change Controller: IESG
- o Specification Document(s): Section 4 of [[this specification]]

- o Device Posture Signal Key: "root_privileges"
- o Device Posture Signal Key Description: Boolean. True if user apps can access root device privileges.
- o Change Controller: IESG
- o Specification Document(s): Section 4 of [[this specification]]

- o Device Posture Signal Key: "full_disk_encryption"
- o Device Posture Signal Key Description: Boolean. True if data stored on the device is fully encrypted at rest.
- o Change Controller: IESG
- o Specification Document(s): Section 4 of [[this specification]]

- o Device Posture Signal Key: "device_id"
- o Device Posture Signal Key Description: String. A unique identifier for the device.
- o Change Controller: IESG
- o Specification Document(s): Section 4 of [[this specification]]

- o Device Posture Signal Key: "device_os"
- o Device Posture Signal Key Description: String. The name of the operating system running on the device such as "iOS" or "Android".
- o Change Controller: IESG
- o Specification Document(s): Section 4 of [[this specification]]

- o Device Posture Signal Key: "device_os_version"
- o Device Posture Signal Key Description: String. The current version of the operating system.
- o Change Controller: IESG
- o Specification Document(s): Section 4 of [[this specification]]

- o Device Posture Signal Key: "device_vendor"
- o Device Posture Signal Key Description: String. The vendor of the device such as "Apple" or "Google".
- o Change Controller: IESG
- o Specification Document(s): Section 4 of [[this specification]]

- o Device Posture Signal Key: "device_model"
- o Device Posture Signal Key Description: String. The model of the device such as "iPhone X" or "Pixel 2"
- o Change Controller: IESG

- o Specification Document(s): Section 4 of [[this specification]]
- o Device Posture Signal Key: "device_attestation"
- o Device Posture Signal Key Description: Dictionary. An attestation from the operating system, containing a signed-statement about the device and/or the app.
- o Change Controller: IESG
- o Specification Document(s): Section 4 of [[this specification]]
- o Device Posture Signal Key: "app_id"
- o Device Posture Signal Key Description: String. The platform-specific identifier (URI) for the application. For Android, the format of the URI is android:apk-key-hash:<hash-of-apk-signing-cert>. For iOS, the format of URI is ios:bundle-id:<ios-bundle-id-of-app>.
- o Change Controller: IESG
- o Specification Document(s): Section 4 of [[this specification]]
- o Device Posture Signal Key: "app_managed"
- o Device Posture Signal Key Description: Boolean. True if the app is managed by a MDM/EMM system.
- o Change Controller: IESG
- o Specification Document(s): Section 4 of [[this specification]]

9. References

9.1. Normative References

- [IANA.OAuth.Parameters]
IANA, "OAuth Parameters",
<<http://www.iana.org/assignments/oauth-parameters>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", RFC 5226, DOI 10.17487/RFC5226, May 2008, <<https://www.rfc-editor.org/info/rfc5226>>.
- [RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <<https://www.rfc-editor.org/info/rfc6749>>.

9.2. Informative References

[RFC6819] Lodderstedt, T., Ed., McGloin, M., and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations", RFC 6819, DOI 10.17487/RFC6819, January 2013, <<https://www.rfc-editor.org/info/rfc6819>>.

Appendix A. Acknowledgements

The following individuals contributed ideas, feedback, and wording that shaped and formed the final specification:

Eric Sachs, John Bradley, and Andy Zmolek.

Authors' Addresses

William Denniss
Google
1600 Amphitheatre Pkwy
Mountain View, CA 94043
USA

Phone: +1 650-253-0000
Email: wdenniss@google.com
URI: <http://google.com/>

Karl McGuinness
Okta
301 Brannan St.
San Francisco, CA 94107
USA

Email: kmcguinness@okta.com
URI: <https://www.okta.com/>

John Bradley
Yubico
530 Lytton Ave, Suite 301
Palo Alto, CA 94301
USA

Phone: +1 202-630-5272
Email: ietf@ve7jtb.com
URI: <https://www.thread-safe.com/>

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 6, 2018

W. Denniss
Google
March 5, 2018

OAuth 2.0 Incremental Authorization
draft-wdenniss-oauth-incremental-auth-01

Abstract

OAuth 2.0 authorization requests that include every scope the client might ever need can result in over-scoped authorization and a sub-optimal end-user consent experience. This specification enhances the OAuth 2.0 authorization protocol by adding incremental authorization, the ability to request specific authorization scopes as needed, when they're needed, removing the requirement to request every possible scope that might be needed upfront.

