OAuth Working Group D. Hardt
Internet-Draft Amazon
Intended status: Standards Track B. Campbell
Expires: December 14, 2018 Ping Identity
N. Sakimura
NRI
June 12, 2018

Distributed OAuth
draft-hardt-oauth-distributed-01

Abstract

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

Status of This Memo

Copyright Notice

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.

```
    +--------+                               +---------------+
    |        |--(A)-- Discovery Request ---->|   Resource    |
    |        |                               |    Server     |
    |        |<-(B)-- Discovery Response ----|               |
    |        |                               +---------------+
    |        |
    |        |  (client obtains authorization grant)
    |        |
    |        |                               +---------------+
    |        |--(C)- Authorization Request ->| Authorization |
    | Client |                               |     Server    |
    |        |<-(D)----- Access Token -------|               |
    |        |                               +---------------+
    |        |
    |        |                               +---------------+
    |        |--(E)----- Access Token ------>|    Resource   |
    |        |                               |     Server    |
    |        |<-(F)--- Protected Resource ---|               |
    +--------+                               +---------------+
```

Figure 1: Abstract Protocol Flow

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

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

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

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

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

2. Authorization Server Discovery

Figure 1, step (A)

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

Figure 1, step (B)

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

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

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

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

3. Authorization Grant

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

4. Access Token Request

Figure 1, step (C)

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

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

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

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

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

Figure 1, step (D)

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

5. Accessing Protected Resource

Figure 1, step (E)

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

Figure 1, step (F)

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

6. Security Considerations

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

6.1. Access Token Reuse

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

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

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

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

6.2. Resource Server Impersonation

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

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

6.3. Malicious Issuer

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

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

7. IANA Considerations

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

- 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

[MTLS] Jones, M., Campbell, B., Bradley, J., and W. Denniss, "OAuth 2.0 Mutual TLS Client Authentication and Certificate Bound Access Tokens", June 2018, <https://datatracker.ietf.org/doc/draft-ietf-oauth-mtls/>.

[OASM] Jones, M., Campbell, B., and J. Bradley, "OAuth 2.0 Authorization Server Metadata", June 2018, <https://datatracker.ietf.org/doc/draft-ietf-oauth-discovery/>.

[OATB] Campbell, B., Bradley, J., Sakimora, N., and T. Lodderstedt, "OAuth 2.0 Token Binding", June 2018, <https://datatracker.ietf.org/doc/draft-ietf-oauth-token-binding/>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC5988] Nottingham, M., "Web Linking", RFC 5988, DOI 10.17487/RFC5988, October 2010, <https://www.rfc-editor.org/info/rfc5988>.

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

[RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, <https://www.rfc-editor.org/info/rfc6750>.

[RFC7519] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <https://www.rfc-editor.org/info/rfc7519>.

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

[RFC8288] Nottingham, M., "Web Linking", RFC 8288, DOI 10.17487/RFC8288, October 2017, <https://www.rfc-editor.org/info/rfc8288>.

Appendix A. Document History

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

- 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

- add client restriction to mitigate access token reuse

- 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

Network Working Group
Internet-Draft
Intended status: Informational
Expires: February 3, 2019

O. Levi Hevroni
Soluto by Asurion
August 02, 2018

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

Copyright Notice

Table of Contents

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 therefor 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:

- o 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.

- o 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):

```
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:

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

After rolling, this will be the payload:

```
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:

- o The key correspond to this client, received on the registration request

- o 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:

- o 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]

- o 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]

- o 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 allows 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 wat, 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

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC2289] Haller, N., Metz, C., Nesser, P., and M. Straw, "A One-Time Password System", STD 61, RFC 2289, DOI 10.17487/RFC2289, February 1998, <https://www.rfc-editor.org/info/rfc2289>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <https://www.rfc-editor.org/info/rfc4086>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <https://www.rfc-editor.org/info/rfc5246>.

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

[RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <https://www.rfc-editor.org/info/rfc7515>.

[RFC7519] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <https://www.rfc-editor.org/info/rfc7519>.

[RFC7521] Campbell, B., Mortimore, C., Jones, M., and Y. Goland, "Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants", RFC 7521, DOI 10.17487/RFC7521, May 2015, <https://www.rfc-editor.org/info/rfc7521>.

[RFC7523] Jones, M., Campbell, B., and C. Mortimore, "JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants", RFC 7523, DOI 10.17487/RFC7523, May 2015, <https://www.rfc-editor.org/info/rfc7523>.

7.2. URIs

[1] https://github.com/Soluto/oauth-seamless-flow

[2] https://www.owasp.org/index.php/Certificate_and_Public_Key_Pinning

[3] https://www.apple.com/business/docs/iOS_Security_Guide.pdf

[4] https://developer.android.com/training/articles/security-tips.html#UserData

[5] https://developer.apple.com/documentation/security/keychain_services/keychains

[6] https://developer.android.com/training/articles/keystore.html

[7] https://doridori.github.io/android-security-the-forgetful-keystore/#sthash.CgPjGF4h.dpbs

Author's Address

Omer Levi Hevroni
Soluto by Asurion

Email: omerlh@gmail.com

Open Authentication Protocol T. Lodderstedt, Ed.
Internet-Draft yes.com AG
Intended status: Standards Track V. Dzhuvinov
Expires: April 21, 2021 Connect2id Ltd.
October 18, 2020

JWT Response for OAuth Token Introspection
draft-ietf-oauth-jwt-introspection-response-10

Abstract

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

Status of This Memo

Copyright Notice

Table of Contents

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 enables deployments to implement opaque 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 token introspection response for an access token, including cases where the authorization server assumes liability for the content of the token introspection response. 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 [RFC7519] as the introspection response. This specification extends the token introspection endpoint with the capability to return responses as JWTs.

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.

3. Resource Server Management

The authorization server (AS) and the resource server (RS) maintain a strong two-way trust relationship. The resource server relies on the authorization server to obtain authorization, user and other data as input to its access control decisions and service delivery. The authorization server relies on the resource server to handle the provided data appropriately.

In the context of this specification, the Token Introspection Endpoint is used to convey such security data and potentially also privacy sensitive data related to an access token.

In order to process the introspection requests in a secure and privacy-preserving manner, the authorization server MUST be able to identify, authenticate and authorize resource servers.

To support encrypted token introspection response JWTs, the authorization server MUST also be provided with the respective resource server encryption keys and algorithms.

The authorization server MUST be able to determine whether an RS is the audience for a particular access token and what data it is entitled to receive, otherwise the RS is not authorized to obtain data for the access token. The AS has the discretion how to fulfil this requirement. The AS could, for example, maintain a mapping between scopes values and resource servers.

The requirements given above imply that the authorization server maintains credentials and other configuration data for each RS.

One way is by utilizing dynamic client registration [RFC7591] and treating every RS as an OAuth client. In this case, the authorization server is assumed to at least maintain "client_id" and "token_endpoint_auth_method" with complementary authentication method metadata, such as "jwks" or "client_secret". In cases where the AS needs to acquire consent to transmit data to a RS, the following client metadata fields are recommended: "client_name", "client_uri", "contacts", "tos_uri", "policy_uri".

The AS MUST restrict the use of client credentials by a RS to the calls it requires, e.g. the AS MAY restrict such a client to call the token introspection endpoint only. How the AS implements this restriction is beyond the scope of this specification.

This specification further introduces client metadata to manage the configuration options required to sign and encrypt token introspection response JWTs.

4. Requesting a JWT Response

A resource server requests a JWT introspection response by including an "Accept" HTTP header "application/token-introspection+jwt" in the introspection request.

The AS SHOULD authenticate the caller at the token introspection endpoint. Authentication can utilize client authentication methods or a separate access token issued to the resource server. Whether a resource server is required to authenticate is determined by the respective RS-specific policy at the AS.

The following is a non-normative example request with client authentication:

```
POST /introspect HTTP/1.1
Host: as.example.com
Accept: application/token-introspection+jwt
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded

token=2YotnFZFEjr1zCsicMWpAA
```

5. JWT Response

The introspection endpoint responds with a JWT, setting the "Content-Type" HTTP header to "application/token-introspection+jwt" and the JWT "typ" ("type") header to "token-introspection+jwt".

The JWT MUST include the following top-level claims:

iss MUST be set to the issuer URL of the authorization server.

aud MUST identify the resource server receiving the token introspection response.

iat MUST be set to the time when the introspection response was created by the authorization server.

token_introspection A JSON object containing the members of the token introspection response as specified in [RFC7662], section 2.2. The separation of the introspection response members into a dedicated containing JWT claim is intended to prevent conflict and confusion with top-level JWT claims that may bear the same name.

If the access token is invalid, expired, revoked, or is not intended for the calling resource server (audience), the authorization server MUST set the value of the "active" member in the "token_introspection" claim to "false" and other members MUST NOT be included. Otherwise, the "active" member is set to "true".

If possible, the AS MUST narrow down the "scope" value to the scopes relevant to the particular RS.

As specified in section 2.2. of [RFC7662], specific implementations MAY extend the token introspection response with service-specific claims. In the context of this specification, such claims will be added as top-level members of the "token_introspection" claim. Response names intended to be used across domains MUST be registered in the OAuth Token Introspection Response registry [IANA.OAuth.Token.Introspection] defined by [RFC7662]. In addition, claims from the JSON Web Token Claims registry [IANA.JWT] established by [RFC7519] MAY be included as members in the "token_introspection" claim. They can serve to convey the privileges delegated to the client, to identify the resource owner or to provide a required contact detail, such as an e-Mail address or phone number. When transmitting such claims the AS acts as an identity provider in regard to the RS. The AS determines based on its RS-specific policy what claims about the resource owner to return in the token introspection response.

The AS MUST ensure the release of any privacy-sensitive data is legally based (see Section 9).

Further content of the introspection response is determined by the RS-specific policy at the AS.

The JWT MAY include other claims, including those from the "JSON Web Token Claims" registry established by [RFC7519]. The JWT SHOULD NOT include the "sub" and "exp" claims as an additional prevention against misuse of the JWT as an access token (see Section 8.1).

Note: Although the JWT format is widely used as an access token format, the JWT returned in the introspection response is not an alternative representation of the introspected access token and is not intended to be used as an access token.

This specification registers the "application/token-introspection+jwt" media type, which is used as value of the "typ" ("type") header parameter of the JWT to indicate that the payload is a token introspection response.

The JWT is cryptographically secured as specified in [RFC7662].

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/token-introspection+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 8.2).

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

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


```
eyJraWQiOiJ3RzZEIiwidHlwIjoidG9rZW4taW50cm9zcGVjdGlvbitqd3QiLCJhbGc
iOiJSUzI1NiJ9.eyJpc3MiOiJodHRwczovL2FzLmV4YW1wbGUuY29tLyIsImF1ZCI6I
mh0dHBzOi8vcnMuZXhhbXBsZS5jb20vcmVzb3VyY2UiLCJpYXQiOjE1MTQ3OTc4OTIs
InRva2VuX2ludHJvc3BlY3Rpb24iOnsiYWN0aXZlIjp0cnVlLCJpc3MiOiJodHRwczo
vL2FzLmV4YW1wbGUuY29tLyIsImF1ZCI6Imh0dHBzOi8vcnMuZXhhbXBsZS5jb20vcm
Vzb3VyY2UiLCJpYXQiOjE1MTQ3OTc4MjIsImV4cCI6MTUxNDc5Nzk0MiwiY2xpZW50X
2lkIjoicGFpQjJnb28wYSIsInNjb3BlIjoicmVhZHdyaXRlZG9scGhpbiIsInN1YiI6
Ilo1TzN1cFBDODhRckFqeDAwZGlzIiwiYmlydGhkYXRlIjoiMTk4Mi0wMi0wMSIsImd
pdmVuX25hbWUiOiJKb2huIiwiZmFtaWx5X25hbWUiOiJEb2UiLCJqdGkiOiJ0MUZvQ0
NhWmQ0WHY0T1JKVVdWVWVUWmZzS2hXMzBDUUNyV0REandYeTZ3In19.przJMU5GhmNz
vwtt1Sr-xa9xTkpiAg5IshbQsRiRVP_7eGR1GHYrNwQh84kxOkHCyje2g5WSRcYosGE
VIiC-eoPJJ-qBwqwSlgx9JEeCDw2W5DjrblOI_N0Jvsq_dUeOyoWVMqlOydOBhKNY0s
mBrI4NZvEExucOm9WUJXMuJtvq1gBes-0go5j4TEv9sOP9uu81gqWTr_LOo6pgT0tFF
yZfWC4kbXPXiQ2YT6mxCiQRRNM-l9cBdF6Jx6IOrsfFhBuYdYQ_mlL19HgDDOFaleyq
mru6lKlASOsaE8dmLSeKcX91FbG79FKN8un24iwIDCbKT9xlUFl54xWVShNDFA
```


The example response JWT header contains the following JSON document:


```
{
  "typ": "token-introspection+jwt",
  "alg": "RS256",
  "kid": "wG6D"
}
```


The example response JWT payload contains the following JSON document:

```
{
  "iss":"https://as.example.com/",
  "aud":"https://rs.example.com/resource",
  "iat":1514797892,
  "token_introspection":
     {
        "active":true,
        "iss":"https://as.example.com/",
        "aud":"https://rs.example.com/resource",
        "iat":1514797822,
        "exp":1514797942,
        "client_id":"paiB2goo0a",
        "scope":"read write dolphin",
        "sub":"Z5O3upPC88QrAjx00dis",
        "birthdate":"1982-02-01",
        "given_name":"John",
        "family_name":"Doe",
        "jti":"t1FoCCaZd4Xv4ORJUWVUeTZfsKhW30CQCrWDDjwXy6w"
     }
}
```


6. Client Metadata

The authorization server determines the algorithm 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 [RFC7591] with the resource server acting as a client, as specified below.

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 OPTIONAL. JWS [RFC7515] algorithm ("alg" value) as defined in JWA [RFC7518] 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 OPTIONAL. JWE [RFC7516] algorithm ("alg" value) as defined in JWA [RFC7518] for content key encryption. If this is specified, the response will be encrypted using JWE and the configured content encryption algorithm ("introspection_encrypted_response_enc"). The default, if omitted, is that no encryption is performed. 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].

introspection_encrypted_response_enc OPTIONAL. JWE [RFC7516] algorithm ("enc" value) as defined in JWA [RFC7518] for content encryption of introspection responses. The default, if omitted, is "A128CBC-HS256". Note: This parameter MUST NOT be specified without setting "introspection_encrypted_response_alg".

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

7. 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. Resource servers use this data to parametrize their client registration requests.

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) as defined in JWA [RFC7518] supported by the introspection endpoint to sign the response.

introspection_encryption_alg_values_supported OPTIONAL. JSON array containing a list of the JWE [RFC7516] encryption algorithms ("alg" values) as defined in JWA [RFC7518] supported by the introspection endpoint to encrypt the content encryption key for introspection responses (content key encryption).

introspection_encryption_enc_values_supported OPTIONAL. JSON array containing a list of the JWE [RFC7516] encryption algorithms ("enc" values) as defined in JWA [RFC7518] supported by the introspection endpoint to encrypt the response (content encryption).

8. Security Considerations

8.1. Cross-JWT Confusion

The "iss" and potentially the "aud" claim of a token introspection JWT can resemble those of a JWT-encoded access token. An attacker could try to exploit this and pass a JWT token introspection response as an access token to the resource server. The "typ" ("type") JWT header "token-introspection+jwt" and the encapsulation of the token introspection members such as "sub" and "scope" in the "token_introspection" claim is intended to prevent such substitution attacks. Resource servers MUST therefore check the "typ" JWT header value of received JWT-encoded access tokens and ensure all minimally required claims for a valid access token are present.

Resource servers MUST additionally apply the countermeasures against replay as described in [I-D.ietf-oauth-security-topics], section 3.2.

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

8.2. Token Data Leakage

The authorization server MUST use Transport Layer Security (TLS) 1.2 (or higher) per BCP 195 [RFC7525] in order to prevent token data leakage.

To prevent introspection of leaked tokens and to present an additional security layer against token guessing attacks the authorization server MAY require all requests to the token introspection endpoint to be authenticated. As an alternative or as an addition to the authentication, the intended recipients MAY be set up 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/token-introspection+jwt". To prevent this attack the authorization server MUST NOT serve requests with a content type other than "application/token-introspection+jwt" if the resource server is set up to receive encrypted responses (see also Section 5).

8.3. Keeping Token Data Confidential from OAuth Clients

Authorization servers with a policy that requires token data to be kept confidential from OAuth clients must require all requests to the token introspection endpoint to be authenticated. As an alternative or as an addition to the authentication, the intended recipients may be set up for encrypted responses.

8.4. Logging and Audit of Introspection Activity

Authorization servers with a policy that requires token introspection activity to be logged and audited must require all requests to the token introspection endpoint to be authenticated.

9. Privacy Considerations

The token introspection response can be used to transfer personal identifiable information from the AS to the RS. The AS MUST ensure a legal basis exists for the data transfer before any data is released to a particular RS. The way the legal basis is established might vary among jurisdictions and MUST consider the legal entities involved.

For example, the classical way to establish the legal basis is by explicit user consent gathered from the resource owner by the AS during the authorization flow.

It is also possible that the legal basis is established out of band, e.g. in an explicit contract or by the client gathering the resource owner's consent.

If the AS and the RS belong to the same legal entity (1st party scenario), there is potentially no need for an explicit user consent but the terms of service and policy of the respective service provider MUST be enforced at all times.

In any case, the AS MUST ensure that the scope of the legal basis is enforced throughout the whole process. The AS MUST retain the scope of the legal basis with the access token, e.g. in the scope value, it MUST authenicate the RS, and the AS MUST determine the data a resource server is allowed to receive based on the resource server's identity and suitable token data, e.g. the scope value.

Implementers should be aware that a token introspection request lets the AS know when the client (and potentially the user) is accessing the RS, which is also an indication of when the user is using the client. If this impliction is not accepatable, implementars MUST use other means to carry access token data, e.g. directly transferring the data needed by the RS within the access token.

10. Acknowledgements

We would like to thank Petteri Stenius, Neil Madden, Filip Skokan, Tony Nadalin, Remco Schaar, Justin Richer and Takahiko Kawasaki for their valuable feedback.

11. IANA Considerations

11.1. 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]:

11.1.1. 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 6 of [[this specification]]

- Client Metadata Name: "introspection_encrypted_response_alg"

- Client Metadata Description: String value specifying the desired introspection response content key encryption algorithm (alg value).

- Change Controller: IESG

- Specification Document(s): Section 6 of [[this specification]]

- Client Metadata Name: "introspection_encrypted_response_enc"

- Client Metadata Description: String value specifying the desired introspection response content encryption algorithm (enc value).

- Change Controller: IESG

- Specification Document(s): Section 6 of [[this specification]]

11.2. 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].

11.2.1. 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.

- Change Controller: IESG

- Specification Document(s): Section 7 of [[this specification]]

- Metadata Name: "introspection_encryption_alg_values_supported"

- Metadata Description: JSON array containing a list of algorithms supported by the authorization server for introspection response content key encryption (alg value).

- Change Controller: IESG

- Specification Document(s): Section 7 of [[this specification]]

- Metadata Name: "introspection_encryption_enc_values_supported"

- Metadata Description: JSON array containing a list of algorithms supported by the authorization server for introspection response content encryption (enc value).

- Change Controller: IESG

- Specification Document(s): Section 7 of [[this specification]]

11.3. Media Type Registration

This section registers the "application/token-introspection+jwt" media type in the "Media Types" registry [IANA.MediaTypes] in the manner described in [RFC6838], which can be used to indicate that the content is a token introspection response in JWT format.

11.3.1. Registry Contents

- o Type name: application
- o Subtype name: token-introspection+jwt
- o Required parameters: N/A
- o Optional parameters: N/A
- o Encoding considerations: binary; A token introspection response is a JWT; JWT values are encoded as a series of base64url-encoded values (with trailing '=' characters removed), some of which may be the empty string, separated by period ('.') characters.
- o Security considerations: See Section 7 of this specification
- o Interoperability considerations: N/A
- o Published specification: Section 4 of this specification
- o Applications that use this media type: Applications that produce and consume OAuth Token Introspection Responses in JWT format
- o Fragment identifier considerations: N/A
- o Additional information:
  - * Magic number(s): N/A
  - * File extension(s): N/A
  - * Macintosh file type code(s): N/A
- o Person & email address to contact for further information: Torsten Lodderstedt, torsten@lodderstedt.net
- o Intended usage: COMMON
- o Restrictions on usage: none
- o Author: Torsten Lodderstedt, torsten@lodderstedt.net
- o Change controller: IESG
- o Provisional registration? No

11.4. JWT Claim Registration

This section registers the "token_introspection" claim in the JSON Web Token (JWT) IANA registry [IANA.JWT] in the manner described in [RFC7519].

11.4.1. Registry Contents

- Claim name: token_introspection

- Claim description: Token introspection response

- Change Controller: IESG

- Specification Document(s): Section 5 of [[this specification]]

12. References

12.1. Normative References

[I-D.ietf-oauth-jwt-bcp]
Sheffer, Y., Hardt, D., and M. Jones, "JSON Web Token Best Current Practices", draft-ietf-oauth-jwt-bcp-06 (work in progress), June 2019.

[I-D.ietf-oauth-security-topics]
Lodderstedt, T., Bradley, J., Labunets, A., and D. Fett, "OAuth 2.0 Security Best Current Practice", draft-ietf-oauth-security-topics-13 (work in progress), July 2019.

[IANA.JWT]
IANA, "JSON Web Token (JWT) claims registry", <https://www.iana.org/assignments/jwt/jwt.xhtml#claims>.

[IANA.MediaTypes]
IANA, "Media Types", <http://www.iana.org/assignments/media-types>.

[IANA.OAuth.Token.Introspection]
IANA, "OAuth Token Introspection Response registry", <https://www.iana.org/assignments/oauth-parameters/oauth-parameters.xhtml#token-introspection-response>.

[OpenID.Registration]
Sakimura, N., Bradley, J., and M. Jones, "OpenID Connect Dynamic Client Registration 1.0 incorporating errata set 1", Nov 2014, <https://openid.net/specs/openid-connect-registration-1_0.html>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, DOI 10.17487/RFC6838, January 2013, <https://www.rfc-editor.org/info/rfc6838>.

[RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <https://www.rfc-editor.org/info/rfc7515>.

[RFC7516] Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)", RFC 7516, DOI 10.17487/RFC7516, May 2015, <https://www.rfc-editor.org/info/rfc7516>.

[RFC7518] Jones, M., "JSON Web Algorithms (JWA)", RFC 7518, DOI 10.17487/RFC7518, May 2015, <https://www.rfc-editor.org/info/rfc7518>.

[RFC7519] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <https://www.rfc-editor.org/info/rfc7519>.

[RFC7525] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 7525, DOI 10.17487/RFC7525, May 2015, <https://www.rfc-editor.org/info/rfc7525>.

[RFC7591] Richer, J., Ed., Jones, M., Bradley, J., Machulak, M., and P. Hunt, "OAuth 2.0 Dynamic Client Registration Protocol", RFC 7591, DOI 10.17487/RFC7591, July 2015, <https://www.rfc-editor.org/info/rfc7591>.

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

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8414] Jones, M., Sakimura, N., and J. Bradley, "OAuth 2.0 Authorization Server Metadata", RFC 8414, DOI 10.17487/RFC8414, June 2018, <https://www.rfc-editor.org/info/rfc8414>.

12.2. 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]]

-10

- o added requirement to authenticate RS if privacy sensitive data is released
- o reworked text on claims from different registries
- o added forward reference to privacy considerations to section 5
- o added text in privacy considerations regarding client/user tracking

-09

- o changes the Accept and Content-Type HTTP headers from "application/json" to "application/token-introspection+jwt" so they match the registered media type
- o moves the token introspection response members into a JSON object claim named "token_introspection" to provide isolation from the top-level JWT-specific claims
- o "iss", "aud" and "iat" MUST be present as top-level JWT claims
- o the "sub" and "exp" claims SHOULD NOT be used as top-level JWT claims as additional prevention against JWT access token substitution attacks

-08

- o made difference between introspected access token and introspection response clearer

- o defined semantics of JWT claims overlapping between introspected access token and introspection response as JWT
- o added section about RS management
- o added text about user claims including a privacy considerations section
- o removed registration of OpenID Connect claims to "Token Introspection Response" registry and refer to "JWT Claims" registry instead
- o added registration of "application/token-introspection+jwt" media type as type identifier of token introspection responses in JWT format
- o more changed to incorporate IESG review feedback

-07

- o fixed wrong description of "locale"
- o added references for ISO and ITU specifications

-06

- o replaced reference to RFC 7159 with reference to RFC 8259

-05

- o improved wording for TLS requirement
- o added RFC 2119 boilerplate
- o fixed and updated some references

-04

- o reworked definition of parameters in section 4
- o added text on data minimization to security considerations section
- o added statement regarding TLS to security considerations section

-03

- o 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
- added text on claims in the token introspection response

-00

- initial version of the WG draft
- defined default signing algorithm
- changed behavior in case resource server is set up for encryption
- Added text on token data leakage prevention to the security considerations
- moved Security Considerations section forward

WG draft

-01

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

-00

- initial version

Authors' Addresses

Torsten Lodderstedt (editor)
yes.com AG

Email: torsten@lodderstedt.net

Vladimir Dzhuvinov
Connect2id Ltd.

Email: vladimir@connect2id.com

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

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

August 22, 2019

OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens

draft-ietf-oauth-mtls-17

Abstract

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

Status of This Memo

Table of Contents

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.

```
     +--------+                                 +----------------+
     |        |                                 |                |
     |        |<--(A)-- Get an access token --->| Authorization  |
     |        |                                 |     Server     |
     |        |                                 |                |
     |        |                                 +----------------+
     |        |                                         ^
     |        |                                         |
     |        |
     |        |                               (C)       |
     | Client |                           Validate the
     |        |                           access token  |
     |        |
     |        |                                         |
     |        |                                         v
     |        |                                 +----------------+
     |        |                                 |      (C)       |
     |        |                                 |                |
     |        |<--(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. Other cases require that the protected resource call out to the authorization server to determine the state of the token and obtain meta-information about it.

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

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

Mutual-TLS certificate-bound access tokens ensure that only the party in possession of the private key corresponding to the certificate can utilize the token to access the associated resources. Such a constraint is sometimes referred to as key confirmation, proof-of-possession, or holder-of-key and is unlike the case of the bearer token described in [RFC6750], where any party in possession of the access token can use it to access the associated resources. Binding an access token to the client's certificate prevents the use of stolen access tokens or replay of access tokens by unauthorized parties.

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

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

1.1. Requirements Notation and Conventions

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

1.2. Terminology

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

2. Mutual TLS for OAuth Client Authentication

This section defines, as an extension of OAuth 2.0, Section 2.3 [RFC6749], two distinct methods of using mutual-TLS X.509 client certificates as client credentials. The requirement of mutual TLS for client authentication is determined by the authorization server based on policy or configuration for the given client (regardless of whether the client was dynamically registered, statically configured, or otherwise established).

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

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

2.1. PKI Mutual-TLS Method

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

2.1.1. PKI Method Metadata Value

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

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

2.1.2. Client Registration Metadata

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

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

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

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

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

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

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

2.2. Self-Signed Certificate Mutual-TLS Method

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

2.2.1. Self-Signed Method Metadata Value

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

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

2.2.2. Client Registration Metadata

For the Self-Signed Certificate method of binding a certificate with a client using mutual TLS client authentication, the existing "jwks_uri" or "jwks" metadata parameters from [RFC7591] are used to convey the client's certificates via JSON Web Key (JWK) in a JWK Set (JWKS) [RFC7517]. The "jwks" metadata parameter is a JWK Set containing the client's public keys as an array of JWKs while the "jwks_uri" parameter is a URL that references a client's JWK Set. A certificate is represented with the "x5c" parameter of an individual JWK within the set. Note that the members of the JWK representing the public key (e.g. "n" and "e" for RSA, "x" and "y" for EC) are required parameters per [RFC7518] so will be present even though they are not utilized in this context. Also note that that Section 4.7 of [RFC7517] requires that the key in the first certificate of the "x5c" parameter match the public key represented by those other members of the JWK.

3. Mutual-TLS Client Certificate-Bound Access Tokens

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

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

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

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

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

3.1. JWT Certificate Thumbprint Confirmation Method

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

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

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

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


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

3.2. Confirmation Method for Token Introspection

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

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

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

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

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

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

3.3. Authorization Server Metadata

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

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

3.4. Client Registration Metadata

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

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

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

4. Public Clients and Certificate-Bound Tokens

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

5. Metadata for Mutual-TLS Endpoint Aliases

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

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

mtls_endpoint_aliases

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

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

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


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

6. Implementation Considerations

6.1. Authorization Server

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

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

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

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

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

6.2. Resource Server

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

6.3. Certificate Expiration and Bound Access Tokens

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

6.4. Implicit Grant Unsupported

This document describes binding an access token to the client certificate presented on the TLS connection from the client to the authorization server's token endpoint, however, such binding of access tokens issued directly from the authorization endpoint via the implicit grant flow is explicitly out of scope. End users interact directly with the authorization endpoint using a web browser and the use of client certificates in user's browsers bring operational and usability issues, which make it undesirable to support certificate-bound access tokens issued in the implicit grant flow. Implementations wanting to employ certificate-bound access tokens should utilize grant types that involve the client making an access token request directly to the token endpoint (e.g. the authorization code and refresh token grant types).

6.5. TLS Termination

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

7. Security Considerations

7.1. Certificate-Bound Refresh Tokens

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

7.2. Certificate Thumbprint Binding

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

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

7.3. TLS Versions and Best Practices

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

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

7.4. X.509 Certificate Spoofing

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

7.5. X.509 Certificate Parsing and Validation Complexity

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

- o checking of Basic Constraints, basic and extended Key Usage constraints, validity periods, and critical extensions;

- o handling of embedded NUL bytes in ASN.1 counted-length strings, and non-canonical or non-normalized string representations in subject names;

- 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 values 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]]

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

9.3. Token Endpoint Authentication Method Registration

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

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

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

9.4. Token Introspection Response Registration

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

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

- o Claim Name: "cnf"

- 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]]

- Client Metadata Name: "tls_client_auth_san_ip"
- Client Metadata Description: String value specifying the expected iPAddress 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_email"
- Client Metadata Description: String value specifying the expected rfc822Name SAN entry in the client certificate.
- Change Controller: IESG

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

10. References

10.1. Normative References

[BCP195] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 7525, DOI 10.17487/RFC7525, May 2015, <http://www.rfc-editor.org/info/bcp195>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC4514] Zeilenga, K., Ed., "Lightweight Directory Access Protocol (LDAP): String Representation of Distinguished Names", RFC 4514, DOI 10.17487/RFC4514, June 2006, <https://www.rfc-editor.org/info/rfc4514>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <https://www.rfc-editor.org/info/rfc5246>.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <https://www.rfc-editor.org/info/rfc5280>.

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

[RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, <https://www.rfc-editor.org/info/rfc6750>.

[RFC7517] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, <https://www.rfc-editor.org/info/rfc7517>.

[RFC7519] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <https://www.rfc-editor.org/info/rfc7519>.

[RFC7591] Richer, J., Ed., Jones, M., Bradley, J., Machulak, M., and P. Hunt, "OAuth 2.0 Dynamic Client Registration Protocol", RFC 7591, DOI 10.17487/RFC7591, July 2015, <https://www.rfc-editor.org/info/rfc7591>.

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

[RFC7800] Jones, M., Bradley, J., and H. Tschofenig, "Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs)", RFC 7800, DOI 10.17487/RFC7800, April 2016, <https://www.rfc-editor.org/info/rfc7800>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8414] Jones, M., Sakimura, N., and J. Bradley, "OAuth 2.0 Authorization Server Metadata", RFC 8414, DOI 10.17487/RFC8414, June 2018, <https://www.rfc-editor.org/info/rfc8414>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

[SHS] National Institute of Standards and Technology, "Secure Hash Standard (SHS)", FIPS PUB 180-4, March 2012, <http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf>.

[X690] International Telephone and Telegraph Consultative Committee, "ASN.1 encoding rules: Specification of basic encoding Rules (BER), Canonical encoding rules (CER) and Distinguished encoding rules (DER)", CCITT Recommendation X.690, July 2015.

10.2. Informative References

[CX5P] Wong, D., "Common x509 certificate validation/creation pitfalls", September 2016, <https://www.cryptologie.net/article/374/common-x509-certificate-validationcreation-pitfalls>.

[DCW] Georgiev, M., Iyengar, S., Jana, S., Anubhai, R., Boneh, D., and V. Shmatikov, "The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software", <http://www.cs.utexas.edu/~shmat/shmat_ccs12.pdf>.

[I-D.ietf-oauth-token-binding]
Jones, M., Campbell, B., Bradley, J., and W. Denniss, "OAuth 2.0 Token Binding", draft-ietf-oauth-token-binding-06 (work in progress), March 2018.

[IANA.JWT.Claims]
IANA, "JSON Web Token Claims", <http://www.iana.org/assignments/jwt>.

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

[OpenID.CIBA]
Fernandez, G., Walter, F., Nennker, A., Tonge, D., and B. Campbell, "OpenID Connect Client Initiated Backchannel Authentication Flow - Core 1.0", January 2019, <https://openid.net/specs/openid-client-initiated-backchannel-authentication-core-1_0.html>.

[RFC4517] Legg, S., Ed., "Lightweight Directory Access Protocol (LDAP): Syntaxes and Matching Rules", RFC 4517, DOI 10.17487/RFC4517, June 2006, <https://www.rfc-editor.org/info/rfc4517>.

[RFC5952] Kawamura, S. and M. Kawashima, "A Recommendation for IPv6 Address Text Representation", RFC 5952, DOI 10.17487/RFC5952, August 2010, <https://www.rfc-editor.org/info/rfc5952>.

[RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <https://www.rfc-editor.org/info/rfc6066>.

[RFC7009] Lodderstedt, T., Ed., Dronia, S., and M. Scurtescu, "OAuth 2.0 Token Revocation", RFC 7009, DOI 10.17487/RFC7009, August 2013, <https://www.rfc-editor.org/info/rfc7009>.

[RFC7518] Jones, M., "JSON Web Algorithms (JWA)", RFC 7518, DOI 10.17487/RFC7518, May 2015, <https://www.rfc-editor.org/info/rfc7518>.

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

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


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


Figure 5: x5t#S256 Confirmation Claim


```
-----BEGIN CERTIFICATE-----
MIIBBjCBrAIBAjAKBggqhkjOPQQDAjAPMQ0wCwYDVQQDDARtdGxzMB4XDTE4MTAx
ODEyMzcwOVoXDTIyMDUwMjEyMzcwOVowDzENMAsGA1UEAwwEbXRsczBZMBMGByqG
SM49AgEGCCqGSM49AwEHA0IABNcnyxwqV6hY8QnhxxzFQ03C7HKW9OylMbnQZjjJ
/Au08/coZwxS7LfA4vOLS9WuneIXhbGGWvsDSb0tH6IxLm8wCgYIKoZIzj0EAwID
SQAwRgIhAP0RC1E+vwJD/D1AGHGzuri+hlV/PpQEKTWUVeORWz83AiEA5x2eXZOV
bUlJSGQgjwD5vaUaKlLR50Q2DmFfQj1L+SY=
-----END CERTIFICATE-----
```


Figure 6: PEM Encoded Self-Signed Certificate


```
{
 "kty":"EC",
 "x":"1yfLHCpXqFjxCeHHHMVDTcLscpb07KUxudBmOMn8C7Q",
 "y":"8_coZwxS7LfA4vOLS9WuneIXhbGGWvsDSb0tH6IxLm8",
 "crv":"P-256",
 "x5c":[
  "MIIBBjCBrAIBAjAKBggqhkjOPQQDAjAPMQ0wCwYDVQQDDARtdGxzMB4XDTE4MTA
   xODEyMzcwOVoXDTIyMDUwMjEyMzcwOVowDzENMAsGA1UEAwwEbXRsczBZMBMGBy
   qGSM49AgEGCCqGSM49AwEHA0IABNcnyxwqV6hY8QnhxxzFQ03C7HKW9OylMbnQZ
   jjJ/Au08/coZwxS7LfA4vOLS9WuneIXhbGGWvsDSb0tH6IxLm8wCgYIKoZIzj0E
   AwIDSQAwRgIhAP0RC1E+vwJD/D1AGHGzuri+hlV/PpQEKTWUVeORWz83AiEA5x2
   eXZOVbUlJSGQgjwD5vaUaKlLR50Q2DmFfQj1L+SY="
   ]
 }
```


Figure 7: JSON Web Key

Appendix B. Relationship to Token Binding

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

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

Appendix C. Acknowledgements

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

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

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

Appendix D. Document(s) History

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

draft-ietf-oauth-mtls-17

- o Updates from IESG ballot position comments.

draft-ietf-oauth-mtls-16

- o Editorial updates from last call review.

draft-ietf-oauth-mtls-15

- o Editorial updates from second AD review.

draft-ietf-oauth-mtls-14

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

draft-ietf-oauth-mtls-13

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

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

draft-ietf-oauth-mtls-12

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

draft-ietf-oauth-mtls-11

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

draft-ietf-oauth-mtls-10

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

draft-ietf-oauth-mtls-09

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

draft-ietf-oauth-mtls-08

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

draft-ietf-oauth-mtls-07

o Update to use the boilerplate from RFC 8174

draft-ietf-oauth-mtls-06

o Add an appendix section describing the relationship of this document to OAuth Token Binding as requested during the Singapore meeting https://datatracker.ietf.org/doc/minutes-100-oauth/
o Add an explicit note that the implicit flow is not supported for obtaining certificate bound access tokens as discussed at the Singapore meeting https://datatracker.ietf.org/doc/minutes-100-oauth/
o Add/incorporate text to the Security Considerations on Certificate Spoofing as suggested https://mailarchive.ietf.org/arch/msg/oauth/V26070X-6OtbVSeUz_7W2k94vCo
o Changed the title to be more descriptive
o Move the Security Considerations section to before the IANA Considerations
o Elaborated on certificate-bound access tokens a bit more in the Abstract
o Update draft-ietf-oauth-discovery reference to -08

draft-ietf-oauth-mtls-05

o Editorial fixes

draft-ietf-oauth-mtls-04

- o Change the name of the 'Public Key method' to the more accurate 'Self-Signed Certificate method' and also change the associated authentication method metadata value to "self_signed_tls_client_auth".
- o Removed the "tls_client_auth_root_dn" client metadata field as discussed in https://mailarchive.ietf.org/arch/msg/oauth/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/bZ6mft0G7D3ccebhOxnEYUv4puI
- o Reorganize the document somewhat in an attempt to more clearly make a distinction between mTLS client authentication and certificate-bound access tokens as well as a more clear delineation between the two (PKI/Public key) methods for client authentication
- o Editorial fixes and clarifications

draft-ietf-oauth-mtls-03

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

draft-ietf-oauth-mtls-02

- o Fixed editorial issue https://mailarchive.ietf.org/arch/msg/oauth/U46UMEh8XIOQnvXY9pHFq1MKPns
- o Changed the title (hopefully "Mutual TLS Profile for OAuth 2.0" is better than "Mutual TLS Profiles for OAuth Clients").

draft-ietf-oauth-mtls-01

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

draft-ietf-oauth-mtls-00

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

draft-campbell-oauth-mtls-01

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

draft-campbell-oauth-mtls-00

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

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

- o Initial draft.

Authors' Addresses

Brian Campbell
Ping Identity

Email: brian.d.campbell@gmail.com

John Bradley
Yubico

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

Nat Sakimura
Nomura Research Institute

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

Torsten Lodderstedt
YES.com AG

Email: torsten@lodderstedt.net

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 28, 2019

J. Bradley
Ping Identity
P. Hunt
Oracle Corporation
M. Jones
Microsoft
H. Tschofenig
Arm Ltd.
M. Meszaros
GITDA

March 27, 2019

OAuth 2.0 Proof-of-Possession: Authorization Server to Client Key Distribution

draft-ietf-oauth-pop-key-distribution-07

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

Copyright Notice

Table of Contents

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.

```
+--------+                                +---------------+
|        |--(A)- Authorization Request ->|   Resource    |
|        |                                |     Owner     |
|        |<-(B)-- Authorization Grant ---|               |
|        |                                +---------------+
|        |
|        |                                +---------------+
|        |--(C)-- Authorization Grant -->|               |
| Client |       (resource, req_cnf)      | Authorization |
|        |                                |     Server    |
|        |<-(D)-- PoP Access Token ------|               |
|        |       (rs_cnf, token_type)     +---------------+
|        |
|        |                                +---------------+
|        |--(E)-- PoP Access Token ----->|               |
|        |     (with proof of private key)|    Resource   |
|        |                                |     Server    |
|        |<-(F)--- Protected Resource ---|               |
+--------+                                +---------------+
```

Figure 1: Augmented OAuth 2.0 Protocol Flow

In OAuth 2.0 [2] access tokens can be obtained via authorization grants and using refresh tokens. The core OAuth specification defines four authorization grants, see Section 1.3 of [2], and [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:

JWA: JSON Web Algorithms[7]

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

RECOMMENDED. 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

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

grant_type=authorization_code
&code=SplxlOBeZQQYbYS6WxSbIA
&scope=calendar%20contacts
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
&resource=https%3A%2F%2Fresource.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.

```
{
  "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":{
      "jwe":
        "eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkExMjhDQkMtSFMyNTYifQ.
        (remainder of JWE omitted for brevity)"
      }
      }
}
```

Figure 3: Example: Encrypted Symmmetric Key

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).

```
{
   "iss": "https://server.example.com",
   "sub": "24400320",
   "aud": "s6BhdRkqt3",
   "exp": 1311281970,
   "iat": 1311280970,
   "cnf":{
     "jwe":
       "eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkExMjhDQkMtSFMyNTYifQ.
       (remainder of JWE omitted for brevity)"
     }
}
```

Figure 4: Example: Access Token in JWT Format

Note: When the JWK inside the access token contains a symmetric key it must be confidentiality protected using a JWE to maintain the security goals of the PoP architecture 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": "18wHLeIgW9wVN6VD1Txgpqy2LszYkMf6J8njVAibvhM",
  "y": "-V4dS4UaLMgP_4fY4j8ir7cl1TXlFdAgcx55o7TkcSA"
 }
```

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.

```
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":"tGzv3JOkF0XG5Qx2TlKWIA"
}
```

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).

```
{
  "iss":"https://authz.example.com",
  "aud":"https://resource.example.com",
  "exp":"1361398824",
  "nbf":"1360189224",
  "cnf":{
     "jwk" : {
       "kty" : "EC",
       "crv" : "P-256",
       "x" : "usWxHK2PmfnHKwXPS54m0kTcGJ90UiglWiGahtagnv8",
       "y" : "IBOL+C3BttVivg+lSreASjpkttcsz+1rb7btKLv8EX4"
     }
   }
 }
```

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

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

- o Name: pop
- o Change controller: IESG
- o 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].

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

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

- o Parameter name: rs_cnf
- o Parameter usage location: token response
- o Change controller: IESG
- o 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].

- o Error name: invalid_token_type
- o Error usage location: implicit grant error response, token error response
- o Related protocol extension: token_type parameter
- o Change controller: IESG
- o 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

[1] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

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

[3] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[4] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <https://www.rfc-editor.org/info/rfc5246>.

[5] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, <https://www.rfc-editor.org/info/rfc7517>.

[6] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <https://www.rfc-editor.org/info/rfc7515>.

[7] Jones, M., "JSON Web Algorithms (JWA)", RFC 7518, DOI 10.17487/RFC7518, May 2015, <https://www.rfc-editor.org/info/rfc7518>.

[8] Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)", RFC 7516, DOI 10.17487/RFC7516, May 2015, <https://www.rfc-editor.org/info/rfc7516>.

[9] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <https://www.rfc-editor.org/info/rfc7519>.

[10] Jones, M., Bradley, J., and H. Tschofenig, "Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs)", RFC 7800, DOI 10.17487/RFC7800, April 2016, <https://www.rfc-editor.org/info/rfc7800>.

[11] Jones, M. and N. Sakimura, "JSON Web Key (JWK) Thumbprint", RFC 7638, DOI 10.17487/RFC7638, September 2015, <https://www.rfc-editor.org/info/rfc7638>.

[12] Jones, M., Seitz, L., Selander, G., Erdtman, S., and H. Tschofenig, "Proof-of-Possession Key Semantics for CBOR Web Tokens (CWTs)", draft-ietf-ace-cwt-proof-of-possession-06 (work in progress), February 2019.

[13] Jones, M., Wahlstroem, E., Erdtman, S., and H. Tschofenig, "CBOR Web Token (CWT)", RFC 8392, DOI 10.17487/RFC8392, May 2018, <https://www.rfc-editor.org/info/rfc8392>.

[14] Schaad, J., "CBOR Object Signing and Encryption (COSE)", RFC 8152, DOI 10.17487/RFC8152, July 2017, <https://www.rfc-editor.org/info/rfc8152>.

[15] Jones, M., Nadalin, A., Campbell, B., Bradley, J., and C. Mortimore, "OAuth 2.0 Token Exchange", draft-ietf-oauth-token-exchange-16 (work in progress), October 2018.

[16] Campbell, B., Bradley, J., and H. Tschofenig, "Resource Indicators for OAuth 2.0", draft-ietf-oauth-resource-indicators-02 (work in progress), January 2019.

8.2. Informative References

[17] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, <https://www.rfc-editor.org/info/rfc6750>.

[18] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[19] Campbell, B., Mortimore, C., Jones, M., and Y. Goland, "Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants", RFC 7521, DOI 10.17487/RFC7521, May 2015, <https://www.rfc-editor.org/info/rfc7521>.

[20] Sakimura, N., Ed., Bradley, J., and N. Agarwal, "Proof Key for Code Exchange by OAuth Public Clients", RFC 7636, DOI 10.17487/RFC7636, September 2015, <https://www.rfc-editor.org/info/rfc7636>.

[21] Richer, J., Ed., Jones, M., Bradley, J., Machulak, M., and P. Hunt, "OAuth 2.0 Dynamic Client Registration Protocol", RFC 7591, DOI 10.17487/RFC7591, July 2015, <https://www.rfc-editor.org/info/rfc7591>.

[22] Hunt, P., Richer, J., Mills, W., Mishra, P., and H. Tschofenig, "OAuth 2.0 Proof-of-Possession (PoP) Security Architecture", draft-ietf-oauth-pop-architecture-08 (work in progress), July 2016.

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

[24] Seitz, L., Selander, G., Wahlstroem, E., Erdtman, S., and H. Tschofenig, "Authentication and Authorization for Constrained Environments (ACE) using the OAuth 2.0 Framework (ACE-OAuth)", draft-ietf-ace-oauth-authz-24 (work in progress), March 2019.

[25] IANA, "OAuth Parameters", October 2018.

[26] IANA, "JSON Web Token Claims", June 2018.

Authors' Addresses

John Bradley
Ping Identity

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

Phil Hunt
Oracle Corporation

Email: phil.hunt@yahoo.com
URI: http://www.indepdentid.com

Michael B. Jones
Microsoft

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

Hannes Tschofenig
Arm Ltd.
Absam 6067
Austria

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

Mihaly Meszaros
GITDA
Debrecen 4033
Hungary

Email: bakfitty@gmail.com
URI: https://github.com/misi

OAuth Working Group D. Hardt
Internet-Draft August 01, 2019
Intended status: Standards Track
Expires: February 2, 2020

Reciprocal OAuth
draft-ietf-oauth-reciprocal-04

Abstract

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

Status of This Memo

Copyright Notice

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 intial OAuth flow, party A obtains consent from the user to grant party B access to a protected resource at party A, and then passes an authorization code to party B using the access token party A obtained from party B to provide party B the context of the user. Party B then exchanges the authorization code for an access token per the usual OAuth flow.

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

1.1. Terminology

In this document, the key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" are to be interpreted as described in BCP 14, RFC 2119 [RFC2119].

2. reciprocal Protocol Flow

```
    Party A                                         Party B
    +---------------+                               +---------------+
    |               |--(A)- Authorization Request ->|   Resource    |
    |               |                               |   Owner B     |
    |               |<-(B)-- Authorization Grant ---|               |
    |               |                               +---------------+
    |   Client A    |
    |               |                               +---------------+
    |               |--(C)-- Authorization Grant -->|               |
    |               |                               | Authorization |
    |               |<-(D)---- Access Token B ------|   Server B    |
    |               |       reciprocal Request      |               |
    +---------------+                               +---------------+
           |
   reciprocal Request
           V
    +---------------+                               +---------------+
    |   Resource    |                               | Authorization |
    |   Owner A     |--(E)--- reciprocal Grant ---->|   Server B    |
    |               |          Access Token B       |               |
    +---------------+                               +---------------+
                                                            |
                                                    reciprocal Grant
                                                            V
    +---------------+                               +---------------+
    |               |<-(F)--- reciprocal Grant -----|               |
    | Authorization |                               |   Client B    |
    |  Server A     |--(G)---- Access Token A ----->|               |
    +---------------+                               +---------------+
```

Figure 1: Abstract reciprocal Protocol Flow

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

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

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

2.1. Reciprocal Scope Request

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

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

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

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

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

{
  "access_token":"2YotnFZFEjr1zCsicMWpAA",
  "token_type":"example",
  "expires_in":3600,
  "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
  "reciprocal":"example_scope",
  "example_parameter":"example_value"
}
```

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

```
HTTP/1.1 302 Found
Location: http://example.com/cb#
    access_token=2YotnFZFEjr1zCsicMWpAA&
    state=xyz&
    token_type=example&
    expires_in=3600&
    reciprocal="example_scope"
```

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

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

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

2.2. Reciprocal Authorization Flow

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

2.2.1. User Consent

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

2.2.2. Reciprocal Authorization Code

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

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

code REQUIRED
    the authorization code generated by party A.

client_id REQUIRED
    party A'a client ID.

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

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

```
POST /token HTTP/1.1
Host: server.example.com
Authorization: Basic ej4hsyfishwssjdusisdhkjsdksusdhjkjsdjk
Content-Type: application/x-www-form-urlencoded

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

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

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

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

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

3. Authorization Update Flow

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

4. IANA Considerations

4.1. Registration of reciprocal

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

- o Parameter Name: reciprocal

- o Parameter usage location: token response

- o Change Controller: IESG

- o Specification Document: Section Section 2.1 of this document

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

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

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

5. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

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

[RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, <https://www.rfc-editor.org/info/rfc6750>.

[RFC6755] Campbell, B. and H. Tschofenig, "An IETF URN Sub-Namespace for OAuth", RFC 6755, DOI 10.17487/RFC6755, October 2012, <https://www.rfc-editor.org/info/rfc6755>.

Appendix A. Document History

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

- o Initial version.

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

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

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

- o Added in diagram to clarify protocol flow

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

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

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

- o completed IANA section

Author's Address

Dick Hardt

Email: dick.hardt@gmail.com

OAuth Working Group B. Campbell
Internet-Draft Ping Identity
Intended status: Standards Track J. Bradley
Expires: March 14, 2020 Yubico
H. Tschofenig
Arm Limited
September 11, 2019

Resource Indicators for OAuth 2.0
draft-ietf-oauth-resource-indicators-08

Abstract

This document specifies 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.

Table of Contents

1. Introduction

Several years of deployment and implementation experience with the OAuth 2.0 Authorization Framework [RFC6749] has uncovered a need, in some circumstances such as an authorization server servicing a significant number of diverse resources, 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 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 to which access is being requested. Its value MUST be an absolute URI, as specified by Section 4.3 of [RFC3986]. The URI MUST NOT include a fragment component. It SHOULD NOT include a query component, but it is recognized that there are cases that make a query component a useful and necessary part of the resource parameter, such as when query parameter(s) are used to scope requests to an application. 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. It 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, missing, 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 an authorization request sent as a JSON Web Token (JWT), 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 apply policy (e.g., refuse the request due to unknown resources), and to 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 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".

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=XzZaJlcwYew1u0QBrRv_Gw
   &redirect_uri=https%3A%2F%2Fclient.example.org%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=tNwzQ87pC6llebpmac_IDeeq-mCR2wLDYljHUZUAWuI
   &redirect_uri=https%3A%2F%2Fclient.example.org%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 downscope 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 a list of scope values is an indication that the resource owner uses the multiple various services listed; downscoping a token to only that which is needed for a particular service can limit the extent to which such information is revealed across different services. 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 (Figure 3) and response (Figure 4) 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).

```
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.example.org%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.eyJpc3MiOi
    JodHRwOi8vYXV0aG9yaXphdGlvbi1zZXJ2ZXIuZXhhbXBsZS5jb20iLCJzdWI
    iOiJfX2JfYyIsImV4cCI6MTU4ODQyMDgwMCwic2NvcGUiOiJjYWxlbmRhciIs
    ImF1ZCI6Imh0dHBzOi8vY2FsLmV4YW1wbGUuY29tLyJ9.nNWJ2dXSxaDRdMUK
    lzs-cYIj8MDoM6Gy7pf_sKrLGsAFf1C2bDhB60DQfW1DZL5npdko1_Mmk5sUf
    zkiQNVpYw",
   "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 czZCaGRSa3F0Mzpoc3FFelFsVW9IQUU5cHg0RlNyNHlJ
Content-Type: application/x-www-form-urlencoded

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

Figure 5: Access Token Request

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

{
   "access_token":"eyJhbGciOiJFUzI1NiIsImtpZCI6Ijc3In0.eyJpc3MiOi
    JodHRwOi8vYXV0aG9yaXphdGlvbi1zZXJ2ZXIuZXhhbXBsZS5jb20iLCJzdWI
    iOiJfX2JfYyIsImV4cCI6MTU4ODQyMDgyNiwic2NvcGUiOiJjb250YWN0cyIs
    ImF1ZCI6Imh0dHBzOi8vY29udGFjdHMuZXhhbXBsZS5jb20vIn0.5f4yhqazc
    OSlJw4y94KPeWNEFQqj2cfeO8x4hr3YbHtIl3nQXnBMw5wREY5O1YbZED-GfH
    UowfmtNaA5EikYAw",
   "token_type":"Bearer",
   "expires_in":3600,
   "scope":"contacts"
}
```

Figure 6: Access Token Response

3. Security Considerations

An audience-restricted access token, legitimately presented to a resource, cannot then be taken by that resource and presented elsewhere for illegitimate access to other 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 where one tenant uses an access token to illegitimately access resources owned by a different tenant, it is important to use a specific resource URI including any portion of the URI that identifies the tenant, such as a path component. This will allow access tokens to be audience-restricted in a way that identifies the tenant and prevent their use, due to an invalid audience, at resources owned by a different tenant.

Although multiple occurrences of the "resource" parameter may be included in a token request, using only a single "resource" parameter is encouraged. A bearer token that has multiple intended recipients (audiences) indicating that the token is valid at more than one protected resource can be used by any one of those protected resources to access any of the other protected resources. 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. Privacy Considerations

In typical OAuth deployments the authorization sever is in a position to observe and track a significant amount of user and client behavior. It is largely just inherent to the nature of OAuth and this document does little to affect that. In some cases, however, such as when access token introspection is not being used, use of the resource parameter defined herein may allow for tracking behavior at a somewhat more granular and specific level than would otherwise be possible in its absence.

5. IANA Considerations

5.1. OAuth Parameters Registration

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

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

5.2. OAuth Extensions Error Registration

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

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

6. References

6.1. Normative References

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

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

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

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

6.2. Informative References

[I-D.ietf-oauth-jwsreq]
Sakimura, N. and J. Bradley, "The OAuth 2.0 Authorization Framework: JWT Secured Authorization Request (JAR)", draft-ietf-oauth-jwsreq-19 (work in progress), June 2019.

[RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, <https://www.rfc-editor.org/info/rfc6750>.

[RFC7519] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <https://www.rfc-editor.org/info/rfc7519>.

[RFC7644] Hunt, P., Ed., Grizzle, K., Ansari, M., Wahlstroem, E., and C. Mortimore, "System for Cross-domain Identity Management: Protocol", RFC 7644, DOI 10.17487/RFC7644, September 2015, <https://www.rfc-editor.org/info/rfc7644>.

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

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, Benjamin Kaduk and Roman Danyliw serving as Security Area Directors. Additionally, the following individuals contributed ideas, feedback, and wording that helped shape this specification:

Vittorio Bertocci, Sergey Beryozkin, Roman Danyliw, William Denniss, Vladimir Dzhuvinov, George Fletcher, Dick Hardt, Phil Hunt, Michael Jones, Benjamin Kaduk, Barry Leiba, Torsten Lodderstedt, Anthony Nadalin, Justin Richer, Adam Roach, Nat Sakimura, Rifaat Shekh-Yusef, Filip Skokan, Eric Vyncke, and Hans Zandbelt.

Appendix B. Document History

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

draft-ietf-oauth-resource-indicators-08

- o One last update from IESG evaluation comments (https://mailarchive.ietf.org/arch/msg/oauth/x87EQ0Dwq3_ERrH5PzDjRSaWBt4).

draft-ietf-oauth-resource-indicators-07

- o One more update from IESG evaluation comments (https://mailarchive.ietf.org/arch/msg/oauth/RS0UZSsguQurHl4P18Zo77BzZnU).

draft-ietf-oauth-resource-indicators-06

- o Expand JWT acronym on first use per Genart last call review.
- o Updates from IESG evaluation comments.

draft-ietf-oauth-resource-indicators-05

- o Remove specific mention of error_uri, which is rarely (if ever) used and seems to only confuse things for readers of extensions like this one.

draft-ietf-oauth-resource-indicators-04

- o Editorial updates from AD review that were overlooked in -03.

draft-ietf-oauth-resource-indicators-03

- Editorial updates from AD review.
- Update draft-ietf-oauth-jwsreq ref to -19.
- Update the IANA requests to say they update the registries.

draft-ietf-oauth-resource-indicators-02

- 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

- 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.
- Change the "invalid_resource" error to "invalid_target" to align with draft-ietf-oauth-token-exchange, which has some overlap in functionality.
- Allow the "resource" parameter value to have a query component (aligning with draft-ietf-oauth-token-exchange).
- Moved the Security Considerations section to before the IANA Considerations.
- Other editorial updates.
- Rework the Acknowledgements section.
- Use RFC 8174 boilerplate.

draft-ietf-oauth-resource-indicators-00

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

draft-campbell-oauth-resource-indicators-02

- No changes.

draft-campbell-oauth-resource-indicators-01

- 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.

draft-campbell-oauth-resource-indicators-00

- 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

Web Authorization Protocol   T. Lodderstedt
Internet-Draft   yes.com
Intended status: Best Current Practice   J. Bradley
Expires: 8 April 2021   Yubico
                                                             A. Labunets

                                                                 D. Fett
                                                                 yes.com
                                                          5 October 2020

OAuth 2.0 Security Best Current Practice
draft-ietf-oauth-security-topics-16

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

Copyright Notice

Table of Contents

1. Introduction

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

* OAuth implementations are being attacked through known implementation weaknesses and anti-patterns. Although most of these threats are discussed in the OAuth 2.0 Threat Model and Security Considerations [RFC6819], continued exploitation demonstrates a need for more specific recommendations, easier to implement mitigations, and more defense in depth.

* OAuth is being used in environments with higher security requirements than considered initially, such as Open Banking, eHealth, eGovernment, and Electronic Signatures. Those use cases call for stricter guidelines and additional protection.

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

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

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

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

1.1. Structure

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

1.2. Conventions and Terminology

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

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

2. Recommendations

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

2.1. Protecting Redirect-Based Flows

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

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

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

In order to prevent mix-up attacks (see Section 4.4), clients MUST only process redirect responses of the authorization server they sent the respective request to and from the same user agent this authorization request was initiated with. Clients MUST store the authorization server they sent an authorization request to and bind this information to the user agent and check that the authorization request was received from the correct authorization server. Clients MUST ensure that the subsequent token request, if applicable, is sent to the same authorization server. Clients SHOULD use distinct redirect URIs for each authorization server as a means to identify the authorization server a particular response came from.

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

2.1.1. Authorization Code Grant

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

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

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

Authorization servers MUST support PKCE [RFC7636].

Authorization servers MUST provide a way to detect their support for PKCE. To this end, they MUST either (a) publish the element "code_challenge_methods_supported" in their AS metadata ([RFC8414]) containing the supported PKCE challenge methods (which can be used by the client to detect PKCE support) or (b) provide a deployment-specific way to ensure or determine PKCE support by the AS.

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

2.1.2. Implicit Grant

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

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

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

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

2.2. Token Replay Prevention

2.2.1. Access Tokens

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

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

2.2.2. Refresh Tokens

Refresh tokens MUST be sender-constrained or use refresh token rotation as described in Section 4.13.

2.3. Access Token Privilege Restriction

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

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

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

2.4. Resource Owner Password Credentials Grant

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

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

2.5. Client Authentication

Authorization servers SHOULD use client authentication if possible.

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

2.6. Other Recommendations

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

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

3. The Updated OAuth 2.0 Attacker Model

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

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

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

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

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

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

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

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

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

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

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

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

* (A4) Attackers that can read, but not modify, the contents of the authorization request (i.e., the authorization request can leak, in the same manner as above, to an attacker).

* (A5) Attackers that can acquire an access token issued by AS. For example, a resource server can be compromised by an attacker, an access token may be sent to an attacker-controlled resource server due to a misconfiguration, or an RO is social-engineered into using a attacker-controlled RS. See also Section 4.9.2.

(A3), (A4) and (A5) typically occur together with either (A1) or (A2).

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

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

4. Attacks and Mitigations

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

4.1. Insufficient Redirect URI Validation

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

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

* by directly sending the user agent to a URI under the attackers control, or

* by exposing the OAuth credentials to an attacker by utilizing an open redirector at the client in conjunction with the way user agents handle URL fragments.

These attacks are shown in detail in the following subsections.

4.1.1. Redirect URI Validation Attacks on Authorization Code Grant

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

Assume the redirect URL pattern "https://*.somesite.example/*" is registered for the client with the client ID "s6BhdRkqt3". The intention is to allow any subdomain of "somesite.example" to be a valid redirect URI for the client, for example "https://app1.somesite.example/redirect". A naive implementation on the authorization server, however, might interpret the wildcard "*" as "any character" and not "any character valid for a domain name". The authorization server, therefore, might permit "https://attacker.example/.somesite.example" as a redirect URI, although "attacker.example" is a different domain potentially controlled by a malicious party.

The attack can then be conducted as follows:

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

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

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

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

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

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

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

Note: Vulnerabilities of this kind can also exist if the authorization server handles wildcards properly. For example, assume that the client registers the redirect URL pattern "https://*.somesite.example/*" and the authorization server interprets this as "allow redirect URIs pointing to any host residing in the domain "somesite.example"". If an attacker manages to establish a host or subdomain in "somesite.example", he can impersonate the legitimate client. This could be caused, for example, by a subdomain takeover attack [subdomaintakeover], where an outdated CNAME record (say, "external-service.somesite.example") points to an external DNS name that does no longer exist (say, "customer-abc.service.example") and can be taken over by an attacker (e.g., by registering as "customer-abc" with the external service).

4.1.2. Redirect URI Validation Attacks on Implicit Grant

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

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

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

The attack can now be conducted as follows:

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

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

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

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

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

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

```
HTTP/1.1 303 See Other
Location: https://attacker.example/
```

Since the redirector at client.somesite.example does not include a fragment in the Location header, the user agent will re-attach the original fragment "#access_token=2YotnFZFEjr1zCsicMWpAA&..." to the URL and will navigate to the following URL:

```
https://attacker.example/#access_token=2YotnFZFEjr1z...
```

The attacker’s page at "attacker.example" can now access the fragment and obtain the access token.

4.1.3. Countermeasures

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

Additional recommendations:

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

* Browsers reattach URL fragments to Location redirection URLs only if the URL in the Location header does not already contain a fragment. Therefore, servers MAY prevent browsers from reattaching fragments to redirection URLs by attaching an arbitrary fragment identifier, for example "#_", to URLs in Location headers.

* Clients SHOULD use the authorization code response type instead of response types causing access token issuance at the authorization endpoint. This offers countermeasures against reuse of leaked credentials through the exchange process with the authorization server and token replay through sender-constraining of the access tokens.

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

4.2. Credential Leakage via Referer Headers

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

4.2.1. Leakage from the OAuth Client

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

* contains links to other pages under the attacker's control and a user clicks on such a link, or

* includes third-party content (advertisements in iframes, images, etc.), for example if the page contains user-generated content (blog).

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

4.2.2. Leakage from the Authorization Server

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

4.2.3. Consequences

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

4.2.4. Countermeasures

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

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

* Suppress the Referer header by applying an appropriate Referrer Policy [webappsec-referrer-policy] to the document (either as part of the "referrer" meta attribute or by setting a Referrer-Policy header). For example, the header "Referrer-Policy: no-referrer" in the response completely suppresses the Referer header in all requests originating from the resulting document.

* Use authorization code instead of response types causing access token issuance from the authorization endpoint.

* Bind authorization code to a confidential client or PKCE challenge. In this case, the attacker lacks the secret to request the code exchange.

* As described in [RFC6749], Section 4.1.2, authorization codes MUST be invalidated by the AS after their first use at the token endpoint. For example, if an AS invalidated the code after the legitimate client redeemed it, the attacker would fail exchanging this code later.

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

* The "state" value SHOULD be invalidated by the client after its first use at the redirection endpoint. If this is implemented, and an attacker receives a token through the Referer header from the client's web site, the "state" was already used, invalidated by the client and cannot be used again by the attacker. (This does not help if the "state" leaks from the AS's web site, since then the "state" has not been used at the redirection endpoint at the client yet.)

* Use the form post response mode instead of a redirect for the authorization response (see [oauth-v2-form-post-response-mode]).

4.3. Credential Leakage via Browser History

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

4.3.1. Authorization Code in Browser History

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

Countermeasures:

* Authorization code replay prevention as described in [RFC6819], Section 4.4.1.1, and Section 4.5.

* Use form post response mode instead of redirect for the authorization response (see [oauth-v2-form-post-response-mode]).

4.3.2. Access Token in Browser History

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

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

Countermeasures:

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

4.4. Mix-Up Attacks

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

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

4.4.1. Attack Description

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

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

* the implicit or authorization code grant are used with multiple AS of which one is considered "honest" (H-AS) and one is operated by the attacker (A-AS),

* 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

* the attacker can intercept and manipulate the first request/response pair from a user's browser to the client (in which the user selects a certain AS and is then redirected by the client to that AS), as in Attacker A2.

The latter ability can, for example, be the result of a man-in-the-middle attack on the user's connection to the client. Note that an attack variant exists that does not require this ability, see below.

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" (see preconditions).

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 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. 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 an authorization code injection attack as described in Section 4.5.

Variants:

* *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, it is assumed 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"). Note that 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.

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

* *Per-AS Redirect URIs*: If clients use different redirect URIs for different ASs, do not store the selected AS in the user's session, and ASs do not check the redirect URIs properly, attackers can mount an attack called "Cross-Social Network Request Forgery". These attacks have been observed in practice. Refer to [oauth_security_jcs_14] for details.

* *OpenID Connect*: There are variants that can be used to attack OpenID Connect. In these attacks, the attacker misuses features of the OpenID Connect Discovery mechanism or replays access tokens or ID Tokens to conduct a Mix-Up Attack. The attacks are described in detail in [arXiv.1704.08539], Appendix A, and [arXiv.1508.04324v2], Section 6 ("Malicious Endpoints Attacks").

4.4.2. Countermeasures

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

To this end, clients SHOULD use distinct redirect URIs for each AS (with alternatives listed below). Clients MUST store, for each authorization request, the AS they sent the authorization request to and bind this information to the user agent. Clients MUST check that the authorization request was received from the correct authorization server and ensure that the subsequent token request, if applicable, is sent to the same authorization server.

Unfortunately, distinct redirect URIs per AS do not work for all kinds of OAuth clients. They are effective for web and JavaScript 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.

If clients cannot use distinct redirect URIs for each AS, the following options exist:

* Authorization servers can be configured to return an AS identitifier ("iss") as a non-standard parameter in the authorization response. This enables complying clients to compare this data to the "iss" identifier of the AS it believed it sent the user agent to.

* In OpenID Connect, if an ID Token is returned in the authorization response, it carries client ID and issuer. It can be used in the same way as the "iss" parameter.

4.5. Authorization Code Injection

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

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

* The code is bound to a particular confidential client and the attacker is unable to obtain the required client credentials to redeem the code himself.

* The attacker wants to access certain functions in this particular client. As an example, the attacker wants to impersonate his victim in a certain app or on a certain web site.

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

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

4.5.1. Attack Description

The attack works as follows:

1. The attacker obtains an authorization code by performing any of the attacks described above.

2. He starts a regular OAuth authorization process with the legitimate client from his device.

3. The attacker injects the stolen authorization code in the response of the authorization server to the legitimate client. Since this response is passing through the attacker's device, the attacker can use any tool that can intercept and manipulate the authorization response to this end. The attacker does not need to control the network.

4. The legitimate client sends the code to the authorization server's token endpoint, along with the client's client ID, client secret and actual "redirect_uri".

5. The authorization server checks the client secret, whether the code was issued to the particular client, and whether the actual redirect URI matches the "redirect_uri" parameter (see [RFC6749]).

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

4.5.2. Discussion

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

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

[RFC6749], Section 4.1.3, requires the AS to "... ensure that the "redirect_uri" parameter is present if the "redirect_uri" parameter was included in the initial authorization request as described in Section 4.1.1, and if included ensure that their values are identical.". In the attack scenario described above, the legitimate client would use the correct redirect URI it always uses for authorization requests. But this URI would not match the tampered redirect URI used by the attacker (otherwise, the redirect would not land at the attackers page). So the authorization server would detect the attack and refuse to exchange the code.

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

* the redirect URI itself needs to contain a nonce or another kind of one-time use, secret data and

* the client has bound this data to this particular instance of the client.

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

Other providers just pattern match the "redirect_uri" parameter against the registered redirect URI pattern. This saves the authorization server from storing the link between the actual redirect URI and the respective authorization code for every transaction. But this kind of check obviously does not fulfill the intent of the specification, since the tampered redirect URI is not considered. So any attempt to inject an authorization code obtained using the "client_id" of a legitimate client or by utilizing the legitimate client on another device will not be detected in the respective deployments.

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

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

4.5.3. Countermeasures

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

4.5.3.1. PKCE

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

4.5.3.2. Nonce

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

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

1. MUST validate the "nonce" in the ID Token obtained from the token endpoint, even if another ID Token was obtained from the authorization response (e.g., "response_type=code+id_token"), and

2. MUST ensure that, unless and until that check succeeds, all tokens (ID Tokens and the access token) are disregarded and not used for any other purpose.

4.5.3.3. Other Solutions

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

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

4.5.4. Limitations

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

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

4.6. Access Token Injection

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

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

4.6.1. Countermeasures

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

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

4.7. Cross Site Request Forgery

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

4.7.1. Countermeasures

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

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

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

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

AS therefore MUST provide a way to detect their support for PKCE either via AS metadata according to [RFC8414] or provide a deployment-specific way to ensure or determine PKCE support.

4.8. PKCE Downgrade Attack

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

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

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

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

4.8.1. Attack Description

1. The user has started an OAuth session using some client at an AS. In the authorization request, the client has set the parameter "code_challenge=sha256(abc)" as the PKCE code challenge. The client is now waiting to receive the authorization response from the user's browse.

2. To conduct the attack, the attacker uses his own device to start an authorization flow with the targeted client. The client now uses another PKCE code challenge, say "code_challenge=sha256(xyz)", in the authorization request. The attacker intercepts the request and removes the entire "code_challenge" parameter from the request. Since this step is performed on the attacker's device, the attacker has full access to the request contents, for example using browser debug tools.

3. If the authorization server allows for flows without PKCE, it will create a code that is not bound to any PKCE code challenge.

4. The attacker now redirects the user's browser to an authorization response URL which contains the code for the attacker's session with the AS.

5. The user's browser sends the authorization code to the client, which will now try to redeem the code for an access token at the AS. The client will send "code_verifier=abc" as the PKCE code verifier in the token request.

6. Since the authorization server sees that this code is not bound to any PKCE code challenge, it will not check the presence or contents of the "code_verifier" parameter. It will issue an access token that belongs to the attacker's resource to the client under the user's control.

4.8.2. Countermeasures

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

Therefore, AS MUST take precautions against this threat.

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

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

* an AS that supports PKCE MUST check whether a code challenge is contained in the authorization request and bind this information to the code that is issued; and

* when a code arrives at the token endpoint, and there was a "code_challenge" in the authorization request for which this code was issued, there must be a valid "code_verifier" in the token request.

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

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

4.9. Access Token Leakage at the Resource Server

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

4.9.1. Access Token Phishing by Counterfeit Resource Server

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

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

4.9.1.1. Countermeasures

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

4.9.1.1.1. Metadata

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

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

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

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

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

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

{
  "access_token":"2YotnFZFEjr1zCsicMWpAA",
  "access_token_resource_server":
    "https://hostedresource.somesite.example/path1",
...
}
```

This mitigation strategy would rely on the client to enforce the security policy and to only send access tokens to legitimate destinations. Results of OAuth related security research (see for example [oauth_security_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.

4.9.1.1.2. Sender-Constrained Access Tokens

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

A typical flow looks like this:

1. The authorization server associates data with the access token that binds this particular token to a certain client. The binding can utilize the client identity, but in most cases the AS utilizes key material (or data derived from the key material) known to the client.

2. This key material must be distributed somehow. Either the key material already exists before the AS creates the binding or the AS creates ephemeral keys. The way pre-existing key material is

distributed varies among the different approaches. For example, X.509 Certificates can be used in which case the distribution happens explicitly during the enrollment process. Or the key material is created and distributed at the TLS layer, in which case it might automatically happen during the setup of a TLS connection.

3. The RS must implement the actual proof of possession check. This is typically done on the application level, often tied to specific material provided by transport layer (e.g., TLS). The RS must also ensure that replay of the proof of possession is not possible.

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

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

* *DPoP* ([I-D.ietf-oauth-dpop]): DPoP (Demonstration of Proof-of-Possession at the Application Layer) outlines an application-level sender-constraining for access and refresh tokens that can be used in cases where neither mTLS nor OAuth Token Binding (see below) are available. It uses proof-of-possession based on a public/private key pair and application-level signing. DPoP can be used with public clients and, in case of confidential clients, can be combined with any client authentication method.

* *OAuth Token Binding* ([I-D.ietf-oauth-token-binding]): In this approach, an access token is, via the token binding ID, bound to key material representing a long term association between a client and a certain TLS host. Negotiation of the key material and proof of possession in the context of a TLS handshake is taken care of by the TLS stack. The client needs to determine the token binding ID of the target resource server and pass this data to the access token request. The authorization server then associates the access token with this ID. The resource server checks on every invocation that the token binding ID of the active TLS connection

and the token binding ID of associated with the access token match. Since all crypto-related functions are covered by the TLS stack, this approach is very client developer friendly. As a prerequisite, token binding as described in [RFC8473] (including federated token bindings) must be supported on all ends (client, authorization server, resource server).

* *Signed HTTP Requests* ([I-D.ietf-oauth-signed-http-request]): This approach utilizes [I-D.ietf-oauth-pop-key-distribution] and represents the elements of the signature in a JSON object. The signature is built using JWS. The mechanism has built-in support for signing of HTTP method, query parameters and headers. It also incorporates a timestamp as basis for replay prevention.

* *JWT Pop Tokens* ([I-D.sakimura-oauth-jpop]): This draft describes different ways to constrain access token usage, namely TLS or request signing. Note: Since the authors of this draft contributed the TLS-related proposal to [RFC8705], this document only considers the request signing part. For request signing, the draft utilizes [I-D.ietf-oauth-pop-key-distribution] and [RFC7800]. The signature data is represented in a JWT and JWS is used for signing. Replay prevention is provided by building the signature over a server-provided nonce, client-provided nonce and a nonce counter.

At the time of writing, OAuth Mutual TLS is the most widely implemented and the only standardized sender-constraining method. The use of OAuth Mutual TLS therefore is RECOMMENDED.

Note that the security of sender-constrained tokens is undermined when an attacker gets access to the token and the key material. This is in particular the case for corrupted client software and cross-site scripting attacks (when the client is running in the browser). If the key material is protected in a hardware or software security module or only indirectly accessible (like in a TLS stack), sender-constrained tokens at least protect against a use of the token when the client is offline, i.e., when the security module or interface is not available to the attacker. This applies to access tokens as well as to refresh tokens (see Section 4.13).

4.9.1.1.3. Audience Restricted Access Tokens

Audience restriction essentially restricts access tokens to a particular resource server. The authorization server associates the access token with the particular resource server and the resource server SHOULD verify the intended audience. If the access token fails the intended audience validation, the resource server must refuse to serve the respective request.

In general, audience restrictions limit the impact of token leakage. In the case of a counterfeit resource server, it may (as described below) also prevent abuse of the phished access token at the legitimate resource server.

The audience can be expressed using logical names or physical addresses (like URLs). In order to prevent phishing, it is necessary to use the actual URL the client will send requests to. In the phishing case, this URL will point to the counterfeit resource server. If the attacker tries to use the access token at the legitimate resource server (which has a different URL), the resource server will detect the mismatch (wrong audience) and refuse to serve the request.

In deployments where the authorization server knows the URLs of all resource servers, the authorization server may just refuse to issue access tokens for unknown resource server URLs.

The client SHOULD tell the authorization server the intended resource server. The proposed mechanism [I-D.ietf-oauth-resource-indicators] could be used or by encoding the information in the scope value.

Instead of the URL, it is also possible to utilize the fingerprint of the resource server's X.509 certificate as audience value. This variant would also allow to detect an attempt to spoof the legitimate resource server's URL by using a valid TLS certificate obtained from a different CA. It might also be considered a privacy benefit to hide the resource server URL from the authorization server.

Audience restriction may seem easier to use since it does not require any crypto on the client-side. Still, since every access token is bound to a specific resource server, the client also needs to obtain a single RS-specific access token when accessing several resource servers. (Resource indicators, as specified in [I-D.ietf-oauth-resource-indicators], can help to achieve this.) [I-D.ietf-oauth-token-binding] has the same property since different token binding ids must be associated with the access token. Using [RFC8705], on the other hand, allows a client to use the access token at multiple resource servers.

It shall be noted that audience restrictions, or generally speaking an indication by the client to the authorization server where it wants to use the access token, has additional benefits beyond the scope of token leakage prevention. It allows the authorization server to create different access token whose format and content is specifically minted for the respective server. This has huge functional and privacy advantages in deployments using structured access tokens.

4.9.2. Compromised Resource Server

An attacker may compromise a resource server to gain access to the resources of the respective deployment. Such a compromise may range from partial access to the system, e.g., its log files, to full control of the respective server.

If the attacker were able to gain full control, including shell access, all controls can be circumvented and all resources be accessed. The attacker would also be able to obtain other access tokens held on the compromised system that would potentially be valid to access other resource servers.

Preventing server breaches by hardening and monitoring server systems is considered a standard operational procedure and, therefore, out of the scope of this document. This section focuses on the impact of OAuth-related breaches and the replaying of captured access tokens.

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

* Sender-constrained access tokens as described in Section 4.9.1.1.2 SHOULD be used to prevent the attacker from replaying the access tokens on other resource servers. Depending on the severity of the penetration, sender-constrained access tokens will also prevent replay on the compromised system.

* Audience restriction as described in Section 4.9.1.1.3 SHOULD be used to prevent replay of captured access tokens on other resource servers.

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

The first and second recommendation also apply to other scenarios where access tokens leak (see Attacker A5).

4.10. Open Redirection

The following attacks can occur when an AS or client has an open redirector. An open redirector is an endpoint that forwards a user's browser to an arbitrary URI obtained from a query parameter.

4.10.1. Client as Open Redirector

Clients MUST NOT expose open redirectors. Attackers may use open redirectors to produce URLs pointing to the client and utilize them to exfiltrate authorization codes and access tokens, as described in Section 4.1.2. Another abuse case is to produce URLs that appear to point to the client. This might trick users into trusting the URL and follow it in their browser. This can be abused for phishing.

In order to prevent open redirection, clients should only redirect if the target URLs are whitelisted or if the origin and integrity of a request can be authenticated. Countermeasures against open redirection are described by OWASP [owasp_redir].

4.10.2. Authorization Server as Open Redirector

Just as with clients, attackers could try to utilize a user's trust in the authorization server (and its URL in particular) for performing phishing attacks. OAuth authorization servers regularly redirect users to other web sites (the clients), but must do so in a safe way.

[RFC6749], Section 4.1.2.1, already prevents open redirects by stating that the AS MUST NOT automatically redirect the user agent in case of an invalid combination of "client_id" and "redirect_uri".

However, an attacker could also utilize a correctly registered redirect URI to perform phishing attacks. The attacker could, for example, register a client via dynamic client registration [RFC7591] and intentionally send an erroneous authorization request, e.g., by using an invalid scope value, thus instructing the AS to redirect the user agent to its phishing site.

The AS MUST take precautions to prevent this threat. Based on its risk assessment, the AS needs to decide whether it can trust the redirect URI and SHOULD only automatically redirect the user agent if it trusts the redirect URI. If the URI is not trusted, the AS MAY inform the user and rely on the user to make the correct decision.

4.11. 307 Redirect

At the authorization endpoint, a typical protocol flow is that the AS prompts the user to enter her credentials in a form that is then submitted (using the HTTP POST method) back to the authorization server. The AS checks the credentials and, if successful, redirects the user agent to the client's redirection endpoint.

In [RFC6749], the HTTP status code 302 is used for this purpose, but "any other method available via the user-agent to accomplish this redirection is allowed". When the status code 307 is used for redirection instead, the user agent will send the user credentials via HTTP POST to the client.

This discloses the sensitive credentials to the client. If the relying party is malicious, it can use the credentials to impersonate the user at the AS.

The behavior might be unexpected for developers, but is defined in [RFC7231], Section 6.4.7. This status code does not require the user agent to rewrite the POST request to a GET request and thereby drop the form data in the POST request body.

In the HTTP standard [RFC7231], only the status code 303 unambigiously enforces rewriting the HTTP POST request to an HTTP GET request. For all other status codes, including the popular 302, user agents can opt not to rewrite POST to GET requests and therefore to reveal the user credentials to the client. (In practice, however, most user agents will only show this behaviour for 307 redirects.)

AS which redirect a request that potentially contains user credentials therefore MUST NOT use the HTTP 307 status code for redirection. If an HTTP redirection (and not, for example, JavaScript) is used for such a request, AS SHOULD use HTTP status code 303 "See Other".

4.12. TLS Terminating Reverse Proxies

A common deployment architecture for HTTP applications is to hide the application server behind a reverse proxy that terminates the TLS connection and dispatches the incoming requests to the respective application server nodes.

This section highlights some attack angles of this deployment architecture with relevance to OAuth and gives recommendations for security controls.

In some situations, the reverse proxy needs to pass security-related data to the upstream application servers for further processing. Examples include the IP address of the request originator, token binding ids, and authenticated TLS client certificates. This data is usually passed in custom HTTP headers added to the upstream request.

If the reverse proxy would pass through any header sent from the outside, an attacker could try to directly send the faked header values through the proxy to the application server in order to circumvent security controls that way. For example, it is standard practice of reverse proxies to accept "X-Forwarded-For" headers and just add the origin of the inbound request (making it a list). Depending on the logic performed in the application server, the attacker could simply add a whitelisted IP address to the header and render a IP whitelist useless.

A reverse proxy must therefore sanitize any inbound requests to ensure the authenticity and integrity of all header values relevant for the security of the application servers.

If an attacker was able to get access to the internal network between proxy and application server, the attacker could also try to circumvent security controls in place. It is, therefore, essential to ensure the authenticity of the communicating entities. Furthermore, the communication link between reverse proxy and application server must be protected against eavesdropping, injection, and replay of messages.

4.13. Refresh Token Protection

Refresh tokens are a convenient and user-friendly way to obtain new access tokens after the expiration of access tokens. Refresh tokens also add to the security of OAuth since they allow the authorization server to issue access tokens with a short lifetime and reduced scope thus reducing the potential impact of access token leakage.

4.13.1. Discussion

Refresh tokens are an attractive target for attackers since they represent the overall grant a resource owner delegated to a certain client. If an attacker is able to exfiltrate and successfully replay a refresh token, the attacker will be able to mint access tokens and use them to access resource servers on behalf of the resource owner.

[RFC6749] already provides a robust baseline protection by requiring

* confidentiality of the refresh tokens in transit and storage,

* the transmission of refresh tokens over TLS-protected connections between authorization server and client,

* the authorization server to maintain and check the binding of a refresh token to a certain client (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 specification gives recommendations beyond the scope of [RFC6749] and clarifications.

4.13.2. Recommendations

Authorization servers SHOULD determine, based on a risk assessment, whether to issue refresh tokens to a certain client. If the authorization server decides not to issue refresh tokens, the client MAY refresh access tokens by utilizing other grant types, such as the authorization code grant type. In such a case, the authorization server may utilize cookies and persistent grants to optimize the user experience.

If refresh tokens are issued, those refresh tokens MUST be bound to the scope and resource servers as consented by the resource owner. This is to prevent privilege escalation by the legitimate client and reduce the impact of refresh token leakage.

For confidential clients, [RFC6749] already requires that refresh tokens can only be used by the client for which they were issued.

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

* *Sender-constrained refresh tokens:* the authorization server cryptographically binds the refresh token to a certain client instance by utilizing [RFC8705] or [I-D.ietf-oauth-token-binding].

* *Refresh token rotation:* the authorization server issues a new refresh token with every access token refresh response. The previous refresh token is invalidated but information about the relationship is retained by the authorization server. If a refresh token is compromised and subsequently used by both the attacker and the legitimate client, one of them will present an invalidated refresh token, which will inform the authorization server of the breach. The authorization server cannot determine which party submitted the invalid refresh token, but it will revoke the active refresh token. This stops the attack at the cost of forcing the legitimate client to obtain a fresh authorization grant.

Implementation note: the grant to which a refresh token belongs may be encoded into the refresh token itself. This can enable an authorization server to efficiently determine the grant to which a refresh token belongs, and by extension, all refresh tokens that need to be revoked. Authorization servers MUST ensure the integrity of the refresh token value in this case, for example, using signatures.

Authorization servers MAY revoke refresh tokens automatically in case of a security event, such as:

* password change

* logout at the authorization server

Refresh tokens SHOULD expire if the client has been inactive for some time, i.e., the refresh token has not been used to obtain fresh access tokens for some time. The expiration time is at the discretion of the authorization server. It might be a global value or determined based on the client policy or the grant associated with the refresh token (and its sensitivity).

4.14. Client Impersonating Resource Owner

Resource servers may make access control decisions based on the identity of the resource owner as communicated in the "sub" claim returned by the authorization server in a token introspection response [RFC7662] or other mechanisms. If a client is able to choose its own "client_id" during registration with the authorization server, then there is a risk that it can register with the same "sub" value as a privileged user. A subsequent access token obtained under the client credentials grant may be mistaken for an access token authorized by the privileged user if the resource server does not perform additional checks.

4.14.1. Countermeasures

Authorization servers SHOULD NOT allow clients to influence their "client_id" or "sub" value or any other claim if that can cause confusion with a genuine resource owner. Where this cannot be avoided, authorization servers MUST provide other means for the resource server to distinguish between access tokens authorized by a resource owner from access tokens authorized by the client itself.

4.15. Clickjacking

As described in Section 4.4.1.9 of [RFC6819], the authorization request is susceptible to clickjacking. An attacker can use this vector to obtain the user's authentication credentials, change the scope of access granted to the client, and potentially access the user's resources.

Authorization servers MUST prevent clickjacking attacks. Multiple countermeasures are described in [RFC6819], including the use of the X-Frame-Options HTTP response header field and frame-busting JavaScript. In addition to those, authorization servers SHOULD also use Content Security Policy (CSP) level 2 [CSP-2] or greater.

To be effective, CSP must be used on the authorization endpoint and, if applicable, other endpoints used to authenticate the user and authorize the client (e.g., the device authorization endpoint, login pages, error pages, etc.). This prevents framing by unauthorized origins in user agents that support CSP. The client MAY permit being framed by some other origin than the one used in its redirection endpoint. For this reason, authorization servers SHOULD allow administrators to configure allowed origins for particular clients and/or for clients to register these dynamically.

Using CSP allows authorization servers to specify multiple origins in a single response header field and to constrain these using flexible patterns (see [CSP-2] for details). Level 2 of this standard provides a robust mechanism for protecting against clickjacking by using policies that restrict the origin of frames (using "frame-ancestors") together with those that restrict the sources of scripts allowed to execute on an HTML page (by using "script-src"). A non-normative example of such a policy is shown in the following listing:

```
HTTP/1.1 200 OK
Content-Security-Policy: frame-ancestors https://ext.example.org:8000
Content-Security-Policy: script-src 'self'
X-Frame-Options: ALLOW-FROM https://ext.example.org:8000
...
```

Because some user agents do not support [CSP-2], this technique SHOULD be combined with others, including those described in [RFC6819], unless such legacy user agents are explicitly unsupported by the authorization server. Even in such cases, additional countermeasures SHOULD still be employed.

5. Acknowledgements

We would like to thank Jim Manico, Phil Hunt, Nat Sakimura, Christian Mainka, Doug McDorman, Johan Peeters, Joseph Heenan, Brock Allen, Vittorio Bertocci, David Waite, Nov Matake, Tomek Stojecki, Dominick Baier, Neil Madden, William Dennis, Dick Hardt, Petteri Stenius, Annabelle Richard Backman, Aaron Parecki, George Fletscher, Brian Campbell, Konstantin Lapine, Tim Würtele, Guido Schmitz, Hans Zandbelt, Jared Jennings, Michael Peck, Pedram Hosseyni, Michael B. Jones, and Travis Spencer for their valuable feedback.

6. IANA Considerations

This draft includes no request to IANA.

7. Security Considerations

All relevant security considerations have been given in the functional specification.

8. Normative References

[RFC8252] Denniss, W. and J. Bradley, "OAuth 2.0 for Native Apps", BCP 212, RFC 8252, DOI 10.17487/RFC8252, October 2017, <https://www.rfc-editor.org/info/rfc8252>.

[RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, <https://www.rfc-editor.org/info/rfc6750>.

[RFC8414] Jones, M., Sakimura, N., and J. Bradley, "OAuth 2.0 Authorization Server Metadata", RFC 8414, DOI 10.17487/RFC8414, June 2018, <https://www.rfc-editor.org/info/rfc8414>.

[OpenID] Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0 incorporating errata set 1", 8 November 2014, <http://openid.net/specs/openid-connect-core-1_0.html>.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <https://www.rfc-editor.org/info/rfc7231>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[oauth-v2-form-post-response-mode] Jones, M. and B. Campbell, "OAuth 2.0 Form Post Response Mode", 27 April 2015, <http://openid.net/specs/oauth-v2-form-post-response-mode-1_0.html>.

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

[RFC6819] Lodderstedt, T., Ed., McGloin, M., and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations", RFC 6819, DOI 10.17487/RFC6819, January 2013, <https://www.rfc-editor.org/info/rfc6819>.

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

[RFC8705] Campbell, B., Bradley, J., Sakimura, N., and T. Lodderstedt, "OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens", February 2020, <https://www.rfc-editor.org/info/rfc8705>.

9. Informative References

[CSP-2] West, M., Barth, A., and D. Veditz, "Content Security Policy Level 2", July 2015, <https://www.w3.org/TR/CSP2>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[arXiv.1901.11520] Fett, D., Hosseyni, P., and R. Küsters, "An Extensive Formal Security Analysis of the OpenID Financial-grade API", 31 January 2019, <http://arxiv.org/abs/1901.11520/>.

[I-D.ietf-oauth-jwsreq] Sakimura, N., Bradley, J., and M. Jones, "The OAuth 2.0 Authorization Framework: JWT Secured Authorization Request (JAR)", Work in Progress, Internet-Draft, draft-ietf-oauth-jwsreq-30, 10 September 2020, <https://tools.ietf.org/html/draft-ietf-oauth-jwsreq-30>.

[oauth_security_ubc]
Sun, S.-T. and K. Beznosov, "The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems", October 2012, <http://passwordresearch.com/papers/paper267.html>.

[oauth_security_cmu]
Chen, E., Pei, Y., Chen, S., Tian, Y., Kotcher, R., and P. Tague, "OAuth Demystified for Mobile Application Developers", November 2014, <http://css.csail.mit.edu/6.858/2012/readings/oauth-sso.pdf>.

[RFC7800] Jones, M., Bradley, J., and H. Tschofenig, "Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs)", RFC 7800, DOI 10.17487/RFC7800, April 2016, <https://www.rfc-editor.org/info/rfc7800>.

[I-D.ietf-oauth-dpop]
Fett, D., Campbell, B., Bradley, J., Lodderstedt, T., Jones, M., and D. Waite, "OAuth 2.0 Demonstration of Proof-of-Possession at the Application Layer (DPoP)", Work in Progress, Internet-Draft, draft-ietf-oauth-dpop-01, 1 May 2020, <https://tools.ietf.org/html/draft-ietf-oauth-dpop-01>.

[I-D.ietf-oauth-signed-http-request]
Richer, J., Bradley, J., and H. Tschofenig, "A Method for Signing HTTP Requests for OAuth", Work in Progress, Internet-Draft, draft-ietf-oauth-signed-http-request-03, 8 August 2016, <https://tools.ietf.org/html/draft-ietf-oauth-signed-http-request-03>.

[RFC7591] Richer, J., Ed., Jones, M., Bradley, J., Machulak, M., and P. Hunt, "OAuth 2.0 Dynamic Client Registration Protocol", RFC 7591, DOI 10.17487/RFC7591, July 2015, <https://www.rfc-editor.org/info/rfc7591>.

[I-D.ietf-oauth-rar]
Lodderstedt, T., Richer, J., and B. Campbell, "OAuth 2.0 Rich Authorization Requests", Work in Progress, Internet-Draft, draft-ietf-oauth-rar-02, 21 August 2020, <https://tools.ietf.org/html/draft-ietf-oauth-rar-02>.

[bug.chromium]
"Referer header includes URL fragment when opening link using New Tab", <https://bugs.chromium.org/p/chromium/issues/detail?id=168213/>.

[I-D.bradley-oauth-jwt-encoded-state]
Bradley, J., Lodderstedt, T., and H. Zandbelt, "Encoding claims in the OAuth 2 state parameter using a JWT", Work in Progress, Internet-Draft, draft-bradley-oauth-jwt-encoded-state-09, 4 November 2018, <https://tools.ietf.org/html/draft-bradley-oauth-jwt-encoded-state-09>.

[RFC7636] Sakimura, N., Ed., Bradley, J., and N. Agarwal, "Proof Key for Code Exchange by OAuth Public Clients", RFC 7636, DOI 10.17487/RFC7636, September 2015, <https://www.rfc-editor.org/info/rfc7636>.

[webauthn] Balfanz, D., Czeskis, A., Hodges, J., Jones, J.C., Jones, M.B., Kumar, A., Liao, A., Lindemann, R., and E. Lundberg, "Web Authentication: An API for accessing Public Key Credentials Level 1", 4 March 2019, <https://www.w3.org/TR/2019/REC-webauthn-1-20190304/>.

[I-D.sakimura-oauth-jpop]
Sakimura, N., Li, K., and J. Bradley, "The OAuth 2.0 Authorization Framework: JWT Pop Token Usage", Work in Progress, Internet-Draft, draft-sakimura-oauth-jpop-05, 22 July 2019, <https://tools.ietf.org/html/draft-sakimura-oauth-jpop-05>.

[webcrypto]
Watson, M., "Web Cryptography API", 26 January 2017, <https://www.w3.org/TR/2017/REC-WebCryptoAPI-20170126/>.

[I-D.ietf-oauth-par]
Lodderstedt, T., Campbell, B., Sakimura, N., Tonge, D., and F. Skokan, "OAuth 2.0 Pushed Authorization Requests", Work in Progress, Internet-Draft, draft-ietf-oauth-par-04, 18 September 2020, <https://tools.ietf.org/html/draft-ietf-oauth-par-04>.

[webappsec-referrer-policy]
Eisinger, J. and E. Stark, "Referrer Policy", 20 April 2017, <https://w3c.github.io/webappsec-referrer-policy>.

[arXiv.1704.08539]
Fett, D., Küsters, R., and G. Schmitz, "The Web SSO Standard OpenID Connect: In-Depth Formal Security Analysis and Security Guidelines", 27 April 2017, <http://arxiv.org/abs/1704.08539/>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[I-D.ietf-oauth-resource-indicators]
Campbell, B., Bradley, J., and H. Tschofenig, "Resource Indicators for OAuth 2.0", Work in Progress, Internet-Draft, draft-ietf-oauth-resource-indicators-08, 11 September 2019, <https://tools.ietf.org/html/draft-ietf-oauth-resource-indicators-08>.

[I-D.ietf-oauth-token-binding]
Jones, M., Campbell, B., Bradley, J., and W. Denniss, "OAuth 2.0 Token Binding", Work in Progress, Internet-Draft, draft-ietf-oauth-token-binding-08, 19 October 2018, <https://tools.ietf.org/html/draft-ietf-oauth-token-binding-08>.

[arXiv.1601.01229]
Fett, D., Küsters, R., and G. Schmitz, "A Comprehensive Formal Security Analysis of OAuth 2.0", 6 January 2016, <http://arxiv.org/abs/1601.01229/>.

[subdomaintakeover]
Liu, D., Hao, S., and H. Wang, "All Your DNS Records Point to Us: Understanding the Security Threats of Dangling DNS Records", 24 October 2016, <https://www.eecis.udel.edu/˜hnw/paper/ccs16a.pdf>.

[owasp_redir]
"OWASP Cheat Sheet Series - Unvalidated Redirects and Forwards", <https://cheatsheetseries.owasp.org/cheatsheets/Unvalidated_Redirects_and_Forwards_Cheat_Sheet.html>.

[oauth_security_jcs_14]
Bansal, C., Bhargavan, K., Delignat-Lavaud, A., and S. Maffeis, "Discovering concrete attacks on website authorization by formal analysis", 23 April 2014, <https://www.doc.ic.ac.uk/˜maffeis/papers/jcs14.pdf>.

[arXiv.1508.04324v2]
Mladenov, V., Mainka, C., and J. Schwenk, "On the security of modern Single Sign-On Protocols: Second-Order Vulnerabilities in OpenID Connect", 7 January 2016, <http://arxiv.org/abs/1508.04324v2/>.

[RFC8473] Popov, A., Nystroem, M., Balfanz, D., Ed., Harper, N., and J. Hodges, "Token Binding over HTTP", RFC 8473, DOI 10.17487/RFC8473, October 2018, <https://www.rfc-editor.org/info/rfc8473>.

[I-D.ietf-oauth-pop-key-distribution]
Bradley, J., Hunt, P., Jones, M., Tschofenig, H., and M. Meszaros, "OAuth 2.0 Proof-of-Possession: Authorization Server to Client Key Distribution", Work in Progress, Internet-Draft, draft-ietf-oauth-pop-key-distribution-07, 27 March 2019, <https://tools.ietf.org/html/draft-ietf-oauth-pop-key-distribution-07>.

Appendix A. Document History

[[To be removed from the final specification]]

-16

* Make MTLS a suggestion, not RECOMMENDED.

* Add important requirements when using nonce for code injection protection.

* Highlight requirements for refresh token sender-constraining.

* Make PKCE a MUST for public clients.

* Describe PKCE Downgrade Attacks and countermeasures.

* Allow variable port numbers in localhost redirect URIs as in RFC8252, Section 7.3.

-15

* Update reference to DPoP

* Fix reference to RFC8414

* Move to xml2rfcv3

-14

* Added info about using CSP to prevent clickjacking

* Changes from WGLC feedback

* Editorial changes

* AS MUST announce PKCE support either in metadata or using deployment-specific ways (before: SHOULD)

-13

* Discourage use of Resource Owner Password Credentials Grant

* Added text on client impersonating resource owner

* Recommend asymmetric methods for client authentication

* Encourage use of PKCE mode "S256"

* PKCE may replace state for CSRF protection

* AS SHOULD publish PKCE support

* Cleaned up discussion on auth code injection

* AS MUST support PKCE

-12

* Added updated attacker model

-11

* Adapted section 2.1.2 to outcome of consensus call

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

-10

* incorporated feedback by Joseph Heenan

* changed occurrences of SHALL to MUST

* added text on lack of token/cert binding support tokens issued in the authorization response as justification to not recommend issuing tokens there at all

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

* added section on refresh tokens

* editorial enhancements to 2.1.2 based on feedback

-09

* changed text to recommend not to use implicit but code

* added section on access token injection

* reworked sections 3.1 through 3.3 to be more specific on implicit grant issues

-08

* added recommendations re implicit and token injection

* uppercased key words in Section 2 according to RFC 2119

-07

* incorporated findings of Doug McDorman

* added section on HTTP status codes for redirects

* added new section on access token privilege restriction based on comments from Johan Peeters

-06

* reworked section 3.8.1

* incorporated Phil Hunt's feedback

* reworked section on mix-up

* extended section on code leakage via referrer header to also cover state leakage

* added Daniel Fett as author

* replaced text intended to inform WG discussion by recommendations to implementors

* modified example URLs to conform to RFC 2606

-05

* Completed sections on code leakage via referrer header, attacks in browser, mix-up, and CSRF

* Reworked Code Injection Section

* Added reference to OpenID Connect spec

* removed refresh token leakage as respective considerations have been given in section 10.4 of RFC 6749

* first version on open redirection

* incorporated Christian Mainka's review feedback

-04

* Restructured document for better readability

* Added best practices on Token Leakage prevention

-03

* Added section on Access Token Leakage at Resource Server

* incorporated Brian Campbell's findings

-02

* Folded Mix up and Access Token leakage through a bad AS into new section for dynamic OAuth threats

* reworked dynamic OAuth section

-01

* Added references to mitigation methods for token leakage

* Added reference to Token Binding for Authorization Code

* incorporated feedback of Phil Hunt

* fixed numbering issue in attack descriptions in section 2

-00 (WG document)

* turned the ID into a WG document and a BCP

* Added federated app login as topic in Other Topics

Authors' Addresses

Torsten Lodderstedt
yes.com

Email: torsten@lodderstedt.net

John Bradley
Yubico

Email: ve7jtb@ve7jtb.com

Andrey Labunets

Email: isciurus@gmail.com

Daniel Fett
yes.com

Email: mail@danielfett.de

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 22, 2019

M. Jones
Microsoft
B. Campbell
Ping Identity
J. Bradley
Yubico
W. Denniss
Google

October 19, 2018

OAuth 2.0 Token Binding
draft-ietf-oauth-token-binding-08

Abstract

This specification enables OAuth 2.0 implementations to apply Token Binding to Access Tokens, Authorization Codes, Refresh Tokens, JWT Authorization Grants, and JWT Client Authentication. This cryptographically binds these tokens to a client's Token Binding key pair, possession of which is proven on the TLS connections over which the tokens are intended to be used. This use of Token Binding protects these tokens from man-in-the-middle and token export and replay attacks.

Table of Contents

1. Introduction

This specification enables OAuth 2.0 [RFC6749] implementations to apply Token Binding (TLS Extension for Token Binding Protocol Negotiation [RFC8472], The Token Binding Protocol Version 1.0 [RFC8471] and Token Binding over HTTP [RFC8473]) to Access Tokens, Authorization Codes, Refresh Tokens, JWT Authorization Grants, and JWT Client Authentication. This cryptographically binds these tokens to a client's Token Binding key pair, possession of which is proven on the TLS connections over which the tokens are intended to be used. This use of Token Binding protects these tokens from man-in-the-middle and token export and replay attacks.

1.1. Requirements Notation and Conventions

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

1.2. Terminology

This specification uses the terms "Access Token", "Authorization Code", "Authorization Endpoint", "Authorization Server", "Client", "Protected Resource", "Refresh Token", and "Token Endpoint" defined by OAuth 2.0 [RFC6749], the terms "Claim" and "JSON Web Token (JWT)" defined by JSON Web Token (JWT) [JWT], the term "User Agent" defined by RFC 7230 [RFC7230], and the terms "Provided", "Referred", "Token

Binding" and "Token Binding ID" defined by Token Binding over HTTP [RFC8473].

2. Token Binding for Refresh Tokens

Token Binding of refresh tokens is a straightforward first-party scenario, applying term "first-party" as used in Token Binding over HTTP [RFC8473]. It cryptographically binds the refresh token to the client's Token Binding key pair, possession of which is proven on the TLS connections between the client and the token endpoint. This case is straightforward because the refresh token is both retrieved by the client from the token endpoint and sent by the client to the token endpoint. Unlike the federation use cases described in Token Binding over HTTP [RFC8473], Section 4, and the access token case described in the next section, only a single TLS connection is involved in the refresh token case.

Token Binding a refresh token requires that the authorization server do two things. First, when refresh token is sent to the client, the authorization server needs to remember the Provided Token Binding ID and remember its association with the issued refresh token. Second, when a token request containing a refresh token is received at the token endpoint, the authorization server needs to verify that the Provided Token Binding ID for the request matches the remembered Token Binding ID associated with the refresh token. If the Token Binding IDs do not match, the authorization server should return an error in response to the request.

How the authorization server remembers the association between the refresh token and the Token Binding ID is an implementation detail that beyond the scope of this specification. Some authorization servers will choose to store the Token Binding ID (or a cryptographic hash of it, such a SHA-256 hash [SHS]) in the refresh token itself, provided it is integrity-protected, thus reducing the amount of state to be kept by the server. Other authorization servers will add the Token Binding ID value (or a hash of it) to an internal data structure also containing other information about the refresh token, such as grant type information. These choices make no difference to the client, since the refresh token is opaque to it.

2.1. Example Token Binding for Refresh Tokens

This section provides an example of what the interactions around a Token Bound refresh token might look like, along with some details of the involved processing. Token Binding of refresh tokens is most useful for native application clients so the example has protocol elements typical of a native client flow. Extra line breaks in all examples are for display purposes only.

A native application client makes the following access token request with an authorization code using a TLS connection where Token Binding has been negotiated. A PKCE "code_verifier" is included because use of PKCE is considered best practice for native application clients [BCP212]. The base64url-encoded representation of the exported keying material (EKM) from that TLS connection is "p6ZuSwfl6pIe8es5KyeV76T4swZmQp0_awd27jHfrbo", which is needed to validate the Token Binding Message.

```
POST /as/token.oauth2 HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded
Sec-Token-Binding: AIkAAgBBQGto7hHRR0Y5nkOWqc9KNfwW95dEFmSI_tCZ_Cbl
  7LWlt6Xjp3DbjiDJavGFiKP2HV_2JSE42VzmKOVVV8m7eqAAQOKiDK1Oi0z6v4X5B
  P7uc0pFestVZ42TTOdJmoHpji06Qq3jsCiCRSJx9ck2fWJYx8tLVXRZPATB3x6c24
  aY0ZEAAA

grant_type=authorization_code&code=4bwcZesc7Xacc330ltc66Wxk8EAfP9j2
  &code_verifier=2x6_ylS390-8V7jaT9wj.8qP9nKmYCf.V-rD9O4r_1
  &client_id=example-native-client-id
```

Figure 1: Initial Request with Code

A refresh token is issued in response to the prior request. Although it looks like a typical response to the client, the authorization server has bound the refresh token to the Provided Token Binding ID from the encoded Token Binding message in the "Sec-Token-Binding" header of the request. In this example, that binding is done by saving the Token Binding ID alongside other information about the refresh token in some server side persistent storage. The base64url-encoded representation of that Token Binding ID is "AgBBQGto7hHRR0Y5nkOWqc9KNfwW95dEFmSI_tCZ_Cbl7LWlt6Xjp3DbjiDJavGFiKP2HV_2JSE42VzmKOVVV8m7eqA".

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

{
 "access_token":"EdRs7qMrLb167Z9fV2dcwoLTC",
 "refresh_token":"ACClZEIQTjW9arT9GOJGGd7QNwqOMmUYfsJTiv8his4",
 "token_type":"Bearer",
 "expires_in":3600
}
```

Figure 2: Successful Response

When the access token expires, the client requests a new one with a refresh request to the token endpoint. In this example, the request is made on a new TLS connection so the EKM (base64url-encoded: "va-84Ukw4Zqfd7uWOtFrAJda96WwgbdaPDX2knoOiAE") and signature in the Token Binding Message are different than in the initial request.

```
POST /as/token.oauth2 HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded
Sec-Token-Binding: AIkAAgBBQGto7hHRR0Y5nkOWqc9KNfwW95dEFmSI_tCZ_Cbl
  7LWlt6Xjp3DbjiDJavGFiKP2HV_2JSE42VzmKOVVV8m7eqAAQCpGbaG_YRf27qOra
  L0UT4fsKKjL6PukuOT00qzamoAXxOq7m_id7O3mLpnb_sM7kwSxLi7iNHzzDgCAkP
  t3lHwAAA

refresh_token=ACClZEIQTjW9arT9GOJGGd7QNwqOMmUYfsJTiv8his4
  &grant_type=refresh_token&client_id=example-native-client-id
```

Figure 3: Refresh Request

However, because the Token Binding ID is long-lived and may span multiple TLS sessions and connections, it is the same as in the initial request. That Token Binding ID is what the refresh token is bound to, so the authorization server is able to verify it and issue a new access token.

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

{
 "access_token":"bwcESCwC4yOCQ8iPsgcn117k7",
 "token_type":"Bearer",
 "expires_in":3600
}
```

Figure 4: Successful Response

3. Token Binding for Access Tokens

Token Binding for access tokens cryptographically binds the access token to the client’s Token Binding key pair, possession of which is proven on the TLS connections between the client and the protected resource. Token Binding is applied to access tokens in a similar manner to that described in Token Binding over HTTP [RFC8473], Section 4 (Federation Use Cases). It also builds upon the mechanisms for Token Binding of ID Tokens defined in OpenID Connect Token Bound Authentication 1.0 [OpenID.TokenBinding].

In the OpenID Connect [OpenID.Core] use case, HTTP redirects are used to pass information between the identity provider and the relying party; this HTTP redirect makes the Token Binding ID of the relying party available to the identity provider as the Referred Token Binding ID, information about which is then added to the ID Token. No such redirect occurs between the authorization server and the protected resource in the access token case; therefore, information about the Token Binding ID for the TLS connection between the client and the protected resource needs to be explicitly communicated by the client to the authorization server to achieve Token Binding of the access token.

This information is passed to the authorization server using the Referred Token Binding ID, just as in the ID Token case. The only difference is that the client needs to explicitly communicate the Token Binding ID of the TLS connection between the client and the protected resource to the Token Binding implementation so that it is sent as the Referred Token Binding ID in the request to the authorization server. This functionality provided by Token Binding implementations is described in Implementation Considerations of Token Binding over HTTP [RFC8473], Section 6.

Note that to obtain this Token Binding ID, the client may need to establish a TLS connection between itself and the protected resource prior to making the request to the authorization server so that the Provided Token Binding ID for the TLS connection to the protected resource can be obtained. How the client retrieves this Token Binding ID from the underlying Token Binding API is implementation and operating system specific. An alternative, if supported, is for the client to generate a Token Binding key to use for the protected resource, use the Token Binding ID for that key, and then later use that key when the TLS connection to the protected resource is established.

3.1. Access Tokens Issued from the Authorization Endpoint

For access tokens returned directly from the authorization endpoint, such as with the implicit grant defined in OAuth 2.0 [RFC6749], Section 4.2, the Token Binding ID of the client's TLS channel to the protected resource is sent with the authorization request as the Referred Token Binding ID in the "Sec-Token-Binding" header, and is used to Token Bind the access token.

Upon receiving the Referred Token Binding ID in an authorization request, the authorization server associates (Token Binds) the ID with the access token in a way that can be accessed by the protected resource. Such methods include embedding the Referred Token Binding ID (or a cryptographic hash of it) in the issued access token itself, possibly using the syntax described in Section 3.4, or through token introspection as described in Section 3.5. The method for associating the referred token binding ID with the access token is determined by the authorization server and the protected resource, and is beyond the scope for this specification.

3.1.1. Example Access Token Issued from the Authorization Endpoint

This section provides an example of what the interactions around a Token Bound access token issued from the authorization endpoint might look like, along with some details of the involved processing. Extra line breaks in all examples are for display purposes only.

The client directs the user-agent to make the following HTTP request to the authorization endpoint. It is a typical authorization request that, because Token Binding was negotiated on the underlying TLS connection and the user-agent was signaled to reveal the Referred Token Binding, also includes the "Sec-Token-Binding" header with a Token Binding Message that contains both a Provided and Referred Token Binding. The base64url-encoded EKM from the TLS connection over which the request was made is "jI5UAyjs5XCPISUGQIwgcSrOiVIWq4fhLVIFTQ4nLxc".

```
GET /as/authorization.oauth2?response_type=token
  &client_id=example-client-id&state=rM8pZxG1c3gKy6rEbsD8s
  &redirect_uri=https%3A%2F%2Fclient%2Eexample%2Eorg%2Fcb HTTP/1.1
Host: server.example.com
Sec-Token-Binding: ARIAAgBBQIEE8mSMtDy2dj9EEBdXaQT9W3Rq1NS-jW8ebPoF
  6FyL0jIfATVE55zlircgOTZmEg1xeIrC3DsGegwjs4bhw14AQGKDlAXFFMyQkZegC
  wlbTlqX3F9HTt-lJxFU_pi16ezka7qVRCpSF0BQLfSqlsxMbYfSSCJX1BDtrIL7PX
  j__fUAAAECAEFA1BNUnP3te5WrwlEwiejEz0OpesmC5PElWc7kZ5nlLSqQTj1ciIp
  5vQ30LLUCyM_a2BYTUPKtd5EdS-PalT4t6ABADgeizRa5NkTMuX4zOdC-R4cLNWVV
  O8lLu2Psko-UJLR_XAH4Q0H7-m0_nQR1zBN78nYMKPvHsz8L3zWKRVyXEgAA
```

Figure 5: Authorization Request

The authorization server issues an access token and delivers it to the client by redirecting the user-agent with the following HTTP response:

```
HTTP/1.1 302 Found
Location: https://client.example.org/cb#state=rM8pZxG1c3gKy6rEbsD8s
  &expires_in=3600&token_type=Bearer
  &access_token=eyJhbGciOiJFUzI[...omitted for brevity...]8xy5W5sQ
```

Figure 6: Authorization Response

The access token is bound to the Referred Token Binding ID from the authorization request, which when represented as a JWT, as described in Section 3.4, contains the SHA-256 hash of the Token Binding ID as the value of the "tbh" (token binding hash) member of the "cnf" (confirmation) claim. The confirmation claim portion of the JWT Claims Set is shown in the following figure.

```
{
  ...other claims omitted for brevity...
  "cnf":{
     "tbh": "vowQESa_MgbGJwIXaFm_BTN2QDPwh8PhuBm-EtUAqxc"
  }
}
```

Figure 7: Confirmation Claim

3.2. Access Tokens Issued from the Token Endpoint

For access tokens returned from the token endpoint, the Token Binding ID of the client’s TLS channel to the protected resource is sent as the Referred Token Binding ID in the "Sec-Token-Binding" header, and is used to Token Bind the access token. This applies to all the grant types from OAuth 2.0 [RFC6749] using the token endpoint, including, but not limited to the refresh and authorization code token requests, as well as some extension grants, such as JWT assertion authorization grants [RFC7523].

Upon receiving the Referred Token Binding ID in a token request, the authorization server associates (Token Binds) the ID with the access token in a way that can be accessed by the protected resource. Such methods include embedding the Referred Token Binding ID (or a cryptographic hash of it) in the issued access token itself, possibly using the syntax described in Section 3.4, or through token introspection as described in Section 3.5. The method for associating the referred token binding ID with the access token is determined by the authorization server and the protected resource, and is beyond the scope for this specification.

Note that if the request results in a new refresh token being generated, it can be Token bound using the Provided Token Binding ID, per Section 2.

3.2.1. Example Access Token Issued from the Token Endpoint

This section provides an example of what the interactions around a Token Bound access token issued from the token endpoint might look like, along with some details of the involved processing. Extra line breaks in all examples are for display purposes only.

The client makes an access token request to the token endpoint and includes the "Sec-Token-Binding" header with a Token Binding Message that contains both Provided and Referred Token Binding IDs. The Provided Token Binding ID is used to validate the token binding of the refresh token in the request (and to Token Bind a new refresh token, if one is issued), and the Referred Token Binding ID is used to Token Bind the access token that is generated. The base64url-encoded EKM from the TLS connection over which the access token request was made is "4jTc5e1QpocqPTZ5l6jsb6pRP18IFKdwwPvasYjn1-E".

```
POST /as/token.oauth2 HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded
Sec-Token-Binding: ARIAAgBBQJFXJir2w4gbJ7grBx9uTYWIrs9V50-PW4ZijegQ
  0LUM-_bGnGT6DizxUK-m5n3dQUIkeH7ybn6wb1C5dGyV_IAAQDDFToFrHt41Zppq7
  u_SEMF_E-KimAB-HewWl2MvZzAQ9QKoWiJCLFiCkjgtr1RrA2-jaJvoB8o51DTGXQ
  ydWYkAAAECAEFAuC1GlYU83rqTGHEau1oqvNwy0fDsdXzIyT_4x1FcldsMxjFkJac
  IBJFGuYcccvnCak_duFi3QKFENuwxql-H9ABAMcU7IjJOUA4IyE6YoEcfz9BMPQqw
  M5M6hw4RZNQd58fsTCCslQE_NmNCl9JXy4NkdkEZBxqvZGPr0y8QZ_bmAwAA

refresh_token=gZR_ZI8EAhLgWR-gWxBimbgZRZi_8EAhLgWRgWxBimbf
 &grant_type=refresh_token&client_id=example-client-id
```

Figure 8: Access Token Request

The authorization server issues an access token bound to the Referred Token Binding ID and delivers it in a response the client.

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

{
 "access_token":"eyJhbGciOiJFUzI1NiIsImtp[...omitted...]1cs29j5c3",
 "token_type":"Bearer",
 "expires_in":3600
}
```

Figure 9: Response

The access token is bound to the Referred Token Binding ID of the access token request, which when represented as a JWT, as described in Section 3.4, contains the SHA-256 hash of the Token Binding ID as the value of the "tbh" (token binding hash) member of the "cnf" (confirmation) claim. The confirmation claim portion of the JWT Claims Set of the access token is shown in the following figure.

```
{
  ...other claims omitted for brevity...
  "cnf":{
     "tbh": "7NRBu9iDdJlYCTOqyeYuLxXv0blEA-yTpmGIrAwKAws"
  }
}
```

Figure 10: Confirmation Claim

3.3. Protected Resource Token Binding Validation

Upon receiving a token bound access token, the protected resource validates the binding by comparing the Provided Token Binding ID to the Token Binding ID for the access token. Alternatively, cryptographic hashes of these Token Binding ID values can be compared. If the values do not match, the resource access attempt MUST be rejected with an error.

3.3.1. Example Protected Resource Request

For example, a protected resource request using the access token from Section 3.2.1 would look something like the following. The base64url-encoded EKM from the TLS connection over which the request was made is "7LsNP3BT1aHHdXdk6meEWjtSkiPVLb7YS6iHp-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: AIkAAgBBQLgtRpWFPN66kxhxGrtaKrzcMtHw7HV8yMk_-MdR
  XJXbDMYxZCWnCASRRrmHHHL5wmpP3bhYt0ChRDbsMapfh_QAQN1He3Ftj4Wa_S_fz
  ZVns4saLfj6aBoMSQW6rLs19IIvHze7LrGjKyCfPTKXjajebxp-TLPFZCc0JTqTY5
  _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:


```
{
 "iss": "https://server.example.com",
 "aud": "https://resource.example.org",
 "sub": "brian@example.com"
 "iat": 1467324320,
 "exp": 1467324920,
 "cnf":{
   "tbh": "7NRBu9iDdJlYCTOqyeYuLxXv0blEA-yTpmGIrAwKAws"
  }
}
```


Figure 12: JWT with Token Binding Hash Confirmation Claim

3.5. Representing Token Binding in Introspection Responses

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

For a token bound access token, the hash of the Token Binding ID to which the token is bound is conveyed to the protected resource as meta-information in a token introspection response. The hash is conveyed using same structure as the token binding hash confirmation method, described in Section 3.4, as a top-level member of the introspection response JSON. The protected resource compares that token binding hash to a hash of the provided Token Binding ID and rejects the request, if they do not match.

The following is an example of an introspection response for an active token bound access token with a "tbh" token binding hash confirmation method.

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

{
  "active": true,
  "iss": "https://server.example.com",
  "aud": "https://resource.example.org",
  "sub": "brian@example.com"
  "iat": 1467324320,
  "exp": 1467324920,
  "cnf":{
    "tbh": "7NRBu9iDdJlYCTOqyeYuLxXv0blEA-yTpmGIrAwKAws"
  }
}
```

Figure 13: Example Introspection Response for a Token Bound Access Token

4. Token Binding Metadata

4.1. Token Binding Client Metadata

Clients supporting Token Binding that also support the OAuth 2.0 Dynamic Client Registration Protocol [RFC7591] use these metadata values to declare their support for Token Binding of access tokens and refresh tokens:

client_access_token_token_binding_supported
: OPTIONAL. Boolean value specifying whether the client supports Token Binding of access tokens. If omitted, the default value is "false".

client_refresh_token_token_binding_supported
: OPTIONAL. Boolean value specifying whether the client supports Token Binding of refresh tokens. If omitted, the default value is "false". Authorization servers MUST NOT Token Bind refresh tokens issued to a client that does not support Token Binding of refresh tokens, but MAY reject requests completely from such clients if token binding is required by authorization server policy by returning an OAuth error response.

4.2. Token Binding Authorization Server Metadata

Authorization servers supporting Token Binding that also support OAuth 2.0 Authorization Server Metadata [RFC8414] use these metadata values to declare their support for Token Binding of access tokens and refresh tokens:

as_access_token_token_binding_supported
OPTIONAL. Boolean value specifying whether the authorization server supports Token Binding of access tokens. If omitted, the default value is "false".

as_refresh_token_token_binding_supported
OPTIONAL. Boolean value specifying whether the authorization server supports Token Binding of refresh tokens. If omitted, the default value is "false".

5. Token Binding for Authorization Codes

There are two variations for Token Binding of an authorization code. One is appropriate for native application clients and the other for web server clients. The nature of where the various components reside for the different client types demands different methods of Token Binding the authorization code so that it is bound to a Token Binding key on the end user’s device. This ensures that a lost or stolen authorization code cannot be successfully utilized from a different device. For native application clients, the code is bound to a Token Binding key pair that the native client itself possesses. For web server clients, the code is bound to a Token Binding key pair on the end user’s browser. Both variations utilize the extensible framework of Proof Key for Code Exchange (PKCE) [RFC7636], which enables the client to show possession of a certain key when exchanging the authorization code for tokens. The following subsections individually describe each of the two PKCE methods respectively.

5.1. Native Application Clients

This section describes a PKCE method suitable for native application clients that cryptographically binds the authorization code to a Token Binding key pair on the client, which the client proves possession of on the TLS connection during the access token request containing the authorization code. The authorization code is bound to the Token Binding ID that the native application client uses to resolve the authorization code at the token endpoint. This binding ensures that the client that made the authorization request is the same client that is presenting the authorization code.

5.1.1. Code Challenge

As defined in Proof Key for Code Exchange [RFC7636], the client sends the code challenge as part of the OAuth 2.0 authorization request with the two additional parameters: "code_challenge" and "code_challenge_method".

For this Token Binding method of PKCE, "TB-S256" is used as the value of the "code_challenge_method" parameter.

The value of the "code_challenge" parameter is the base64url encoding (per Section 5 of [RFC4648] with all trailing padding ('=') characters omitted and without the inclusion of any line breaks or whitespace) of the SHA-256 hash of the Provided Token Binding ID that the client will use when calling the authorization server's token endpoint. Note that, prior to making the authorization request, the client may need to establish a TLS connection between itself and the authorization server's token endpoint in order to establish the appropriate Token Binding ID.

When the authorization server issues the authorization code in the authorization response, it associates the code challenge and method values with the authorization code so they can be verified later when the authorization code is presented in the access token request.

5.1.1.1. Example Code Challenge

For example, a native application client sends an authorization request by sending the user's browser to the authorization endpoint. The resulting HTTP request looks something like the following (with extra line breaks for display purposes only).

```
GET /as/authorization.oauth2?response_type=code
  &client_id=example-native-client-id&state=oUC2jyYtzRCrMyWrVnGj
  &code_challenge=rBlgOyMY4teiuJMDgOwkrpsAjPyI07D2WsEM-dnq6eE
  &code_challenge_method=TB-S256 HTTP/1.1
Host: server.example.com
```

Figure 14: Authorization Request with PKCE Challenge

5.1.2. Code Verifier

Upon receipt of the authorization code, the client sends the access token request to the token endpoint. The Token Binding Protocol [RFC8471] is negotiated on the TLS connection between the client and the authorization server and the "Sec-Token-Binding" header, as defined in Token Binding over HTTP [RFC8473], is included in the access token request. The authorization server extracts the Provided Token Binding ID from the header value, hashes it with SHA-256, and compares it to the "code_challenge" value previously associated with the authorization code. If the values match, the token endpoint continues processing as normal (as defined by OAuth 2.0 [RFC6749]). If the values do not match, an error response indicating "invalid_grant" MUST be returned.

The "Sec-Token-Binding" header contains sufficient information for verification of the authorization code and its association to the original authorization request. However, PKCE [RFC7636] requires that a "code_verifier" parameter be sent with the access token request, so the static value "provided_tb" is used to meet that requirement and indicate that the Provided Token Binding ID is used for the verification.

5.1.2.1. Example Code Verifier

An example access token request, correlating to the authorization request in the previous example, to the token endpoint over a TLS connection for which Token Binding has been negotiated would look like the following (with extra line breaks for display purposes only). The base64url-encoded EKM from the TLS connection over which the request was made is "pNVKtPuQFvylNYn000QowWrQKoeMkeX9H32hVuU71Bs".

```
POST /as/token.oauth2 HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded
Sec-Token-Binding: AIkAAgBBQEOO9GRFP-LM0hoWw6-2i318BsuuUum5AL8bt1sz
  lr1EFfp5DMXMNW3O8WjcIXr2DKJnI4xnuGsE6GywQd9RbD0AQJDb3xyo9PBxj8M6Y
  jLt-6OaxgDkyoBoTkyrnNbLc8tJQ0JtXomKzBbj5qPtHDduXc6xz_lzvNpxSPxi42
  8m7wkAAA

grant_type=authorization_code&code=mJAReTWKX7zI3oHUNd4o3PeNqNqxKGp6
  &code_verifier=provided_tb&client_id=example-native-client-id
```

Figure 15: Token Request with PKCE Verifier

5.2. Web Server Clients

This section describes a PKCE method suitable for web server clients, which cryptographically binds the authorization code to a Token Binding key pair on the browser. The authorization code is bound to the Token Binding ID that the browser uses to deliver the authorization code to a web server client, which is sent to the authorization server as the Referred Token Binding ID during the authorization request. The web server client conveys the Token Binding ID to the authorization server when making the access token request containing the authorization code. This binding ensures that the authorization code cannot successfully be played or replayed to the web server client from a different browser than the one that made the authorization request.

5.2.1. Code Challenge

As defined in Proof Key for Code Exchange [RFC7636], the client sends the code challenge as part of the OAuth 2.0 Authorization Request with the two additional parameters: "code_challenge" and "code_challenge_method".

The client must send the authorization request through the browser such that the Token Binding ID established between the browser and itself is revealed to the authorization server's authorization endpoint as the Referred Token Binding ID. Typically, this is done with an HTTP redirection response and the "Include-Referred-Token-Binding-ID" header, as defined in Token Binding over HTTP [RFC8473], Section 5.3.

For this Token Binding method of PKCE, "referred_tb" is used for the value of the "code_challenge_method" parameter.

The value of the "code_challenge" parameter is "referred_tb". The static value for the required PKCE parameter indicates that the authorization code is to be bound to the Referred Token Binding ID from the Token Binding Message sent in the "Sec-Token-Binding" header of the authorization request.

When the authorization server issues the authorization code in the authorization response, it associates the Token Binding ID (or hash thereof) and code challenge method with the authorization code so they can be verified later when the authorization code is presented in the access token request.

5.2.1.1. Example Code Challenge

For example, the web server client sends the authorization request by redirecting the browser to the authorization endpoint. That HTTP redirection response looks like the following (with extra line breaks for display purposes only).

```
HTTP/1.1 302 Found
Location: https://server.example.com?response_type=code
  &client_id=example-web-client-id&state=P4FUFqYzs1ij3ffsYCP34d3
  &redirect_uri=https%3A%2F%2Fclient%2Eexample%2Eorg%2Fcb
  &code_challenge=referred_tb&code_challenge_method=referred_tb
Include-Referred-Token-Binding-ID: true
```

Figure 16: Redirect the Browser

The redirect includes the "Include-Referred-Token-Binding-ID" response header field that signals to the user-agent that it should reveal, to the authorization server, the Token Binding ID used on the connection to the web server client. The resulting HTTP request to the authorization server looks something like the following (with extra line breaks for display purposes only). The base64url-encoded EKM from the TLS connection over which the request was made is "7gOdRzMhPeO-1YwZGmnVHyReN5vd2CxcsRBN69Ue4cI".

```
GET /as/authorization.oauth2?response_type=code
  &client_id=example-web-client-id&state=dryo8YFpWacbUPjhBf4Nvt51
  &redirect_uri=https%3A%2F%2Fclient%2Eexample%2Eorg%2Fcb
  &code_challenge=referred_tb
  &code_challenge_method=referred_tb HTTP/1.1
Host: server.example.com
Sec-Token-Binding: ARIAAgBBQB-XOPf5ePlf7ikATiAFEGOS503lPmRfkyymzdWw
  HCxl0njjxC3D0E_OVfBNqrIQxzIfkF7tWby2ZfyaE6XpwTsAQBYqhFX78vMOgDX_F
  d_b2dlHyHlMmkIz8iMVBY_reM98OUaJFz5IB7PG9nZ11j58LoG5QhmQoI9NXYktKZ
  RXxrYAAAECAEFAdUFTnfQADkn1uDbQnvJEk6oQs38L92gv-KO-qlYadLoDIKe2h53
  hSiKwIP98iRj_unedkNkAMyg9e2mY4Gp7WwBAeDUOwaSXNz1e6gKohwN4SAZ5eNyx
  45Mh8VI4woL1BipLoqrJRoK6dxFkWgHRMuBROcLGUj5PiOoxybQH_Tom3gAA
```

Figure 17: Authorization Request

5.2.2. Code Verifier

The web server client receives the authorization code from the browser and extracts the Provided Token Binding ID from the "Sec-Token-Binding" header of the request. The client sends the base64url-encoded (per Section 5 of [RFC4648] with all trailing padding ('=') characters omitted and without the inclusion of any line breaks or whitespace) Provided Token Binding ID as the value of the "code_verifier" parameter in the access token request to the authorization server's token endpoint. The authorization server compares the value of the "code_verifier" parameter to the Token Binding ID value previously associated with the authorization code. If the values match, the token endpoint continues processing as normal (as defined by OAuth 2.0 [RFC6749]). If the values do not match, an error response indicating "invalid_grant" MUST be returned.

5.2.2.1. Example Code Verifier

Continuing the example from the previous section, the authorization server sends the code to the web server client by redirecting the browser to the client's "redirect_uri", which results in the browser making a request like the following (with extra line breaks for display purposes only) to the web server client over a TLS channel for which Token Binding has been established. The base64url-encoded EKM from the TLS connection over which the request was made is "EzW60vyINbsb_tajt8ij3tV6cwy2KH-i8BdEMYXcNn0".

```
GET /cb?state=dryo8YFpWacbUPjhBf4Nvt51&code=jwD3oOa5cQvvLc81bwc4CMw
Host: client.example.org
Sec-Token-Binding: AIkAAgBBQHVBU530AA5J9bg20J7yRJOqELN_C_doL_ijvqpW
  GnS6AyCntoed4UoisCD_fIkY_7p3nZDZADMoPXtpmOBqe1sAQEwgC9Zpg7QFCDBib
  6GlZki3MhH32KNfLefLJc1vR1xE8l7OMfPLZHP2Woxh6rEtmgBcAABubEbTz7muNl
  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 b3JnLmV4YW1wbGUuY2xpZW50OmlldGY5OGNoaWNhZ28=

grant_type=authorization_code&code=jwD3oOa5cQvvLc81bwc4CMw
  &redirect_uri=https%3A%2F%2Fclient%2Eexample%2Eorg%2Fcb
  &client_id=example-web-client-id
  &code_verifier=AgBBQHVBU530AA5J9bg20J7yRJOqELN_C_doL_ijv
  qpWGnS6AyCntoed4UoisCD_fIkY_7p3nZDZADMoPXtpmOBqe1s
```

Figure 19: Exchange Authorization Code

6. Token Binding JWT Authorization Grants and Client Authentication

The JWT Profile for OAuth 2.0 Client Authentication and Authorization Grants [RFC7523] defines the use of bearer JWTs as a means for requesting an OAuth 2.0 access token as well as for client authentication. This section describes extensions to that specification enabling the application of Token Binding to JWT client authentication and JWT authorization grants.

6.1. JWT Format and Processing Requirements

In addition the requirements set forth in Section 3 of RFC 7523 [RFC7523], the following criteria must also be met for token bound JWTs used as authorization grants or for client authentication.

- o The JWT MUST contain a "cnf" (confirmation) claim with a "tbh" (token binding hash) member identifying the Token Binding ID of the Provided Token Binding used by the client on the TLS connection to the authorization server. The authorization server MUST reject any JWT that has a token binding hash confirmation that does not match the corresponding hash of the Provided Token Binding ID from the "Sec-Token-Binding" header of the request.

6.2. Token Bound JWTs for Client Authentication

To use a token bound JWT for client authentication, the client uses the parameter values and encodings from Section 2.2 of RFC 7523 [RFC7523] with one exception: the value of the "client_assertion_type" is "urn:ietf:params:oauth:client-assertion-type:jwt-token-bound".

The "OAuth Token Endpoint Authentication Methods" registry [IANA.OAuth.Parameters] contains values, each of which specify a method of authenticating a client to the authorization server. The values are used to 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 Toking 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

- o Specification Document(s): Section 4.1 of [[this specification]]

8.2. OAuth Authorization Server Metadata Registration

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

8.2.1. Registry Contents

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

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

8.3. PKCE Code Challenge Method Registration

This specification requests registration of the following Code Challenge Method Parameter Names in the IANA "PKCE Code Challenge Methods" registry [IANA.OAuth.Parameters] established by [RFC7636].

8.3.1. Registry Contents

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

- o Code Challenge Method Parameter Name: referred_tb
- o Change controller: IESG
- o Specification document(s): Section 5.2.1 of [[this specification]]

9. Token Endpoint Authentication Method Registration

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

9.1. Registry Contents

- 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]]

- URN: urn:ietf:params:oauth:client-assertion-type:jwt-token-bound
- 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>.

[JWT] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <http://tools.ietf.org/html/rfc7519>.

[OpenID.TokenBinding]
Jones, M., Bradley, J., and B. Campbell, "OpenID Connect Token Bound Authentication 1.0", October 2017, <http://openid.net/specs/openid-connect-token-bound-authentication-1_0-03.html>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

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

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <https://www.rfc-editor.org/info/rfc7230>.

[RFC7523] Jones, M., Campbell, B., and C. Mortimore, "JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants", RFC 7523, DOI 10.17487/RFC7523, May 2015, <https://www.rfc-editor.org/info/rfc7523>.

[RFC7591] Richer, J., Ed., Jones, M., Bradley, J., Machulak, M., and P. Hunt, "OAuth 2.0 Dynamic Client Registration Protocol", RFC 7591, DOI 10.17487/RFC7591, July 2015, <https://www.rfc-editor.org/info/rfc7591>.

[RFC7636] Sakimura, N., Ed., Bradley, J., and N. Agarwal, "Proof Key for Code Exchange by OAuth Public Clients", RFC 7636, DOI 10.17487/RFC7636, September 2015, <https://www.rfc-editor.org/info/rfc7636>.

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

[RFC7800] Jones, M., Bradley, J., and H. Tschofenig, "Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs)", RFC 7800, DOI 10.17487/RFC7800, April 2016, <https://www.rfc-editor.org/info/rfc7800>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8414] Jones, M., Sakimura, N., and J. Bradley, "OAuth 2.0 Authorization Server Metadata", RFC 8414, DOI 10.17487/RFC8414, June 2018, <https://www.rfc-editor.org/info/rfc8414>.

[RFC8471] Popov, A., Ed., Nystroem, M., Balfanz, D., and J. Hodges, "The Token Binding Protocol Version 1.0", RFC 8471, DOI 10.17487/RFC8471, October 2018, <https://www.rfc-editor.org/info/rfc8471>.

[RFC8472] Popov, A., Ed., Nystroem, M., and D. Balfanz, "Transport Layer Security (TLS) Extension for Token Binding Protocol Negotiation", RFC 8472, DOI 10.17487/RFC8472, October 2018, <https://www.rfc-editor.org/info/rfc8472>.

[RFC8473] Popov, A., Nystroem, M., Balfanz, D., Ed., Harper, N., and J. