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
Internet-Draft              Distributed OAuth                  June 2018

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.
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 an 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.

- **Relation Name**: oauth_server_metadata_uri
  - **Description**: An OAuth 2.0 Server Metadata URI.
  - **Reference**: This specification

- **Relation Name**: resource_uri
  - **Description**: An OAuth 2.0 Resource Endpoint specified in [RFC6750] section 3.2.
  - **Reference**: This specification

8. Acknowledgements

TBD.

9. Normative References


Appendix A.  Document History

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

- Initial version.

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

- resource identity expanded from just a hostname "host", to a URI that contains the hostname "resource URI"
- use oauth discovery document to obtain token endpoint rather than explicitly returning token endpoint
- use [RFC8288] to provide resource and discovery URIs
- allow any authorization grant type be used to obtain an authorization grant
- change attribute "host" to "resource"
- 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
Seamless OAuth 2.0 Client Assertion Grant
draft-hevroni-oauth-seamless-flow-01

Abstract

This specification defines the use of a One Time Password, encoded as JSON Web Token (JWS) Bearer Token, as a means for requesting an OAuth 2.0 access token as well as for client authentication.

Status of This Memo

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

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at 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 3, 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.
1. Introduction

1.1. Motivation

Authentication is a crucial part of modern application. There are various authentication methods for client side applications, and all those methods requires user interaction (e.g. login). This is due to the fact that there is no secure way to embed credentials in the application code.

While asking the user to login in order to authenticate the app is a strong authentication solution, it has impact on the application behavior. A login is just another step the user has to complete in order to use the apps, which users don’t always like to fulfill.
Also, there are cases for applications without any UI, for example - Internet of Things applications. For those applications, adding a login steps could be a challenge.

In this document, we propose an extension to OAuth 2.0 protocol that provides a new authentication grant dedicated for those cases. This grant will allow an application to use strong authentication solution without user interaction.

This document defines how a One Time Password, encoded in a JWS, can be used to authenticate the client. In order for the client to perform an authentication request, an initial registration step is required. This registration step is not part of this protocol, and should be defined by the authorization server.

1.2. Target Audience

The protocol requires the app to be able to persist state in a secure, sand-boxed, persisted storage. It is possible to use this protocol for web application, although it is not recommended. This protocol is targeted for mobile or IoT devices where it is possible (although not always simple) to achieve such storage. See Security Consideration section for more details.

1.3. Existing Solutions

There are alternatives to this protocol, this section will discuss them. Interactive grants (authorization code, resource owner etc) will not be discussed.

1.3.1. Client Credentials grant

This grant (as defined in [RFC6749]) allows applications to authenticate without user interaction. It is intend to be used by applications running on trusted environment. Mobile applications are not running on trusted environment, and therefore should not use this grant. See the Security section for discussion on the various threat and how this protocol mitigate them. Also refer to section 10.1 in [RFC6749], which strongly advise against using this grant on native applications.

1.3.2. Device grant

This grant is for Browserless and Input Constrained Devices. In this grant the login is performed on a different device, which could handle interactive login. Therefore, it still requires user interaction, which this protocol aims to avoid.
1.3.3. JWT Client Assertion

This grant (as defined in [RFC7523]) could be used by mobile application for seamless authentication. The grant used signed JWT (see [RFC7519]) to authenticate the client. It has two disadvantages when compared with this grant:

- Significant part of the security of the protocol is the expiration date of the JWT. In case a hacker was able to obtain a JWT, she will be able to perform authentication request until the JWT expires. Therefore, it is advised to use as shorter expiration time as possible. Time can be a challenge on mobile devices, which are not always synchronized with the global time. Usage of JWT would require the authorization server to allow very long JWT expiration time.

- Detecting Compromised Signing Key. As discussed on the security section, this protocol allows the authorization server to detect compromised signing key. See the discussion there for reference. This mitigation does not exist in JWT client assertion grant.

1.4. 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 [RFC2119].

The term "device" used in this document refer to the physical appliance used by the user, which the application code is running on.

2. Note to Readers

*Note to the RFC Editor:* Please remove this section prior to publication.

Development of this draft takes place on Github at: https://github.com/Soluto/oauth-seamless-flow [1].

3. HTTP Parameter Bindings for Transporting Assertions

The OAuth Assertion Framework [RFC7521] defines generic HTTP parameters for transporting assertions (a.k.a. security tokens) during interactions with a token endpoint. This section defines specific parameters and treatments of those parameters for use with JWS (as defined in [RFC7515]) Bearer Tokens.
3.1. Using OTP JWS for client authentication

To use a OTP JWS, the client first need to generate the OTP as defined in section "JWS format and request processing". Than, the client need to use the following parameter values and encodings.

The value of the "client_assertion_type" is "urn:ietf:params:oauth:client-assertion-type:JWS-otp".

The value of the "client_assertion" parameter contains a single JWS, as defined in [RFC7515]. It MUST NOT contain more than one JWS.

The following example demonstrates client authentication using a JWS during the presentation of an authorization code grant in an access token request (with extra line breaks for display purposes only):

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

grant_type=token id_token&&
client_assertion_type=urn%3Aietf%3Aparams%3Aoauth%3A
client-assertion-type%3AJWS-otp&
client_assertion=eyJhbGciOiJSUzI1NiIsImtpZCI6IjIyIn0.
eyJpc3Mi[...omitted for brevity...].
cC4hiUPo[...omitted for brevity...]
```

4. JWS format and request processing

4.1. One Time Password generation

To generate one time password (OTP) as defined in [RFC2289], the client use its state, created during the registration request, which is not covered in this document. The state consist from 2 numbers: "previous" and "next". Each of those numbers can hold signed int, up to 64 bytes length. In order to generate a new JWS, the client has to roll this payload. The rolling is done by setting the value of "previous" to the value of "current", and setting new crypto random, as defined in [RFC4086], value to "next". For example, assuming this is the current state of the app:

```plaintext
previous: 1
next: 2
```

After rolling, this will be the payload:

```plaintext
previous: 2
next: 5
```
4.2. Creating the JWS

After rolling the payload, the client can create the JWS. This is the format of the JWS payload:

```
{
  previous: 2
  next: 5
  client-id: 89
}
```

Where "client-id" is the id used when this client first registered. All the fields are required. Any other fields besides those will be ignored. To sign the JWS, the client use its own key, which was generated during the registration of this client.

4.3. Request processing

In order to issue an access token response as described in OAuth 2.0 [RFC6749], the authorization server MUST validate the JWS according to the criteria below. Application of additional restrictions and policy are at the discretion of the authorization server. After decoding the JWS and extracting the "client-id", the server will fetch:

- The key correspond to this client, received on the registration request
- The current state of this client, from the last successful request, or from the registration

The server verifies that the JWS is valid, by using the client’s key. If the signature is valid, the server can validate the payload:

- If the client’s "previous" is equals to the server "new", the request is valid. The server will issue a token, as specified in OAuth 2.0 [RFC6749]
- If the client "previous" equals to the server "previous", and the client "next" equals to the server "next", the server construct an error response as defined in OAuth 2.0 [RFC6749]
- Any other case will be treated by the server as an indication of a malicious attack, and should be reported accordingly. The server construct an error response as defined in OAuth 2.0 [RFC6749]
5. Security Considerations

This protocol was designed for mobile application. The following sections will discuss threats which are relevant for mobile applications and are mitigated by this protocol.

5.1. Replay Attacks

Due to the usage of OTP, a replay attack is not feasible. If an attacker will try to replay authentication request, an error response will return. Also, because of how the OTP is generated, guessing it is almost impossible (see the OTP Generation section). Refer to the Request processing section for more details.

5.2. Compromised Signing key

As the application is running on a mobile device, an attacker can gain physical access to the device. In such a scenario, the attacker will be able to compromise it and retrieve the state and the signing key. This will allow the attacker to impersonate the device and request an access token. The attacker will be able to authenticate as until the first time the device will try to authenticate. When the device will try to authenticate, the request will fail. It will fail because the state on the authorization server will match the attacker’s state, not the one on the device.

The device authentication request will revoke the client (see Request processing section). This will cause both the device and the attacker to not be able to perform authentication request. In such cases, an alternative flow is required in order to allow the device to authenticate. Such a flow is not part of this standard.

In order for this mitigation to be effective, the device must to perform an authentication request on a regular basis. The period between authentication requests should be 24 hours or less, depend on the client.

5.3. Man in the Middle

Performing Man in the Middle (MitM) attack on mobile application is relatively simple. It is highly recommended to use TLS [RFC5246] for all authentication requests. It is also recommended to implement Certificate Pinning for all the requests. For more details, please refer to this guide [2] by OWASP.
5.4. Reverse Engineering

The mobile application code is publicly available, which make reverse engineering a simple task. This attack is irrelevant to this protocol. No sensitive data should be embedded in the application code. All that is required for the authentication request should be generated on the device.

5.5. OTP Generation

The security of the OTP is as strong as the randomness used to generate it. Only strong, secure random implementation (as described in [RFC4086]) should be used. Usage of weak random protocol will allow the attacker to guess the numbers generated by the client, and by that generates the OTP herself. The state ("next" and "new") is not considered a secret. Compromise of state only, without the signing key, will not allows the attacker to perform authentication request. It is still advised to store them securely, and follow the operating system recommendation (iOS [3], Android [4]).

5.6. Signing Key Consideration

5.6.1. Generation and Storage

A fundamental part of the security of the protocol is the key used to sign the JWS. The key should be generated and stored in a secure way, and if possible to use the tools provided by the OS. On iOS, use Keychain [5] to generate and store the key. On Android, the best option is the Keystore [6], but due to implementation limitations (see this post [7] for example), it is advised to use OpenSSL.

5.6.2. Algorithm

Asymmetric encryption and signing algorithms are preferred over symmetric ones. The main advantages of such protocol is that the private key never leaves the device. Even if an attacker was able to capture the public key (either in transit or by compromising the authorization server), she will not be able to use it to perform authentication request. For any algorithm that is chosen, a strong key should be generated. In case of RSA, 2048 bytes is the minimum key size.

6. IANA Considerations

TODO IANA
7. References

7.1. Normative References


7.2. URIs


Author’s Address

Omer Levi Hevroni
Soluto by Asurion

Email: omerlh@gmail.com
JWT Response for OAuth Token Introspection
draft-ietf-oauth-jwt-introspection-response-03

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 22, 2019.

Copyright Notice

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

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.
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 authorization server issued the access token, including cases where the authorization server assumes liability for the token’s content. An example is a resource server using verified person data to create certificates, which in turn are used to create qualified electronic signatures.

In such use cases it may be useful or even required 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=2YotnFZFEjr1zCsicMWpAA

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 6.1).

This JWT MAY furthermore contain all other claims described in Section 2.2. of [RFC7662] and beyond (e.g. as defined in [OpenID.Core]).

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

HTTP/1.1 200 OK
Content-Type: application/jwt
eyJraWQiOiIxIiwiYWxnIjoiUlMyNTYifQ.eyJzdWIiOiJaNU8zdXBQQzg4UXJBa
ngwMGRpcysImFiI2CI6Imh0dHBzOlwvXC9wcmc93ZWN0ZWQuZXhhbXBsZS5uZXRL
3Jic9Zlc9zcmNlIiwiZHd2XzU5ZzaW9uX2ZpZWFkIjoidmlldlZnLmFsbwUwIiwi
GU0iOiJyZWFkIiwiX1RincZaYXRlIjoiMjIiLCJpc3MiOiJodHRwczpcIiwiZXRcL
mV4YW1wbGU0Iy9xtXCI6aHRtbC9QcnBmb3JtZC90YWlsLmNvbSIsImF1ZCI6
NjE5MzUxMDA0MDExMGQ1LWY3Zy00Y2I0LTc1NjctZTFkMzQ5MGMtNjA0OS0xMjFl
OGI1NjY4OWQ0ZTliZjBlNzI3ZmY1NzhmNTk1NjFiNzMyM2MxZmQ0OTIyZjI3MzI1
ZDc2NzY2MmU2ZTY1ODg0ZjE4ODUifQ.eyJhY3RpdmUiOnRydWUsImV4cCI6MTQxOTM1
NjIzOCwiaWF0IjoxNzA2NzIyMjU0LCJpZCI6IjI3NDE1NDQxNjY5MDIzLTA1MjUt
NGU5OS00ODA1LjMyNzIzNGU5MjAwMyIiwic3RvZHNVc1wiXCI6IjIiLCJpc3Mi
OiJodHRwczpcIiwiZXRcLmV4YW1wbGU0Iy9xtXCI6aHRtbC9QcnBmb3JtZC90YWls
LmNvbSIsImF1ZCI6NjE5MzUxMDA0MDExMGQ1LWY3Zy00Y2I0LTc1NjctZTFkMzQ5
MGMtNjA0OS0xMjFlOGI1NjY4OWQ0ZTliZjBlNzI3ZmY1NzhmNTk1NjFiNzMyM2Mx
ZmQ0OTIyZjI3MzI1ZDc2NzY2MmU2ZTY1ODg0ZjE4ODUifQ.eyJhY3RpdmUiOnRydWUs
ImV4cCI6MTQxOTM1NjIzOCwiaWF0IjoxNzA2NzIyMjU0LCJpZCI6IjI3NDE1NDQx
NjY5MDIzLTA1MjUtNGU5OS00ODA1LjMyNzIzNGU5MjAwMyIiwicGx5IjoxLCJpZCI6
IjI3NDE1NDQxNjY5MDIzLTA1MjUtNGU5OS00ODA1LjMyNzIzNGU5MjAwMyIiwidXNl
cm5hbWUxIjoiMTg5ODE2NjU1MTc3NGEwOGQ2ZjVlNzQ2MDU3OTUwMzU2N2ZkMTk0
ZDc2NzY2MmU2ZTY1ODg0ZjE4ODUifQ.eyJhY3RpdmUiOnRydWUsImV4cCI6MTQxOTM1
NjIzOCwiaWF0IjoxNzA2NzIyMjU0LCJpZCI6IjI3NDE1NDQxNjY5MDIzLTA1MjUt
NGU5OS00ODA1LjMyNzIzNGU5MjAwMyIiwic3RvZHNVc1wiXCI6IjIiLCJpc3Mi
OiJodHRwczpcIiwiZXRcLmV4YW1wbGU0Iy9xtXCI6aHRtbC9QcnBmb3JtZC90YWls
LmNvbSIsImF1ZCI6NjE5MzUxMDA0MDExMGQ1LWY3Zy00Y2I0LTc1NjctZTFkMzQ5
MGMtNjA0OS0xMjFlOGI1NjY4OWQ0ZTliZjBlNzI3ZmY1NzhmNTk1NjFiNzMyM2Mx
ZmQ0OTIyZjI3MzI1ZDc2NzY2MmU2ZTY1ODg0ZjE4ODUifQ.eyJhY3RpdmUiOnRydWUs
ImV4cCI6MTQxOTM1NjIzOCwiaWF0IjoxNzA2NzIyMjU0LCJpZCI6IjI3NDE1NDQx
NjY5MDIzLTA1MjUtNGU5OS00ODA1LjMyNzIzNGU5MjAwMyIiwic3RvZHNVc1wiXCI6Ij

The example response contains the following JSON document:


```
{
    "sub": "Z5O3upPC88QrAjx00dis",
    "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"
}
```

Depending on the specific resource server policy the JWT is either signed, or signed and encrypted. If the JWT is signed and encrypted it MUST be a Nested JWT, as defined in JWT [RFC7519].

Note: If the resource server policy requires a signed and encrypted response and the authorization server receives an unauthenticated request containing an Accept header with content type other than "application/jwt", it MUST refuse to serve the request and return an HTTP status code 400. This is done to prevent downgrading attacks to obtain token data intended for release to legitimate recipients only (see Section 6.2).

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, as defined by this draft.

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 signed using JWS and the configured algorithm. The default, if omitted, is "RS256".
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 2.0 Authorization Server Metadata [RFC8414]
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
descriptor 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
descriptor 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
descriptor to encrypt the response.

6. Security Considerations

6.1. Cross-JWT Confusion

JWT introspection responses and OpenID Connect ID Tokens are
syntactically similar. An attacker could therefore attempt to
Impersonate an end-user at an OpenID Connect relying party by passing the JWT as an ID token.

Such an attack can be prevented like any other token substitution attack. The authorization server MUST include the claims "iss" and "aud" in each JWT introspection response, with the "iss" value set to the authorization server’s issuer URL and the "aud" value set to the resource server’s identifier. This allows a correctly implemented OpenID Connect relying party to detect substitution by checking the "iss" and "aud" claims as described in Section 3.1.3.7. of [OpenID.Core]. Relying parties 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. 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 involving JWTs are discussed in [I-D.ietf-oauth-jwt-bcp].

6.2. Token Data Leakage

If the authorization server supports unauthenticated requests an attacker could potentially retrieve token data which must be kept confidential. This attack can be prevented by either authenticating any request to the token introspection endpoint or by setting up the respective recipient for encrypted responses.

In the latter case, confidentiality is ensured by the fact that only the legitimate recipient is able to decrypt the response. An attacker could try to circumvent this measure by requesting a plain JSON response, using an Accept header with the content type set to, for example, "application/json" instead of "application/jwt". To prevent this attack the authorization server MUST NOT serve requests with content type other than "application/jwt" if the resource server is set up to receive encrypted responses (see also Section 3).

7. Acknowledgements

We would like to thank Petteri Stenius, Neil Madden, Filip Skokan, and Tony Nadalin for their valuable feedback.

8. IANA Considerations
8.1. 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]:

8.1.1. Registry Contents

- Client Metadata Name: "introspection_signed_response_alg"
  - Client Metadata Description: String value indicating the client’s desired introspection response signing algorithm.
  - Change Controller: IESG
  - Specification Document(s): Section 4 of [[ this specification ]]

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

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

8.2. OAuth 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].

8.2.1. Registry Contents

- Metadata Name: "introspection_signing_alg_values_supported"
  - Metadata Description: JSON array containing a list of algorithms supported by the authorization server for introspection response signing.
8.3. OAuth Token Introspection Response

This specification requests registration of the following claim values as defined in [OpenID.Core], Section 5.1, in the IANA "OAuth Token Introspection Response" registry. [IANA.OAuth.Parameters] established by [RFC8414].

8.3.1. Registry Contents

- Name: "name"
  - Description: End-User’s full name in displayable form including all name parts, possibly including titles and suffixes, ordered according to the End-User’s locale and preferences.
  - Change Controller: IESG
  - Specification Document(s): [OpenID.Core], Section 5.1
  - Name: "given_name"
  - Description: Given name(s) or first name(s) of the End-User. Note that in some cultures, people can have multiple given names; all
can be present, with the names being separated by space characters.

- Change Controller: IESG
- Specification Document(s): [OpenID.Core], Section 5.1
- Name: "family_name"
- Description: Surname(s) or last name(s) of the End-User. Note that in some cultures, people can have multiple family names or no family name; all can be present, with the names being separated by space characters.

- Change Controller: IESG
- Specification Document(s): [OpenID.Core], Section 5.1
- Name: "middle_name"
- Description: Middle name(s) of the End-User. Note that in some cultures, people can have multiple middle names; all can be present, with the names being separated by space characters. Also note that in some cultures, middle names are not used.

- Change Controller: IESG
- Specification Document(s): [OpenID.Core], Section 5.1
- Name: "nickname"
- Description: Casual name of the End-User that may or may not be the same as the given_name. For instance, a nickname value of Mike might be returned alongside a given_name value of Michael.

- Change Controller: IESG
- Specification Document(s): [OpenID.Core], Section 5.1
- Name: "preferred_username"
- Description: Shorthand name by which the End-User wishes to be referred to at the RP, such as janedoe or j.doe. This value MAY be any valid JSON string including special characters such as $, /, or whitespace.

- Change Controller: IESG
o Specification Document(s): [OpenID.Core], Section 5.1

o Name: "profile"

o Description: URL of the End-User’s profile page. The contents of this Web page SHOULD be about the End-User.

o Change Controller: IESG

o Specification Document(s): [OpenID.Core], Section 5.1

o Name: "picture"

o Description: URL of the End-User’s profile picture. This URL MUST refer to an image file (for example, a PNG, JPEG, or GIF image file), rather than to a Web page containing an image. Note that this URL SHOULD specifically reference a profile photo of the End-User suitable for displaying when describing the End-User, rather than an arbitrary photo taken by the End-User.

o Change Controller: IESG

o Specification Document(s): [OpenID.Core], Section 5.1

o Name: "website"

o Description: URL of the End-User’s Web page or blog. This Web page SHOULD contain information published by the End-User or an organization that the End-User is affiliated with.

o Change Controller: IESG

o Specification Document(s): [OpenID.Core], Section 5.1

o Name: "email"

o Description: End-User’s preferred e-mail address. Its value MUST conform to the RFC 5322 [RFC5322] addr-spec syntax.

o Change Controller: IESG

o Specification Document(s): [OpenID.Core], Section 5.1

o Name: "email_verified"

o Description: True if the End-User’s e-mail address has been verified; otherwise false. When this Claim Value is true, this means that the OP took affirmative steps to ensure that this
The e-mail address was controlled by the End-User at the time the verification was performed. The means by which an e-mail address is verified is context-specific, and dependent upon the trust framework or contractual agreements within which the parties are operating.

- **Change Controller:** IESG
- **Specification Document(s):** [OpenID.Core], Section 5.1
- **Name:** "gender"
- **Description:** End-User’s gender. Values defined by this specification are female and male. Other values MAY be used when neither of the defined values are applicable.

- **Change Controller:** IESG
- **Specification Document(s):** [OpenID.Core], Section 5.1
- **Name:** "birthdate"
- **Description:** Time the End-User’s information was last updated. Its value is a JSON number representing the number of seconds from 1970-01-01T0:0:0Z as measured in UTC until the date/time.

- **Change Controller:** IESG
- **Specification Document(s):** [OpenID.Core], Section 5.1
- **Name:** "zoneinfo"
- **Description:** String from zoneinfo [zoneinfo] time zone database representing the End-User’s time zone. For example, Europe/Paris or America/Los_Angeles.

- **Change Controller:** IESG
- **Specification Document(s):** [OpenID.Core], Section 5.1
- **Name:** "locale"
- **Description:** Time the End-User’s information was last updated. Its value is a JSON number representing the number of seconds from 1970-01-01T0:0:0Z as measured in UTC until the date/time.
o Specification Document(s): [OpenID.Core], Section 5.1

- Name: "phone_number"

- Description: End-User’s preferred telephone number. E.164 [E.164] is RECOMMENDED as the format of this Claim, for example, +1 (425) 555-1212 or +56 (2) 687 2400. If the phone number contains an extension, it is RECOMMENDED that the extension be represented using the RFC 3966 [RFC3966] extension syntax, for example, +1 (604) 555-1234;ext=5678.

- Change Controller: IESG

- Specification Document(s): [OpenID.Core], Section 5.1

- Name: "phone_number_verified"

- Description: True if the End-User’s phone number has been verified; otherwise false. When this Claim Value is true, this means that the OP took affirmative steps to ensure that this phone number was controlled by the End-User at the time the verification was performed. The means by which a phone number is verified is context-specific, and dependent upon the trust framework or contractual agreements within which the parties are operating. When true, the phone_number Claim MUST be in E.164 format and any extensions MUST be represented in RFC 3966 format.

- Change Controller: IESG

- Specification Document(s): [OpenID.Core], Section 5.1

- Name: "address"

- Description: End-User’s preferred postal address. The value of the address member is a JSON [RFC4627] structure containing some or all of the members defined in [OpenID.Core], Section 5.1.1.

- Change Controller: IESG

- Specification Document(s): [OpenID.Core], Section 5.1

- Name: "updated_at"

- Description: Time the End-User’s information was last updated. Its value is a JSON number representing the number of seconds from 1970-01-01T0:0:02 as measured in UTC until the date/time.

- Change Controller: IESG
9. References

9.1. Normative References

[I-D.ietf-oauth-jwt-bcp]

[I-D.ietf-oauth-security-topics]

[OpenID.Core]

[OpenID.Registration]


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 ]]

-03

- added registration for OpenID Connect Standard Claims to OAuth
  Token Introspection Response registry

-02

- updated references

-01

- adapted wording to preclude any accept header except "application/
  jwt" if encrypted responses are required

- use registered alg value RS256 for default signing algorithm
o added text on claims in the token introspection response
-00

o initial version of the WG draft

o defined default signing algorithm

o changed behavior in case resource server is set up for encryption

o Added text on token data leakage prevention to the security considerations

o moved Security Considerations section forward

WG draft
-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
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 October 13, 2019.
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.

```
+--------+                                 +---------------+
|        |                                 | Authorization |
|        |<--(A)-- Get an access token --->|     Server    |
|        |                                 |               |
|        |                                 +---------------+
|        |                                         ^
|        |                                         |
|        |                                         |
|        |<--(B)-- Use the access token -->|   Protected   |
|        |                                 |    Resource   |
|        |                                 |               |
+--------+                                 +---------------+
```

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. While 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). While 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 bearing the access 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.

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 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.

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 [RFC4291] section 2.2) that is expected to be present as an iPAddress SAN entry in the certificate, which the OAuth client will use in mutual TLS authentication.

**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 pre-requisite, the client registers its X.509 certificates (using "jwks" defined in [RFC7591]) or 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 sec 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.


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.

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 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.
{  "iss": "https://server.example.com",  "sub": "ty.webb@example.com",  "exp": 1493726400,  "nbf": 1493722800,  "cnf": {    "x5t#S256": "bwcK0esc3ACC3DB2Y5_lESsXE8o91tc05O89jdN-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.
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_lESsXE8o91tc05O89jdN-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 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".

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 choose to isolate the server side mutual TLS behaviour 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_endpointAliases

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 SNI or actual distinct hosts) to differentiate its TLS behavior as appropriate.

```json
{
  "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"],
  "mtls_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, 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 decouples it 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 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 preimage and 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 the IANA "JWT Confirmation Methods" registry [IANA.JWT.Claims] for JWT "cnf" member values.

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].

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 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]:

- checking of Basic Constraints, basic and extended Key Usage constraints, validity periods, and critical extensions;
- handling of null-terminator bytes and non-canonical string representations in subject names;
- handling of wildcard patterns in subject names;
- 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].

- Confirmation Method Value: "x5t#S256"
- Confirmation Method Description: X.509 Certificate SHA-256 Thumbprint
- Change Controller: IESG
- Specification Document(s): Section 3.1 of [[ this specification ]]
9.2. 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 [RFC8414].

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

- Metadata Name: "mtls_endpoint_aliases"
- 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.
- Change Controller: IESG
- Specification Document(s): Section 5 of [[ this specification ]]

9.3. Token Endpoint Authentication Method Registration

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

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

- Token Endpoint Authentication Method Name:
  "self_signed_tls_client_auth"
- Change Controller: IESG
- 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]
derived 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].