Status of This Memo

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

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

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

This Internet-Draft will expire on September 6, 2018.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of

the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
2. Notational Conventions	3
3. Terminology	3
4. Incremental Auth for Confidential Clients	3
5. Incremental Auth for Public Clients	4
6. Usability Considerations	4
6.1. Handling Denials	4
7. Alternative Approaches	5
7.1. Alternative for Public Clients	5
7.2. Alternative for Confidential Clients	5
8. Privacy Considerations	5
8.1. Requesting Authorization In Context	5
8.2. Preventing Overbroad Authorization Requests	6
8.3. Authorization Correlation	6
9. Security Considerations	6
9.1. Public Client Impersonation	6
10. IANA Considerations	7
10.1. OAuth Parameters Registry	7
11. Normative References	7
Appendix A. Acknowledgements	8
Appendix B. Document History	8
Author's Address	8

1. Introduction

OAuth 2.0 clients may offer multiple features that requiring user authorization, but commonly not every user will use each feature. Without incremental authentication, applications need to either request all the possible scopes they need upfront, potentially resulting in a bad user experience, or track each authorization grant separately, complicating development.

The goal of incremental authorization is to allow clients to request just the scopes they need, when they need them, while allowing them to store a single authorization grant for the user that contains the sum of the scopes granted. Thus, each new authorization request increments the scope of the authorization grant, without the client needing to track a separate authorization grant for each group of scopes.

2. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in Key words for use in RFCs to Indicate Requirement Levels [RFC2119]. If these words are used without being spelled in uppercase then they are to be interpreted with their normal natural language meanings.

3. Terminology

In addition to the terms defined in referenced specifications, this document uses the following terms:

"OAuth" In this document, OAuth refers to OAuth 2.0 [RFC6749].

4. Incremental Auth for Confidential Clients

For confidential clients, such as web servers that can keep secrets, the authorization endpoint SHOULD treat scopes that the user already granted differently on the consent user interface. Typically such scopes are hidden for new authorization requests, or at least there is an indication that the user already approved them.

By itself, this property of the authorization endpoint enables incremental authorization. The client can track every scope they've ever requested, and include those scopes on every new authorization request.

To avoid the need for confidential clients to re-request already authorized scopes, authorization servers MAY support an additional "include_granted_scopes" parameter in the authorization request. This parameter, enables the client to request tokens during the authorization grant exchange that represent the full scope of the user's grant to the application including any previous grants, without the app needing to track the scopes directly.

The client indicates they wish the new authorization grant to include previously granted scopes by sending the following additional parameter in the OAuth 2.0 Authorization Request (Section 4.1.1 of [RFC6749].) using the following additional parameter:

include_granted_scopes OPTIONAL. Either "true" or "false". When "true", the authorization server SHOULD include previously granted scopes for this client in the new authorization grant.

5. Incremental Auth for Public Clients

Unlike with confidential clients, it is NOT RECOMMEND to automatically approve OAuth requests for public clients without user consent (see Section 10.2 of OAuth 2.0 [RFC6749], and Section 8.6 of OAuth 2.0 [RFC8252]), thus authorization grants shouldn't contain previously authorized scopes in the manner described above for confidential clients.

Public clients (and confidential clients using this technique) should instead track the scopes for every authorization grant, and only request yet to be granted scopes during incremental authorization. In the past, this would result in multiple discrete authorization grants that would need to be tracked. To enable incrementing a single authorization grant for public clients, the client supplies their existing refresh token during the authorization code exchange, and receives new authorization tokens with the scope of the previous and current authorization grants.

The client sends the previous refresh token in the OAuth 2.0 Access Token Request (Section 4.1.3 of [RFC6749].) using the following additional parameter:

`existing_grant` OPTIONAL. The refresh token from the existing authorization grant.

When processing the token exchange, in addition to the normal processing of such a request, the token endpoint MUST verify that token provided in the "existing_grant" parameter is unexpired and unrevoked, and was issued to the same client id and relates to the same user as the current authorization grant. If this verification succeeds, the new refresh token issued in the Access Token Response (Section 4.1.4 of) SHOULD include authorization for the scopes in the previous grant.

6. Usability Considerations

6.1. Handling Denials

A core principle of OAuth is that users may deny authorization requests for any reason. This remains true for incremental authorization requests. In the case of incremental authorization, clients may already have a valid authorization and receive a denial for an incremental authorization request (that is, an "access_denied" error code as defined in Section 4.1.2.1 of OAuth 2.0 [RFC6749]). Clients should SHOULD handle such errors gracefully and not discard any existing authorization grants if the user denies an incremental authorization request. Clients SHOULD NOT immediately request the

same incremental authorization again, as this may result in an infinite denial loop (and the end-user feeling badgered).

7. Alternative Approaches

7.1. Alternative for Public Clients

It is possible for OAuth clients to maintain multiple authorizations per user for feature-specific scopes without needing the feature documented in this specification. For example, an app could maintain an authorization for the contacts and one for calendar, and store them separately.

This specification offers a convenience that a single authorization grant can be managed that represents all the scope granted so far, rather than needing to maintain multiple, however it does require that all grants are made from a single end-user account (as authorization servers cannot typically combine grants from multiple users). Apps where users may wish to authorize separate end-user accounts for different features should consider using the alternative documented above.

7.2. Alternative for Confidential Clients

An alternative incremental auth design for confidential clients is to ask for authorization scopes as they are needed and keep a running record of all granted scopes. In this way each incremental authorization request would include all scopes granted so far, plus the new scope needed. Authorization servers can see the existing scopes and only display the new scopes for approval (and likely to inform the user of the existing grants). This approach can be performed using RFC 6749 without additions, but requires the client to keep track of every authorization grant.

Confidential clients can also use the alternative documented for public clients in Section 7.1.

8. Privacy Considerations

8.1. Requesting Authorization In Context

The goal of incremental authorization is to enhance end-user privacy by allowing clients to request only the authorization scopes needed in the context of a particular user action, rather than asking for ever possible scope upfront. For example, an app may offer calendar and contacts integration, and an extension of OAuth like OpenID Connect for sign-in. Such an app should first sign the user in with just the scopes needed for that. If later the user interacts with

the calendar or contacts features then, and only then, should the requires scopes be requested. By using this specification, apps can improve the privacy choices of end-users by only requesting the scopes they need in context.

Clients authorizing the user with an authorization server that supports incremental auth SHOULD ask for the minimal authorization scope for the user's current context, and use this specification to add authorization scope as required.

8.2. Preventing Overbroad Authorization Requests

When this specification is implemented, clients should have no technical reason to make overbroad authorization requests (i.e. requesting every possible scope, even ones they don't immediately need). It is therefore RECOMMENDED for authorization servers to limit the authorization scope that can be requested in a single authorization to what would reasonably be needed by a single feature.

8.3. Authorization Correlation

Incremental authorization is designed for use-cases where it's the same user authorizing each request, and thus all incremental authorization grants are correlated to that one user (by being merged into a single authorization grant). For applications where users may wish to connect different user accounts for different features (e.g. contacts from one account, and calendar from another) it is RECOMMENDED to instead allow multiple unrelated authorizations, as documented in Section 7.1.

The goal of this specification is to improve end-user privacy by giving them more choice over which scopes they grant access to. Previously many apps would request an overly large number of scopes upfront (typically for all the features of the app, rather than the subset that the user is currently wishing to use). The scopes in such authorization grants are necessarily correlated with the same user as they are contained in a single authorization grant. Implementing this specification doesn't change that attribute, but it does improve user privacy overall by empowering the user to grant access in a more granular way.

9. Security Considerations

9.1. Public Client Impersonation

As documented in Section 8.6 of RFC 8252 [RFC8252], some public clients are susceptible to client impersonation, depending on the type of redirect URI used. If the "include_granted_scopes" feature

documented in Section 4 is used by an impersonating client, it may receive a greater authorization grant than the user specifically approved for that client. For this reason, the "include_granted_scopes" feature MUST NOT be enabled for such public client requests.

Note that there is no such restriction on the use of "existing_grant" feature documented in Section 5. While it is designed for public clients, it MAY be supported for all client types.

10. IANA Considerations

This specification makes a registration request as follows:

10.1. OAuth Parameters Registry

This specification registers the following parameters in the IANA OAuth Parameters registry defined in OAuth 2.0 [RFC6749].

- o Parameter name: include_granted_scopes
- o Parameter usage location: authorization request
- o Change controller: IESG
- o Specification document(s): this document
- o Parameter name: existing_grant
- o Parameter usage location: token request
- o Change controller: IESG
- o Specification document(s): this document

11. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <<https://www.rfc-editor.org/info/rfc6749>>.

[RFC8252] Denniss, W. and J. Bradley, "OAuth 2.0 for Native Apps", BCP 212, RFC 8252, DOI 10.17487/RFC8252, October 2017, <<https://www.rfc-editor.org/info/rfc8252>>.

Appendix A. Acknowledgements

The following individuals contributed ideas, feedback, and wording that shaped and formed the final specification:

Yanna Wu, Marius Scurtescu, Jason Huang, Nicholas Watson, and Breno de Medeiros.

Appendix B. Document History

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

-01

- o Added usability, privacy, and security considerations.
- o Documented alternative approaches.

-00

- o Initial draft based on the implementation of incremental and "appcremental" auth at Google.

Author's Address

William Denniss
Google
1600 Amphitheatre Pkwy
Mountain View, CA 94043
USA

Email: wdenniss@google.com
URI: <http://wdenniss.com/incremental-auth>