

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.

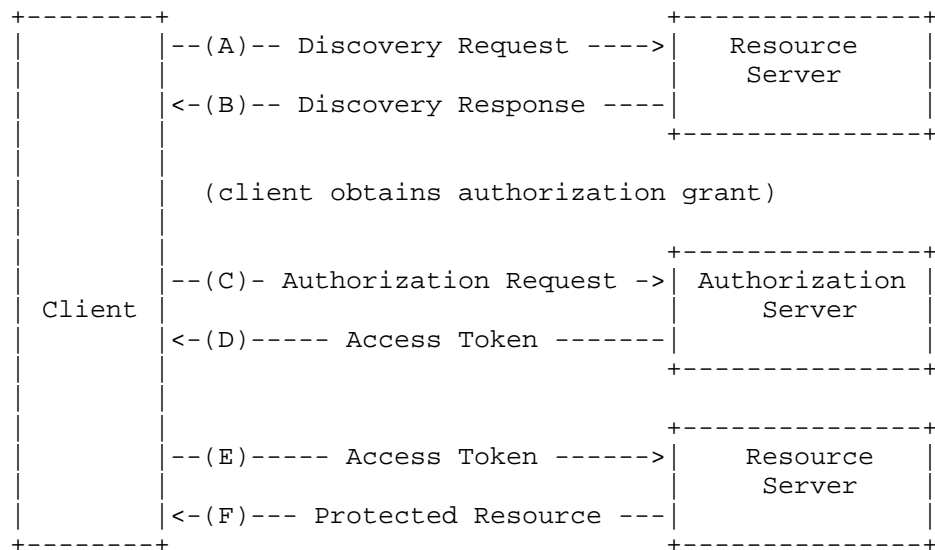


Figure 1: Abstract Protocol Flow

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 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 Working Group
Internet-Draft
Intended status: Standards Track
Expires: November 4, 2020

W. Denniss
Google
May 3, 2020

OAuth 2.0 Incremental Authorization
draft-ietf-oauth-incremental-authorization-04

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 November 4, 2020.

Copyright Notice

Copyright (c) 2020 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
6.2. Handling Scope Reductions	5
7. Alternative Approaches	5
7.1. Alternative for Public Clients	6
7.2. Alternative for Confidential Clients	6
8. Privacy Considerations	6
8.1. Requesting Authorization In Context	6
8.2. Preventing Overbroad Authorization Requests	7
8.3. Authorization Correlation	7
8.4. Previously Granted Scopes	8
9. Discovery Metadata	8
10. Security Considerations	8
10.1. Public Client Impersonation	8
11. IANA Considerations	9
11.1. OAuth Parameters Registry	9
11.2. OAuth Extensions Error Registration	9
11.3. OAuth 2.0 Authorization Server Metadata	9
12. Normative References	10
Appendix A. Acknowledgements	10
Appendix B. Document History	10
Author's Address	11

1. Introduction

OAuth 2.0 clients may offer multiple features that require 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 client 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 access and refresh tokens issued in the Access Token Response (Section 4.1.4 of) MUST include authorization for the scopes in the previous grant, unless the authorization server is exercising its prerogative to "fully or partially ignore the scope requested by the client" per Section 3.3 of OAuth 2.0 [RFC6749].

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).

6.2. Handling Scope Reductions

As specified by Section 5.1 of OAuth 2.0 [RFC6749], a successful response may not always include all the scope that was asked for, a fact indicated by the "scope" response parameter when it happens. This is still true in the case of incremental auth. The success response may include less scope than what was requested, or even less scope than before the incremental authorization request (say, if the user was given an opportunity to revise the grant down). Clients MUST check for the "scope" parameter in success responses and react accordingly.

For the purposes of an incremental auth request, a success response to an incremental authorization request that contains the same scope granted prior to the request being made, and an error response (for example, in the case of a denial) can have the same effect: the client retains a grant with the same scope as before. In the case of the approved request but with the same scope, they have a new grant, but with the same scope. In the case of the denied incremental authorization request, they still have the old grant with the same scope (although in some cases it may have been revoked or reduced in scope out of band).

An incremental authorization request isn't the only time that scope can be reduced for a grant. As specified by section 6 of OAuth 2.0 [RFC6749], scope can be reduced during a token refresh as well. So it's a good practice for clients to retain the current scope of the grant, update it during authorization, incremental authorization and token refreshes, and take action at any time based on the current scope by presenting an incremental authorization if a non-present scope is needed.

7. Alternative Approaches

This non-normative section discusses some alternative ways to achieve the incremental authorization result purely on the client side. These options are somewhat more complex and burdensome to client developers.

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, a public client (such as a mobile 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). Clients where users may wish to authorize separate end-user accounts for different features should consider using the alternative documented in this section.

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 they might ever expect need, rather than ones related to the user's current activity). To improve privacy, 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. The authorization server MAY deny such authorization requests with the following error code.

overbroad_scope

The scope of the request is considered overbroad by the authorization server. Consult the documentation of your authorization server to determine acceptable scope combinations, and consider using [[This Specification]] to perform incremental authorization requests in the context that the scope is needed.

Determining what constitutes an overbroad request is the purview of the authorization server. As an example, say an authorization supported "calendar" and "mail" scopes to access a user's calendar and inbox respectively. They may decide that their users should have the chance to grant such requests in context through incremental authorization, rather than all at once upfront, and deny the request for being overly broad.

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.

8.4. Previously Granted Scopes

When the authorization server displays the list of scopes on page and prompts the user to consent to sharing access, users may assume that the displayed list of scopes on such a page is the full and complete list being granted to the application. It may be desirable for such a consent page to list previously granted scopes, provided that the client is confidential, or one that cannot be impersonated.

9. Discovery Metadata

Support for the incremental authorization MAY be declared in the OAuth 2.0 Authorization Server Metadata [RFC8414] with the following metadata:

`incremental_authz_types_supported`

OPTIONAL. JSON array of OAuth 2.0 client types that are supported for incremental authorization. The possible types are "confidential", and "public".

Specifically, "confidential" indicates that the behavior documented in Section 4 (Incremental Auth for Confidential Clients) is supported, and "public" indicates that the behavior documented in Section 5 (Incremental Auth for Public Clients) is supported.

A server which supports both forms of incremental auth documented in this specification would declare support like so:

```
"incremental_authz_types_supported": ["confidential", "public"]
```

10. Security Considerations

10.1. 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.

11. IANA Considerations

This specification makes a registration request as follows:

11.1. 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

11.2. OAuth Extensions Error Registration

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

Name: overbroad_scope
Usage Location: authorization code grant error response
Protocol Extension: [[This Specification]]
Change Controller: IETF
Reference: Section 8.2 of [[This Specification]]

11.3. 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].

- o Metadata Name: incremental_authz_types_supported

- o Metadata Description: JSON array containing a list of client types that support OAuth 2.0 Incremental Authorization [[this specification]]. The possible types are "confidential", and "public".
- o Change controller: IESG
- o Specification Document: Section 9 of [[this specification]]

12. 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>>.
- [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>>.
- [IANA.OAuth.Parameters] IANA, "OAuth Parameters", <<http://www.iana.org/assignments/oauth-parameters>>.

Appendix A. Acknowledgements

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

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

Yanna Wu, Marius Scurtescu, Jason Huang, Nicholas Watson, Breno de Medeiros, Naveen Agarwal, Brian Campbell, and Aaron Parecki.

Appendix B. Document History

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

01

- o Changed a SHOULD to a MUST in Section 5 regarding the protocol behavior of incremental auth for public clients, while clarifying that the authorization server retains the prerogative to do whatever it wants.
- o Defined an OAuth Metadata entry.

00

- o Now a working group draft.

draft-wdenniss-oauth-incremental-auth-01

- o Added usability, privacy, and security considerations.
- o Documented alternative approaches.

draft-wdenniss-oauth-incremental-auth-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: <https://wdenniss.com/incremental-auth>

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.

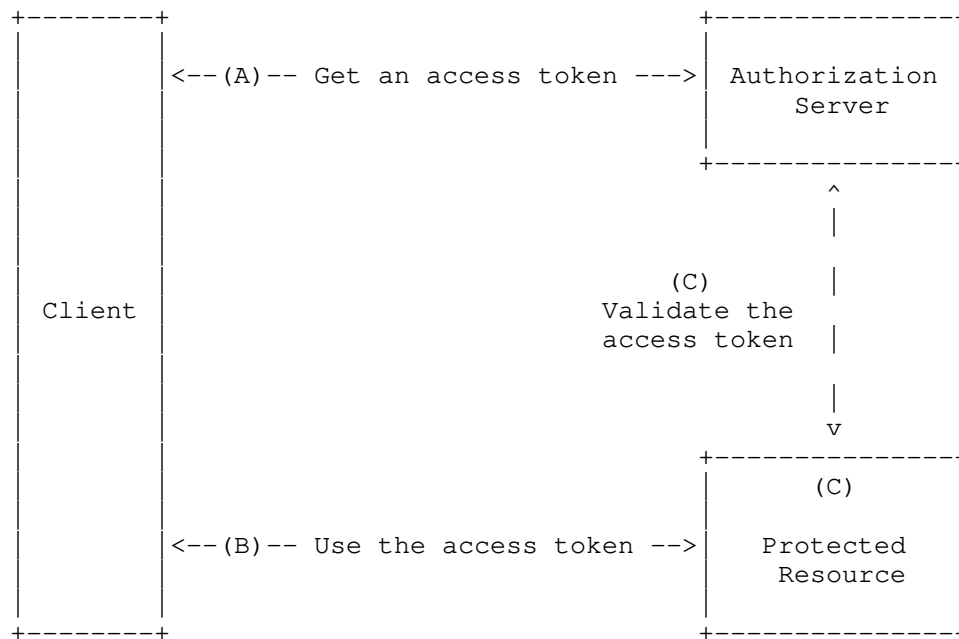


Figure 1: Abstract OAuth 2.0 Protocol Flow

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
ODEyMzcwOVoxDTIyMDUwMjEyMzcwOVowDzENMAsGA1UEAwEbXRsczBZMBMBGByqG
SM49AgEGCCqGSM49AwEHA0IABNcnxwqV6hY8QnhxxzFQ03C7HKW9OylMbnQZjjJ
/Au08/coZwxS7LFA4vOLS9WuneIXhbGGWvsDSb0tH6IxLm8wCgYIKoZIZj0EAWID
SQAwrGIhAP0RC1E+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
    xODEyMzcwOVoxDTIyMDUwMjEyMzcwOVowDzENMAsGA1UEAwEbXRsczBZMBMBGByqG
    qGSM49AgEGCCqGSM49AwEHA0IABNcnxwqV6hY8QnhxxzFQ03C7HKW9OylMbnQZjjJ
    /Au08/coZwxS7LFA4vOLS9WuneIXhbGGWvsDSb0tH6IxLm8wCgYIKoZIZj0EAWID
    SQAwrGIhAP0RC1E+vwJD/D1AGHGzuri+hlV/PpQEKTWUveORWz83AiEA5x2eXZOV
    bUlJSGQgjd5vaUaK1LR50Q2DmFfQj1L+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

OAuth
Internet-Draft
Intended status: Informational
Expires: January 9, 2017

P. Hunt, Ed.
Oracle Corporation
J. Richer

W. Mills

P. Mishra
Oracle Corporation
H. Tschofenig
ARM Limited
July 8, 2016

OAuth 2.0 Proof-of-Possession (PoP) Security Architecture
draft-ietf-oauth-pop-architecture-08.txt

Abstract

The OAuth 2.0 bearer token specification, as defined in RFC 6750, allows any party in possession of a bearer token (a "bearer") to get access to the associated resources (without demonstrating possession of a cryptographic key). To prevent misuse, bearer tokens must be protected from disclosure in transit and at rest.

Some scenarios demand additional security protection whereby a client needs to demonstrate possession of cryptographic keying material when accessing a protected resource. This document motivates the development of the OAuth 2.0 proof-of-possession security mechanism.

Status of This Memo

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

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

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

This Internet-Draft will expire on January 9, 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	3
2. Terminology	3
3. Use Cases	3
3.1. Preventing Access Token Re-Use by the Resource Server . .	4
3.2. TLS and DTLS Channel Binding Support	4
3.3. Access to a Non-TLS Protected Resource	4
3.4. Offering Application Layer End-to-End Security	5
4. Security and Privacy Threats	5
5. Requirements	6
6. Threat Mitigation	10
6.1. Confidentiality Protection	11
6.2. Sender Constraint	11
6.3. Key Confirmation	12
6.4. Summary	13
7. Architecture	14
7.1. Client and Authorization Server Interaction	15
7.1.1. Symmetric Keys	15
7.1.2. Asymmetric Keys	16
7.2. Client and Resource Server Interaction	17
7.3. Resource and Authorization Server Interaction (Token Introspection)	18
8. Security Considerations	19
9. IANA Considerations	19
10. Acknowledgments	19
11. References	20
11.1. Normative References	20
11.2. Informative References	21
Authors' Addresses	22

1. Introduction

The OAuth 2.0 protocol family ([RFC6749], [RFC6750], and [RFC6819]) offer a single token type known as the "bearer" token to access protected resources. RFC 6750 [RFC6750] specifies the bearer token mechanism and defines it as follows:

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

The bearer token meets the security needs of a number of use cases the OAuth 2.0 protocol had originally been designed for. There are, however, other scenarios that require stronger security properties and ask for active participation of the OAuth client in form of cryptographic computations when presenting an access token to a resource server.

This document outlines additional use cases requiring stronger security protection in Section 3, identifies threats in Section 4, proposes different ways to mitigate those threats in Section 6, outlines an architecture for a solution that builds on top of the existing OAuth 2.0 framework in Section 7, and concludes with a requirements list in Section 5.

2. Terminology

The key words 'MUST', 'MUST NOT', 'REQUIRED', 'SHALL', 'SHALL NOT', 'SHOULD', 'SHOULD NOT', 'RECOMMENDED', 'MAY', and 'OPTIONAL' in this specification are to be interpreted as described in [RFC2119], with the important qualification that, unless otherwise stated, these terms apply to the design of the protocol, not its implementation or application.

3. Use Cases

The main use case that motivates improvement upon "bearer" token security is the desire of resource servers to obtain additional assurance that the client is indeed authorized to present an access token. The expectation is that the use of additional credentials (symmetric or asymmetric keying material) will encourage developers to take additional precautions when transferring and storing access token in combination with these credentials.

Additional use cases listed below provide further requirements for the solution development. Note that a single solution does not necessarily need to offer support for all use cases.

3.1. Preventing Access Token Re-Use by the Resource Server

In a scenario where a resource server receives a valid access token, the resource server then re-uses it with other resource server. The reason for re-use may be malicious or may well be legitimate. In a legitimate case, the intent is to support chaining of computations whereby a resource server needs to consult other third party resource servers to complete a requested operation. In both cases it may be assumed that the scope and audience of the access token is sufficiently defined that to allow such a re-use. For example, imagine a case where a company operates email services as well as picture sharing services and that company had decided to issue access tokens with a scope and audience that allows access to both services.

With this use case the desire is to prevent such access token re-use. This also implies that the legitimate use cases require additional enhancements for request chaining.

3.2. TLS and DTLS Channel Binding Support

In this use case we consider the scenario where an OAuth 2.0 request to a protected resource is secured using TLS or DTLS (see [RFC4347]), but the client and the resource server demand that the underlying TLS/DTLS exchange is bound to additional application layer security to prevent cases where the TLS/DTLS connection is terminated at a TLS/DTLS intermediary, which splits the TLS/DTLS connection into two separate connections.

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

3.3. Access to a Non-TLS Protected Resource

This use case is for a web client that needs to access a resource that makes data available (such as videos) without offering integrity and confidentiality protection using TLS. Still, the initial resource request using OAuth, which includes the access token, must be protected against various threats (e.g., token replay, token modification).

While it is possible to utilize bearer tokens in this scenario with TLS protection when the request to the protected resource is made, as described in [RFC6750], there may be the desire to avoid using TLS

between the client and the resource server at all. In such a case the bearer token approach is not possible since it relies on TLS for ensuring integrity and confidentiality protection of the access token exchange since otherwise replay attacks are possible: First, an eavesdropper may steal an access token and present it at a different resource server. Second, an eavesdropper may steal an access token and replay it against the same resource server at a later point in time. In both cases, if the attack is successful, the adversary gets access to the resource owners data or may perform an operation selected by the adversary (e.g., sending a message). Note that the adversary may obtain the access token (if the recommendations in [RFC6749] and [RFC6750] are not followed) using a number of ways, including eavesdropping the communication on the wireless link.

Consequently, the important assumption in this use case is that a resource server does not have TLS support and the security solution should work in such a scenario. Furthermore, it may not be necessary to provide authentication of the resource server towards the client.

3.4. Offering Application Layer End-to-End Security

In Web deployments resource servers are often placed behind load balancers, which are deployed by the same organization that operates the resource servers. These load balancers may terminate the TLS connection setup and HTTP traffic is transmitted without TLS protection from the load balancer to the resource server. With application layer security in addition to the underlying TLS security it is possible to allow application servers to perform cryptographic verification on an end-to-end basis.

The key aspect in this use case is therefore to offer end-to-end security in the presence of load balancers via application layer security. Enterprise networks also deploy proxies that inspect traffic and thereby break TLS.

4. Security and Privacy Threats

The following list presents several common threats against protocols utilizing some form of token. This list of threats is based on NIST Special Publication 800-63 [NIST800-63]. We exclude a discussion of threats related to any form of identity proofing and authentication of the resource owner to the authorization server since these procedures are not part of the OAuth 2.0 protocol specification itself.

Token manufacture/modification:

An attacker may generate a bogus token or modify the token content (such as authentication or attribute statements) of an existing token, causing resource server to grant inappropriate access to the client. For example, an attacker may modify the token to extend the validity period. A client, which MAY be a normal client or MAY be assumed to be constrained (see [RFC7252]), may modify the token to have access to information that they should not be able to view.

Token disclosure:

Tokens may contain personal data, such as real name, age or birthday, payment information, etc.

Token redirect:

An attacker uses the token generated for consumption by the resource server to obtain access to another resource server.

Token reuse:

An attacker attempts to use a token that has already been used once with a resource server. The attacker may be an eavesdropper who observes the communication exchange or, worse, one of the communication end points. A client may, for example, leak access tokens because it cannot keep secrets confidential. A client may also reuse access tokens for some other resource servers. Finally, a resource server may use a token it had obtained from a client and use it with another resource server that the client interacts with. A resource server, offering relatively unimportant application services, may attempt to use an access token obtained from a client to access a high-value service, such as a payment service, on behalf of the client using the same access token.