- Claim Name: "cnf"
- Claim Description: Confirmation
- Change Controller: IESG
- 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]:

- Client Metadata Name: "tls_client_certificate_bound_access_tokens"
- Client Metadata Description: Indicates the client’s intention to use mutual TLS client certificate-bound access tokens.
- Change Controller: IESG
- Specification Document(s): Section 3.4 of [[ this specification ]]

- Client Metadata Name: "tls_client_auth_subject_dn"
- Client Metadata Description: String value specifying the expected subject DN of the client certificate.
- Change Controller: IESG
- Specification Document(s): Section 2.1.2 of [[ this specification ]]

- Client Metadata Name: "tls_client_auth_san_dns"
- Client Metadata Description: String value specifying the expected dNSName SAN entry in the client certificate.
- Change Controller: IESG
- Specification Document(s): Section 2.1.2 of [[ this specification ]]

- Client Metadata Name: "tls_client_auth_san_uri"
- Client Metadata Description: String value specifying the expected uniformResourceIdentifier SAN entry in the client certificate.
- Change Controller: IESG
- 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


10.2. Informative References


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 example figures.  A JWK representation of the certificate's public key along with the "x5c" member is also provided.

"cnf":{"x5t#S256":"A4DtL2JmUMhAsvJj5tKyn64GqzmuXbMrJa0n761y5v0"}

-----BEGIN CERTIFICATE-----
MIIBBjCBrAIBAjAKBggqhkjOPQDAjAPMQ0wCwYDVQQDDARtdGxzMB4XDTE4MTAx
ODEyMzczOVxDTIyMDUwMi4EYwczOWXwDzENMAsGA1UEAwEBRsczBZMBMGByqG
SM49agECCcgGS49AwEHAA0IABNcnxywqV6Y8QnhxxxFQ03C7HKW90y1MbnQzj
jj/AuO8/coZwxS7Laf4vOLS9WuneIXhbGGWvsDSb0tH6IxLm8wCgYIKoZIzj0EawID
SQAwRg1hAP0Rc1E+vwJD/1AAGHzuri+hLV/PPQK7WUVeORWz83AiEA5x2eXZO
bUJSGQgjw5vaUaKLR5Q2DmFQj1L+SY=
-----END CERTIFICATE-----

-----BEGIN CERTIFICATE-----
MIIBBjCBrAIBAjAKBggqhkjOPQDAjAPMQ0wCwYDVQQDDARtdGxzMB4XDTE4MTAx
ODEyMzczOVxDTIyMDUwMi4EYwczOWXwDzENMAsGA1UEAwEBRsczBZMBMGByqG
SM49agECCcgGS49AwEHAA0IABNcnxywqV6Y8QnhxxxFQ03C7HKW90y1MbnQzj
jj/AuO8/coZwxS7Laf4vOLS9WuneIXhbGGWvsDSb0tH6IxLm8wCgYIKoZIzj0EawID
SQAwRg1hAP0Rc1E+vwJD/1AAGHzuri+hLV/PPQK7WUVeORWz83AiEA5x2eXZO
bUJSGQgjw5vaUaKLR5Q2DmFQj1L+SY=
-----END CERTIFICATE-----

Figure 6: PEM Encoded Self-Signed Certificate

{  
"kty":"EC",
"x":"1yfLHCpXqFjxCeHHHMVTcLscpb07KUxudBmOMn8C7Q",
"y":"8_coZwxS7Laf4vOLS9WuneIXhbGGWvsDSb0tH6IxLm8",
"crv":"P-256",
"x5c":[
"MIIBBjCBrAIBAjAKBggqhkjOPQDAjAPMQ0wCwYDVQQDDARtdGxzMB4XDTE4MTAx
ODEyMzczOVxDTIyMDUwMi4EYwczOWXwDzENMAsGA1UEAwEBRsczBZMBMGByqG
SM49agECCcgGS49AwEHAA0IABNcnxywqV6Y8QnhxxxFQ03C7HKW90y1MbnQzj
jj/AuO8/coZwxS7Laf4vOLS9WuneIXhbGGWvsDSb0tH6IxLm8wCgYIKoZIzj0EawID
SQAwRg1hAP0Rc1E+vwJD/1AAGHzuri+hLV/PPQK7WUVeORWz83AiEA5x2eXZO
bUJSGQgjw5vaUaKLR5Q2DmFQj1L+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 predated 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 and Benjamin Kaduk serving as Security Area Directors. Additionally, the following individuals contributed ideas, feedback, and wording that helped shape this specification: Vittorio Bertocci,
Appendix D. Document(s) History

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

draft-ietf-oauth-mtls-14

- Editorial clarifications around there being only a single subject registered/configured per client for the tls_client_auth method.
- 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.
- Add mention of refresh tokens in the abstract.

draft-ietf-oauth-mtls-13

- 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.
- A little bit less of that German influence.
- 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’.
- Move the explanation about "cnf" introspection registration into the IANA Considerations.
- Add CIBA as an informational reference and additional example of an OAuth extension that defines an endpoint that utilizes client authentication.
- Shorten a few of the section titles.
- Add new client metadata values to allow for the use of a SAN in the PKI MTLS client authentication method.
- Add privacy considerations attempting to discuss the implications of the client cert being sent in the clear in TLS 1.2.
- 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.
- Reword/restructure the main PKI method section somewhat to (hopefully) improve readability.
- Reword/restructure the Self-Signed method section a bit to (hopefully) make it more comprehensible.
Reword the AS and RS Implementation Considerations somewhat to (hopefully) improve readability.

Clarify that the protected resource obtains the client certificate used for mutual TLS from its TLS implementation layer.

Add Security Considerations section about the certificate thumbprint binding that includes the hash algorithm agility recommendation.

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.

Minor editorial updates.

draft-ietf-oauth-mtls-12

Add an example certificate, JWK, and confirmation method claim.

Minor editorial updates based on implementer feedback.

Additional Acknowledgements.

draft-ietf-oauth-mtls-11

Editorial updates.

Mention/reference TLS 1.3 RFC8446 in the TLS Versions and Best Practices section.

draft-ietf-oauth-mtls-10

Update draft-ietf-oauth-discovery reference to RFC8414

draft-ietf-oauth-mtls-09

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

draft-ietf-oauth-mtls-08

Incorporate clarifications and editorial improvements from Justin Richer’s WGLC review

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)

Add a new security considerations section on X.509 parsing and validation per WGLC feedback from Neil Madden and Benjamin Kaduk

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 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-60tbVSeuzW2k94vCo
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/swDV2y0be6o8czGKQi1eJV-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/
bZ6mft0G7D3cceb0xnEYUv4puI

- 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
- Editorial fixes and clarifications

draft-ietf-oauth-mtls-03

- Introduced metadata and client registration parameter to publish
  and request support for mutual TLS sender constrained access
tokens
- Added description of two methods of binding the cert and client,
  PKI and Public Key.
- 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
- 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
- Added new section to security considerations on cert spoofing
- 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

- Fixed editorial issue https://mailarchive.ietf.org/arch/msg/oauth/
  U46UMEh8XIOQnvXY9pHFq1MKPns
- 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

- Added more explicit details of using RFC 7662 token introspection
  with mutual TLS sender constrained access tokens.
- Added an IANA OAuth Token Introspection Response Registration
  request for "cnf".
- Specify that tls_client_auth_subject_dn and
tls_client_auth_root_dn are RFC 4514 String Representation of
  Distinguished Names.
- Changed tls_client_authIssuer_dn to tls_client_auth_root_dn.
- 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/
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.

Status of This Memo

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

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

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

This Internet-Draft will expire on September 28, 2019.
1. Introduction

The work on proof-of-possession tokens, an extended token security mechanisms for OAuth 2.0, is motivated in [22]. This document defines the ability for the client request and to obtain PoP tokens from the authorization server. After successfully completing the exchange the client is in possession of a PoP token and the keying material bound to it. Clients that access protected resources then need to demonstrate knowledge of the secret key that is bound to the PoP 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). To demonstrate possession of the private/secret key to the resource server protocol mechanisms outside the scope of this document are used.

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 [19] 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 specification 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 4.1 and the procedure for dealing with asymmetric keys is described in Section 4.2.

This document describes how the client requests and obtains a PoP access token from the authorization server for use with HTTPS-based

transport. The use of alternative transports, such as Constrained Application Protocol (CoAP), is described in [24].

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:

In the context of this specification '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:

JWT: JSON Web Token[9]
JWS: JSON Web Signature[6]
JWK: JSON Web Key[5]
JWE: JSON Web Encryption[8]
CWT: CBOR Web Token[13]
COSE: CBOR Object Signing and Encryption[14]

3. 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 use the resource identicator parameter, defined in [16], or the audience parameter, defined in [15], when symmetric PoP tokens are used. For asymmetric PoP tokens the use of resource indicators and audience is optional but
The parameters ‘audience’ and ‘resource’ both allow the client to express the location of the target service and the difference between the two is described in [15]. As a summary, ‘audience’ allows expressing a logical name while ‘resource’ contains an absolute URI. More details about the ‘resource’ parameter can be found in [16].

Step (3): The authorization server parses the request from the server and determines the suitable response based on OAuth 2.0 and the PoP token credential procedures.

Note that PoP access tokens may be encoded in a variety of ways:

**JWT** The access token may be encoded using the JSON Web Token (JWT) format [9]. The proof-of-possession token functionality is described in [10]. A JWT encoded PoP token MUST be protected against modification by either using a digital signature or a keyed message digest, as described in [6]. The JWT may also be encrypted using [8].

**CWT** [13] defines an alternative token format based on CBOR. The proof-of-possession token functionality is defined in [12]. A CWT encoded PoP token MUST be protected against modification by either using a digital signature or a keyed message digest, as described in [12].

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 [23].

Note that the OAuth 2.0 framework nor this specification does not mandate a specific PoP token format but using a standardized format will improve interoperability and will lead to better code re-use.

Application layer interactions between the client and the resource server are beyond the scope of this document.

4. Examples

This section provides a number of examples.

4.1. Symmetric Key Transport

4.1.1. Client-to-AS Request

The client starts with a request to the authorization server indicating that it is interested to obtain a token for https://resource.example.com
Example Request to the Authorization Server

4.1.2. 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.

Figure 2 shows a response containing a token and a "cnf" parameter with a symmetric proof-of-possession key both encoded in a JSON-based serialization format. The "cnf" parameter contains the RFC 7517 [5] encoded key element.
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",
   "cnf": {
      "keys": [
         {"kty": "oct",
          "alg": "A128KW",
          "k": "GawgguFyGrWKav7AX4VKUg"
         }
      ]
   }
}

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

Note that the cnf payload in Figure 2 is not encrypted at the application layer since Transport Layer Security is used between the AS and the client and the content of the cnf payload is consumed by the client itself. Alternatively, a JWE could be used to encrypt the key distribution, as shown in Figure 3.
The content of the ‘access_token’ in JWT format contains the ‘cnf’ (confirmation) claim. 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 has to be present in a real deployment to mitigate a number of security threats.

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).

The digital signature or the keyed message digest offering integrity protection is not shown in this example but has to be present in a real deployment to mitigate a number of security threats.

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).
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 since content is meant for consumption by the selected resource server only. The details are described in [22].

4.2. Asymmetric Key Transport

4.2.1. Client-to-AS Request

This example illustrates the case where an asymmetric key shall be bound to an access token. The client makes the following HTTPS request shown in Figure 5. 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=SplxlOBeZQQYbYS6WxSbIA
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
&token_type=pop
&req_cnf=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 'req_cnf' parameter contains the ECC public key the client would like to associate with the access token (in JSON format).

```
"jwk":{
  "kty": "EC",
  "use": "sig",
  "crv": "P-256",
  "x": "18wHLe1gW9wVN6VD1TxgPqy2LszYkMf6J8njVAibvhM",
  "y": "-V4dS4uAMgP_4fY4j8ir7cl1TX1FdAgcx55o7TkcsA"
}
```

Figure 6: Client Providing Public Key to Authorization Server
4.2.2. 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. 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 "pop" is placed into the "token_type" parameter.

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

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

{
    "access_token":"2YotnFZFE....jr1zCsicMWpAA",
    "token_type":"pop",
    "expires_in":3600,
    "refresh_token":"tGzv3J0kF0XG5Qx2T1KWIA"
}
```

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

The content of the 'access_token' field contains an encoded JWT, as shown in Figure 8. The digital signature covering the access token offering authenticity and integrity protection is not shown below (but must be present).
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 key (as well as the public key).

5. Security Considerations

[22] 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 [17] 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.
6. IANA Considerations

6.1. OAuth Access Token Types

This specification registers the following error in the IANA "OAuth Access Token Types" [25] established by [17].

- Name: pop
- Change controller: IESG
- Specification document(s): [[ this specification ]]

6.2. OAuth Parameters Registration

This specification registers the following value in the IANA "OAuth Parameters" registry [25] established by [2].

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

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

- Parameter name: rs_cnf
- Parameter usage location: token response
- Change controller: IESG
- Specification document(s): [[ this specification ]]

6.3. OAuth Extensions Error Registration

This specification registers the following error in the IANA "OAuth Extensions Error Registry" [25] established by [2].

- Error name: invalid_token_type
- Error usage location: implicit grant error response, token error response
- Related protocol extension: token_type parameter
- Change controller: IESG
- Specification document(s): [[ this specification ]]

7. Acknowledgements

We would like to thank Chuck Mortimore and James Manger for their review comments.
8. References

8.1. Normative References


8.2. Informative References


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/
Reciprocal OAuth
draft-ietf-oauth-reciprocal-01

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 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
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.

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 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:
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":"tGzv3JOkF0XG5Qx2TkWIA",
  "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"

3. 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.

3.1. 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 party B had preconfigured at party A.

3.2. 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 to provide user context.

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 ej4hsyfishwssjdisdhjksusdhjkjsdjk
Content-Type: application/x-www-form-urlencoded

grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Areiprocal&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.

4. 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 3.2.

5. IANA Considerations

TBD.

6. Acknowledgements

TBD.
7. Normative References


Appendix A. Document History

A.1. draft-ietf-oauth-reciprical-00
   o Initial version.

A.2. draft-ietf-oauth-reciprical-01
   o changed reciprocal scope request to be in access token response rather than authorization request

Author’s Address

Dick Hardt
Amazon

Email: dick.hardt@gmail.com
Resource Indicators for OAuth 2.0

draft-ietf-oauth-resource-indicators-02

Abstract

An extension to the OAuth 2.0 Authorization Framework defining request parameters that enable a client to explicitly signal to an authorization server about the identity of the protected resource(s) to which it is requesting access.

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 August 1, 2019.

Copyright Notice

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

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

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

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

Specific knowledge of the intended recipient(s) of the access token also helps facilitate improved security characteristics of the token itself. Bearer tokens, currently the most commonly utilized type of OAuth access token, allow any party in possession of a token to get access to the associated resources. To prevent misuse, several important security assumptions must hold, one of which is that an access token must only be valid for use at a specific protected...
resource and for a specific scope of access. Section 5.2 of [RFC6750], for example, prescribes including the token’s intended recipients within the token to prevent token redirect. When the authorization server is informed of the resource that will process the access token, it can restrict the intended audience of that token to the given resource such that the token cannot be used successfully at other resources.

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

In some circumstances and for some deployments, a means for the client to signal to the authorization server where it intends to use the access token it’s requesting is important and useful. A number of implementations and deployments of OAuth 2.0 have already employed proprietary parameters toward that end. Going forward, this specification aspires to provide a standardized and interoperable alternative to the proprietary approaches.

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

2. Resource Parameter

In requests to the authorization server, a client MAY indicate the protected resource (a.k.a. resource server, application, API, etc.) to which it is requesting access by including the following parameter in the request.
resource
Indicates the target service or resource at which access is being requested. Its value MUST be an absolute URI, as specified by Section 4.3 of [RFC3986], which MAY include a query component but MUST NOT include a fragment component. The "resource" parameter URI value is an identifier representing the identity of the resource, which MAY be a locator that corresponds to a network addressable location where the target resource is hosted. Multiple "resource" parameters MAY be used to indicate that the requested token is intended to be used at multiple resources.

The parameter value identifies a resource to which the client is requesting access. The parameter can carry the location of a protected resource, typically as an https URL, or a more abstract identifier. This enables the authorization server to apply policy as appropriate for the resource, such as determining the type and content of tokens to be issued, if and how tokens are encrypted, and applying appropriate audience restrictions.

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

The following error code is provided for an authorization server to indicate problems with the requested resource(s) in response to an authorization request or access token request. And can also be used to inform the client that it has requested an invalid combination of resource and scope.

invalid_target
The requested resource is invalid, unknown, or malformed.

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

2.1. Authorization Request

When the "resource" parameter is used in an authorization request to the authorization endpoint, it indicates the identity of the protected resource(s) to which access is being requested. When an access token will be returned directly from the authorization endpoint via the implicit flow (Section 4.2 of OAuth 2.0 [RFC6749]), the requested resource is applicable to that access token. In the code flow (Section 4.1 of OAuth 2.0 [RFC6749]) where an intermediate representation of the authorization grant (the authorization code) is returned from the authorization endpoint, the requested resource is applicable to the full authorization grant.

For authorization requests sent as a JWTs, such as when using JWT Secured Authorization Request [I-D.ietf-oauth-jwsreq], a single "resource" parameter value is represented as a JSON string while multiple values are represented as an array of strings.

If the client omits the "resource" parameter when requesting authorization, the authorization server MAY process the request with no specific resource or by using a pre-defined default resource value. Alternatively, the authorization server MAY require clients to specify the resource(s) they intend to access and MAY fail requests that omit the parameter with an "invalid_target" error. The authorization server might use this data to inform the user about the resources the client is going to access on her behalf, to meet policy decision (e.g. refuse the request due to unknown resources), and determine the set of resources that can be used in subsequent access token requests.

If the authorization server fails to parse the provided value(s) or does not consider the resource(s) acceptable, it should reject the request with an an error response using the error code "invalid_target" as the value of the "error" parameter and can provide additional information regarding the reasons for the error using the "error_description" and/or "error_uri" parameters.

An example of an authorization request where the client tells the authorization server that it wants an access token for use at
"https://api.example.com/app/" is shown in Figure 1 below (extra line breaks and indentation are for display purposes only).

```
GET /as/authorization.oauth2?response_type=token
 &client_id=example-client
 &state=XzZaJ1cwYewlu0QBBrVw
 &redirect_uri=https%3A%2F%2Fclient%2Eexample%2Eorg%2Fcb
 &resource=https%3A%2F%2Fapi.example.com%2Fapp%2F HTTP/1.1
Host: authorization-server.example.com
```

Figure 1: Implicit Flow Authorization Request

Below in Figure 2 is an example of an authorization request using the "code" response type where the client is requesting access to the resource owner’s contacts and calendar data at "https://cal.example.com/*" and "https://contacts.example.com/*" (extra line breaks and indentation are for display purposes only).

```
GET /as/authorization.oauth2?response_type=code
 &client_id=s6BhdRkqt3
 &state=tNwzQ87pC61lepmac_IDeeq-mCR2wLDY1jHUZUAWuI
 &redirect_uri=https%3A%2F%2Fclient%2Eexample%2Eorg%2Fcb
 &scope=calendar%20contacts
 &resource=https%3A%2F%2Fcal.example.com%2F
 &resource=https%3A%2F%2Fcontacts.example.com%2F HTTP/1.1
Host: authorization-server.example.com
```

Figure 2: Code Flow Authorization Request

2.2. Access Token Request

When the "resource" parameter is used on an access token request made to the token endpoint, for all grant types, it indicates the target service or protected resource where the client intends to use the requested access token.

The resource value(s) that are acceptable to an authorization server in fulfilling an access token request are at its sole discretion based on local policy or configuration. In the case of a "refresh_token" or "authorization_code" grant type request, such policy may limit the acceptable resources to those that were originally granted by the resource owner or a subset thereof. In the "authorization_code" case where the requested resources are a subset of the set of resources originally granted, the authorization server will issue an access token based on that subset of requested resources while any refresh token that is returned is bound to the full original grant.
When requesting a token, the client can indicate the desired target service(s) where it intends to use that token by way of the "resource" parameter and can indicate 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. To the extent possible, when issuing access tokens, the authorization server should adapt the scope value associated with an access token to the value the respective resource is able to process and needs to know. This further improves privacy as scope values give an indication of what services the resource owner uses and it improves security as scope values may contain confidential data. As specified in Section 5.1 of [RFC6749], the authorization server must indicate the access token’s effective scope to the client in the "scope" response parameter value when it differs from the scope requested by the client.

Following from the code flow authorization request shown in Figure 2, the below examples show an "authorization_code" grant type access token request and response where the client tells the authorization server that it wants the access token for use at "https://cal.example.com/" (extra line breaks and indentation are for display purposes only).

```plaintext
POST /as/token.oauth2 HTTP/1.1
Host: authorization-server.example.com
Authorization: Basic czZCaGRSa3F0Mzpoc3FFelFsVW9IQUU5cHg0RlNyNHlJ
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Eorg%2Fcb
&code=10esc29BWC2qZB0acc9v8zAv9ltc2pko105tQauZ
&resource=https%3A%2F%2Fcal.example.com%2F

Figure 3: Access Token Request
```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-cache, no-store

{
   "access_token": "eyJhbGciOiJFUzI1NiIsImtpZCI6Ijc3In0.eyJpc3MiOiJodHRwOi8vYXV0aG9yaXphdGlvbi1zZXJ2ZXIuZXhhbXBsZS5jb20iLCJ0b19fXyIsImV4cCI6MTU4ODQyMDgwMCwic2NvcGUiOiJhYyJ9.nNWJ2dXSxaDRdMUKlzs-cYi8MDoM6Gy7pf_sKrLgSAFlC2bDhB60DQfW1DZL5npdkol_Mmk5sUFzkivNQnYw",
   "token_type": "Bearer",
   "expires_in": 3600,
   "refresh_token": "4LTC8lb0acc6Oy4esc1Nk9BWC0imAwH7kic16BDC2",
   "scope": "calendar"
}

Figure 4: Access Token Response

A subsequent access token request, using the refresh token, where the client tells the authorization server that it wants an access token for use at "https://contacts.example.com/" is shown in Figure 5 below with the response shown in Figure 6 (extra line breaks and indentation are for display purposes only).

POST /as/token.oauth2 HTTP/1.1
Host: authorization-server.example.com
Authorization: Basic czZCaGRSa3F0Mzpc3FFe1FsVW9IU5cHg0RLMyNHIJ
Content-Type: application/x-www-form-urlencoded

grant_type=refresh_token
&refresh_token=4LTC8lb0acc6Oy4esc1Nk9BWC0imAwH7kic16BDC2
&resource=https%3A%2F%2Fcontacts.example.com%2Fapp%2F

Figure 5: Access Token Request
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-cache, no-store

{
  "access_token":"eyJhbGciOiJFUzI1NiIsImtpZCI6Ijc3In0.eyJpc3MiOiJodHRwOi8vYXV0aG9yaXphdGlvbi1zZXJ2ZXIuZXhhbXBsb2dSb25yYWN0cyIsImF1ZCI6Imh0dHBzOi8vY29udGFjdHMuZXhhbXBsb2dSb25yYWN0cyIsImF1ZCI6ImF1ZCI6Mn0.eyJodHRwOi8vYXV0aG9yaXphdGlvbi1zZXJ2ZXIuZXhhbXBsb2dSb25yYWN0cyI6W10.5f4yhqazcOSlJw4y94KPeWNEFQqj2cefeO8x4hr3YbHtIl3nQXnBMw5wREY501YbZED-GfH
  "token_type":"Bearer",
  "expires_in":3600,
  "scope":"contacts"
}

Figure 6: Access Token Response

3. Security Considerations

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

Some servers may host user content or be multi-tenant. In order to avoid attacks that might confuse a client into sending an access token to a resource that is user controlled or is owned by a different tenant, it is important to use a specific resource URI including a path component. This will cause any access token issued for accessing the user controlled resource to have a invalid audience if replayed against the legitimate resource API.

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

Whenever feasible, the "resource" parameter should correspond to the network addressable location of the protected resource. This makes it possible for the client to validate that the resource being
requested controls the corresponding network location, reducing the
risk of malicious endpoints obtaining tokens meant for other
resources. If the "resource" parameter contains an abstract
identifier, it is the client’s responsibility to validate out of band
that any network endpoint to which tokens are sent are the intended
audience for that identifier.

4. IANA Considerations

4.1. OAuth Parameters Registration

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

- Parameter name: resource
- Parameter usage location: authorization request, token request
  [[TODO: draft-ietf-oauth-token-exchange will have already
  registered this for 'token request' and this draft has a more
  generalized usage and needs to somehow either update that
  registration or do a partial registration and reference]]
- Change controller: IESG
- Specification document(s): [[ this specification ]]

4.2. OAuth Extensions Error Registration

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

- Error name: invalid_target
- Error usage location: implicit grant error response, token error
  response [[TODO: draft-ietf-oauth-token-exchange will have already
  registered this for 'token error response' and this draft has a more
  generalized usage and needs to somehow either update that
  registration or do a partial registration and reference]]
- Related protocol extension: resource parameter
- Change controller: IESG
- Specification document(s): [[ this specification ]]

5. References

5.1. Normative References

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

[I-D.ietf-oauth-jwsreq]


Appendix A. Acknowledgements

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

Vittorio Bertocci, Sergey Beryozkin, William Denniss, Vladimir Dzhuvinov, George Fletcher, Dick Hardt, Phil Hunt, Michael Jones, Torsten Lodderstedt, Anthony Nadalin, Justin Richer, Nat Sakimura, Filip Skokan, and Hans Zandbelt.

Appendix B. Document History

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

draft-ietf-oauth-resource-indicators-02

  o Clarify that the value of the "resource" parameter is a URI which can be an abstract identifier for the target resource and doesn’t necessarily have to correspond to a network addressable location.

draft-ietf-oauth-resource-indicators-01

  o Significant rework of the main section of the document attempting to clarify a number of things that came up at, around and after IETF 102 and the call for adoption.
  o Change the "invalid_resource" error to "invalid_target" to align with draft-ietf-oauth-token-exchange, which has some overlap in functionality.
  o Allow the "resource" parameter value to have a query component (aligning with draft-ietf-oauth-token-exchange).
  o Moved the Security Considerations section to before the IANA Considerations.
  o Other editorial updates.
  o Rework the Acknowledgements section.
  o Use RFC 8174 boilerplate.

draft-ietf-oauth-resource-indicators-00

  o First version of the working group document. A replica of draft-campbell-oauth-resource-indicators-02.

draft-campbell-oauth-resource-indicators-02

  o No changes.

draft-campbell-oauth-resource-indicators-01

  o Move Hannes Tschofenig, who wrote https://tools.ietf.org/html/draft-tschofenig-oauth-audience in ’13, from Acknowledgements to Authors.
Added IANA Considerations to register the "resource" parameter and "invalid_resource" error code.

Initial draft to define a resource parameter for OAuth 2.0.

Authors' Addresses

Brian Campbell
Ping Identity

Email: brian.d.campbell@gmail.com

John Bradley
Yubico

Email: ve7jtb@ve7jtb.com

Hannes Tschofenig
Arm Limited

Email: hannes.tschofenig@gmx.net
OAuth 2.0 Security Best Current Practice
draft-ietf-oauth-security-topics-12

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 September 9, 2019.

Copyright Notice

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

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

Table of Contents

1. Introduction .................................................. 3
   1.1. Structure ................................................ 4
   1.2. Conventions and Terminology ............................. 4
2. The Updated OAuth 2.0 Attacker Model ......................... 4
3. Recommendations ............................................... 6
   3.1. Protecting Redirect-Based Flows .......................... 6
       3.1.1. Authorization Code Grant .......................... 7
       3.1.2. Implicit Grant ................................. 7
   3.2. Token Replay Prevention .................................. 8
   3.3. Access Token Privilege Restriction ........................ 8
4. Attacks and Mitigations ....................................... 9
   4.1. Insufficient Redirect URI Validation ..................... 9
       4.1.1. Redirect URI Validation Attacks on Authorization Code Grant ........................................ 9
       4.1.2. Redirect URI Validation Attacks on Implicit Grant .................................................. 10
       4.1.3. Proposed Countermeasures ........................... 12
   4.2. Credential Leakage via Referrer Headers .................. 12
       4.2.1. Leakage from the OAuth Client ..................... 13
       4.2.2. Leakage from the Authorization Server .............. 13
       4.2.3. Consequences .................................... 13
       4.2.4. Proposed Countermeasures ........................... 13
   4.3. Attacks through the Browser History ...................... 14
       4.3.1. Code in Browser History ........................... 14
       4.3.2. Access Token in Browser History ................... 15
   4.4. Mix-Up .................................................. 15
       4.4.1. Attack Description ................................ 15
       4.4.2. Countermeasures ................................. 17
   4.5. Authorization Code Injection ............................. 18
       4.5.1. Attack Description ................................ 18
       4.5.2. Discussion ..................................... 19
       4.5.3. Proposed Countermeasures ........................... 20
   4.6. Access Token Injection ................................... 21
       4.6.1. Proposed Countermeasures ........................... 22
   4.7. Cross Site Request Forgery ................................ 22
       4.7.1. Proposed Countermeasures ........................... 22
       4.8.1. Access Token Phishing by Counterfeit Resource Server ........................................... 22
       4.8.2. Compromised Resource Server ........................ 28
   4.9. Open Redirection ......................................... 28
       4.9.1. Authorization Server as Open Redirector ........... 29
1. Introduction

Since its publication in [RFC6749] and [RFC6750], OAuth 2.0 has gotten massive traction in the market and became the standard for API protection and, as foundation of OpenID Connect [OpenID], identity providing. While OAuth was used in a variety of scenarios and different kinds of deployments, the following challenges could be observed:

- OAuth implementations are being attacked through known implementation weaknesses and anti-patterns (CSRF, referrer header). Although most of these threats are discussed in the OAuth 2.0 Threat Model and Security Considerations [RFC6819], continued exploitation demonstrates there may be a need for more specific recommendations or that the existing mitigations are too difficult to deploy.

- Technology has changed, e.g., the way browsers treat fragments in some situations, which may change the implicit grant’s underlying security model.

- OAuth is 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], [RFC6749], and [RFC6819].

Moreover, OAuth is being adopted in use cases 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 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 serves as a frontend to a particular tenant in a multi-tenancy. Extensions of OAuth, such as [RFC7591] and [RFC8414] were developed in order to support the usage of OAuth in dynamic scenarios. As a challenge to the community, such usage scenarios open up new attack angles, which are discussed in this document.

1.1. Structure

The remainder of the document is organized as follows: The next section updates the OAuth attacker model. Afterwards, the most important recommendations of the OAuth working group for every OAuth implementor are summarized. Subsequently, a detailed analysis of the threats and implementation issues which 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.

2. The Updated OAuth 2.0 Attacker Model

In [RFC6819], an attacker model was laid out that described the capabilities of attackers against which OAuth deployments must defend. In the following, this attacker model is updated to account for the potentially dynamic relationships involving multiple parties (as described above), to include new types of attackers, and to make it more clearly defined.

OAuth 2.0 aims to ensure that the authorization of the resource owner (RO) (with a user agent) at an 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 control an arbitrary number of network endpoints (except for RO, AS, and RS). Web attackers may set up web sites that are visited by the RO, operate their own user
agents, participate in the protocol using their own user credentials, etc. 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 ROs. It must also be assumed that web attackers can lure the user to open arbitrary attacker-chosen URIs at any time. This can be achieved through many ways, for example, by injecting malicious advertisements into advertisement networks, or by sending legitimate-looking emails.

- (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 attacker can also block specific messages.

These attackers conform to the attacker model that was used in formal analysis efforts for OAuth [arXiv.1601.01229]. Previous attacks on OAuth have shown that OAuth deployments should protect against an even strong attacker model that is described as follows:

- (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), so-called mix-up attacks, 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 control a resource server used by RO with 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.

Note that in this attacker model, an attacker can be a RO or act as one (see A1). 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 discusses the additional threats resulting from these attackers in detail and recommends suitable mitigations.

This is a minimal attacker model. Implementers MUST take into account all possible attackers in the environment in which their OAuth implementations are expected to run.

3. Recommendations

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

3.1. Protecting Redirect-Based Flows

Authorization servers MUST utilize exact matching of client redirect URIs against pre-registered URIs. This measure contributes to the prevention of leakage of authorization codes and access tokens (depending on the grant type). It also helps to detect mix-up attacks.

Clients SHOULD avoid forwarding the user’s browser to a URI obtained from a query parameter since such a function could be utilized to exfiltrate authorization codes and access tokens. If there is a strong need for this kind of redirects, clients are advised to implement appropriate countermeasures against open redirection, e.g., as described by the OWASP [owasp].

Clients MUST prevent CSRF and ensure that each authorization response is only accepted once. One-time use CSRF tokens carried in the "state" parameter, which are securely bound to the user agent, SHOULD be used for that purpose.

In order to prevent mix-up attacks, clients MUST only process redirect responses of the OAuth authorization server they sent the respective request to and from the same user agent this authorization request was initiated with. Clients MUST memorize which authorization server they sent an authorization request to and bind this information to the user agent and ensure any subsequent messages are sent to the same authorization server. Clients SHOULD use AS-specific redirect URIs as a means to identify the AS a particular response came from.

Note: [I-D.bradley-oauth-jwt-encoded-state] gives advice on how to implement CSRF prevention and AS matching using signed JWTs in the "state" parameter.
AS which redirect a request that potentially contains user credentials MUST avoid forwarding these user credentials accidentally (see Section 4.10).

3.1.1. Authorization Code Grant

Clients utilizing the authorization grant type MUST use PKCE [RFC7636] in order to (with the help of the authorization server) detect and prevent attempts to inject (replay) authorization codes into the authorization response. The PKCE challenges must be transaction-specific and securely bound to the user agent in which the transaction was started and the respective client. OpenID Connect clients MAY use the "nonce" parameter of the OpenID Connect authentication request as specified in [OpenID] in conjunction with the corresponding ID Token claim for the same purpose.

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

Authorization servers SHOULD use client authentication if possible.

Authorization servers SHOULD furthermore consider the recommendations given in [RFC6819], Section 4.4.1.1, on authorization code replay prevention.

3.1.2. 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 3.2. 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 any other response type issuing access tokens in the authorization response, such as "token id_token" and "code token id_token", unless the issued access tokens are sender-constrained and access token injection in the authorization response is prevented.

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).

Clients SHOULD instead use the response type "code" (aka authorization code grant type) as specified in Section 3.1.1 or any other response type that causes the authorization server to issue access tokens in the token response. This allows the authorization server to detect replay attempts 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.

3.2. Token Replay Prevention

Authorization servers SHOULD use TLS-based methods for sender-constrained access tokens as described in Section 4.8.1.2, such as token binding [I-D.ietf-oauth-token-binding] or Mutual TLS for OAuth 2.0 [I-D.ietf-oauth-mtls] in order to prevent token replay. Refresh tokens MUST be sender-constrained or use refresh token rotation as described in Section 4.12. It is also recommended to use end-to-end TLS whenever possible.

3.3. 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 limit the impact of token leakage although more effective counter-measures are described in Section 3.2.

In particular, access tokens SHOULD be restricted to certain resource servers, 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] to determine those resources and/or actions.

4. Attacks and Mitigations

This section gives a detailed description of attacks on OAuth implementations, along with potential countermeasures. This section complements and enhances the description given in [RFC6819].

4.1. Insufficient Redirect URI Validation

Some authorization servers allow clients to register redirect URI patterns instead of complete redirect URIs. In those cases, the authorization server, at runtime, matches the actual redirect URI parameter value at the authorization endpoint against this pattern. This approach allows clients to encode transaction state into additional redirect URI parameters or to register just a single pattern for multiple redirect URIs. As a downside, it turned out to be more complex to implement and error prone to manage than exact redirect URI matching. Several successful attacks, utilizing 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.

4.1.1. Redirect URI Validation Attacks on Authorization Code Grant

For a public client using the grant type code, an attack would look as follows:

Let’s assume the redirect URL pattern "https://*.somesite.example/*" had been registered for the client "s6BhdRkqt3". This pattern allows redirect URIs pointing to any host residing in the domain somesite.example. So if an attacker manages to establish a host or subdomain in somesite.example he can impersonate the legitimate client. Assume the attacker sets up the host "evil.somesite.example".
The attack can then be conducted as follows:

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

This URL initiates an authorization request with the client id of a legitimate client to the authorization endpoint. This is the example authorization request (line breaks are for display purposes only):

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

Afterwards, the authorization server validates the redirect URI in order to identify the client. Since the pattern allows arbitrary domains host names in "somesite.example", the authorization request is processed under the legitimate client’s identity. This includes the way the request for user consent is presented to the user. If auto-approval is allowed (which is not recommended for public clients according to [RFC6749]), the attack can be performed even easier.

If the user does not recognize the attack, the code is issued and immediately sent to the attacker’s client.

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

Note: This attack will not work as easily for confidential clients, since the code exchange requires authentication with the legitimate client’s secret. The attacker will need to impersonate or utilize the legitimate client to redeem the code (e.g., by performing a code injection attack). This kind of injections is covered in Section 4.5.

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 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 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, which launches a page under the attacker’s control, say "https://www.evil.example".

Afterwards, the website initiates an authorization request, which 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://client.evil.example" into the redirect URI and it uses the response type "token" (line breaks are for display purposes 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%252Fclient.evil.example%252Fcb HTTP/1.1
Host: server.somesite.example

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

HTTP/1.1 303 See Other
Location: https://client.somesite.example/cb?
  redirect_to=https%3A%2F%2Fclient.evil.example%2Fcb
  #access_token=2YotnFZFEjr1zCsicMWpAA&...

At example.com, the request arrives at the open redirector. It will read the redirect parameter and will issue an HTTP 303 Location header redirect to the URL "https://client.evil.example/cb".

HTTP/1.1 303 See Other
Location: https://client.evil.example/cb
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=2YotnFZFEjr1zCsiMcMWPAA&..." to the URL and will navigate to the following URL:

https://client.evil.example/cb#access_token=2YotnFZFEjr1z...

The attacker’s page at “client.evil.example” can now access the fragment and obtain the access token.

4.1.3. Proposed Countermeasures

The complexity of implementing and managing pattern matching correctly obviously causes security issues. This document therefore proposes to simplify the required logic and configuration by using exact redirect URI matching only. This means the authorization server must compare the two URIs using simple string comparison as defined in [RFC3986], Section 6.2.1.

Additional recommendations:

- Servers on which callbacks are hosted must not expose open redirectors (see Section 4.9).
- Clients MAY drop fragments via intermediary URLs with "fix fragments" (see [fb_fragments]) to prevent the user agent from appending any unintended fragments.
- 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 certificate binding of the access tokens.

As an alternative to exact redirect URI matching, the AS could also authenticate clients, e.g., using [I-D.ietf-oauth-jwsreq].

4.2. Credential Leakage via Referrer Headers

Authorization codes or values of "state" can unintentionally be disclosed to attackers through the referrer header, by leaking either from a client’s web site or from an AS’s web site. Note: even if 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 (ads, faq, ...) and a user clicks on such a link, or

- includes third-party content (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", "access token", or "state".

4.2.2. Leakage from the Authorization Server

In a similar way, an attacker can learn "state" 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. Proposed 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:

- 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 referrer 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.)

- Suppress the referrer header by adding the attribute "rel="noreferrer" to HTML links or 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).

- Use authorization code instead of response types causing access token issuance from the authorization endpoint. This provides countermeasures against leakage on the OAuth protocol level through the code exchange process with the authorization server.

- Additionally, one might use the form post response mode instead of redirect for authorization response (see [oauth-v2-form-post-response-mode]).

4.3. Attacks through the 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. 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.

Proposed countermeasures:

- Authorization code replay prevention as described in [RFC6819], Section 4.4.1.1, and Section 4.5
4.3.2. Access Token in Browser History

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

In case of 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.

Proposed countermeasures:

- Replace implicit flow with postmessage communication or the authorization code grant
- Never pass access tokens in URL query parameters

4.4. Mix-Up

Mix-up is an attack on scenarios where an OAuth client interacts with multiple authorization servers, as is usually the case when dynamic registration is used. The goal of the attack is to obtain an authorization code or an access token by tricking the client into sending those credentials to the attacker instead of using them at the respective endpoint at the authorization/resource server.

4.4.1. Attack Description

For a detailed attack description, refer to [arXiv.1601.01229] and [I-D.ietf-oauth-mix-up-mitigation]. The description here closely follows [arXiv.1601.01229], with variants of the attack outlined below.

Preconditions: For 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),
o 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, and

o the attacker can 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.

Some of the attack variants described below require different preconditions.

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").

Attack on the authorization code grant:

1. The user selects to start the grant using H-AS (e.g., by clicking on a button at the client’s website).

2. The attacker intercepts this request and changes the user’s selection to "A-AS".

3. The client stores in the user’s session that the user selected "A-AS" and redirects the user to A-AS’s authorization endpoint by sending the response code "303 See Other" with a Location header containing the URL "https://attacker.example/authorize?response_type=code&client_id=666RVZJTA".

4. Now the attacker intercepts this response and changes the redirection such that the user is being redirected to H-AS. The attacker also 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"

5. Now, the user authorizes the client to access her resources at H-AS. H-AS issues a code and sends it (via the browser) back to the client.

6. 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.

7. The attacker therefore obtains code and can either exchange the code for an access token (for public clients) or perform a code injection attack as described in Section 4.5.
Variants:

- *Implicit Grant*: In the implicit grant, the attacker receives an access token instead of the code; the rest of the attack works as above.

- *Mix-Up Without Interception*: A variant of the above attack works even if the first request/response pair cannot be intercepted (for example, because TLS is used to protect these messages): Here, we assume that the user wants to start the grant using A-AS (and not H-AS, see Attacker A1). After the client redirected the user to the authorization endpoint at A-AS, the attacker immediately redirects the user to H-AS (changing the client id to "7ZGZldHQ"). (A vigilant user might at this point detect that she intended to use A-AS instead of H-AS.) The attack now proceeds exactly as in Steps 3ff. of the attack description 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". Refer to [oauth_security_jcs_14] for details.

- *OpenID Connect*: There are several variants that can be used to attack OpenID Connect. They are described in detail in [arXiv.1704.08539], Appendix A, and [arXiv.1508.04324v2], Section 6 ("Malicious Endpoints Attacks").

4.4.2. Countermeasures

In scenarios where an OAuth client interacts with multiple authorization servers, clients MUST prevent mix-up attacks.

Potential countermeasures:

- Configure authorization servers to return an AS identitifier ("iss") and the "client_id" for which a code or token was issued in the authorization response. This enables clients to compare this data to their own client id and the "iss" identifier of the AS it believed it sent the user agent to. This mitigation is discussed in detail in [I-D.ietf-oauth-mix-up-mitigation]. In OpenID Connect, if an ID token is returned in the authorization response, it carries client id and issuer. It can be used for this mitigation.

- As it can be seen in the preconditions of the attacks above, clients can prevent mix-up attack by (1) using AS-specific redirect URIs with exact redirect URI matching, (2) storing, for
each authorization request, the intended AS, and (3) comparing the intended AS with the actual redirect URI where the authorization response was received.

4.5. Authorization Code Injection

In such an attack, the adversary attempts to inject a stolen authorization code into a legitimate client on a device under his control. In the simplest case, the attacker would want to use the code in his own client. But there are situations where this might not be possible or intended. Examples are:

- 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 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 authorization or resource servers are limited to certain networks that the attacker is unable to access directly.

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. It performs a regular OAuth authorization process with the legitimate client on his device.
3. The attacker injects the stolen authorization code in the response of the authorization server to the legitimate client.
4. The client sends the code to the authorization server’s token endpoint, along with 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. If all checks succeed, the authorization server issues access and other tokens to the client, so now the attacker is able to impersonate the legitimate user.
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 malware pretending to be the legitimate client 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 a 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.

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 spec, since the tampered redirect URI is not considered. So any attempt to inject a code obtained using the "client_id" of a legitimate client or by utilizing the legitimate client on another device won’t 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 memorize or re-construct the correct redirect URI for the call to the tokens 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.

4.5.3. Proposed Countermeasures

There are multiple technical solutions to achieve this goal:

- **Nonce**: OpenID Connect’s existing "nonce" parameter can be used for the purpose of detecting authorization code injection attacks. 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. The main advantage of this option is that "nonce" is an existing feature used in the wild. On the other hand, leveraging "nonce" by the broader OAuth community would require AS and clients to adopt ID Tokens.

- **Code-bound State**: The "state" parameter as specified in [RFC6749] could be used similarly to what is described above. This would require to add a further parameter "state" to the code exchange token endpoint request. The authorization server would then compare the "state" value it associated with the code and the "state" value in the parameter. If those values do not match, it is considered an attack and the request fails. The advantage of this approach would be to utilize an existing OAuth parameter. But it would also mean to re-interpret the purpose of "state" and to extend the token endpoint request.

- **PKCE**: The PKCE parameter "code_challenge" along with the corresponding "code_verifier" as specified in [RFC7636] could be used in the same way as "nonce" or "state". In contrast to its original intention, the verifier check would fail although the client uses its correct verifier but the code is associated with a challenge, which does not match. PKCE is a deployed OAuth feature, even though it is used today to secure native apps only.
*Token Binding*: Token binding [I-D.ietf-oauth-token-binding] could also be used. In this case, the code would need to be bound to two legs, between user agent and AS and the user agent and the client. This requires further data (extension to response) to manifest binding id for particular code. Token binding is promising as a secure and convenient mechanism (due to its browser integration). As a challenge, it requires broad browser support and use with native apps is still under discussion.

*Per-instance client id/secret*: One could use per instance "client_id" and secrets and bind the code to the respective "client_id". Unfortunately, this does not fit into the web application programming model (would need to use per-user client IDs).

PKCE seems to be the most obvious solution for OAuth clients as it available and effectively used today for similar purposes for OAuth native apps whereas "nonce" is appropriate for OpenID Connect clients.

Note on pre-warmed secrets: An attacker can circumvent the countermeasures described above if he is able to create or capture the respective secret or code_challenge on a device under his control, which is then used in the victim’s authorization request.

Exact redirect URI matching of authorization requests can prevent the attacker from using the pre-warmed secret in the faked authorization transaction on the victim’s device.

Unfortunately, it does not work for all kinds of OAuth clients. It is effective for web and JS apps and for native apps with claimed URLs. Attacks on native apps using custom schemes or redirect URIs on localhost cannot be prevented this way, except if the AS enforces one-time use for PKCE verifier or "nonce" values.

### 4.6. Access Token Injection

In such an attack, the adversary attempts to inject a stolen access token into a legitimate client on a device under his 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 adversary starts an OAuth flow with the client 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 and will use the access token injected by the attacker.

4.6.1. Proposed 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. 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.

4.7.1. Proposed Countermeasures

Standard CSRF defenses should be used to protect the redirection endpoint, for example:

- *CSRF Tokens*: Use of CSRF tokens which are bound to the user agent and passed in the "state" parameter to the authorization server.

- *Origin Header*: The Origin header can be used to detect and prevent CSRF attacks. Since this feature, at the time of writing, is not consistently supported by all browsers, CSRF tokens should be used in addition to Origin header checking.

For more details see [owasp_csrf].

4.8. Access Token Leakage at the Resource Server

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

4.8.1. 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.

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

4.8.1.1. 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 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 return parameter "access_token_resource_server":


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_ubc] 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.

4.8.1.2. 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 which 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
3. The RS must implement the actual proof of possession check. This is typically done on the application level, it may utilize capabilities of the transport layer (e.g., TLS). Note: replay prevention is required as well!

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

- **OAuth Token Binding** ([I-D.ietf-oauth-token-binding]): In this approach, an access token is, via the so-called 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 than 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).

- **OAuth Mutual TLS** ([I-D.ietf-oauth-mtls]): 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.

- **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.
o *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 [I-D.ietf-oauth-mlts], 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.

Mutual TLS and OAuth Token Binding are built on top of TLS and this way continue the successful OAuth 2.0 philosophy to leverage TLS to secure OAuth wherever possible. Both mechanisms allow prevention of access token leakage in a fairly client developer friendly way.

There are some differences between both approaches: To start with, for OAuth Token Binding, all key material is automatically managed by the TLS stack whereas mTLS requires the developer to create and maintain the key pairs and respective certificates. Use of self-signed certificates, which is supported by the draft, significantly reduces the complexity of this task. Furthermore, OAuth Token Binding allows to use different key pairs for different resource servers, which is a privacy benefit. On the other hand, [I-D.ietf-oauth-mlts] only requires widely deployed TLS features, which means it might be easier to adopt in the short term.

Application level signing approaches, like [I-D.ietf-oauth-signed-http-request] and [I-D.sakimura-oauth-jpop] have been debated for a long time in the OAuth working group without a clear outcome.

As one advantage, application-level signing allows for end-to-end protection including non-repudiation even if the TLS connection is terminated between client and resource server. But deployment experiences have revealed challenges regarding robustness (e.g., reproduction of the signature base string including correct URL) as well as state management (e.g., replay prevention).

This document therefore recommends implementors to consider one of TLS-based approaches wherever possible.

4.8.1.3. Audience Restricted Access Tokens

An audience restriction essentially restricts the resource server a particular access token can be used at. The authorization server associates the access token with a certain resource server and every

resource server is obliged to verify for every request, whether the
access token sent with that request was meant to be used at the
particular resource server. If not, the resource server must refuse
to serve the respective request. In the general case, audience
restrictions limit the impact of a token leakage. In the case of a
counterfeit resource server, it may (as described see below) also
prevent abuse of the phished access token at the legitimate resource
server.

The audience can basically 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 needs to tell the authorization server, at which URL it
will use the access token it is requesting. It could use the
mechanism proposed [I-D.ietf-oauth-resource-indicators] or encode 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 legit
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 seems easy to use since it does not require any
crypto on the client side. But since every access token is bound to
a certain resource server, the client also needs to obtain different
RS-specific access tokens, if it wants to access several resource
services. [I-D.ietf-oauth-token-binding] has the same property,
since different token binding ids must be associated with the access
token. [I-D.ietf-oauth-mlts] 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.

4.8.2. Compromised Resource Server

An attacker may compromise a resource server in order to get access to its resources and other resources of the respective deployment. Such a compromise may range from partial access to the system, e.g., its logfiles, to full control of the respective server.

If the attacker was able to take over full control including shell access it will be able to circumvent all controls in place and access resources without access control. It will also get access to access tokens, which are sent to the compromised system and which potentially are valid for access to other resource servers as well. Even if the attacker "only" is able to access logfiles or databases of the server system, it may get access to valid access tokens.

Preventing server breaches by way of hardening and monitoring server systems is considered a standard operational procedure and therefore out of scope of this document. This section will focus on the impact of such breaches on OAuth-related parts of the ecosystem, which is the replay of captured access tokens on the compromised resource server and other resource servers of the respective deployment.

The following measures should be taken into account by implementors in order to cope with access token replay:

- The resource server must treat access tokens like any other credentials. It is considered good practice to not log them and not to store them in plain text.

- Sender-constrained access tokens as described in Section 4.8.1.2 will prevent the attacker from replaying the access tokens on other resource servers. Depending on the severity of the penetration, it will also prevent replay on the compromised system.

- Audience restriction as described in Section 4.8.1.3 may be used to prevent replay of captured access tokens on other resource servers.

4.9. Open Redirection

The following attacks can occur when an AS or client has an open redirector, i.e., a URL which causes an HTTP redirect to an attacker-controlled web site.
4.9.1. Authorization Server as Open Redirector

Attackers could try to utilize a user’s trust in the authorization server (and its URL in particular) for performing phishing attacks. [RFC6749], Section 4.1.2.1, already prevents open redirects by stating the AS MUST NOT automatically redirect the user agent in case of an invalid combination of client_id and redirect_uri.

However, as described in [I-D.ietf-oauth-closing-redirectors], an attacker could also utilize a correctly registered redirect URI to perform phishing attacks. It 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, to cause the AS to automatically 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 or not and SHOULD only automatically redirect the user agent, if it trusts the redirect URI. If not, it MAY inform the user that it is about to redirect her to the another site and rely on the user to decide or MAY just inform the user about the error.

4.9.2. Clients as Open Redirector

Client MUST NOT expose URLs which could be utilized as open redirector. Attackers may use an open redirector to produce URLs which appear to point to the client, which might trick users to trust the URL and follow it in her browser. Another abuse case is to produce URLs pointing to the client and utilize them to impersonate a client with an authorization server.

In order to prevent open redirection, clients should only expose such a function, if the target URLs are whitelisted or if the origin of a request can be authenticated.

4.10. 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". However, when the status code 307 is used...
for redirection, the user agent will send the form data (user credentials) via HTTP POST to the client since this status code does not require the user agent to rewrite the POST request to a GET request (and thereby dropping the form data in the POST request body). If the relying party is malicious, it can use the credentials to impersonate the user at the AS.

In the HTTP standard [RFC6749], 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".

4.11. TLS Terminating Reverse Proxies

A common deployment architecture for HTTP applications is to have the application server sitting behind a reverse proxy, which 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, which are relevant to OAuth, and give 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.

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 "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 would be able to get access to the internal network between proxy and application server, it could also try to circumvent security controls in place. It is therefore important to ensure the authenticity of the communicating entities. Furthermore, the communication link between reverse proxy and application server must therefore be protected against tapping and injection (including replay prevention).

4.12. Refresh Token Protection

Refresh tokens are a convenient and UX-friendly way to obtain new access tokens after the expiration of older 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.

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 (i.e., "client_id"),
- 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 draft gives recommendations beyond the scope of [RFC6749] and clarifications.

Authorization servers MUST determine based on their 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 legit client and reduce the impact of refresh token leakage.

Authorization server MUST utilize one of these methods to detect refresh token replay for public clients:

- *Sender-constrained refresh tokens:* the authorization server cryptographically binds the refresh token to a certain client instance by utilizing [I-D.ietf-oauth-token-binding] or [I-D.ietf-oauth-mtls].

- *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 can revoke the active refresh token. This stops the attack at the cost of forcing the legit client to obtain a fresh authorization grant. Implementation note: refresh tokens belonging to the same grant may share a common id. If any of those refresh tokens is used at the authorization server, the authorization server uses this common id to look up the currently active refresh token and can revoke it.

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).

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 Matake, Tomek Stojekci, Dominick Baier, Neil Madden, William Dennis, Dick Hardt, Petteri Stenius, Annabelle Richard Backman, Aaron Parecki, George Fletscher, and Brian Campbell 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. References

8.1. Normative References

[oauth-v2-form-post-response-mode]


8.2. Informative References

[arXiv.1508.04324v2]

[arXiv.1601.01229]

[arXiv.1704.08539]

[bug.chromium]

[fb_fragments]

[I-D.bradley-oauth-jwt-encoded-state]
[I-D.ietf-oauth-closing-redirectors]

[I-D.ietf-oauth-jwsreq]

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

[I-D.ietf-oauth-mtls]

[I-D.ietf-oauth-pop-key-distribution]

[I-D.ietf-oauth-resource-indicators]

[I-D.ietf-oauth-signed-http-request]

[I-D.ietf-oauth-token-binding]

[I-D.sakimura-oauth-jpop]


Appendix A. Document History

[[ To be removed from the final specification ]]

-12

o Added updated attacker model

-11

o Adapted section 2.1.2 to outcome of consensus call

o more text on refresh token inactivity and implementation note on refresh token replay detection via refresh token rotation

-10

o incorporated feedback by Joseph Heenan

o changed occurrences of SHALL to MUST

o 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

o added requirement to authenticate clients during code exchange (PKCE or client credential) to 2.1.1.

o added section on refresh tokens

o editorial enhancements to 2.1.2 based on feedback

-09
o changed text to recommend not to use implicit but code
o added section on access token injection
o reworked sections 3.1 through 3.3 to be more specific on implicit grant issues
-08
o added recommendations re implicit and token injection
o uppercased key words in Section 2 according to RFC 2119
-07
o incorporated findings of Doug McDorman
o added section on HTTP status codes for redirects
o added new section on access token privilege restriction based on comments from Johan Peeters
-06
o reworked section 3.8.1
o incorporated Phil Hunt’s feedback
o reworked section on mix-up
o extended section on code leakage via referrer header to also cover state leakage
o added Daniel Fett as author
o replaced text intended to inform WG discussion by recommendations to implementors
o modified example URLs to conform to RFC 2606
-05
o Completed sections on code leakage via referrer header, attacks in browser, mix-up, and CSRF
o Reworked Code Injection Section
o 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
Facebook
Email: isciurus@fb.com

Daniel Fett
yes.com
Email: mail@danielfett.de
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.
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
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_Cbl7LWlt6Xjp3DbjiDjavGF1KP2HV_2JSE42VzmKOVVV8m7eqA
grant_type=authorization_code&code=4bwcZesc7Xacc330ltc66Wxk8EAfP9j2&code_verifier=2x6_yIS390-8V7jaT9wj.8p9nKmY Cf.V-rD904r_l&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 "AgBBQto7hHRR0Y5nkOWqc9KNfwW95dEFmSI_tCZ_Cbl7LWlt6Xjp3DbjiDjavGF1KP2HV_2JSE42VzmKOVVV8m7eqA".

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-cache, no-store
{
"access_token":"EdRs7qMrLb167Z9fV2dcwoLTC",
"refresh_token":"ACC1ZEIQTjW9arT9GOJGgd7QNwqOMmUYfsJTiv8his4",
"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-84Ukw42qfd7uWOTFrAjd96WgbdaPDX2kno0iAE") and signature in the Token Binding Message are different than in the initial request.

```plaintext
POST /as/token.oauth2 HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded
Sec-Token-Binding: AIkAAgBBQGto7hHR0Y5nkOWq9KNfwW95dEFmSI_tCZ_Cbl7LWlt6Xjp3Dbj1DjavGF1KP2HV_2JSE42VzmKOVVVe7eqAQCPgbaG_YRf27qOraL0UT4fsKKjL6PukuGT00qzamoAXxOq7m_id7O3mLpnb_sM7kwSxLi7iNHzzDgCAkPt3IlHwAAAA
refresh_token=ACClZEIQTj9arT9GOJGGd7QNwqOMmUYfsJtiv8his4
&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 "jI5UAyjs5XCPISUGQIwgcSrO1VIWq4fhlVIFQTQ4nLxic".

GET /as/authorization.oauth2?response_type=token
&client_id=example-client-id&state=rM8pZxG1c3gKy6rEbsD8s
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Eorg%2Fcob HTTP/1.1
Host: server.example.com
Sec-Token-Binding: ARIAAgBBQIEE8mSMtDy2dj9EEBdXaqT9W3Rq1NS--jW8ebPoF
6FyL0jIfATVE55zlircgOTZmEgIxeIrC3DsGegwjs4bhw14AQQKd1AXXFMyQkZegC
w1bT1gX3F9HtG-1jxFU_pi16ezka7qVRCpSFOQlfsQ15sxM6yfS5SCJ1BdtrIL7PX
j__FAAAECAMFA1BNUnP3te5rwlEwiejEzOopasmC5PE1Wc7k25n1lsqtj1ciUp
5vQ30LLUCyM_a2BYTUPKtd5EdS-PalT4t6ABADgeizRa5NkTMu4zOdC-R4cLNWVV
081Lu2Psko-UJLR_XAH4Q0H7-mo_nQR1zBN78nYMKpHs8L3zWKRVYxEgAA

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=eyJhbGciOiJFUzI[

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.

```
{ ...
  "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 "4jTc5e1QpcqPTZ5l6jsh6RPls18IFKdwwPvasYjn1-E".

POST /as/token.oauth2 HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded
Sec-Token-Binding: ARIAAgBBQJFXJir2w4gbj7grBx9uYWN1s9V50-PW4ZijegQ0JUM--bgGNT6DizxUK-m5n3dQUlKeH7ybn6wb1C5dGoYV_IAAQDDFToPrHt412ppq7u_SEMP-E-KimAB-HeWw1zZgZAg9QOSWiJClFiCkgtr1RrA2-jajVoB8o51DTGxQydWyKAACEAFuC1GLYU83rgTQEau1ogVdWw0fDsdXzIyT_41IcldsMwjFkJacIBJFGuYccvncAk_duFi3QKFENwxwq1-H9ABAMcU7IjJOUA4IyE6YoEcfz9BMPQqw
M5M6hW4RZNQd58fSTCCs1OE_NmNC19JXy4NkdkE2BxqVZGPrt0y8QZ_bmAwAA
refresh_token=gZR_ZI8EAhLgWR-gWbImbfgZRBZi_8EAhLgWRgWbImbf&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": "eyJhbGciOiJFUzI1NiIsImtp...
"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.
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 "7LsNP3BTlaHHdXdk6meEWjtSkIPVLCb7YS6iHp-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 eyJhbGciOiJFUzI1NiIsI...omitted...]1cs29j5c3
Sec-Token-Binding: AIkAAgBBQLgtRpWFPN66khhxGrtarKrzrMtHw7HV8yNyK_-MdRXJXbDMYxZCWcmCA5RRrHmHHI5wmpF3bhYt0ChRDbMapfh_QAQN1He3Ftj4Wa_s_fzZVns4safj6aBoMSQW6rL1s19IIvHze7LrGjKyCFPTkXjaJebxp-TLPF2Cc0JTqTY5_0MBAAAA

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:

```json
{
    "iss": "https://server.example.com",
    "aud": "https://resource.example.org",
    "sub": "brian@example.com",
    "iat": 1467324320,
    "exp": 1467324920,
    "cnf": {
        "tbh": "7NRBu9iDdJlYCT0qyeYuLxXv0blEA-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

5.1.2. 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 "pNVKtPuQFvy1NYn00QowWrQKoeMkeX9H32hVuU71Bs".

```plaintext
POST /as/token.oauth2 HTTP/1.1
Host: server.example.com
Sec-Token-Binding: AIkAAgBBQEOO9GRFP-LM0hoWw6-2i318BsuuUum5AL8btlsz
                 lr1EFfp5DMXMNW0i8Wjc1Xr2DKJnI4xnuGsE6GywQd9Rb0OQJDb3xyo9Pf8xj8M6Y
                 jlt-6oaxgDkyoBoTkyrnNbLc8tJQ0JtXomKzBbj5qPtHDduXc6xz_1zvNpxSPxi42
                 8m7wAAA
grant_type=authorization_code&code=mJAReTWKK7zI3oHUNdQo3PeNqNqzKGp6
&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
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-1YwZGmnVHyReN5vd2ChcsRBN69Ue4cI".

GET /as/authorization.oauth2?response_type=code
&client_id=example-web-client-id&state=dryo8YFpWacbUPjhBf4Nvt51
&redirect_uri=https%3A%2F%2Fexample%2Eorg%2Fcb
&code_challenge=referred_tb
&code_challenge_method=referred_tb HTTP/1.1
Host: server.example.com
Sec-Token-Binding: ARIAAgBBQB-XOPf5ePlf7ikATiAFEGOS5031PmR6kymzdWw
HCx10njxc3D0E_0VfBNqrIQzzFf7tWby2ZfyE6XpwTsAQBqFX78vMQgDX_F
d_b2d1HyH1MkFz8iMvBY_reh9800uJaFz1IB7Pc9nZ1lj58LoG5QhmQo19NYktKZ
RXrYAAAECaEAdUftfQADkn1uDbQnvrJEx6oQs38L92gy-kO-q1yAdLoD1ke2h53
h5ikWzIP98iRj_unedkNkAMyg9e2mY4Gp7WwBAeDUOwaSNzle6gKohwN4SAZ5eNy
45Mh8VI4woL1BipLogrJRokK6dxFkWgHRMuBROcLGUj5PiOoxybQH_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_tajt8ij3tV6cwyy2KH-i8BdEMYXcNn0".
GET /cb?state=dryo8YPwacbUPjhBf4Nvt51&code=jwD3oOa5cQvvLc81bwc4CMw
Host: client.example.org
Sec-Token-Binding: AIkAAgBBQHVBU530AA5J9bg20J7yRJOqELN_C_doL_i_jvqW
GnS6AyCnfo6d4UoisCD_FIkY_7p3nZD2ADBMoPZtmOBqe1sAQEwgc9Zpg7QFCDDbib
6G1Zk13Mh32KNfLefLJc1vR1x8170MfPLZHF2Woxh6rEtmgBcAAbubEbTz7muN1
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 b3JnLmV4YW1wbGUvY2xpZW50OmlldGV4YXlv
grant_type=authorization_code&code=jwD3oOa5cQvvLc81bwc4CMw
&redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
&client_id=example-web-client-id
&code_verifier=AgBBQHVBU530AA5J9bg20J7yRJOqELN_C_doL_i_jvqW
GnS6AyCnfo6d4UoisCD_FIkY_7p3nZD2ADBMoPZtmOBqe1s

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 as for authentication.

- 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 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 indicated 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 of 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

- Client Metadata Name: "client_access_token_token_binding_supported"
- Client Metadata Description: Boolean value specifying whether the client supports Token Binding of access tokens
- Change Controller: IESG
- Specification Document(s): Section 4.1 of [[[ this specification ]]]

- Client Metadata Name: "client_refresh_token_token_binding_supported"
- Client Metadata Description: Boolean value specifying whether the client supports Token Binding of refresh tokens
- Change Controller: IESG
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

- Metadata Name: "as_access_token_token_binding_supported"
- Metadata Description: Boolean value specifying whether the authorization server supports Token Binding of access tokens
- Change Controller: IESG
- Specification Document(s): Section 4.2 of [[ this specification ]]

- Metadata Name: "as_refresh_token_token_binding_supported"
- Metadata Description: Boolean value specifying whether the authorization server supports Token Binding of refresh tokens
- Change Controller: IESG
- 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

- Code Challenge Method Parameter Name: TB-S256
- Change controller: IESG
- Specification document(s): Section 5.1.1 of [[ this specification ]]

- Code Challenge Method Parameter Name: referred_tb
- Change controller: IESG
- 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

- Token Endpoint Authentication Method Name: "client_secret_token_bound_jwt"
- Change Controller: IESG
- Specification Document(s): Section 6 of [[ this specification ]]

- Token Endpoint Authentication Method Name: "private_key_token_bound_jwt"
- Change Controller: IESG
- 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

- URN: urn:ietf:params:oauth:grant-type:jwt-token-bound
  - Common Name: Token Bound JWT Grant Type for OAuth 2.0
  - Change controller: IESG
  - Specification Document: Section 6 of [[ this specification ]]

  - Common Name: Token Bound JWT for OAuth 2.0 Client Authentication
  - Change controller: IESG
  - 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>.


- [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>.

Jones, et al. Expires April 22, 2019


11.2. Informative References


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
-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.
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 2.0 Token Exchange
draft-ietf-oauth-token-exchange-16

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 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. Delegation vs. Impersonation Semantics ................. 4
   1.2. Requirements Notation and Conventions .................. 5
   1.3. Terminology .............................................. 6
2. Token Exchange Request and Response .......................... 6
   2.1. Request ................................................ 6
       2.1.1. Relationship Between Resource, Audience and Scope . 8
   2.2. Response ............................................... 9
       2.2.1. Successful Response ................................ 9
       2.2.2. Error Response ................................... 10
   2.3. Example Token Exchange ................................ 11
3. Token Type Identifiers ........................................ 13
4. JSON Web Token Claims and Introspection Response Parameters . 14
   4.1. "act" (Actor) Claim .................................... 14
   4.2. "scope" (Scopes) Claim .................................. 16
   4.3. "client_id" (Client Identifier) Claim ................. 17
   4.4. "may_act" (May Act For) Claim ......................... 17
5. Security Considerations ....................................... 18
6. Privacy Considerations ........................................ 19
7. IANA Considerations .......................................... 19
   7.1. OAuth URI Registration ................................... 19
       7.1.1. Registry Contents .................................. 19
   7.2. OAuth Parameters Registration ........................... 20
       7.2.1. Registry Contents .................................. 20
   7.3. OAuth Access Token Type Registration ................... 21
       7.3.1. Registry Contents .................................. 21
   7.4. JSON Web Token Claims Registration ..................... 21
       7.4.1. Registry Contents .................................. 21
   7.5. OAuth Token Introspection Response Registration ....... 22
       7.5.1. Registry Contents .................................. 22
   7.6. OAuth Extensions Error Registration .................... 22
       7.6.1. Registry Contents .................................. 22
8. References ................................................... 22
   8.1. Normative References .................................... 22
   8.2. Informative References .................................. 23
Appendix A. Additional Token Exchange Examples .................. 24
   A.1. Impersonation Token Exchange Example ................... 24
       A.1.1. Token Exchange Request ............................ 24
       A.1.2. Subject Token Claims .............................. 25
       A.1.3. Token Exchange Response ........................... 25
       A.1.4. Issued Token Claims ................................ 26
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 and issuing security tokens, 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, which 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

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 in so far 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.
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 OAuth 2.0 [RFC6749].

Client authentication to the authorization server is done using the normal mechanisms provided by OAuth 2.0. Section 2.3.1 of The OAuth 2.0 Authorization Framework [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 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.

The client makes a token exchange request to the token endpoint with an extension grant type by including the following parameters using the "application/x-www-form-urlencoded" format with a character encoding of UTF-8 in the HTTP request entity-body:

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

resource  
OPTIONAL. Indicates the location of 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 the location 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.

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 rather than a location. 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], or a URI are examples of things that might be used as "audience" parameter values. 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 locations.

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 this 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 validate the subject token as appropriate for the indicated token type and, if the actor token is present, also validate it according to its 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 validity of the subject token or actor token have no impact on the validity of the issued token after the exchange has occurred.

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 [RFC7159], 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 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 typically used with this meaning, 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 all the target services 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" and/or
"error_uri" parameters.

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
token exchange in order to trade an access token that it received in
a protected resource request for a token that it will use to call to
a backend service (extra line breaks and indentation in the examples
are for display purposes only).

The resource server receives the following request containing an
OAuth access token in the Authorization request header, as specified
in Section 2.1 of [RFC6750].

GET /resource HTTP/1.1
Host: frontend.example.com
Authorization: Bearer accVjcJyb4BWCxGsndESCJQbdFMogUC5PbRDqceLTC

Figure 1: Protected Resource Request

The resource server assumes the role of the client for the token
exchange and the access token from the request above 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 cnMwODpsb25nLXNlY3VyZS1yW5kb20tc2VjcmV0
Content-Type: application/x-www-form-urlencoded

grant_type=urn%3Aiietf%3Aparams%3Aoauth%3Agrant-type%3Atoken-exchange
&resource=https%3A%2F%2Fbackend.example.com%2Fapi
&subject_token=accVkjcJyb4BWCxGsndESCJQbdFMogUC5PbRDqceLTC
&subject_token_type=
    urn%3Aiietf%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 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": "eyJhbGciOiJFUzI1NiIsIiQiLCJpc3MiOiJodHRwczovL2FzLmV4YW1wbGUUy29tIiwidXNlcl9pZCI6IjllciJ9.eyJhdWQiOiJo
   dHRwczoL2JhY2t1bmQuZXhhbXBsZS5jb20ic2VjcmV0
   4YW1wbGUUy29tIiwidXNlcl9pZCI6IjllciJ9.eyJhdWQiOiJo
dHRwczoL2JhY2t1bmQuZXhhbXBsZS5jb20ic2VjcmV0
   4YW1wbGUUy29tIiwidXNlcl9pZCI6IjllciJ9.eyJhdWQiOiJo
dHRwczoL2JhY2t1bmQuZXhhbXBsZS5jb20ic2VjcmV0
   4YW1wbGUUy29tIiwidXNlcl9pZCI6IjllciJ9.eyJhdWQiOiJo
   dHRwczoL2JhY2t1bmQuZXhhbXBsZS5jb20ic2VjcmV0
   4YW1wbGUUy29tIiwidXNlcl9pZCI6IjllciJ9.eyJhdWQiOiJo
dHRwczoL2JhY2t1bmQuZXhhbXBsZS5jb20ic2VjcmV0
   4YW1wbGUUy29tIiwidXNlcl9pZCI6IjllciJ9.eyJhdWQiOiJo
dHRwczoL2JhY2t1bmQuZXhhbXBsZS5jb20ic2VjcmV0
   4YW1wbGUUy29tIiwidXNlcl9pZCI6IjllciJ9.eyJhdWQiOiJo
dHRwczoL2JhY2t1bmQuZXhhbXBsZS5jb20ic2VjcmV0
   4YW1wbGUUy29tIiwidXNlcl9pZCI6IjllciJ9.eyJhdWQiOiJo
dHRwczoL2JhY2t1bmQuZXhhbXBsZS5jb20ic2VjcmV0
   4YW1wbGUUy29tIiwidXNlcl9pZCI6IjllciJ9.eyJhdWQiOiJo
dHRwczoL2JhY2t1bmQuZXhhbXBsZS5jb20ic2VjcmV0
   4YW1wbGUUy29tIiwidXNlcl9pZCI6IjllciJ9.eyJhdWQiOiJo
dHRwczoL2JhY2t1bmQuZXhhbXBsZS5jb20ic2VjcmV0
   4YW1wbGUUy29tIiwidXNlcl9pZCI6IjllciJ9.eyJhdWQiOiJo
dHRwczoL2JhY2t1bmQuZXhhbXBsZS5jb20ic2VjcmV0
   4YW1wbGUUy29tIiwidXNlcl9pZCI6IjllciJ9.eyJhdWQiOiJo
dHRwczoL2JhY2t1bmQuZXhhbXBsZS5jb20ic2VjcmV0
   4YW1wbGUUy29tIiwidXNlcl9pZCI6IjllciJ9.eyJhdWQiOiJo
dHRwczoL2JhY2t1bmQuZXhhbXBsZS5jb20ic2VjcmV0
   4YW1wbGUUy29tIiwidXNlcl9pZCI6IjllciJ9.eyJhdWQiOiJo
dHRwczoL2JhY2t1bmQuZXhhbXBsZS5jb20ic2VjcmV0
   4YW1wbGUUy29tIiwic2NvcGUiOiJhcGkifQ.K4Ik-igqOKl_4C
   nBu4dG3-gGUofbgv-rJhgXVDCOJwW_MHgVwddhgVLLQf_bm3xlpQM6wHrLbMaZC4
   LicsQC23g",
   "issued_token_type":
     "urn%3Aiietf%3Aparams%3Aoauth: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.

GET /api HTTP/1.1
Host: backend.example.com
Authorization: Bearer eyJhbGciOiJFUzI1NiIsImtpZCI6IjllciJ9.eyJhbGciOiJFUzI1NiIsImtpZCI6IjllciJ9.eyJhdWQiOiJodHRwczovL2JhY2tlbmQvZXRhbnBseS5jb20iLCJpc3MiOiJodHRwczovL2FzLmV4YW1wbGUuY29tIiwiZXhwIjoxNDQxOTE3NTkzLCJpYXQiOjE0NDE5MTc1MzMsInN1YiI6ImJjQGV4YW1wbGUuY29tIiwic2NwIjpbImFwaSJdfQ.MXgnpvPMo0nhcePwnQbunDgw_pDyCFA-Saob16gyLAdyPbaALFuA0yFc4XTwPeNHV_LGmXk1STpz0yC7h1SQ

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]).

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 must not be used.

The following example 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.

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 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.

```json
{
    "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 OAuth 2.0 [RFC6749].
The following example illustrates the "scope" claim within a JWT Claims Set.

```json
{
    "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 illustrates the "client_id" claim within a JWT Claims Set indicating an OAuth 2.0 client with "s6BhdRkqt3" as its identifier.

```json
{
    "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" (May Act For) 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 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 should not be used.

The following example 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: May Act For 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

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 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, should 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 should 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

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

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

  - Common Name: Token type URI for an OAuth 2.0 refresh token
  - Change controller: IESG
  - Specification Document: Section 3 of [[this specification]]

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

- URN: urn:ietf:params:oauth:token-type:saml1
  - Common Name: Token type URI for a base64url-encoded SAML 1.1 assertion
  - Change Controller: IESG
  - 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

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

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

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

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

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

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

- Parameter name: actor_token_type
  - Parameter usage location: token request
  - Change controller: IESG
  - Specification document(s): Section 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

- Type name: N_A
- Additional Token Endpoint Response Parameters: (none)
- HTTP Authentication Scheme(s): (none)
- Change controller: IESG
- 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

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

- Claim Name: "scope"
- Claim Description: Scope Values
- Change Controller: IESG
- Specification Document(s): Section 4.2 of [[ this specification ]]

- Claim Name: "client_id"
- Claim Description: Client Identifier
- Change Controller: IESG
- Specification Document(s): Section 4.3 of [[ this specification ]]

- Claim Name: "may_act"
- Claim Description: May Act For
- Change Controller: IESG
- 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

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

- Claim Name: "may_act"
  - Claim Description: May Act For
  - Change Controller: IESG
  - 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

- Error Name: "invalid_target"
  - Error Usage Location: token error response
  - Related Protocol Extension: OAuth 2.0 Token Exchange
  - Change Controller: IETF
  - 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>.

8.2. Informative References


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. 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=eyJhbGciOiJFUzI1NiIsImtpZCI6IjE2In0.eyJhdWQiOiJodHRwczovL2FzLmV4YW1wbGUuY29tIiwiaXNzIjoiaHR0cHM6Ly9vcmlnaW5hbCI6N1ZXIuZXhhbXBsb2dSdW50eXBlIiwia2V5IjoiYm9vYmFzdCIsImF1ZCI6IjEzNjI5MTUyNWU1MjQ0ZmNiY2YyOTNkZDdjODIyZjEiLCJpYXQiOjE0NDE5MTA2MDAsImV4cCI6MTQ0MTk3NTQzM30.eyJhdWQiOiJodHRwczovL2FzLmV4YW1wbGUuY29tIiwiaXNzIjoiaHR0cHM6Ly9vcmlnaW5hbCI6N1ZXIuZXhhbXBsb2dSdW50eXBlIiwia2V5IjoiYm9vYmFzdCIsImF1ZCI6IjEzNjI5MTUyNWU1MjQ0ZmNiY2YyOTNkZDdjODIyZjEiLCJpYXQiOjE0NDE5MTA2MDAsImV4cCI6MTQ0MTk3NTQzM30&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 ("bc@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":"bc@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

```json
{
  "access_token": "eyJhbGciOiJFUzI1NiIsImtpZCI6IjcyIn0.eyJhdWQiOiJ1cm5hbXBlYXgsI forsMjMzNjM2IiwiaWF0IjoxMjM2MjIwMzQ1LCJleHAiOjE2MzY3OTUwNzB9.5Jd8z7l4X31W4D7
rI9HqQp_23FbQ",
  "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.

```json
{
  "aud": "urn:example:cooperation-context",
  "iss": "https://as.example.com",
  "exp": 1441913610,
  "sub": "bc@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:ietf:params:oauth:grant-type:token-exchange
&auidence=urn%3Aexample%3Acooperation-context
&subject_token=eyJhbGciOiJFUzI1NiIsImtpZCI6IjE2In0.eyJhdWQiOiJodHRwcizovL2FzLmV4YW1wbGUuY29tIiwiaXNzIjoiaHR0cHM6Ly9vcm1naW5hbClpc3N1ZXIuZXhhbXBsZS5uZXQidmlldy1zdXJ2ZXItY2F0aW9uc19kZXZpY2c6MjI2IiwiaWQiOiIiLCJleHAiOjE0NjgwMTQ1NjMsImF1ZCI6IjE2In0.4rPRSWihQbpMiQmgAaAqRZjz5d4C3T5DGlhX6ZnC5Rb5Jk8nX2w
&subject_token_type=urn%3Aietf%3Aparams%3Aoauth%3Atoken-type%3Ajwt
&actor_token=eyJhbGciOiJFUzI1NiIsImtpZCI6IjE2In0.eyJhdWQiOiJodHRwcizovL2FzLmV4YW1wbGUuY29tIiwiaXNzIjoiaHR0cHM6Ly9vcm1naW5hbClpc3N1ZXIuZXhhbXBsZS5uZXQidmlldy1zdXJ2ZXItY2F0aW9uc19kZXZpY2c6MjI2IiwiaWQiOiIiLCJleHAiOjE0NjgwMTQ1NjMsImF1ZCI6IjE2In0.7YQ-3zPfhUvzje5oqw8COCvN5uF6NskK9CVV6cAof4Km-tKfi0WcZoUuDL2tEs6tqPlcB1MjizEjm3yBg
&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.

```json
{
    "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.

```json
{
    "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":"eyJhbGciOiJFUzI1NiIsImtpZCI6IjcyIn0.eyJhdWQiOiJ1cm46ZXhhbXBsb29wb250ZXRhOlNlc2NvcmU6NTIiLCJhbGciOiJIUl0sImNvbSIsIiwiZXN1bWVudF90b2tlbiI6eyJ0b2tlbiI6Ikxzc2lkIn0.3paKl9UySKYB5ng6_cUtq108RC_y7Mea7IwEXTyY3Zf9-G1EKCFe5fW3HohwX-M5Z49WpcbBi1AzaQnTf",
    "issued_token_type":"urn:ietf:params:oauth:token-type:jwt",
    "token_type":null,
    "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.
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, and Benjamin Kaduk serving as Security Area Directors. The following individuals contributed ideas, feedback, and wording to this specification:


Appendix C. Document History

-16

-15

Fixed typo and added an AD to Acknowledgements.
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_U5uk3k6zpYF-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.
-08

-07

-06

-05

-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 6749 registration requests for request/response parameters.

o Removed the Implementation Considerations and the requirement to support JRTs.

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 JWSs 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