Token repudiation:

Token repudiation refers to a property whereby a resource server is given an assurance that the authorization server cannot deny to have created a token for the client.

5. Requirements

RFC 4962 [RFC4962] gives useful guidelines for designers of authentication and key management protocols. While RFC 4962 was written with the AAA framework used for network access authentication in mind the offered suggestions are useful for the design of other key management systems as well. The following requirements list

applies OAuth 2.0 terminology to the requirements outlined in RFC 4962.

These requirements include

Cryptographic Algorithm Independent:

The key management protocol MUST be cryptographic algorithm independent.

Strong, fresh session keys:

Session keys MUST be strong and fresh. Each session deserves an independent session key, i.e., one that is generated specifically for the intended use. In context of OAuth this means that keying material is created in such a way that can only be used by the combination of a client instance, protected resource, and authorization scope.

Limit Key Scope:

Following the principle of least privilege, parties MUST NOT have access to keying material that is not needed to perform their role. Any protocol that is used to establish session keys MUST specify the scope for session keys, clearly identifying the parties to whom the session key is available.

Replay Detection Mechanism:

The key management protocol exchanges MUST be replay protected. Replay protection allows a protocol message recipient to discard any message that was recorded during a previous legitimate dialogue and presented as though it belonged to the current dialogue.

Authenticate All Parties:

Each party in the key management protocol MUST be authenticated to the other parties with whom they communicate. Authentication mechanisms MUST maintain the confidentiality of any secret values used in the authentication process. Secrets MUST NOT be sent to another party without confidentiality protection.

Authorization:

Client and resource server authorization MUST be performed. These entities MUST demonstrate possession of the appropriate keying material, without disclosing it. Authorization is REQUIRED

whenever a client interacts with an authorization server.
Authorization checking prevents an elevation of privilege attack.

Keying Material Confidentiality and Integrity:

While preserving algorithm independence, confidentiality and integrity of all keying material MUST be maintained.

Confirm Cryptographic Algorithm Selection:

The selection of the "best" cryptographic algorithms SHOULD be securely confirmed. The mechanism SHOULD detect attempted roll-back attacks.

Uniquely Named Keys:

Key management proposals require a robust key naming scheme, particularly where key caching is supported. The key name provides a way to refer to a key in a protocol so that it is clear to all parties which key is being referenced. Objects that cannot be named cannot be managed. All keys MUST be uniquely named, and the key name MUST NOT directly or indirectly disclose the keying material.

Prevent the Domino Effect:

Compromise of a single client MUST NOT compromise keying material held by any other client within the system, including session keys and long-term keys. Likewise, compromise of a single resource server MUST NOT compromise keying material held by any other Resource Server within the system. In the context of a key hierarchy, this means that the compromise of one node in the key hierarchy must not disclose the information necessary to compromise other branches in the key hierarchy. Obviously, the compromise of the root of the key hierarchy will compromise all of the keys; however, a compromise in one branch MUST NOT result in the compromise of other branches. There are many implications of this requirement; however, two implications deserve highlighting. First, the scope of the keying material must be defined and understood by all parties that communicate with a party that holds that keying material. Second, a party that holds keying material in a key hierarchy must not share that keying material with parties that are associated with other branches in the key hierarchy.

Bind Key to its Context:

Keying material MUST be bound to the appropriate context. The context includes the following.

- * The manner in which the keying material is expected to be used.
- * The other parties that are expected to have access to the keying material.
- * The expected lifetime of the keying material. Lifetime of a child key SHOULD NOT be greater than the lifetime of its parent in the key hierarchy.

Any party with legitimate access to keying material can determine its context. In addition, the protocol MUST ensure that all parties with legitimate access to keying material have the same context for the keying material. This requires that the parties are properly identified and authenticated, so that all of the parties that have access to the keying material can be determined. The context will include the client and the resource server identities in more than one form.

Authorization Restriction:

If client authorization is restricted, then the client SHOULD be made aware of the restriction.

Client Identity Confidentiality:

A client has identity confidentiality when any party other than the resource server and the authorization server cannot sufficiently identify the client within the anonymity set. In comparison to anonymity and pseudonymity, identity confidentiality is concerned with eavesdroppers and intermediaries. A key management protocol SHOULD provide this property.

Resource Owner Identity Confidentiality:

Resource servers SHOULD be prevented from knowing the real or pseudonymous identity of the resource owner, since the authorization server is the only entity involved in verifying the resource owner's identity.

Collusion:

Resource servers that collude can be prevented from using information related to the resource owner to track the individual. That is, two different resource servers can be prevented from determining that the same resource owner has authenticated to both

of them. Authorization servers MUST bind different keying material to access tokens used for resource servers from different origins (or similar concepts in the app world).

AS-to-RS Relationship Anonymity:

For solutions using asymmetric key cryptography the client MAY conceal information about the resource server it wants to interact with. The authorization server MAY reject such an attempt since it may not be able to enforce access control decisions.

Channel Binding:

A solution MUST enable support for channel bindings. The concept of channel binding, as defined in [RFC5056], allows applications to establish that the two end-points of a secure channel at one network layer are the same as at a higher layer by binding authentication at the higher layer to the channel at the lower layer.

There are performance concerns with the use of asymmetric cryptography. Although symmetric key cryptography offers better performance asymmetric cryptography offers additional security properties. A solution MUST therefore offer the capability to support both symmetric as well as asymmetric keys.

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

A number of the threats listed in Section 4 demand protection of the access token content and a standardized solution, for example, in the form of a JSON-based format, is available with the JWT [RFC7519].

6. Threat Mitigation

A large range of threats can be mitigated by protecting the content of the token, for example using a digital signature or a keyed message digest. Alternatively, the content of the token could be passed by reference rather than by value (requiring a separate message exchange to resolve the reference to the token content).

To simplify discussion in the following example we assume that the token itself cannot be modified by the client, either due to cryptographic protection (such as signature or encryption) or use of a reference value with sufficient entropy and associated secure lookup. The token remains opaque to the client. These are characteristics shared with bearer tokens and more information on

best practices can be found in [RFC6819] and in the security considerations section of [RFC6750].

To deal with token redirect it is important for the authorization server to include the identifier of the intended recipient - the resource server. A resource server must not be allowed to accept access tokens that are not meant for its consumption.

To provide protection against token disclosure two approaches are possible, namely (a) not to include sensitive information inside the token or (b) to ensure confidentiality protection. The latter approach requires at least the communication interaction between the client and the authorization server as well as the interaction between the client and the resource server to experience confidentiality protection. As an example, TLS with a ciphersuite that offers confidentiality protection has to be applied as per [RFC7525]. Encrypting the token content itself is another alternative. In our scenario the authorization server would, for example, encrypt the token content with a symmetric key shared with the resource server.

To deal with token reuse more choices are available.

6.1. Confidentiality Protection

In this approach confidentiality protection of the exchange is provided on the communication interfaces between the client and the resource server, and between the client and the authorization server. No eavesdropper on the wire is able to observe the token exchange. Consequently, a replay by a third party is not possible. An authorization server wants to ensure that it only hands out tokens to clients it has authenticated first and who are authorized. For this purpose, authentication of the client to the authorization server will be a requirement to ensure adequate protection against a range of attacks. This is, however, true for the description in Section 6.2 and Section 6.3 as well. Furthermore, the client has to make sure it does not distribute (or leak) the access token to entities other than the intended the resource server. For that purpose the client will have to authenticate the resource server before transmitting the access token.

6.2. Sender Constraint

Instead of providing confidentiality protection, the authorization server could also put the identifier of the client into the protected token with the following semantic: 'This token is only valid when presented by a client with the following identifier.' When the access token is then presented to the resource server how does it

know that it was provided by the client? It has to authenticate the client! There are many choices for authenticating the client to the resource server, for example by using client certificates in TLS [RFC5246], or pre-shared secrets within TLS [RFC4279]. The choice of the preferred authentication mechanism and credential type may depend on a number of factors, including

- o security properties
- o available infrastructure
- o library support
- o credential cost (financial)
- o performance
- o integration into the existing IT infrastructure
- o operational overhead for configuration and distribution of credentials

This long list hints to the challenge of selecting at least one mandatory-to-implement client authentication mechanism.

6.3. Key Confirmation

A variation of the mechanism of sender authentication, described in Section 6.2, is to replace authentication with the proof-of-possession of a specific (session) key, i.e., key confirmation. In this model the resource server would not authenticate the client itself but would rather verify whether the client knows the session key associated with a specific access token. Examples of this approach can be found with the OAuth 1.0 MAC token [RFC5849], and Kerberos [RFC4120] when utilizing the AP_REQ/AP_REP exchange (see also [I-D.hardjono-oauth-kerberos] for a comparison between Kerberos and OAuth).

To illustrate key confirmation, the first example is borrowed from Kerberos and use symmetric key cryptography. Assume that the authorization server shares a long-term secret with the resource server, called $K(\text{Authorization Server-Resource Server})$. This secret would be established between them out-of-band. When the client requests an access token the authorization server creates a fresh and unique session key K_s and places it into the token encrypted with the long term key $K(\text{Authorization Server-Resource Server})$. Additionally, the authorization server attaches K_s to the response message to the client (in addition to the access token itself) over a

confidentiality protected channel. When the client sends a request to the resource server it has to use K_s to compute a keyed message digest for the request (in whatever form or whatever layer). The resource server, when receiving the message, retrieves the access token, verifies it and extracts $K(\text{Authorization Server-Resource Server})$ to obtain K_s . This key K_s is then used to verify the keyed message digest of the request message.

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

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

6.4. Summary

As a high level message, there are various ways the threats can be mitigated. While the details of each solution are somewhat different, they all accomplish the goal of mitigating the threats.

The three approaches are:

Confidentiality Protection:

The weak point with this approach, which is briefly described in Section 6.1, is that the client has to be careful to whom it discloses the access token. What can be done with the token entirely depends on what rights the token entitles the presenter and what constraints it contains. A token could encode the identifier of the client but there are scenarios where the client is not authenticated to the resource server or where the identifier of the client rather represents an application class rather than a single application instance. As such, it is possible that certain deployments choose a rather liberal approach to security and that everyone who is in possession of the access token is granted access to the data.

Sender Constraint:

The weak point with this approach, which is briefly described in Section 6.2, is to setup the authentication infrastructure such that clients can be authenticated towards resource servers. Additionally, the authorization server must encode the identifier of the client in the token for later verification by the resource server. Depending on the chosen layer for providing client-side authentication there may be additional challenges due to Web server load balancing, lack of API access to identity information, etc.

Key Confirmation:

The weak point with this approach, see Section 6.3, is the increased complexity: a complete key distribution protocol has to be defined.

In all cases above it has to be ensured that the client is able to keep the credentials secret.

7. Architecture

The proof-of-possession security concept assumes that the authorization server acts as a trusted third party that binds keys to access tokens. These keys are then used by the client to demonstrate the possession of the secret to the resource server when accessing the resource. The resource server, when receiving an access token, needs to verify that the key used by the client matches the one included in the access token.

There are slight differences between the use of symmetric keys and asymmetric keys when they are bound to the access token and the subsequent interaction between the client and the authorization server when demonstrating possession of these keys. Figure 1 shows the symmetric key procedure and Figure 2 illustrates how asymmetric keys are used. While symmetric cryptography provides better performance properties the use of asymmetric cryptography allows the client to keep the private key locally and never expose it to any other party.

For example, with the JSON Web Token (JWT) [RFC7519] a standardized format for access tokens is available. The necessary elements to bind symmetric or asymmetric keys to a JWT are described in [I-D.ietf-oauth-proof-of-possession].

Note: The negotiation of cryptographic algorithms between the client and the authorization server is not shown in the examples below and

assumed to be present in a protocol solution to meet the requirements for crypto-agility.

7.1. Client and Authorization Server Interaction

7.1.1. Symmetric Keys

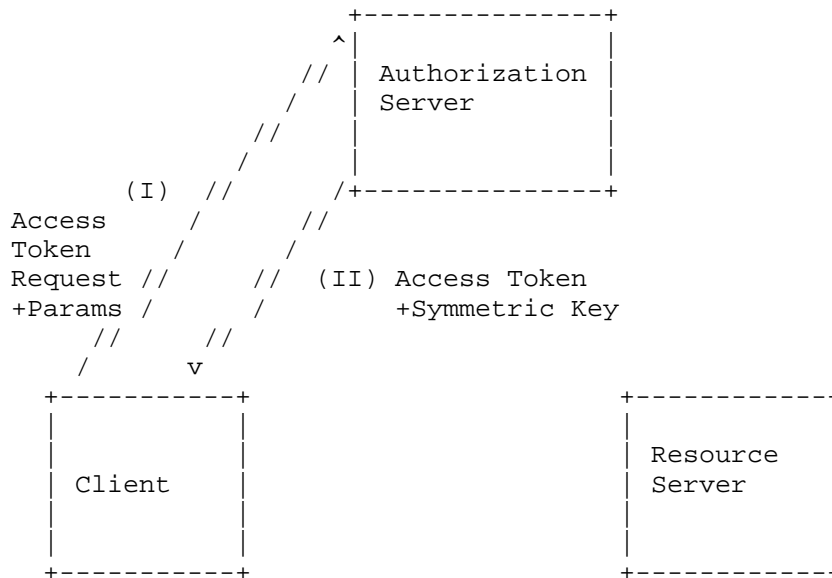


Figure 1: Interaction between the Client and the Authorization Server (Symmetric Keys).

In order to request an access token the client interacts with the authorization server as part of the a normal grant exchange, as shown in Figure 1. However, it needs to include additional information elements for use with the PoP security mechanism, as depicted in message (I). In message (II) the authorization server then returns the requested access token. In addition to the access token itself, the symmetric key is communicated to the client. This symmetric key is a unique and fresh session key with sufficient entropy for the given lifetime. Furthermore, information within the access token ties it to this specific symmetric key.

Note: For this security mechanism to work the client as well as the resource server need to have access to the session key. While the key transport mechanism from the authorization server to the client has been explained in the previous paragraph there are three ways for communicating this session key from the authorization server to the resource server, namely

Embedding the symmetric key inside the access token itself. This requires that the symmetric key is confidentiality protected.

The resource server queries the authorization server for the symmetric key. This is an approach envisioned by the token introspection endpoint [RFC7662].

The authorization server and the resource server both have access to the same back-end database. Smaller, tightly coupled systems might prefer such a deployment strategy.

7.1.2. Asymmetric Keys

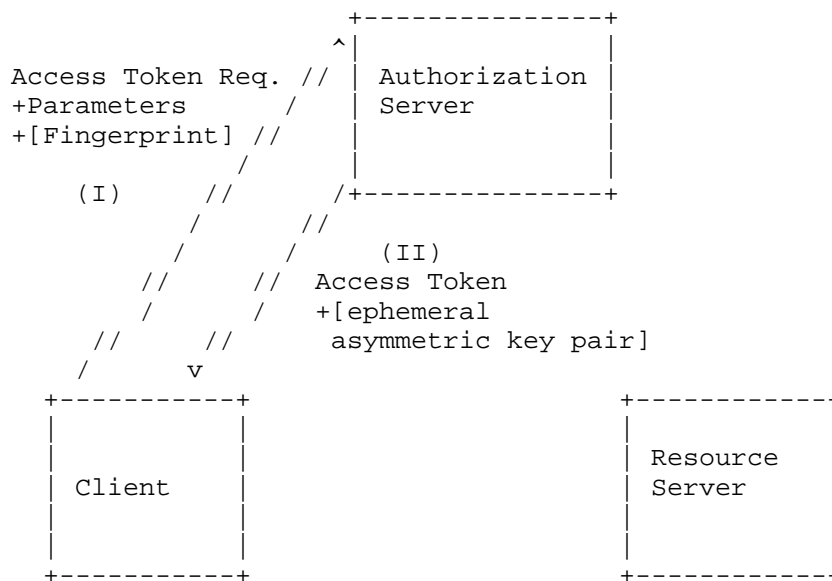


Figure 2: Interaction between the Client and the Authorization Server (Asymmetric Keys).

The use of asymmetric keys is slightly different since the client or the server could be involved in the generation of the ephemeral key pair. This exchange is shown in Figure 1. If the client generates the key pair it either includes a fingerprint of the public key or the public key in the request to the authorization server. The authorization server would include this fingerprint or public key in the confirmation claim inside the access token and thereby bind the asymmetric key pair to the token. If the client did not provide a fingerprint or a public key in the request then the authorization server is asked to create an ephemeral asymmetric key pair, binds the fingerprint of the public key to the access token, and returns the

asymmetric key pair (public and private key) to the client. Note that there is a strong preference for generating the private/public key pair locally at the client rather than at the server.

7.2. Client and Resource Server Interaction

The specification describing the interaction between the client and the authorization server, as shown in Figure 1 and in Figure 2, can be found in [I-D.ietf-oauth-pop-key-distribution].

Once the client has obtained the necessary access token and keying material it can start to interact with the resource server. To demonstrate possession of the key bound to the access token it needs to apply this key to the request by computing a keyed message digest (i.e., a symmetric key-based cryptographic primitive) or a digital signature (i.e., an asymmetric cryptographic computation). When the resource server receives the request it verifies it and decides whether access to the protected resource can be granted. This exchange is shown in Figure 3.

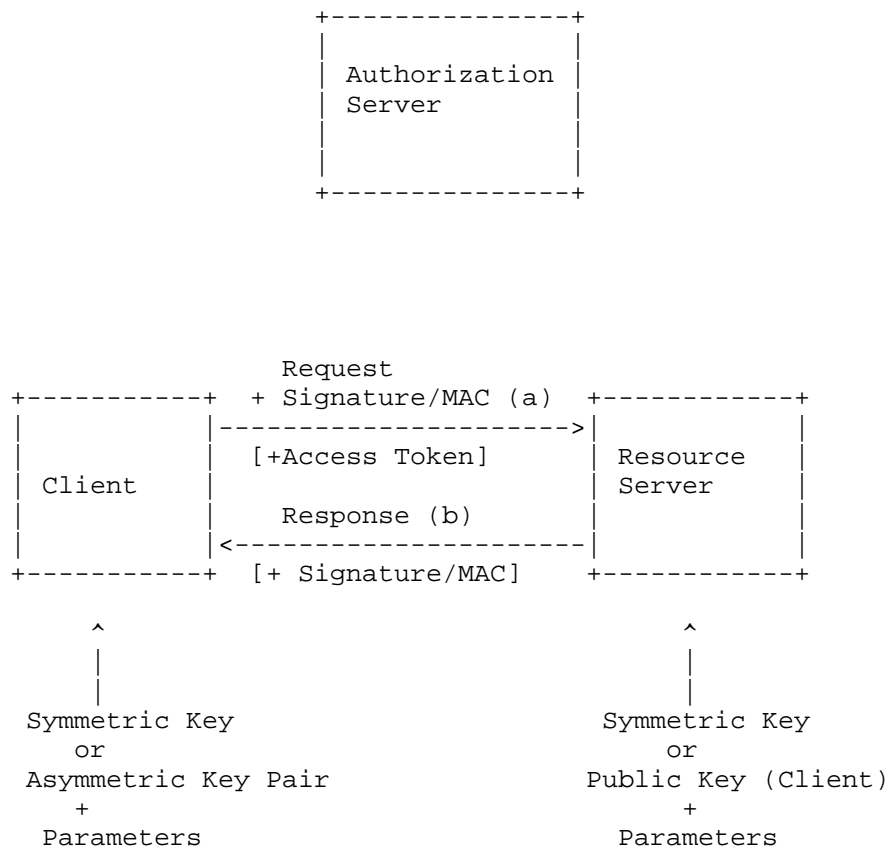


Figure 3: Client Demonstrates PoP.

The specification describing the ability to sign the HTTP request from the client to the resource server can be found in [I-D.ietf-oauth-signed-http-request].

7.3. Resource and Authorization Server Interaction (Token Introspection)

So far the examples talked about access tokens that are passed by value and allow the resource server to make authorization decisions immediately after verifying the request from the client. In some deployments a real-time interaction between the authorization server and the resource server is envisioned that lowers the need to pass self-contained access tokens around. In that case the access token merely serves as a handle or a reference to state stored at the authorization server. As a consequence, the resource server cannot autonomously make an authorization decision when receiving a request

from a client but has to consult the authorization server. This can, for example, be done using the token introspection endpoint (see [RFC7662]). Figure 4 shows the protocol interaction graphically. Despite the additional token exchange previous descriptions about associating symmetric and asymmetric keys to the access token are still applicable to this scenario.

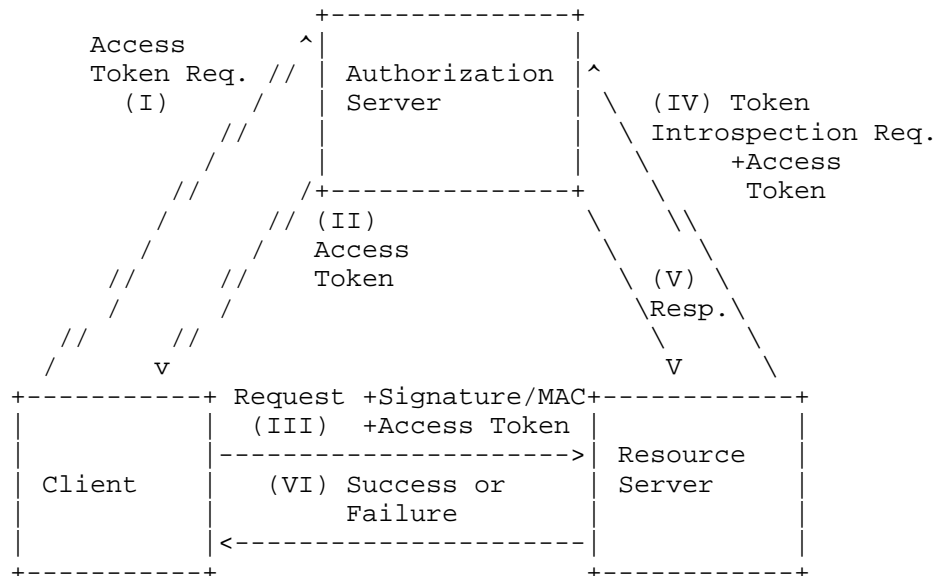


Figure 4: Token Introspection and Access Token Handles.

8. Security Considerations

The purpose of this document is to provide use cases, requirements, and motivation for developing an OAuth security solution extending Bearer Tokens. As such, this document is only about security.

9. IANA Considerations

This document does not require actions by IANA.

10. Acknowledgments

This document is the result of conference calls late 2012/early 2013 and in design team conference calls February 2013 of the IETF OAuth working group. The following persons (in addition to the OAuth WG chairs, Hannes Tschofenig, and Derek Atkins) provided their input during these calls: Bill Mills, Justin Richer, Phil Hunt, Prateek Mishra, Mike Jones, George Fletcher, Leif Johansson, Lucy Lynch, John

Bradley, Tony Nadalin, Klaas Wierenga, Thomas Hardjono, Brian Campbell

In the appendix of this document we reuse content from [RFC4962] and the authors would like thank Russ Housely and Bernard Aboba for their work on RFC 4962.

We would like to thank Reddy Tirumaleswar for his review.

11. References

11.1. Normative References

- [I-D.ietf-oauth-pop-key-distribution]
Bradley, J., Hunt, P., Jones, M., and H. Tschofenig, "OAuth 2.0 Proof-of-Possession: Authorization Server to Client Key Distribution", draft-ietf-oauth-pop-key-distribution-02 (work in progress), October 2015.
- [I-D.ietf-oauth-proof-of-possession]
Jones, M., Bradley, J., and H. Tschofenig, "Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs)", draft-ietf-oauth-proof-of-possession-11 (work in progress), December 2015.
- [I-D.ietf-oauth-signed-http-request]
Richer, J., Bradley, J., and H. Tschofenig, "A Method for Signing HTTP Requests for OAuth", draft-ietf-oauth-signed-http-request-02 (work in progress), February 2016.
- [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>>.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <<http://www.rfc-editor.org/info/rfc5246>>.
- [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>>.
- [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>>.

- [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, <<http://www.rfc-editor.org/info/rfc7525>>.
- [RFC7662] Richer, J., Ed., "OAuth 2.0 Token Introspection", RFC 7662, DOI 10.17487/RFC7662, October 2015, <<http://www.rfc-editor.org/info/rfc7662>>.

11.2. Informative References

- [I-D.hardjono-oauth-kerberos]
Hardjono, T., "OAuth 2.0 support for the Kerberos V5 Authentication Protocol", draft-hardjono-oauth-kerberos-01 (work in progress), December 2010.
- [NIST800-63]
Burr, W., Dodson, D., Perlner, R., Polk, T., Gupta, S., and E. Nabbus, "NIST Special Publication 800-63-1, INFORMATION SECURITY", December 2008.
- [RFC4120] Neuman, C., Yu, T., Hartman, S., and K. Raeburn, "The Kerberos Network Authentication Service (V5)", RFC 4120, DOI 10.17487/RFC4120, July 2005, <<http://www.rfc-editor.org/info/rfc4120>>.
- [RFC4279] Eronen, P., Ed. and H. Tschofenig, Ed., "Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)", RFC 4279, DOI 10.17487/RFC4279, December 2005, <<http://www.rfc-editor.org/info/rfc4279>>.
- [RFC4347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security", RFC 4347, DOI 10.17487/RFC4347, April 2006, <<http://www.rfc-editor.org/info/rfc4347>>.
- [RFC4962] Housley, R. and B. Aboba, "Guidance for Authentication, Authorization, and Accounting (AAA) Key Management", BCP 132, RFC 4962, DOI 10.17487/RFC4962, July 2007, <<http://www.rfc-editor.org/info/rfc4962>>.
- [RFC5056] Williams, N., "On the Use of Channel Bindings to Secure Channels", RFC 5056, DOI 10.17487/RFC5056, November 2007, <<http://www.rfc-editor.org/info/rfc5056>>.
- [RFC5849] Hammer-Lahav, E., Ed., "The OAuth 1.0 Protocol", RFC 5849, DOI 10.17487/RFC5849, April 2010, <<http://www.rfc-editor.org/info/rfc5849>>.

- [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, <<http://www.rfc-editor.org/info/rfc6125>>.
- [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>>.
- [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, <<http://www.rfc-editor.org/info/rfc6819>>.
- [RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, DOI 10.17487/RFC7252, June 2014, <<http://www.rfc-editor.org/info/rfc7252>>.

Authors' Addresses

Phil Hunt (editor)
Oracle Corporation

Email: phil.hunt@yahoo.com

Justin Richer

Email: ietf@justin.richer.org

William Mills

Email: wmills@yahoo-inc.com

Prateek Mishra
Oracle Corporation

Email: prateek.mishra@oracle.com

Hannes Tschofenig
ARM Limited
Hall in Tirol 6060
Austria

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

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: August 28, 2017

J. Bradley
Ping Identity
P. Hunt
Oracle Corporation
M. Jones
Microsoft
H. Tschofenig
ARM Limited
February 24, 2017

OAuth 2.0 Proof-of-Possession: Authorization Server to Client Key
Distribution
draft-ietf-oauth-pop-key-distribution-03

Abstract

RFC 6750 specified the bearer token concept for securing access to protected resources. Bearer tokens need to be protected in transit as well as at rest. When a client requests access to a protected resource it hands-over the bearer token to the resource server.

The OAuth 2.0 Proof-of-Possession security concept extends bearer token security and requires the client to demonstrate possession of a key when accessing a protected resource.

This document describes how the client obtains this keying material from the authorization server.

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 28, 2017.

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

Table of Contents

1. Introduction	2
2. Terminology	4
3. Audience	4
3.1. Audience Parameter	5
3.2. Processing Instructions	5
4. Symmetric Key Transport	6
4.1. Client-to-AS Request	6
4.2. Client-to-AS Response	7
5. Asymmetric Key Transport	9
5.1. Client-to-AS Request	9
5.2. Client-to-AS Response	11
6. Token Types and Algorithms	12
7. Security Considerations	13
8. IANA Considerations	14
9. Acknowledgements	15
10. References	15
10.1. Normative References	15
10.2. Informative References	16
Appendix A. Augmented Backus-Naur Form (ABNF) Syntax	17
A.1. 'aud' Syntax	17
A.2. 'key' Syntax	18
A.3. 'alg' Syntax	18
Authors' Addresses	18

1. Introduction

The work on additional security mechanisms beyond OAuth 2.0 bearer tokens [12] is motivated in [17], which also outlines use cases, requirements and an architecture. This document defines the ability for the client indicate support for this functionality and to obtain keying material from the authorization server. As an outcome of the

exchange between the client and the authorization server is an access token that is bound to keying material. Clients that access protected resources then need to demonstrate knowledge of the secret key that is bound to the access token.

To best describe the scope of this specification, the OAuth 2.0 protocol exchange sequence is shown in Figure 1. The extension defined in this document piggybacks on the message exchange marked with (C) and (D).

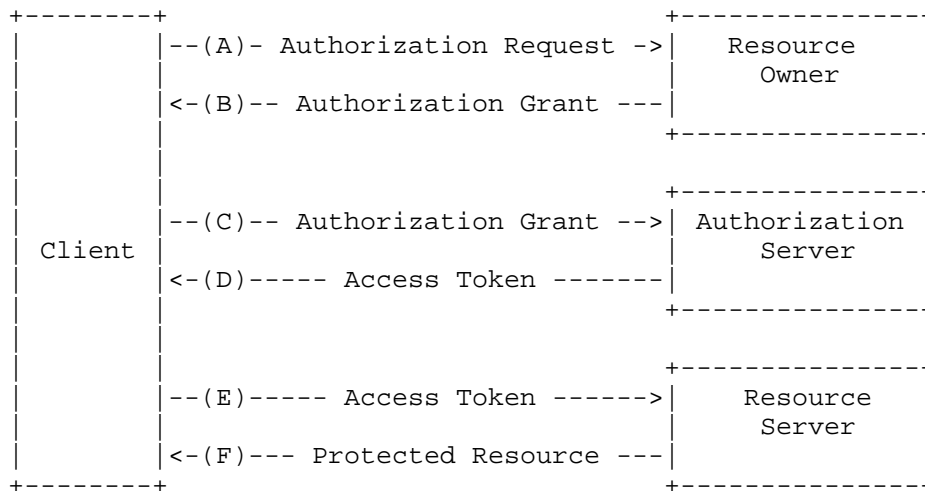


Figure 1: Abstract OAuth 2.0 Protocol Flow

In OAuth 2.0 [2] access tokens can be obtained via authorization grants and using refresh tokens. The core OAuth specification defines four authorization grants, see Section 1.3 of [2], and [14] adds an assertion-based authorization grant to that list. The token endpoint, which is described in Section 3.2 of [2], is used with every authorization grant except for the implicit grant type. In the implicit grant type the access token is issued directly.

This document extends the functionality of the token endpoint, i.e., the protocol exchange between the client and the authorization server, to allow keying material to be bound to an access token. Two types of keying material can be bound to an access token, namely symmetric keys and asymmetric keys. Conveying symmetric keys from the authorization server to the client is described in Section 4 and the procedure for dealing with asymmetric keys is described in Section 5.

2. Terminology

The key words 'MUST', 'MUST NOT', 'REQUIRED', 'SHALL', 'SHALL NOT', 'SHOULD', 'SHOULD NOT', 'RECOMMENDED', 'MAY', and 'OPTIONAL' in this specification are to be interpreted as described in [1].

Session Key:

The term session key refers to fresh and unique keying material established between the client and the resource server. This session key has a lifetime that corresponds to the lifetime of the access token, is generated by the authorization server and bound to the access token.

This document uses the following abbreviations:

JWA: JSON Web Algorithms (JWA) [7]

JWT: JSON Web Token (JWT) [9]

JWS: JSON Web Signature (JWS) [6]

JWK: JSON Web Key (JWK) [5]

JWE: JSON Web Encryption (JWE) [8]

3. Audience

When an authorization server creates an access token, according to the PoP security architecture [17], it may need to know which resource server will process it. This information is necessary when the authorization server applies integrity protection to the JWT using a symmetric key and has to select the key of the resource server that has to verify it. The authorization server also requires this audience information if it has to encrypt a symmetric session key inside the access token using a long-term symmetric key.

This section defines a new header that is used by the client to indicate what protected resource at which resource server it wants to access. This information may subsequently also be communicated by the authorization server securely to the resource server, for example within the audience field of the access token.

QUESTION: A benefit of asymmetric cryptography is to allow clients to request a PoP token for use with multiple resource servers. The downside of that approach is linkability since different resource servers will be able to link individual requests to the same client.

(The same is true if the a single public key is linked with PoP tokens used with different resource servers.) Nevertheless, to support the functionality the audience parameter could carry an array of values. Is this desirable?

3.1. Audience Parameter

The client constructs the access token request to the token endpoint by adding the 'aud' parameter using the "application/x-www-form-urlencoded" format with a character encoding of UTF-8 in the HTTP request entity-body.

The URI included in the aud parameter 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 for the 'aud' element is defined in Appendix A.

3.2. Processing Instructions

Step (0): As an initial step the client typically determines the resource server it wants to interact with. This may, for example, happen as part of a discovery procedure or via manual configuration.

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 authorization server who obtains the request from the client needs to parse it to determine whether the provided audience value matches any of the 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 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 [9] the "aud" claim MUST be used. The access token, if

passed per value, MUST be protected against modification by either using a digital signature or a keyed message digest. Access tokens can also be passed by reference, which then requires the token introspection endpoint (or a similiar, proprietary protocol mechanism) to be used. The authorization server returns the access token to the client, as specified in [2].

Subsequent steps for the interaction between the client and the resource server are beyond the scope of this document.

4. Symmetric Key Transport

4.1. Client-to-AS Request

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

token_type: OPTIONAL. See Section 6 for more details.

alg: OPTIONAL. See Section 6 for more details.

These two new parameters are optional in the case where the authorization server has prior knowledge of the capabilities of the client otherwise these two parameters are required. This prior knowledge may, for example, be set by the use of a dynamic client registration protocol exchange.

QUESTION: Should we register these two parameters for use with the dynamic client registration protocol?

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

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

grant_type=authorization_code
&code=Splxl0BeZQQYbYS6WxSbIA
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
&token_type=pop
&alg=HS256
```

Example Request to the Authorization Server

4.2. Client-to-AS Response

If the access token request has been successfully verified by the authorization server and the client is authorized to obtain a PoP token for the indicated resource server, the authorization server issues an access token and optionally a refresh token. If client authentication failed or is invalid, the authorization server returns an error response as described in Section 5.2 of [2].

The authorization server MUST include an access token and a 'key' element in a successful response. The 'key' parameter either contains a plain JWK structure or a JWK encrypted with a JWE. The difference between the two approaches is the following:

Plain JWK: If the JWK container is placed in the 'key' element then the security of the overall PoP architecture relies on Transport Layer Security (TLS) between the authorization server and the client. Figure 2 illustrates an example response using a plain JWK for key transport from the authorization server to the client.

JWK protected by a JWE: If the JWK container is protected by a JWE then additional security protection at the application layer is provided between the authorization server and the client beyond the use of TLS. This approach is a reasonable choice, for example, when a hardware security module is available on the client device and confidentiality protection can be offered directly to this hardware security module.

Note that there are potentially two JSON-encoded structures in the response, namely the access token (with the recommended JWT encoding) and the actual key transport mechanism itself. Note, however, that the two structures serve a different purpose and are consumed by different parties. The access token is created by the authorization server and processed by the resource server (and opaque to the

client) whereas the key transport payload is created by the authorization server and processed by the client; it is never forwarded to the resource server.

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

{
  "access_token": "SlAV32hkKG ...
    (remainder of JWT omitted for brevity;
    JWT contains JWK in the cnf claim)",
  "token_type": "pop",
  "expires_in": 3600,
  "refresh_token": "8xLOxBtZp8",
  "key": "eyJhbGciOiJSU0ExXzUi ...
    (remainder of plain JWK omitted for brevity)"
}
```

Figure 2: Example: Response from the Authorization Server (Symmetric Variant)

The content of the key parameter, which is a JWK in our example, is shown in Figure 3.

```
{
  "kty": "oct",
  "kid": "id123",
  "alg": "HS256",
  "k": "ZoRSOrFzN_FzUA5XKMYoVHyzzff5oRJxl-IXRtztJ6uE"
}
```

Figure 3: Example: Key Transport to Client via a JWK

The content of the 'access_token' in JWT format contains the 'cnf' (confirmation) claim, as shown in Figure 4. The confirmation claim is defined in [10]. The digital signature or the keyed message digest offering integrity protection is not shown in this example but MUST be present in a real deployment to mitigate a number of security threats. Those security threats are described in [17].

The JWK in the key element of the response from the authorization server, as shown in Figure 2, contains the same session key as the JWK inside the access token, as shown in Figure 4. It is, in this

example, protected by TLS and transmitted from the authorization server to the client (for processing by the client).

```
{
  "iss": "https://server.example.com",
  "sub": "24400320",
  "aud": "s6BhdRkqt3",
  "nonce": "n-0S6_WzA2Mj",
  "exp": 1311281970,
  "iat": 1311280970,
  "cnf": {
    "jwk":
      "JDLUhTMjU2IiwiY3R5Ijoi ...
      (remainder of JWK protected by JWE omitted for brevity)"
  }
}
```

Figure 4: Example: Access Token in JWT Format

Note: When the JWK inside the access token contains a symmetric key it MUST be confidentiality protected using a JWE to maintain the security goals of the PoP architecture, as described in [17] since content is meant for consumption by the selected resource server only.

Note: This document does not impose requirements on the encoding of the access token. The examples used in this document make use of the JWT structure since this is the only standardized format.

If the access token is only a reference then a look-up by the resource server is needed, as described in the token introspection specification [18].

5. Asymmetric Key Transport

5.1. Client-to-AS Request

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

token_type: OPTIONAL. See Section 6 for more details.

alg: OPTIONAL. See Section 6 for more details.

key: OPTIONAL. This field contains information about the public key the client would like to bind to the access token in the JWK format. If the client does not provide a public key then the authorization server MUST create an ephemeral key pair (considering the information provided by the client) or alternatively respond with an error message. The client may also convey the fingerprint of the public key to the authorization server instead of passing the entire public key along (to conserve bandwidth). [11] defines a way to compute a thumbprint for a JWK and to embed it within the JWK format.

The 'token_type' and the 'alg' parameters are optional in the case where the authorization server has prior knowledge of the capabilities of the client otherwise these two parameters are required.

For example, the client makes the following HTTP request using TLS (extra line breaks are for display purposes only) shown in Figure 5.

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

grant_type=authorization_code
&code=SpIxl0BeZQQYbYS6WxSbIA
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
&token_type=pop
&alg=RS256
&key=eyJhbGciOiJSU0ExXzUi ...
(remainder of JWK omitted for brevity)
```

Figure 5: Example Request to the Authorization Server (Asymmetric Key Variant)

As shown in Figure 6 the content of the 'key' parameter contains the RSA public key the client would like to associate with the access token.

```
{
  "kty": "RSA",
  "n": "0vx7agoebGcQSuuPiLJXZptN9nndrQmbXEps2aiAFbWhM78LhWx
4cbbfAAAtVT86zwulRK7aPFFxuhDR1L6tSoc_BJECPEbWKRXjBZCiFV4n3oknjhMs
tn64tZ_2W-5JsGY4Hc5n9yBXArwl93lqt7_RN5w6Cf0h4QyQ5v-65YGjQR0_FDW2
QvzqY368QQMicAtaSqzs8KJZgnYb9c7d0zgdAZHzu6QMqvRL5hajrnln91CbOpbI
SD08qNLyrdkt-bFTWhAI4vMQFh6WeZu0fM4lFd2NcRwr3XPksINHaQ-G_xBniIqb
w0LsljF44-csFCur-kEgU8awapJzKnqDKgw",
  "e": "AQAB",
  "alg": "RS256",
  "kid": "id123"
}
```

Figure 6: Client Providing Public Key to Authorization Server

5.2. Client-to-AS Response

If the access token request is valid and authorized, the authorization server issues an access token and optionally a refresh token. If the request client authentication failed or is invalid, the authorization server returns an error response as described in Section 5.2 of [2].

The authorization server also places information about the public key used by the client into the access token to create the binding between the two. The new token type "public_key" is placed into the 'token_type' parameter.

An example of a successful response is shown in Figure 7.

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

{
  "access_token": "2YotnFZFE....jrlzCsicMWpAA",
  "token_type": "pop",
  "alg": "RS256",
  "expires_in": 3600,
  "refresh_token": "tGzv3JOkF0XG5Qx2TlKWIA"
}
```

Figure 7: Example: Response from the Authorization Server (Asymmetric Variant)

The content of the 'access_token' field contains an encoded JWT with the following structure, as shown in Figure 8. The digital signature

or the keyed message digest offering integrity protection is not shown (but must be present).

```
{
  "iss": "xas.example.com",
  "aud": "http://auth.example.com",
  "exp": "1361398824",
  "nbf": "1360189224",
  "cnf": {
    "jwk": { "kty": "RSA",
              "n": "0vx7agoebGcQSuuPiLJXZptN9nndrQmbXEps2aiAFbWhM78LhWx
4cbbfAAAtVT86zwulRK7aPFFxuhDR1L6tSoc_BJECPEbWKRXjBZCiFV4n3oknjhMs
tn64tZ_2W-5JsGY4Hc5n9yBXArwl93lqt7_RN5w6Cf0h4QyQ5v-65YGjQR0_FDW2
QvzqY368QQMicAtaSqzs8KJZgnYb9c7d0zgdAZHzu6QMqvRL5hajrnl9lCbOpbI
SD08qNLyrdkt-bFTWhAI4vMQFh6WeZu0fM4lFd2NcRwr3XPksINHaQ-G_xBniIqb
w0LsljF44-csFCur-kEgU8awapJzKnqDKgw",
              "e": "AQAB",
              "alg": "RS256",
              "kid": "id123"
            }
  }
}
```

Figure 8: Example: Access Token Structure (Asymmetric Variant)

Note: In this example there is no need for the authorization server to convey further keying material to the client since the client is already in possession of the private RSA key.

6. Token Types and Algorithms

To allow clients to indicate support for specific token types and respective algorithms they need to interact with authorization servers. They can either provide this information out-of-band, for example, via pre-configuration or up-front via the dynamic client registration protocol [16].

The value in the 'alg' parameter together with value from the 'token_type' parameter allow the client to indicate the supported algorithms for a given token type. The token type refers to the specification used by the client to interact with the resource server to demonstrate possession of the key. The 'alg' parameter provides further information about the algorithm, such as whether a symmetric or an asymmetric crypto-system is used. Hence, a client supporting a specific token type also knows how to populate the values to the 'alg' parameter.

The value for the 'token_type' MUST be taken from the 'OAuth Access Token Types' registry created by [2].

This document does not register a new value for the OAuth Access Token Types registry nor does it define values to be used for the 'alg' parameter since this is the responsibility of specifications defining the mechanism for clients interacting with resource servers. An example of such specification can be found in [19].

The values in the 'alg' parameter are case-sensitive. If the client supports more than one algorithm then each individual value MUST be separated by a space.

7. Security Considerations

[17] describes the architecture for the OAuth 2.0 proof-of-possession security architecture, including use cases, threats, and requirements. This requirements describes one solution component of that architecture, namely the mechanism for the client to interact with the authorization server to either obtain a symmetric key from the authorization server, to obtain an asymmetric key pair, or to offer a public key to the authorization. In any case, these keys are then bound to the access token by the authorization server.

To summarize the main security recommendations: A large range of threats can be mitigated by protecting the contents of the access token by using a digital signature or a keyed message digest. Consequently, the token integrity protection MUST be applied to prevent the token from being modified, particularly since it contains a reference to the symmetric key or the asymmetric key. If the access token contains the symmetric key (see Section 2.2 of [10] for a description about how symmetric keys can be securely conveyed within the access token) this symmetric key MUST be encrypted by the authorization server with a long-term key shared with the resource server.

To deal with token redirect, it is important for the authorization server to include the identity of the intended recipient (the audience), typically a single resource server (or a list of resource servers), in the token. Using a single shared secret with multiple authorization server to simplify key management is NOT RECOMMENDED since the benefit from using the proof-of-possession concept is significantly reduced.

Token replay is also not possible since an eavesdropper will also have to obtain the corresponding private key or shared secret that is bound to the access token. Nevertheless, it is good practice to

limit the lifetime of the access token and therefore the lifetime of associated key.

The authorization server MUST offer confidentiality protection for any interactions with the client. This step is extremely important since the client will obtain the session key from the authorization server for use with a specific access token. Not using confidentiality protection exposes this secret (and the access token) to an eavesdropper thereby making the OAuth 2.0 proof-of-possession security model completely insecure. OAuth 2.0 [2] relies on TLS to offer confidentiality protection and additional protection can be applied using the JWK [5] offered security mechanism, which would add an additional layer of protection on top of TLS for cases where the keying material is conveyed, for example, to a hardware security module. Which version(s) of TLS ought to be implemented will vary over time, and depend on the widespread deployment and known security vulnerabilities at the time of implementation. At the time of this writing, TLS version 1.2 [4] is the most recent version. The client MUST validate the TLS certificate chain when making requests to protected resources, including checking the validity of the certificate.

Similarly to the security recommendations for the bearer token specification [12] developers MUST ensure that the ephemeral credentials (i.e., the private key or the session key) is not leaked to third parties. An adversary in possession of the ephemeral credentials bound to the access token will be able to impersonate the client. Be aware that this is a real risk with many smart phone app and Web development environments.

Clients can at any time request a new proof-of-possession capable access token. Using a refresh token to regularly request new access tokens that are bound to fresh and unique keys is important. Keeping the lifetime of the access token short allows the authorization server to use shorter key sizes, which translate to a performance benefit for the client and for the resource server. Shorter keys also lead to shorter messages (particularly with asymmetric keying material).

When authorization servers bind symmetric keys to access tokens then they SHOULD scope these access tokens to a specific permissions.

8. IANA Considerations

This specification registers the following parameters in the OAuth Parameters Registry established by [2].

Parameter name: alg

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

Change controller: IETF

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

Related information: None

Parameter name: key

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

Change controller: IETF

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

Related information: None

Parameter name: aud

Parameter usage location: token request

Change controller: IETF

Specification document(s): [[This document.]]

Related information: None

9. Acknowledgements

We would like to thank Chuck Mortimore for his review comments.

10. References

10.1. Normative References

- [1] 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>>.
- [2] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <<http://www.rfc-editor.org/info/rfc6749>>.

- [3] 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>>.
- [4] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <<http://www.rfc-editor.org/info/rfc5246>>.
- [5] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, <<http://www.rfc-editor.org/info/rfc7517>>.
- [6] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <<http://www.rfc-editor.org/info/rfc7515>>.
- [7] Jones, M., "JSON Web Algorithms (JWA)", RFC 7518, DOI 10.17487/RFC7518, May 2015, <<http://www.rfc-editor.org/info/rfc7518>>.
- [8] Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)", RFC 7516, DOI 10.17487/RFC7516, May 2015, <<http://www.rfc-editor.org/info/rfc7516>>.
- [9] 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>>.
- [10] 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, <<http://www.rfc-editor.org/info/rfc7800>>.
- [11] Jones, M. and N. Sakimura, "JSON Web Key (JWK) Thumbprint", RFC 7638, DOI 10.17487/RFC7638, September 2015, <<http://www.rfc-editor.org/info/rfc7638>>.

10.2. Informative References

- [12] 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>>.

- [13] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <<http://www.rfc-editor.org/info/rfc5234>>.
- [14] Campbell, B., Mortimore, C., Jones, M., and Y. Goland, "Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants", RFC 7521, DOI 10.17487/RFC7521, May 2015, <<http://www.rfc-editor.org/info/rfc7521>>.
- [15] 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, <<http://www.rfc-editor.org/info/rfc7636>>.
- [16] 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, <<http://www.rfc-editor.org/info/rfc7591>>.
- [17] Hunt, P., Richer, J., Mills, W., Mishra, P., and H. Tschofenig, "OAuth 2.0 Proof-of-Possession (PoP) Security Architecture", draft-ietf-oauth-pop-architecture-08 (work in progress), July 2016.
- [18] Richer, J., Ed., "OAuth 2.0 Token Introspection", RFC 7662, DOI 10.17487/RFC7662, October 2015, <<http://www.rfc-editor.org/info/rfc7662>>.
- [19] Richer, J., Bradley, J., and H. Tschofenig, "A Method for Signing HTTP Requests for OAuth", draft-ietf-oauth-signed-http-request-03 (work in progress), August 2016.

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

This section provides Augmented Backus-Naur Form (ABNF) syntax descriptions for the elements defined in this specification using the notation of [13].

A.1. 'aud' Syntax

The ABNF syntax is defined as follows where by the "URI-reference" definition is taken from [3]:

aud = URI-reference

A.2. 'key' Syntax

The "key" element is defined in Section 4 and Section 5:

key = 1*VSCHAR

A.3. 'alg' Syntax

The "alg" element is defined in Section 6:

alg = alg-token *(SP alg-token)

alg-token = 1*NQCHAR

Authors' Addresses

John Bradley
Ping Identity

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

Phil Hunt
Oracle Corporation

Email: phil.hunt@yahoo.com
URI: <http://www.indepdentid.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: February 2, 2020

D. Hardt
August 01, 2019

Reciprocal OAuth
draft-ietf-oauth-reciprocal-04

Abstract

There are times when a user has a pair of protected resources that would like to request access to each other. While OAuth flows typically enable the user to grant a client access to a protected resource, granting the inverse access requires an additional flow. Reciprocal OAuth enables a more seamless experience for the user to grant access to a pair of protected resources.

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 2, 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.

1. Introduction

In the usual three legged, authorization code grant, the OAuth flow enables a resource owner (user) to enable a client (party A) to be granted authorization to access a protected resource (party B). If party A also has a protected resource that the user would like to let party B access, then a second complete OAuth flow, but in the reverse direction, must be performed. In practice, this is a complicated user experience as the user is at Party A, but the OAuth flow needs to start from Party B. This requires the second flow to send the user back to party B, which then sends the user to Party A as the first step in the flow. At the end, the user is at Party B, even though the original flow started at Party A.

Reciprocal OAuth simplifies the user experience by eliminating the redirections in the second OAuth flow. After the initial OAuth flow, party A obtains consent from the user to grant party B access to a protected resource at party A, and then passes an authorization code to party B using the access token party A obtained from party B to provide party B the context of the user. Party B then exchanges the authorization code for an access token per the usual OAuth flow.

For example, a user would like their voice assistant (party A) and music service (party B) to work together. The voice assistant wants to call the music service to play music, and the music service wants to call the voice assistant with music information to present to the user. The user starts the OAuth flow at the voice assistant, and is redirected to the music service. The music services obtains consent from the user and the redirects back to the voice assistant. At this point the voice assistant is able to obtain an access token for the music service. The voice assistant can then get consent from the user to authorize the music service to access the voice assistant, and then the voice assistant can create an authorization code and send it to the music service, which then exchanges the authorization code for an access token, all without further user interaction. Note that either the voice assistant or the music service can initiate the flow, so that either can prompt the user for the two parties to work together.

1.1. Terminology

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, RFC 2119 [RFC2119].

2. reciprocal Protocol Flow

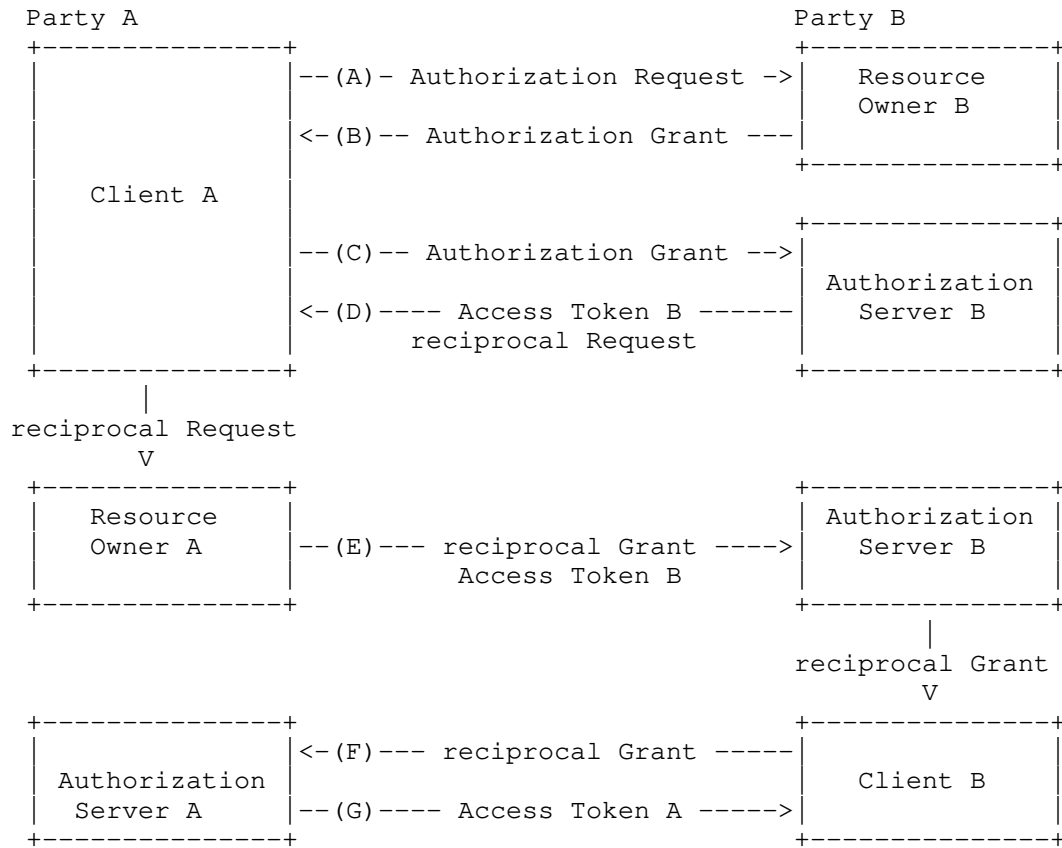


Figure 1: Abstract reciprocal Protocol Flow

The reciprocal authorization between party A and party B are abstractly represented in Figure 1 and includes the following steps:

- o (A - C) are the same as in [RFC6749] 1.2
- o (D) Party B optionally includes the reciprocal scope in the response. See Section 2.1 for details.
- o (E) Party A sends the reciprocal authorization grant to party B. See Section 2.2.2 for details.
- o (F) Party B requests an access token, mirroring step (B)
- o (G) Party A issues an access token, mirroring step (C)

Note that Resource Owner A and Resource Owner B are the respective resource owner interaction systems controlled by the same owner.

2.1. Reciprocal Scope Request

When party B is providing an access token response per [RFC6749] 4.1.4, 4.2.1, 4.3.3 or 4.4.3, party B MAY include an additional query component in the redirection URI to indicate the scope requested in the reciprocal grant:

reciprocal OPTIONAL

The scope of party B's reciprocal access request per [RFC6749] 3.3.

If party B does not provide a reciprocal parameter in the access token response, the reciprocal scope will be a value previously preconfigured by party A and party B.

If an authorization code grant access token response per [RFC6749] 4.1.4, an example successful response (with extra line breaks for display purposes only):

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

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

If an authorization code grant access token response per [RFC6749] 4.2.2, an example successful response (with extra line breaks for display purposes only):

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

When party B is providing an authorization response per [RFC6749] 4.1.2, party B MAY include an additional query component in the redirection URI to indicate the scope requested in the reciprocal grant.

reciprocal OPTIONAL. The scope of party B's reciprocal access request per [RFC6749] 3.3.

If party B does not provide a reciprocal parameter in the authorization response, the reciprocal scope will be a value previously preconfigured by party A and party B.

2.2. Reciprocal Authorization Flow

The reciprocal authorization flow starts after the client (party A) has obtained an access token from the authorization server (party B) per [RFC6749] 4.1 Authorization Code Grant.

2.2.1. User Consent

Party A obtains consent from the user to grant Party B access to protected resources at party A. The consent represents the scopes requested by party B from party A per Section 2.1.

2.2.2. Reciprocal Authorization Code

Party A generates an authorization code representing the access granted to party B by the user. Party A then makes a request to party B's token endpoint authenticating per [RFC6749] 2.3 and sending the following parameters using the "application/x-www-form-urlencoded" format per [RFC6749] Appendix B with a character encoding of UTF-8 in the HTTP request entity-body:

grant_type REQUIRED

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

code REQUIRED

the authorization code generated by party A.

client_id REQUIRED

party A's client ID.

access_token REQUIRED the access token obtained from Party B. Used by Party B to identify which user authorization is being requested.

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

```
POST /token HTTP/1.1
Host: server.example.com
Authorization: Basic ej4hsyfishwssjdusisdhjkjsdksusdhjkjsdjk
Content-Type: application/x-www-form-urlencoded

grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Areciprocal
&code=hasdyubasdjahsbdkjbasd
&client_id=example.com
&access_token=sadadojsadlkjasdkljxxlkjdas
```

Party B MUST verify the authentication provided by Party A per [RFC6749] 2.3

Party B MUST then verify the access token was granted to the client identified by the `client_id`.

Party B MUST respond with either an HTTP 200 (OK) response if the request is valid, or an HTTP 400 "Bad Request" if it is not.

Party B then plays the role of the client to make an access token request per [RFC6749] 4.1.3.

3. Authorization Update Flow

After the initial authorization, the user may add or remove scopes available to the client at the authorization server. For example, the user may grant additional scopes to the client using a voice interface, or revoke some scopes. The authorization server can update the client with the new authorization by sending a new authorization code per Section 2.2.2.

4. IANA Considerations

4.1. Registration of reciprocal

This section registers the value "reciprocal" in the IANA "OAuth Parameters" registry established by "The OAuth 2.0 Authorization Framework" [RFC6749].

- o Parameter Name: reciprocal
- o Parameter usage location: token response
- o Change Controller: IESG
- o Specification Document: Section 2.1 of this document

4.2. Sub-Namespace Registration of urn:ietf:params:oauth:grant-type:reciprocal

This section registers the value "grant-type:reciprocal" in the IANA "OAuth URI" registry established by "An IETF URN Sub-Namespace for OAuth" [RFC6755].

- o URN: urn:ietf:params:oauth:grant-type:reciprocal
- o Common Name: reciprocal grant for OAuth 2.0
- o Change Controller: IESG
- o Specification Document: Section 2.2.2 of this document

5. 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>>.
- [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>>.
- [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>>.

Appendix A. Document History

A.1. draft-ietf-oauth-reciprocal-00

- o Initial version.

A.2. draft-ietf-oauth-reciprocal-01

- o Changed reciprocal scope request to be in access token response rather than authorization request

A.3. draft-ietf-oauth-reciprocal-02

- o Added in diagram to clarify protocol flow

A.4. draft-ietf-oauth-reciprocal-03

- o fixed spelling of reciprocal
- o added example use case in introduction
- o resource owner is the same in Party A and Party B

A.5. draft-ietf-oauth-reciprocal-04

- o completed IANA section

Author's Address

Dick Hardt

Email: dick.hardt@gmail.com

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 Referrer 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 Referrer 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 Referrer 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=eyJhbGciOiJIUzI1IiwiaXNjaWkiOiJ1IiwiaWF0IjoiMTYxMjM0OTYyIn08xy5W5sQ
```

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 "4jTc5elQpocqPTZ5l6jsb6pRP18IFKdwwPvasYjnl-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-_bGnGT6DizxUK-m5n3dQUIkeH7ybn6wb1C5dGyV_IAAQDDFTtoFrHt41Zppq7
  u_SEMF_E-KimAB-HewWl2MvZzAQ9QKoWiJCLFiCkjpgtr1RrA2-jaJvoB8o51DTGXQ
  ydWYkAAAECAEFaUc1G1YU83rqTGHEauloqvNwy0fDsdXzIyT_4x1FcldsMxjFkJac
  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": "eyJhbGciOiJIJFZlIiwiaXNpZmtpIjoiYm9vaW8o51DTGXQ",
  "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=dry08YFpWacBUPjhBf4Nvt51
  &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_OVfBNqrIQxzIfkF7tWby2ZfyaE6XpwTsAQBYqhFX78vM0gDX_F
  d_b2dlHyHlMmkIz8iMVBY_reM98OUaJFz5IB7PG9nZ11j58LoG5QhmQoI9NXYktKZ
  RXxrYAAAECAEFAdUFTnfQADknluDbQnvJEk6oQs38L92gv-KO-qlYadLoDIKe2h53
  hSiKwIP98iRj_unedkNkAMyg9e2mY4Gp7WwBAeDUOwaSXNz1e6gKohwN4SAZ5eNyx
  45Mh8VI4woLlBipLoqrJR0K6dxFkWGHRMuBR0cLGUj5PiOoxybQH_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: January 21, 2020

M. Jones
A. Nadalin
Microsoft
B. Campbell, Ed.
Ping Identity
J. Bradley
Yubico
C. Mortimore
Salesforce
July 20, 2019

OAuth 2.0 Token Exchange
draft-ietf-oauth-token-exchange-19

Abstract

This specification defines a protocol for an HTTP- and JSON- based Security Token Service (STS) by defining how to request and obtain security tokens from OAuth 2.0 authorization servers, including security tokens employing impersonation and delegation.

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 January 21, 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. Delegation vs. Impersonation Semantics	4
1.2. Requirements Notation and Conventions	6
1.3. Terminology	6
2. Token Exchange Request and Response	6
2.1. Request	6
2.1.1. Relationship Between Resource, Audience and Scope . .	9
2.2. Response	9
2.2.1. Successful Response	9
2.2.2. Error Response	11
2.3. Example Token Exchange	11
3. Token Type Identifiers	13
4. JSON Web Token Claims and Introspection Response Parameters .	15
4.1. "act" (Actor) Claim	15
4.2. "scope" (Scopes) Claim	17
4.3. "client_id" (Client Identifier) Claim	18
4.4. "may_act" (Authorized Actor) Claim	18
5. Security Considerations	19
6. Privacy Considerations	20
7. IANA Considerations	20
7.1. OAuth URI Registration	20
7.1.1. Registry Contents	20
7.2. OAuth Parameters Registration	21
7.2.1. Registry Contents	21
7.3. OAuth Access Token Type Registration	22
7.3.1. Registry Contents	22
7.4. JSON Web Token Claims Registration	22
7.4.1. Registry Contents	22
7.5. OAuth Token Introspection Response Registration	23
7.5.1. Registry Contents	23
7.6. OAuth Extensions Error Registration	23
7.6.1. Registry Contents	23
8. References	24
8.1. Normative References	24
8.2. Informative References	24
Appendix A. Additional Token Exchange Examples	26
A.1. Impersonation Token Exchange Example	26
A.1.1. Token Exchange Request	26
A.1.2. Subject Token Claims	26
A.1.3. Token Exchange Response	27

A.1.4. Issued Token Claims	27
A.2. Delegation Token Exchange Example	28
A.2.1. Token Exchange Request	28
A.2.2. Subject Token Claims	29
A.2.3. Actor Token Claims	29
A.2.4. Token Exchange Response	29
A.2.5. Issued Token Claims	30
Appendix B. Acknowledgements	31
Appendix C. Document History	31
Authors' Addresses	35

1. Introduction

A security token is a set of information that facilitates the sharing of identity and security information in heterogeneous environments or across security domains. Examples of security tokens include JSON Web Tokens (JWTs) [JWT] and SAML 2.0 Assertions [OASIS.saml-core-2.0-os]. Security tokens are typically signed to achieve integrity and sometimes also encrypted to achieve confidentiality. Security tokens are also sometimes described as Assertions, such as in [RFC7521].

A Security Token Service (STS) is a service capable of validating security tokens provided to it and issuing new security tokens in response, which enables clients to obtain appropriate access credentials for resources in heterogeneous environments or across security domains. Web Service clients have used WS-Trust [WS-Trust] as the protocol to interact with an STS for token exchange. While WS-Trust uses XML and SOAP, the trend in modern Web development has been towards RESTful patterns and JSON. The OAuth 2.0 Authorization Framework [RFC6749] and OAuth 2.0 Bearer Tokens [RFC6750] have emerged as popular standards for authorizing third-party applications' access to HTTP and RESTful resources. The conventional OAuth 2.0 interaction involves the exchange of some representation of resource owner authorization for an access token, which has proven to be an extremely useful pattern in practice. However, its input and output are somewhat too constrained as is to fully accommodate a security token exchange framework.

This specification defines a protocol extending OAuth 2.0 that enables clients to request and obtain security tokens from authorization servers acting in the role of an STS. Similar to OAuth 2.0, this specification focuses on client developer simplicity and requires only an HTTP client and JSON parser, which are nearly universally available in modern development environments. The STS protocol defined in this specification is not itself RESTful (an STS doesn't lend itself particularly well to a REST approach) but does

utilize communication patterns and data formats that should be familiar to developers accustomed to working with RESTful systems.

A new grant type for a token exchange request and the associated specific parameters for such a request to the token endpoint are defined by this specification. A token exchange response is a normal OAuth 2.0 response from the token endpoint with a few additional parameters defined herein to provide information to the client.

The entity that makes the request to exchange tokens is considered the client in the context of the token exchange interaction. However, that does not restrict usage of this profile to traditional OAuth clients. An OAuth resource server, for example, might assume the role of the client during token exchange in order to trade an access token that it received in a protected resource request for a new token that is appropriate to include in a call to a backend service. The new token might be an access token that is more narrowly scoped for the downstream service or it could be an entirely different kind of token.

The scope of this specification is limited to the definition of a basic request-and-response protocol for an STS-style token exchange utilizing OAuth 2.0. Although a few new JWT claims are defined that enable delegation semantics to be expressed, the specific syntax, semantics and security characteristics of the tokens themselves (both those presented to the authorization server and those obtained by the client) are explicitly out of scope and no requirements are placed on the trust model in which an implementation might be deployed. Additional profiles may provide more detailed requirements around the specific nature of the parties and trust involved, such as whether signing and/or encryption of tokens is needed or if proof-of-possession style tokens will be required or issued; however, such details will often be policy decisions made with respect to the specific needs of individual deployments and will be configured or implemented accordingly.

The security tokens obtained may be used in a number of contexts, the specifics of which are also beyond the scope of this specification.

1.1. Delegation vs. Impersonation Semantics

One common use case for an STS (as alluded to in the previous section) is to allow a resource server A to make calls to a backend service C on behalf of the requesting user B. Depending on the local site policy and authorization infrastructure, it may be desirable for A to use its own credentials to access C along with an annotation of some form that A is acting on behalf of B ("delegation"), or for A to be granted a limited access credential to C but that continues to

identify B as the authorized entity ("impersonation"). Delegation and impersonation can be useful concepts in other scenarios involving multiple participants as well.

When principal A impersonates principal B, A is given all the rights that B has within some defined rights context and is indistinguishable from B in that context. Thus, when principal A impersonates principal B, then insofar as any entity receiving such a token is concerned, they are actually dealing with B. It is true that some members of the identity system might have awareness that impersonation is going on, but it is not a requirement. For all intents and purposes, when A is impersonating B, A is B within the context of the rights authorized by the token. A's ability to impersonate B could be limited in scope or time, or even with a one-time-use restriction, whether via the contents of the token or an out-of-band mechanism.

Delegation semantics are different than impersonation semantics, though the two are closely related. With delegation semantics, principal A still has its own identity separate from B and it is explicitly understood that while B may have delegated some of its rights to A, any actions taken are being taken by A representing B. In a sense, A is an agent for B.

Delegation and impersonation are not inclusive of all situations. When a principal is acting directly on its own behalf, for example, neither delegation nor impersonation are in play. They are, however, the more common semantics operating for token exchange and, as such, are given more direct treatment in this specification.

Delegation semantics are typically expressed in a token by including information about both the primary subject of the token as well as the actor to whom that subject has delegated some of its rights. Such a token is sometimes referred to as a composite token because it is composed of information about multiple subjects. Typically, in the request, the "subject_token" represents the identity of the party on behalf of whom the token is being requested while the "actor_token" represents the identity of the party to whom the access rights of the issued token are being delegated. A composite token issued by the authorization server will contain information about both parties. When and if a composite token is issued is at the discretion of the authorization server and applicable policy and configuration.

The specifics of representing a composite token and even whether or not such a token will be issued depend on the details of the implementation and the kind of token. The representations of composite tokens that are not JWTs are beyond the scope of this

specification. The "actor_token" request parameter, however, does provide a means for providing information about the desired actor and the JWT "act" claim can provide a representation of a chain of delegation.

1.2. 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.

1.3. Terminology

This specification uses the terms "access token type", "authorization server", "client", "client identifier", "resource server", "token endpoint", "token request", and "token response" defined by OAuth 2.0 [RFC6749], and the terms "Base64url Encoding", "Claim", and "JWT Claims Set" defined by JSON Web Token (JWT) [JWT].

2. Token Exchange Request and Response

2.1. Request

A client requests a security token by making a token request to the authorization server's token endpoint using the extension grant type mechanism defined in Section 4.5 of [RFC6749].

Client authentication to the authorization server is done using the normal mechanisms provided by OAuth 2.0. Section 2.3.1 of [RFC6749] defines password-based authentication of the client, however, client authentication is extensible and other mechanisms are possible. For example, [RFC7523] defines client authentication using bearer JSON Web Tokens (JWTs) [JWT]. The supported methods of client authentication and whether or not to allow unauthenticated or unidentified clients are deployment decisions that are at the discretion of the authorization server. Note that omitting client authentication allows for a compromised token to be leveraged via an STS into other tokens by anyone possessing the compromised token. Thus client authentication allows for additional authorization checks by the STS as to which entities are permitted to impersonate or receive delegations from other entities.

The client makes a token exchange request to the token endpoint with an extension grant type using the HTTP "POST" method. The following parameters are included in the HTTP request entity-body using the

"application/x-www-form-urlencoded" format with a character encoding of UTF-8 as described in Appendix B of RFC6749 [RFC6749].

grant_type

REQUIRED. The value "urn:ietf:params:oauth:grant-type:token-exchange" indicates that a token exchange is being performed.

resource

OPTIONAL. A URI that indicates the target service or resource where the client intends to use the requested security token. This enables the authorization server to apply policy as appropriate for the target, such as determining the type and content of the token to be issued or if and how the token is to be encrypted. In many cases, a client will not have knowledge of the logical organization of the systems with which it interacts and will only know a URI of the service where it intends to use the token. The "resource" parameter allows the client to indicate to the authorization server where it intends to use the issued token by providing the location, typically as an https URL, in the token exchange request in the same form that will be used to access that resource. The authorization server will typically have the capability to map from a resource URI value to an appropriate policy. The value of the "resource" parameter MUST be an absolute URI, as specified by Section 4.3 of [RFC3986], which MAY include a query component and MUST NOT include a fragment component. Multiple "resource" parameters may be used to indicate that the issued token is intended to be used at the multiple resources listed. See [I-D.ietf-oauth-resource-indicators] for additional background and uses of the "resource" parameter.

audience

OPTIONAL. The logical name of the target service where the client intends to use the requested security token. This serves a purpose similar to the "resource" parameter, but with the client providing a logical name for the target service. Interpretation of the name requires that the value be something that both the client and the authorization server understand. An OAuth client identifier, a SAML entity identifier [OASIS.saml-core-2.0-os], an OpenID Connect Issuer Identifier [OpenID.Core], are examples of things that might be used as "audience" parameter values. However, "audience" values used with a given authorization server must be unique within that server, to ensure that they are properly interpreted as the intended type of value. Multiple "audience" parameters may be used to indicate that the issued token is intended to be used at the multiple audiences listed. The "audience" and "resource" parameters may be used together to indicate multiple target services with a mix of logical names and resource URIs.

scope

OPTIONAL. A list of space-delimited, case-sensitive strings, as defined in Section 3.3 of [RFC6749], that allow the client to specify the desired scope of the requested security token in the context of the service or resource where the token will be used. The values and associated semantics of scope are service specific and expected to be described in the relevant service documentation.

requested_token_type

OPTIONAL. An identifier, as described in Section 3, for the type of the requested security token. If the requested type is unspecified, the issued token type is at the discretion of the authorization server and may be dictated by knowledge of the requirements of the service or resource indicated by the "resource" or "audience" parameter.

subject_token

REQUIRED. A security token that represents the identity of the party on behalf of whom the request is being made. Typically, the subject of this token will be the subject of the security token issued in response to the request.

subject_token_type

REQUIRED. An identifier, as described in Section 3, that indicates the type of the security token in the "subject_token" parameter.

actor_token

OPTIONAL. A security token that represents the identity of the acting party. Typically, this will be the party that is authorized to use the requested security token and act on behalf of the subject.

actor_token_type

An identifier, as described in Section 3, that indicates the type of the security token in the "actor_token" parameter. This is REQUIRED when the "actor_token" parameter is present in the request but MUST NOT be included otherwise.

In processing the request, the authorization server MUST perform the appropriate validation procedures for the indicated token type and, if the actor token is present, also perform the appropriate validation procedures for its indicated token type. The validity criteria and details of any particular token are beyond the scope of this document and are specific to the respective type of token and its content.

In the absence of one-time-use or other semantics specific to the token type, the act of performing a token exchange has no impact on the validity of the subject token or actor token. Furthermore, the exchange is a one-time event and does not create a tight linkage between the input and output tokens, so that (for example) while the expiration time of the output token may be influenced by that of the input token, renewal or extension of the input token is not expected to be reflected in the output token's properties. It may still be appropriate or desirable to propagate token revocation events. However, doing so is not a general property of the STS protocol and would be specific to a particular implementation, token type or deployment.

2.1.1. Relationship Between Resource, Audience and Scope

When requesting a token, the client can indicate the desired target service(s) where it intends to use that token by way of the "audience" and "resource" parameters, as well as indicating the desired scope of the requested token using the "scope" parameter. The semantics of such a request are that the client is asking for a token with the requested scope that is usable at all the requested target services. Effectively, the requested access rights of the token are the cartesian product of all the scopes at all the target services.

An authorization server may be unwilling or unable to fulfill any token request but the likelihood of an unfulfillable request is significantly higher when very broad access rights are being solicited. As such, in the absence of specific knowledge about the relationship of systems in a deployment, clients should exercise discretion in the breadth of the access requested, particularly the number of target services. An authorization server can use the "invalid_target" error code, defined in Section 2.2.2, to inform a client that it requested access to too many target services simultaneously.

2.2. Response

The authorization server responds to a token exchange request with a normal OAuth 2.0 response from the token endpoint, as specified in Section 5 of [RFC6749]. Additional details and explanation are provided in the following subsections.

2.2.1. Successful Response

If the request is valid and meets all policy and other criteria of the authorization server, a successful token response is constructed by adding the following parameters to the entity-body of the HTTP

response using the "application/json" media type, as specified by [RFC8259], and an HTTP 200 status code. The parameters are serialized into a JavaScript Object Notation (JSON) structure by adding each parameter at the top level. Parameter names and string values are included as JSON strings. Numerical values are included as JSON numbers. The order of parameters does not matter and can vary.

access_token

REQUIRED. The security token issued by the authorization server in response to the token exchange request. The "access_token" parameter from Section 5.1 of [RFC6749] is used here to carry the requested token, which allows this token exchange protocol to use the existing OAuth 2.0 request and response constructs defined for the token endpoint. The identifier "access_token" is used for historical reasons and the issued token need not be an OAuth access token.

issued_token_type

REQUIRED. An identifier, as described in Section 3, for the representation of the issued security token.

token_type

REQUIRED. A case-insensitive value specifying the method of using the access token issued, as specified in Section 7.1 of [RFC6749]. It provides the client with information about how to utilize the access token to access protected resources. For example, a value of "Bearer", as specified in [RFC6750], indicates that the issued security token is a bearer token and the client can simply present it as is without any additional proof of eligibility beyond the contents of the token itself. Note that the meaning of this parameter is different from the meaning of the "issued_token_type" parameter, which declares the representation of the issued security token; the term "token type" is more typically used with the aforementioned meaning as the structural or syntactical representation of the security token, as it is in all "*_token_type" parameters in this specification. If the issued token is not an access token or usable as an access token, then the "token_type" value "N_A" is used to indicate that an OAuth 2.0 "token_type" identifier is not applicable in that context.

expires_in

RECOMMENDED. The validity lifetime, in seconds, of the token issued by the authorization server. Oftentimes the client will not have the inclination or capability to inspect the content of the token and this parameter provides a consistent and token-type-agnostic indication of how long the token can be expected to be

valid. For example, the value 1800 denotes that the token will expire in thirty minutes from the time the response was generated.

scope

OPTIONAL, if the scope of the issued security token is identical to the scope requested by the client; otherwise, REQUIRED.

refresh_token

OPTIONAL. A refresh token will typically not be issued when the exchange is of one temporary credential (the subject_token) for a different temporary credential (the issued token) for use in some other context. A refresh token can be issued in cases where the client of the token exchange needs the ability to access a resource even when the original credential is no longer valid (e.g., user-not-present or offline scenarios where there is no longer any user entertaining an active session with the client). Profiles or deployments of this specification should clearly document the conditions under which a client should expect a refresh token in response to "urn:ietf:params:oauth:grant-type:token-exchange" grant type requests.

2.2.2. Error Response

If the request itself is not valid or if either the "subject_token" or "actor_token" are invalid for any reason, or are unacceptable based on policy, the authorization server MUST construct an error response, as specified in Section 5.2 of [RFC6749]. The value of the "error" parameter MUST be the "invalid_request" error code.

If the authorization server is unwilling or unable to issue a token for any target service indicated by the "resource" or "audience" parameters, the "invalid_target" error code SHOULD be used in the error response.

The authorization server MAY include additional information regarding the reasons for the error using the "error_description" as discussed in Section 5.2 of [RFC6749].

Other error codes may also be used, as appropriate.

2.3. Example Token Exchange

The following example demonstrates a hypothetical token exchange in which an OAuth resource server assumes the role of the client during the exchange. It trades an access token, which it received in a protected resource request, for a new token that it will use to call to a backend service (extra line breaks and indentation in the examples are for display purposes only).

Figure 1 shows the resource server receiving a protected resource request containing an OAuth access token in the Authorization header, as specified in Section 2.1 of [RFC6750].

```
GET /resource HTTP/1.1
Host: frontend.example.com
Authorization: Bearer accVkjcJyb4BWCxGsndESCJQbdfMogUC5PbRDqceLTC
```

Figure 1: Protected Resource Request

In Figure 2, the resource server assumes the role of client for the token exchange and the access token from the request in Figure 1 is sent to the authorization server using a request as specified in Section 2.1. The value of the "subject_token" parameter carries the access token and the value of the "subject_token_type" parameter indicates that it is an OAuth 2.0 access token. The resource server, acting in the role of the client, uses its identifier and secret to authenticate to the authorization server using the HTTP Basic authentication scheme. The "resource" parameter indicates the location of the backend service, `https://backend.example.com/api`, where the issued token will be used.

```
POST /as/token.oauth2 HTTP/1.1
Host: as.example.com
Authorization: Basic cnMwODpsb25nLXNlY3VyZS1yYW5kb20tc2VjcmV0
Content-Type: application/x-www-form-urlencoded

grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Atoken-exchange
&resource=https%3A%2F%2Fbackend.example.com%2Fapi
&subject_token=accVkjcJyb4BWCxGsndESCJQbdfMogUC5PbRDqceLTC
&subject_token_type=
urn%3Aietf%3Aparams%3Aoauth%3Atoken-type%3Aaccess_token
```

Figure 2: Token Exchange Request

The authorization server validates the client credentials and the "subject_token" presented in the token exchange request. From the "resource" parameter, the authorization server is able to determine the appropriate policy to apply to the request and issues a token suitable for use at `https://backend.example.com`. The "access_token" parameter of the response shown in Figure 3 contains the new token, which is itself a bearer OAuth access token that is valid for one minute. The token happens to be a JWT; however, its structure and format are opaque to the client so the "issued_token_type" indicates only that it is an access token.

```

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

{
  "access_token": "eyJhbGciOiJFUzI1NiIsImtpZCI6IjllciJ9.eyJhdWQiOiJodHRwczovL2JhY2t1bmQuZXhhbXBsZS5jb20iLCJpc3MiOiJodHRwczovL2FzLmV4YW1wbGUuY29tIiwiaXhwIjoxNDQxOTE3NTkzLCJpYXQiOiJlbnN1YiI6ImJkY0bleGFtcGxlLmNvbSIsInNjb3BlIjoieXBpIn0.40y3ZgQedw6rxf59WlwHDD9jryFOr0_Wh3CGozQBihNBhnXEQgU85AI9x3KmsPottVMLPIWvmDCMy5-kdXjwhw",
  "issued_token_type":
    "urn:ietf:params:oauth:token-type:access_token",
  "token_type": "Bearer",
  "expires_in": 60
}

```

Figure 3: Token Exchange Response

The resource server can then use the newly acquired access token in making a request to the backend server as illustrated in Figure 4.

```

GET /api HTTP/1.1
Host: backend.example.com
Authorization: Bearer eyJhbGciOiJFUzI1NiIsImtpZCI6IjllciJ9.eyJhdWQiOiJodHRwczovL2JhY2t1bmQuZXhhbXBsZS5jb20iLCJpc3MiOiJodHRwczovL2FzLmV4YW1wbGUuY29tIiwiaXhwIjoxNDQxOTE3NTkzLCJpYXQiOiJlbnN1YiI6ImJkY0bleGFtcGxlLmNvbSIsInNjb3BlIjoieXBpIn0.40y3ZgQedw6rxf59WlwHDD9jryFOr0_Wh3CGozQBihNBhnXEQgU85AI9x3KmsPottVMLPIWvmDCMy5-kdXjwhw

```

Figure 4: Backend Protected Resource Request

Additional examples can be found in Appendix A.

3. Token Type Identifiers

Several parameters in this specification utilize an identifier as the value to describe the token in question. Specifically, they are the "requested_token_type", "subject_token_type", "actor_token_type" parameters of the request and the "issued_token_type" member of the response. Token type identifiers are URIs. Token Exchange can work with both tokens issued by other parties and tokens from the given authorization server. For the former the token type identifier indicates the syntax (e.g., JWT or SAML 2.0) so the authorization server can parse it; for the latter it indicates what the given authorization server issued it for (e.g., access_token or refresh_token).

The following token type identifiers are defined by this specification. Other URIs MAY be used to indicate other token types.

`urn:ietf:params:oauth:token-type:access_token`

Indicates that the token is an OAuth 2.0 access token issued by the given authorization server.

`urn:ietf:params:oauth:token-type:refresh_token`

Indicates that the token is an OAuth 2.0 refresh token issued by the given authorization server.

`urn:ietf:params:oauth:token-type:id_token`

Indicates that the token is an ID Token, as defined in Section 2 of [OpenID.Core].

`urn:ietf:params:oauth:token-type:saml1`

Indicates that the token is a base64url-encoded SAML 1.1 [OASIS.saml-core-1.1] assertion.

`urn:ietf:params:oauth:token-type:saml2`

Indicates that the token is a base64url-encoded SAML 2.0 [OASIS.saml-core-2.0-os] assertion.

The value `"urn:ietf:params:oauth:token-type:jwt"`, which is defined in Section 9 of [JWT], indicates that the token is a JWT.

The distinction between an access token and a JWT is subtle. An access token represents a delegated authorization decision, whereas JWT is a token format. An access token can be formatted as a JWT but doesn't necessarily have to be. And a JWT might well be an access token but not all JWTs are access tokens. The intent of this specification is that `"urn:ietf:params:oauth:token-type:access_token"` be an indicator that the token is a typical OAuth access token issued by the authorization server in question, opaque to the client, and usable the same manner as any other access token obtained from that authorization server. (It could well be a JWT, but the client isn't and needn't be aware of that fact.) Whereas, `"urn:ietf:params:oauth:token-type:jwt"` is to indicate specifically that a JWT is being requested or sent (perhaps in a cross-domain use-case where the JWT is used as an authorization grant to obtain an access token from a different authorization server as is facilitated by [RFC7523]).

Note that for tokens which are binary in nature, the URI used for conveying them needs to be associated with the semantics of a base64 or other encoding suitable for usage with HTTP and OAuth.

4. JSON Web Token Claims and Introspection Response Parameters

It is useful to have defined mechanisms to express delegation within a token as well as to express authorization to delegate or impersonate. Although the token exchange protocol described herein can be used with any type of token, this section defines claims to express such semantics specifically for JWTs and in an OAuth 2.0 Token Introspection [RFC7662] response. Similar definitions for other types of tokens are possible but beyond the scope of this specification.

Note that the claims not established herein but used in examples and descriptions, such as "iss", "sub", "exp", etc., are defined by [JWT].

4.1. "act" (Actor) Claim

The "act" (actor) claim provides a means within a JWT to express that delegation has occurred and identify the acting party to whom authority has been delegated. The "act" claim value is a JSON object and members in the JSON object are claims that identify the actor. The claims that make up the "act" claim identify and possibly provide additional information about the actor. For example, the combination of the two claims "iss" and "sub" might be necessary to uniquely identify an actor.

However, claims within the "act" claim pertain only to the identity of the actor and are not relevant to the validity of the containing JWT in the same manner as the top-level claims. Consequently, non-identity claims (e.g., "exp", "nbf", and "aud") are not meaningful when used within an "act" claim, and therefore are not used.

Figure 5 illustrates the "act" (actor) claim within a JWT Claims Set. The claims of the token itself are about user@example.com while the "act" claim indicates that admin@example.com is the current actor.

```
{
  "aud": "https://consumer.example.com",
  "iss": "https://issuer.example.com",
  "exp": 1443904177,
  "nbf": 1443904077,
  "sub": "user@example.com",
  "act": {
    "sub": "admin@example.com"
  }
}
```

Figure 5: Actor Claim

A chain of delegation can be expressed by nesting one "act" claim within another. The outermost "act" claim represents the current actor while nested "act" claims represent prior actors. The least recent actor is the most deeply nested. The nested "act" claims serve as a history trail that connects the initial request and subject through the various delegation steps undertaken before reaching the current actor. In this sense, the current actor is considered to include the entire authorization/delegation history, leading naturally to the nested structure described here.

For the purpose of applying access control policy, the consumer of a token MUST only consider the token's top-level claims and the party identified as the current actor by the "act" claim. Prior actors identified by any nested "act" claims are informational only and are not to be considered in access control decisions.

The following example in Figure 6 illustrates nested "act" (actor) claims within a JWT Claims Set. The claims of the token itself are about user@example.com while the "act" claim indicates that the system https://service16.example.com is the current actor and https://service77.example.com was a prior actor. Such a token might come about as the result of service16 receiving a token in a call from service77 and exchanging it for a token suitable to call service26 while the authorization server notes the situation in the newly issued token.

```
{
  "aud": "https://service26.example.com",
  "iss": "https://issuer.example.com",
  "exp": 1443904100,
  "nbf": 1443904000,
  "sub": "user@example.com",
  "act": {
    {
      "sub": "https://service16.example.com",
      "act": {
        {
          "sub": "https://service77.example.com"
        }
      }
    }
  }
}
```

Figure 6: Nested Actor Claim

When included as a top-level member of an OAuth token introspection response, "act" has the same semantics and format as the claim of the same name.

4.2. "scope" (Scopes) Claim

The value of the "scope" claim is a JSON string containing a space-separated list of scopes associated with the token, in the format described in Section 3.3 of [RFC6749].

Figure 7 illustrates the "scope" claim within a JWT Claims Set.

```
{
  "aud": "https://consumer.example.com",
  "iss": "https://issuer.example.com",
  "exp": 1443904177,
  "nbf": 1443904077,
  "sub": "dgaf4mvfs75Fci_FL3heQA",
  "scope": "email profile phone address"
}
```

Figure 7: Scopes Claim

OAuth 2.0 Token Introspection [RFC7662] already defines the "scope" parameter to convey the scopes associated with the token.

4.3. "client_id" (Client Identifier) Claim

The "client_id" claim carries the client identifier of the OAuth 2.0 [RFC6749] client that requested the token.

The following example in Figure 8 illustrates the "client_id" claim within a JWT Claims Set indicating an OAuth 2.0 client with "s6BhdRkqt3" as its identifier.

```
{
  "aud": "https://consumer.example.com",
  "iss": "https://issuer.example.com",
  "exp": 1443904177,
  "sub": "user@example.com",
  "client_id": "s6BhdRkqt3"
}
```

Figure 8: Client Identifier Claim

OAuth 2.0 Token Introspection [RFC7662] already defines the "client_id" parameter as the client identifier for the OAuth 2.0 client that requested the token.

4.4. "may_act" (Authorized Actor) Claim

The "may_act" claim makes a statement that one party is authorized to become the actor and act on behalf of another party. The claim might be used, for example, when a "subject_token" is presented to the token endpoint in a token exchange request and "may_act" claim in the subject token can be used by the authorization server to determine whether the client (or party identified in the "actor_token") is authorized to engage in the requested delegation or impersonation.

The claim value is a JSON object and members in the JSON object are claims that identify the party that is asserted as being eligible to act for the party identified by the JWT containing the claim. The claims that make up the "may_act" claim identify and possibly provide additional information about the authorized actor. For example, the combination of the two claims "iss" and "sub" are sometimes necessary to uniquely identify an authorized actor, while the "email" claim might be used to provide additional useful information about that party.

However, claims within the "may_act" claim pertain only to the identity of that party and are not relevant to the validity of the containing JWT in the same manner as top-level claims. Consequently, claims such as "exp", "nbf", and "aud" are not meaningful when used within a "may_act" claim, and therefore are not used.

Figure 9 illustrates the "may_act" claim within a JWT Claims Set. The claims of the token itself are about user@example.com while the "may_act" claim indicates that admin@example.com is authorized to act on behalf of user@example.com.

```
{
  "aud": "https://consumer.example.com",
  "iss": "https://issuer.example.com",
  "exp": 1443904177,
  "nbf": 1443904077,
  "sub": "user@example.com",
  "may_act": {
    "sub": "admin@example.com"
  }
}
```

Figure 9: Authorized Actor Claim

When included as a top-level member of an OAuth token introspection response, "may_act" has the same semantics and format as the claim of the same name.

5. Security Considerations

Much of the guidance from Section 10 of [RFC6749], the Security Considerations in The OAuth 2.0 Authorization Framework, is also applicable here. Furthermore, [RFC6819] provides additional security considerations for OAuth and [I-D.ietf-oauth-security-topics] has updated security guidance based on deployment experience and new threats that have emerged since OAuth 2.0 was originally published.

All of the normal security issues that are discussed in [JWT], especially in relationship to comparing URIs and dealing with unrecognized values, also apply here.

In addition, both delegation and impersonation introduce unique security issues. Any time one principal is delegated the rights of another principal, the potential for abuse is a concern. The use of the "scope" claim (in addition to other typical constraints such as a limited token lifetime) is suggested to mitigate potential for such abuse, as it restricts the contexts in which the delegated rights can be exercised.

6. Privacy Considerations

Tokens employed in the context of the functionality described herein may contain privacy-sensitive information and, to prevent disclosure of such information to unintended parties, MUST only be transmitted over encrypted channels, such as Transport Layer Security (TLS). In cases where it is desirable to prevent disclosure of certain information to the client, the token MUST be encrypted to its intended recipient. Deployments SHOULD determine the minimally necessary amount of data and only include such information in issued tokens. In some cases, data minimization may include representing only an anonymous or pseudonymous user.

7. IANA Considerations

7.1. OAuth URI Registration

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

7.1.1. Registry Contents

- o URN: urn:ietf:params:oauth:grant-type:token-exchange
- o Common Name: Token exchange grant type for OAuth 2.0
- o Change controller: IESG
- o Specification Document: Section 2.1 of [[this specification]]

- o URN: urn:ietf:params:oauth:token-type:access_token
- o Common Name: Token type URI for an OAuth 2.0 access token
- o Change controller: IESG
- o Specification Document: Section 3 of [[this specification]]

- o URN: urn:ietf:params:oauth:token-type:refresh_token
- o Common Name: Token type URI for an OAuth 2.0 refresh token
- o Change controller: IESG
- o Specification Document: Section 3 of [[this specification]]

- o URN: urn:ietf:params:oauth:token-type:id_token
- o Common Name: Token type URI for an ID Token
- o Change controller: IESG
- o Specification Document: Section 3 of [[this specification]]

- o URN: urn:ietf:params:oauth:token-type:saml1
- o Common Name: Token type URI for a base64url-encoded SAML 1.1 assertion
- o Change Controller: IESG
- o Specification Document: Section 3 of [[this specification]]

- o URN: urn:ietf:params:oauth:token-type:saml2
- o Common Name: Token type URI for a base64url-encoded SAML 2.0 assertion
- o Change Controller: IESG
- o Specification Document: Section 3 of [[this specification]]

7.2. OAuth Parameters Registration

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

7.2.1. Registry Contents

- o Parameter name: resource
- o Parameter usage location: token request
- o Change controller: IESG
- o Specification document(s): Section 2.1 of [[this specification]]

- o Parameter name: audience
- o Parameter usage location: token request
- o Change controller: IESG
- o Specification document(s): Section 2.1 of [[this specification]]

- o Parameter name: requested_token_type
- o Parameter usage location: token request
- o Change controller: IESG
- o Specification document(s): Section 2.1 of [[this specification]]

- o Parameter name: subject_token
- o Parameter usage location: token request
- o Change controller: IESG
- o Specification document(s): Section 2.1 of [[this specification]]

- o Parameter name: subject_token_type
- o Parameter usage location: token request
- o Change controller: IESG

- o Specification document(s): Section 2.1 of [[this specification]]
- o Parameter name: actor_token
- o Parameter usage location: token request
- o Change controller: IESG
- o Specification document(s): Section 2.1 of [[this specification]]
- o Parameter name: actor_token_type
- o Parameter usage location: token request
- o Change controller: IESG
- o Specification document(s): Section 2.1 of [[this specification]]
- o Parameter name: issued_token_type
- o Parameter usage location: token response
- o Change controller: IESG
- o Specification document(s): Section 2.2.1 of [[this specification]]

7.3. OAuth Access Token Type Registration

This specification registers the following access token type in the IANA "OAuth Access Token Types" registry [IANA.OAuth.Parameters] established by [RFC6749].

7.3.1. Registry Contents

- o Type name: N_A
- o Additional Token Endpoint Response Parameters: (none)
- o HTTP Authentication Scheme(s): (none)
- o Change controller: IESG
- o Specification document(s): Section 2.2.1 of [[this specification]]

7.4. JSON Web Token Claims Registration

This specification registers the following Claims in the IANA "JSON Web Token Claims" registry [IANA.JWT.Claims] established by [JWT].

7.4.1. Registry Contents

- o Claim Name: "act"
- o Claim Description: Actor
- o Change Controller: IESG
- o Specification Document(s): Section 4.1 of [[this specification]]
- o Claim Name: "scope"
- o Claim Description: Scope Values
- o Change Controller: IESG

- o Specification Document(s): Section 4.2 of [[this specification]]
- o Claim Name: "client_id"
- o Claim Description: Client Identifier
- o Change Controller: IESG
- o Specification Document(s): Section 4.3 of [[this specification]]
- o Claim Name: "may_act"
- o Claim Description: Authorized Actor - the party that is authorized to become the actor
- o Change Controller: IESG
- o Specification Document(s): Section 4.4 of [[this specification]]

7.5. OAuth Token Introspection Response Registration

This specification registers the following values in the IANA "OAuth Token Introspection Response" registry [IANA.OAuth.Parameters] established by [RFC7662].

7.5.1. Registry Contents

- o Claim Name: "act"
- o Claim Description: Actor
- o Change Controller: IESG
- o Specification Document(s): Section 4.1 of [[this specification]]
- o Claim Name: "may_act"
- o Claim Description: Authorized Actor - the party that is authorized to become the actor
- o Change Controller: IESG
- o Specification Document(s): Section 4.4 of [[this specification]]

7.6. OAuth Extensions Error Registration

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

7.6.1. Registry Contents

- o Error Name: "invalid_target"
- o Error Usage Location: token error response
- o Related Protocol Extension: OAuth 2.0 Token Exchange
- o Change Controller: IETF
- o Specification Document(s): Section 2.2.2 of [[this specification]]

8. References

8.1. Normative References

- [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>>.
- [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>>.
- [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>>.
- [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>>.
- [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>>.
- [RFC7662] Richer, J., Ed., "OAuth 2.0 Token Introspection", RFC 7662, DOI 10.17487/RFC7662, October 2015, <<https://www.rfc-editor.org/info/rfc7662>>.
- [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>>.

8.2. Informative References

- [I-D.ietf-oauth-resource-indicators]
Campbell, B., Bradley, J., and H. Tschofenig, "Resource Indicators for OAuth 2.0", draft-ietf-oauth-resource-indicators-02 (work in progress), January 2019.
- [I-D.ietf-oauth-security-topics]
Lodderstedt, T., Bradley, J., Labunets, A., and D. Fett, "OAuth 2.0 Security Best Current Practice", draft-ietf-oauth-security-topics-13 (work in progress), July 2019.
- [OASIS.saml-core-1.1]
Maler, E., Mishra, P., and R. Philpott, "Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V1.1", OASIS Standard oasis-sstc-saml-core-1.1, September 2003.
- [OASIS.saml-core-2.0-os]
Cantor, S., Kemp, J., Philpott, R., and E. Maler, "Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V2.0", OASIS Standard saml-core-2.0-os, March 2005.
- [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>.
- [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>>.
- [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>>.
- [RFC7521] Campbell, B., Mortimore, C., Jones, M., and Y. Goland, "Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants", RFC 7521, DOI 10.17487/RFC7521, May 2015, <<https://www.rfc-editor.org/info/rfc7521>>.

[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>>.

[WS-Trust]

Nadalin, A., Goodner, M., Gudgin, M., Barbir, A., and H. Granqvist, "WS-Trust 1.4", February 2012, <<http://docs.oasis-open.org/ws-sx/ws-trust/v1.4/ws-trust.html>>.

Appendix A. Additional Token Exchange Examples

Two example token exchanges are provided in the following sections illustrating impersonation and delegation, respectively (with extra line breaks and indentation for display purposes only).

A.1. Impersonation Token Exchange Example

A.1.1. Token Exchange Request

In the following token exchange request, a client is requesting a token with impersonation semantics (with only a "subject_token" and no "actor_token", delegation is impossible). The client tells the authorization server that it needs a token for use at the target service with the logical name "urn:example:cooperation-context".

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

grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Atoken-exchange
&audience=urn%3Aexample%3Acooperation-context
&subject_token=eyJhbGciOiJIUzI1NiIsImtpZCI6IjE2In0.eyJhdWQiOiJodHRwczovL2FzLmV4YW1wbGUuY29tIiwiaXNzIjoiaHR0cHM6Ly9vcmlnaW5hbC1pc3N1ZXIuZXhhbXBsZS5uZXQlLCJleHAiOjE0NDE5MTA2MDAsIm5iZiI6MTQ0MTkwOTAwMCwic3ViIjoiYmRjQGV4YW1wbGUubmV0Iiwic2NvcGUiOiJvcmlcnMgcHJvZmlsZSBoaXN0b3J5In0.PRBg-jXn4cJujlgmYXFiGkZzRuzbXZ_sDxdE98ddW44ufsbWLKd3JJ1VZhF64pbTtfjy4VXFVBdaQpKjn5JzAw
&subject_token_type=urn%3Aietf%3Aparams%3Aoauth%3Atoken-type%3Ajwt
```

Figure 10: Token Exchange Request

A.1.2. Subject Token Claims

The "subject_token" in the prior request is a JWT and the decoded JWT Claims Set is shown here. The JWT is intended for consumption by the authorization server within a specific time window. The subject of

the JWT ("bdc@example.net") is the party on behalf of whom the new token is being requested.

```
{
  "aud": "https://as.example.com",
  "iss": "https://original-issuer.example.net",
  "exp": 1441910600,
  "nbf": 1441909000,
  "sub": "bdc@example.net",
  "scope": "orders profile history"
}
```

Figure 11: Subject Token Claims

A.1.3. Token Exchange Response

The "access_token" parameter of the token exchange response shown below contains the new token that the client requested. The other parameters of the response indicate that the token is a bearer access token that expires in an hour.

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

```
{
  "access_token": "eyJhbGciOiJIUzI1NiIsImtpZCI6IjcyIn0.eyJhdWQiOiJlcm42ZXhhbXBzTjBjb29wZXhjdGlvbiljb250ZXh0IiwiaXNzIjoiaHR0cHM6Ly9hcycy5leGFtcGxlLmNvbSIsImV4cCI6MTQ0MTkxMzYxMCwic3ViIjoieYmRjQGv4YWlwGUubmV0Iiwic2NvcGUiOiJvcmlcnMgcHJvZmlsZSBoaXN0b3J5In0.rMdWpSGNACTvnFuOL74sYZ6MVuld2Z2WkGLmQeR9ztj6w2OXraQlkJmGjyiCq24kcB7AI2VqVxl3wSWnVKh85A",
  "issued_token_type":
    "urn:ietf:params:oauth:token-type:access_token",
  "token_type": "Bearer",
  "expires_in": 3600
}
```

Figure 12: Token Exchange Response

A.1.4. Issued Token Claims

The decoded JWT Claims Set of the issued token is shown below. The new JWT is issued by the authorization server and intended for consumption by a system entity known by the logical name "urn:example:cooperation-context" any time before its expiration. The subject ("sub") of the JWT is the same as the subject the token used to make the request, which effectively enables the client to

impersonate that subject at the system entity known by the logical name of "urn:example:cooperation-context" by using the token.

```
{
  "aud": "urn:example:cooperation-context",
  "iss": "https://as.example.com",
  "exp": 1441913610,
  "sub": "bdc@example.net",
  "scope": "orders profile history"
}
```

Figure 13: Issued Token Claims

A.2. Delegation Token Exchange Example

A.2.1. Token Exchange Request

In the following token exchange request, a client is requesting a token and providing both a "subject_token" and an "actor_token". The client tells the authorization server that it needs a token for use at the target service with the logical name "urn:example:cooperation-context". Policy at the authorization server dictates that the issued token be a composite.

```
POST /as/token.oauth2 HTTP/1.1
```

Host: as.example.com

Content-Type: application/x-www-form-urlencoded

```
grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Atoken-exchange
&audience=urn%3Aexample%3Acooperation-context
&subject_token=eyJhbGciOiJIUzI1NiIsImtpZCI6IjE2In0.eyJhdWQiOiJodHRwczovL2FzLmV4YW1wbGUuY29tIiwiaXNzIjoiaHR0cHM6Ly9vcmlnaW5hbC1pc3NlZXIuZSxhhbXBsZS5uZXQlLCJleHAiOjE0NDE5MTAwNjAsInNjb3BlIjoic3Rh dHVzIGZlZWQlLCJzdWIiOiJlc2VyQG V4YW1wbGUubmV0IiwibWF5X2FjdCI6eyJzdWIiOiJhZGlpbkBLEGFtcGxlLm5ldCJ9fQ.4rPRSWihQbpMIgAmAoqaJoJAxj-p2X8_fAtAGTXrvMxU-eEZhnXqY0_AOZgLdxw5DyLzua8H_I10MCcckF-Q_g
&subject_token_type=urn%3Aietf%3Aparams%3Aoauth%3Atoken-type%3Ajwt
&actor_token=eyJhbGciOiJIUzI1NiIsImtpZCI6IjE2In0.eyJhdWQiOiJodHRwczovL2FzLmV4YW1wbGUuY29tIiwiaXNzIjoiaHR0cHM6Ly9vcmlnaW5hbC1pc3NlZXIuZSxhhbXBsZS5uZXQlLCJleHAiOjE0NDE5MTAwNjAsInNlYiI6ImFkbWluQG V4YW1wbGUubmV0In0.7YQ-3zpFfhUvzje5oqw8COcvN5uP6NsKik9CVV6cAO f4QKgM-tKfiOwcgZoUDL2tEs6tqPlcB1MjiSzEjm3yBg
&actor_token_type=urn%3Aietf%3Aparams%3Aoauth%3Atoken-type%3Ajwt
```

Figure 14: Token Exchange Request

A.2.2. Subject Token Claims

The "subject_token" in the prior request is a JWT and the decoded JWT Claims Set is shown here. The JWT is intended for consumption by the authorization server before a specific expiration time. The subject of the JWT ("user@example.net") is the party on behalf of whom the new token is being requested.

```
{
  "aud": "https://as.example.com",
  "iss": "https://original-issuer.example.net",
  "exp": 1441910060,
  "scope": "status feed",
  "sub": "user@example.net",
  "may_act": {
    "sub": "admin@example.net"
  }
}
```

Figure 15: Subject Token Claims

A.2.3. Actor Token Claims

The "actor_token" in the prior request is a JWT and the decoded JWT Claims Set is shown here. This JWT is also intended for consumption by the authorization server before a specific expiration time. The subject of the JWT ("admin@example.net") is the actor that will wield the security token being requested.

```
{
  "aud": "https://as.example.com",
  "iss": "https://original-issuer.example.net",
  "exp": 1441910060,
  "sub": "admin@example.net"
}
```

Figure 16: Actor Token Claims

A.2.4. Token Exchange Response

The "access_token" parameter of the token exchange response shown below contains the new token that the client requested. The other parameters of the response indicate that the token is a JWT that expires in an hour and that the access token type is not applicable since the issued token is not an access token.

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

{
  "access_token": "eyJhbGciOiJIUzI1NiIsImtpZCI6IjcyIn0.eyJhdWQiOiJ1cm46ZXhhdXBsZTpjb29wZXJhdGlvbiIjb250ZXh0IiwiaXNzIjoiaHR0cHM6Ly9hcy5leGFtcGxlLmNvbSIsImV4cCI6MTQ0MTkxMzYxMCwic2NvcGUOiJzdGF0dXMgZmVlZCIsInN1YiI6InVzZXJAZXhhdXBsZS5uZXQlLCJhY3QiOiOnsic3ViIjojYWWRtaW5AZXhhdXBsZS5uZXQifX0.3paKl9UySKYB5ng6_cUtQ2ql08Rc_y7Mea7IwEXTcYbNdwG9-G1EKCFe5fw3H0hwX-MSZ49Wpcb1SiAZaOQBtw",
  "issued_token_type": "urn:ietf:params:oauth:token-type:jwt",
  "token_type": "N_A",
  "expires_in": 3600
}
```

Figure 17: Token Exchange Response

A.2.5. Issued Token Claims

The decoded JWT Claims Set of the issued token is shown below. The new JWT is issued by the authorization server and intended for consumption by a system entity known by the logical name "urn:example:cooperation-context" any time before its expiration. The subject ("sub") of the JWT is the same as the subject of the "subject_token" used to make the request. The actor ("act") of the JWT is the same as the subject of the "actor_token" used to make the request. This indicates delegation and identifies "admin@example.net" as the current actor to whom authority has been delegated to act on behalf of "user@example.net".

```
{
  "aud": "urn:example:cooperation-context",
  "iss": "https://as.example.com",
  "exp": 1441913610,
  "scope": "status feed",
  "sub": "user@example.net",
  "act":
  {
    "sub": "admin@example.net"
  }
}
```

Figure 18: Issued Token Claims

Appendix B. Acknowledgements

This specification was developed within the OAuth Working Group, which includes dozens of active and dedicated participants. It was produced under the chairmanship of Hannes Tschofenig, Derek Atkins, and Rifaat Shekh-Yusef with Kathleen Moriarty, Stephen Farrell, Eric Rescorla, Roman Danyliw, and Benjamin Kaduk serving as Security Area Directors. The following individuals contributed ideas, feedback, and wording to this specification:

Caleb Baker, Vittorio Bertocci, Mike Brown, Thomas Broyer, Roman Danyliw, William Denniss, Vladimir Dzhuvinov, Eric Fazendin, Phil Hunt, Benjamin Kaduk, Jason Keglovitz, Torsten Lodderstedt, Barry Leiba, Adam Lewis, James Manger, Nov Mataka, Matt Miller, Hilarie Orman, Matthew Perry, Eric Rescorla, Justin Richer, Adam Roach, Rifaat Shekh-Yusef, Scott Tomilson, and Hannes Tschofenig.

Appendix C. Document History

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

-19

- o Fix-up changes introduced in -18.
- o Fix invalid JSON in the Nested Actor Claim example.
- o Reference figure numbers in text when introducing the examples in Section 2 and 4.
- o Editorial updates from additional IESG evaluation comments.
- o Add an informational reference to ietf-oauth-resource-indicators
- o Update ietf-oauth-security-topics ref to 13

-18

- o Editorial updates based on a few more IESG evaluation comments.

-17

- o Editorial improvements and example fixes resulting from IESG evaluation comments.
- o Added a pointer to RFC6749's Appendix B. on the "Use of application/x-www-form-urlencoded Media Type" as a way of providing a normative citation (by reference) for the media type.
- o Strengthened some of the wording in the privacy considerations to bring it inline with RFC 7519 Sec. 12 and RFC 6749 Sec. 10.8.

-16

- o Fixed typo and added an AD to Acknowledgements.

-15

- o Updated the nested actor claim example to (hopefully) be more straightforward.
- o Reworked Privacy Considerations to say to use TLS in transit, minimize the amount of information in the token, and encrypt the token if disclosure of its information to the client is a concern per https://mailarchive.ietf.org/arch/msg/secdir/KJhx4aq_U5uk3k6zpYP-CEHbpVM
- o Moved the Security and Privacy Considerations sections to before the IANA Considerations.

-14

- o Added text in Section 4.1 about the "act" claim stating that only the top-level claims and the current actor are to be considered in applying access control decisions.

-13

- o Updated the claim name and value syntax for scope to be consistent with the treatment of scope in RFC 7662 OAuth 2.0 Token Introspection.
- o Updated the client identifier claim name to be consistent with the treatment of client id in RFC 7662 OAuth 2.0 Token Introspection.

-12

- o Updated to use the boilerplate from RFC 8174.

-11

- o Added new WG chair and AD to the Acknowledgements.
- o Applied clarifications suggested during AD review by EKR.

-10

- o Defined token type URIs for base64url-encoded SAML 1.1 and SAML 2.0 assertions.
- o Applied editorial fixes.

-09

- o Changed "security tokens obtained could be used in a number of contexts" to "security tokens obtained may be used in a number of contexts" per a WGLC suggestion.

- o Clarified that the validity of the subject or actor token have no impact on the validity of the issued token after the exchange has occurred per a WGLC comment.
- o Changed use of `invalid_target` error code to a SHOULD per a WGLC comment.
- o Clarified text about non-identity claims within the "act" claim being meaningless per a WGLC comment.
- o Added brief Privacy Considerations section per WGLC comments.

-08

- o Use the bibxml reference for OpenID.Core rather than defining it inline.
- o Added editor role for Campbell.
- o Minor clarification of the text for `actor_token`.

-07

- o Fixed typo (desecration -> discretion).
- o Added an explanation of the relationship between scope, audience and resource in the request and added an "invalid_target" error code enabling the AS to tell the client that the requested audiences/resources were too broad.

-06

- o Drop "An STS for the REST of Us" from the title.
- o Drop "heavyweight" and "lightweight" from the abstract and introduction.
- o Clarifications on the language around `xxxxxx_token_type`.
- o Remove the `want_composite` parameter.
- o Add a short mention of proof-of-possession style tokens to the introduction and remove the respective open issue.

-05

- o Defined the JWT claim "cid" to express the OAuth 2.0 client identifier of the client that requested the token.
- o Defined and requested registration for "act" and "may_act" as Token introspection response parameters (in addition to being JWT claims).
- o Loosen up the language about `refresh_token` in the response to OPTIONAL from NOT RECOMMENDED based on feedback from real world deployment experience.
- o Add clarifying text about the distinction between JWT and access token URIs.
- o Close out (remove) some of the Open Issues bullets that have been resolved.

-04

- o Clarified that the "resource" and "audience" request parameters can be used at the same time (via <http://www.ietf.org/mail-archive/web/oauth/current/msg15335.html>).
- o Clarified subject/actor token validity after token exchange and explained a bit more about the recommendation to not issue refresh tokens (via <http://www.ietf.org/mail-archive/web/oauth/current/msg15318.html>).
- o Updated the examples appendix to use an issuer value that doesn't imply that the client issued and signed the tokens and used "Bearer" and "urn:ietf:params:oauth:token-type:access_token" in one of the responses (via <http://www.ietf.org/mail-archive/web/oauth/current/msg15335.html>).
- o Defined and registered urn:ietf:params:oauth:token-type:id_token, since some use cases perform token exchanges for ID Tokens and no URI to indicate that a token is an ID Token had previously been defined.

-03

- o Updated the document editors (adding Campbell, Bradley, and Mortimore).
- o Added to the title.
- o Added to the abstract and introduction.
- o Updated the format of the request to use application/x-www-form-urlencoded request parameters and the response to use the existing token endpoint JSON parameters defined in OAuth 2.0.
- o Changed the grant type identifier to urn:ietf:params:oauth:grant-type:token-exchange.
- o Added RFC 6755 registration requests for urn:ietf:params:oauth:token-type:refresh_token, urn:ietf:params:oauth:token-type:access_token, and urn:ietf:params:oauth:grant-type:token-exchange.
- o Added RFC 6749 registration requests for request/response parameters.
- o Removed the Implementation Considerations and the requirement to support JWTs.
- o Clarified many aspects of the text.
- o Changed "on_behalf_of" to "subject_token", "on_behalf_of_token_type" to "subject_token_type", "act_as" to "actor_token", and "act_as_token_type" to "actor_token_type".
- o Added an "audience" request parameter used to indicate the logical names of the target services at which the client intends to use the requested security token.
- o Added a "want_composite" request parameter used to indicate the desire for a composite token rather than trying to infer it from the presence/absence of token(s) in the request.

- o Added a "resource" request parameter used to indicate the URLs of resources at which the client intends to use the requested security token.
- o Specified that multiple "audience" and "resource" request parameter values may be used.
- o Defined the JWT claim "act" (actor) to express the current actor or delegation principal.
- o Defined the JWT claim "may_act" to express that one party is authorized to act on behalf of another party.
- o Defined the JWT claim "scp" (scopes) to express OAuth 2.0 scope-token values.
- o Added the "N_A" (not applicable) OAuth Access Token Type definition for use in contexts in which the token exchange syntax requires a "token_type" value, but in which the token being issued is not an access token.
- o Added examples.

-02

- o Enabled use of Security Token types other than JWTs for "act_as" and "on_behalf_of" request values.
- o Referenced the JWT and OAuth Assertions RFCs.

-01

- o Updated references.

-00

- o Created initial working group draft from draft-jones-oauth-token-exchange-01.

Authors' Addresses

Michael B. Jones
Microsoft

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

Anthony Nadalin
Microsoft

Email: tonynad@microsoft.com

Brian Campbell (editor)
Ping Identity

Email: brian.d.campbell@gmail.com

John Bradley
Yubico

Email: ve7jtb@ve7jtb.com

Chuck Mortimore
Salesforce

Email: cmortimore@salesforce.com

Open Authentication Protocol
Internet-Draft
Intended status: Standards Track
Expires: November 29, 2018

T. Lodderstedt, Ed.
YES.com AG
V. Dzhuvinov
Connect2id Ltd.
May 28, 2018

JWT Response for OAuth Token Introspection
draft-lodderstedt-oauth-jwt-introspection-response-01

Abstract

This draft proposes an additional JSON Web Token (JWT) based response for OAuth 2.0 Token Introspection.

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 November 29, 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. Requesting a JWT Response	2
3. JWT Response	3
4. Client Metadata	4
5. Authorization Server Metadata	5
6. Acknowledgements	5
7. IANA Considerations	5
7.1. OAuth Dynamic Client Registration Metadata Registration	5
7.1.1. Registry Contents	5
7.2. OAuth Authorization Server Metadata Registration	6
7.2.1. Registry Contents	6
7.3. OAuth Token Introspection Response	7
8. Security Considerations	7
8.1. Cross-JWT Confusion	7
9. References	7
9.1. Normative References	8
9.2. Informative References	9
Appendix A. Document History	9
Authors' Addresses	10

1. Introduction

OAuth 2.0 Token Introspection [RFC7662] specifies a method for a protected resource to query an OAuth 2.0 authorization server to determine the state of an access token and obtain data associated with the access token. This allows deployments to implement identifier-based access tokens in an interoperable way.

The introspection response as specified in OAuth 2.0 Token Introspection [RFC7662] is a plain JSON object. However, there are use cases where the resource server requires stronger assurance that the authorisation server issued the access token, including cases where the authorisation server assumes liability for the token's content. An example is a resource server using verified person data to create qualified electronic signatures.

In such use cases, it would be useful to return a signed JWT as the introspection response. This specification extends the Token Introspection endpoint with the capability to return responses as JWTs.

2. Requesting a JWT Response

A resource server requests to receive a JWT introspection response by including an Accept header with content type "application/jwt" in the introspection request.

The following is a non-normative example request:

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

token=2YotnFZFEjrlzCsicMWpAA
```

3. JWT Response

The introspection endpoint responds with a JWT, setting the Content-Type header to "application/jwt".

This JWT MUST contain the claims "iss" and "aud" in order to prevent misuse of the JWT as ID or access token (see Section 8.1).

This JWT may furthermore contain all other claims described in Section 2.2. of [RFC7662].

The following is a non-normative example response (with line breaks for display purposes only):

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

```
eyJraWQiOiIxIiwiaWxzIjoiUlMyNTYifQ.eyJzdWIiOiJaNU8zdXBQZG4UXJBa
ngwMGRpcyIsImF1ZCI6Imh0dHBzOlwvXC9wcm90ZWN0ZWQuZXhhbXBsZS5uZXRcL
3Jlc291cmNlIiwiaXN0ZW5zaW9uX2ZpZWxkIjoidHdlbnR5LXNldmVuIiwic2Nvc
GUiOiJyZWZkIHdyZXRLIGRvbHB0aW4iLCJpc3MiOiJodHRwczpcL1wvc2VydmVyL
mV4YW1wbGUuY29tXC8iLCJhY3RpdmUiOnRydWUsImV4cCI6MTQxOTM1NjIzOCwia
WF0IjojoxNDE5MzUwMjM4LCJjbGllbnRfaWQiOiJjMyM4ajMyM2RzLTlzaWo0Iiwid
XNlcm5hbWUiOiJqZG91In0.HEQHf05vqVvWVnWuEjbzUnPz6JDQVR69QkxgzBNq5
kk-sK54ieglSTazXGsdFAT8nUhiiv1f_Z4HOKNnBs8TLKaFXokhA0MqNBOYI--2u
nVHDqI_RPmC3p0NmP02Xmv4hzzFmTmPgjSy3vpKQDihOjhwNBh7G81JNaJqjJQTR
v_ldHUPJotQjMK3k8_5FyiO2p64Y2VyxyQn1VWVlgOHlJwhj6BaGHk4Qf5F8DHQZ
1WCPg2p_-hwfINfXh1_buSjxyDRF4oe9pKy6ZB3ejh9qIMm-WrwltuUluWMXxN6e
S6tUtpKo8UCHBwLWCHmJN7KU6ZojmaISspds231ELAlw
```

The example response contains the following JSON document:

```
{
  "sub": "Z503upPC88QrAjx00dis",
  "aud": "https://protected.example.net/resource",
  "scope": "read write dolphin",
  "iss": "https://server.example.com/",
  "active": true,
  "exp": 1419356238,
  "iat": 1419350238,
  "client_id": "1238j323ds-23ij4",
  "given_name": "John",
  "family_name": "Doe",
  "birthdate": "1982-02-01"
}
```

4. Client Metadata

The authorization server determines what algorithm to employ to secure the JWT for a particular introspection response. This decision can be based on registered metadata parameters for the resource server, supplied via dynamic client registration with the resource server posing as the client.

The parameter names follow the pattern established by OpenID Connect Dynamic Client Registration [OpenID.Registration] for configuring signing and encryption algorithms for JWT responses at the UserInfo endpoint.

The following client metadata parameters are introduced by this specification:

`introspection_signed_response_alg` JWS [RFC7515] "alg" algorithm JWA [RFC7518] REQUIRED for signing introspection responses. If this is specified, the response will be JWT [RFC7519] serialized, and signed using JWS. The default, if omitted, is for the introspection response to return the Claims as a UTF-8 encoded JSON object using the "application/json" content type, as defined in [RFC7662].

`introspection_encrypted_response_alg` JWE [RFC7516] "alg" algorithm JWA [RFC7518] REQUIRED for encrypting introspection responses. If both signing and encryption are requested, the response will be signed then encrypted, with the result being a Nested JWT, as defined in JWT [RFC7519]. The default, if omitted, is that no encryption is performed.

`introspection_encrypted_response_enc` JWE [RFC7516] "enc" algorithm JWA [RFC7518] REQUIRED for encrypting introspection responses. If "introspection_encrypted_response_alg" is

specified, the default for this value is A128CBC-HS256. When "introspection_encrypted_response_enc" is included, "introspection_encrypted_response_alg" MUST also be provided.

Resource servers may register their public encryption keys using the "jwks_uri" or "jwks" metadata parameters.

5. Authorization Server Metadata

Authorization servers SHOULD publish the supported algorithms for signing and encrypting the JWT of an introspection response by utilizing OAuth Authorization Server Metadata parameters.

The following parameters are introduced by this specification:

introspection_signing_alg_values_supported OPTIONAL. JSON array containing a list of the JWS [RFC7515] signing algorithms ("alg" values) JWA [RFC7518] supported by the Introspection Endpoint to sign the response.

introspection_encryption_alg_values_supported OPTIONAL. JSON array containing a list of the JWE [RFC7516] encryption algorithms ("alg" values) JWA [RFC7518] supported by the Introspection Endpoint to encrypt the response.

introspection_encryption_enc_values_supported OPTIONAL. JSON array containing a list of the JWE [RFC7516] encryption algorithms ("enc" values) JWA [RFC7518] supported by the Introspection Endpoint to encrypt the response.

6. Acknowledgements

We would like to thank Petteri Stenius and Neil Madden for their valuable feedback.

7. IANA Considerations

7.1. OAuth 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]:

7.1.1. Registry Contents

o Client Metadata Name: "introspection_signed_response_alg"

- o Client Metadata Description: String value indicating the client's desired introspection response signing algorithm.
- o Change Controller: IESG
- o Specification Document(s): Section 4 of [[this specification]]
- o Client Metadata Name: "introspection_encrypted_response_alg"
- o Client Metadata Description: String value specifying the desired introspection response encryption algorithm (alg value).
- o Change Controller: IESG
- o Specification Document(s): Section 4 of [[this specification]]
- o Client Metadata Name: "introspection_encrypted_response_enc"
- o Client Metadata Description: String value specifying the desired introspection response encryption algorithm (enc value).
- o Change Controller: IESG
- o Specification Document(s): Section 4 of [[this specification]]

7.2. OAuth Authorization Server Metadata Registration

This specification requests registration of the following value in the IANA "OAuth Authorization Server Metadata" registry [IANA.OAuth.Parameters] established by [I-D.ietf-oauth-discovery].

7.2.1. Registry Contents

- o Metadata Name: "introspection_signing_alg_values_supported"
- o Metadata Description: JSON array containing a list of algorithms supported by the authorization server for introspection response signing.
- o Change Controller: IESG
- o Specification Document(s): Section 5 of [[this specification]]
- o Metadata Name: "introspection_encryption_alg_values_supported"
- o Metadata Description: JSON array containing a list of algorithms supported by the authorization server for introspection response encryption (alg value).

- o Change Controller: IESG
- o Specification Document(s): Section 5 of [[this specification]]
- o Metadata Name: "introspection_encryption_enc_values_supported"
- o Metadata Description: JSON array containing a list of algorithms supported by the authorization server for introspection response encryption (enc value).
- o Change Controller: IESG
- o Specification Document(s): Section 5 of [[this specification]]

7.3. OAuth Token Introspection Response

TBD: add all OpenID Connect standard claims.

8. Security Considerations

8.1. Cross-JWT Confusion

JWT introspection responses and OpenID Connect ID Tokens are syntactically more or less equivalent. An attacker could therefore try to misuse an JWT obtained from an introspection response to impersonate the user whose claims are included in this JWT at a OpenID Connect RP. Such an attack is treated and prevented like any other token substitution attack. The AS MUST include the claims "iss" and "aud" into every JWT introspection response. This allows every well behaving OpenID Connect RP to detect substitution by checking the "iss" and "aud" claims as described in Section 3.1.3.7. of [OpenID.Core]. RPs should also use and check the "nonce" parameter and claim to prevent token and code replay.

Resource servers utilizing JWTs to represent structured access tokens could be susceptible to replay attacks as well. Resource servers should therefore apply proper counter measures against replay as described in [I-D.ietf-oauth-security-topics], section 2.2.

JWT Confusion and other attacks on JWTs are discussed in detail in [I-D.ietf-oauth-jwt-bcp].

9. References

9.1. Normative References

- [I-D.ietf-oauth-discovery]
Jones, M., Sakimura, N., and J. Bradley, "OAuth 2.0 Authorization Server Metadata", draft-ietf-oauth-discovery-10 (work in progress), March 2018.
- [I-D.ietf-oauth-jwt-bcp]
Sheffer, Y., Hardt, D., and M. Jones, "JSON Web Token Best Current Practices", draft-ietf-oauth-jwt-bcp-03 (work in progress), May 2018.
- [I-D.ietf-oauth-security-topics]
Lodderstedt, T., Bradley, J., Labunets, A., and D. Fett, "OAuth 2.0 Security Best Current Practice", draft-ietf-oauth-security-topics-06 (work in progress), May 2018.
- [OpenID.Core]
NRI, Ping Identity, Microsoft, Google, and Salesforce, "OpenID Connect Core 1.0 incorporating errata set 1", Nov 2014, <http://openid.net/specs/openid-connect-core-1_0.html>.
- [OpenID.Registration]
NRI, Ping Identity, and Microsoft, "OpenID Connect Dynamic Client Registration 1.0 incorporating errata set 1", Nov 2014, <https://openid.net/specs/openid-connect-registration-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>>.
- [RFC2246] Dierks, T. and C. Allen, "The TLS Protocol Version 1.0", RFC 2246, DOI 10.17487/RFC2246, January 1999, <<https://www.rfc-editor.org/info/rfc2246>>.
- [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>>.
- [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>>.

9.2. Informative References

- [IANA.OAuth.Parameters]
IANA, "OAuth Parameters",
<<http://www.iana.org/assignments/oauth-parameters>>.

Appendix A. Document History

[[To be removed from the final specification]]

-01

- o fixed typos in client meta data field names
- o added OAuth Server Metadata parameters to publish algorithms supported for signing and encrypting the introspection response
- o added registration of new parameters for OAuth Server Metadata and Client Registration
- o added explicit request for JWT introspection response
- o made iss and aud claims mandatory in introspection response
- o Stylistic and clarifying edits, updates references

-00

- o initial version

Authors' Addresses

Torsten Lodderstedt (editor)
YES.com AG

Email: torsten@lodderstedt.net

Vladimir Dzhuvinov
Connect2id Ltd.

Email: vladimir@connect2id.com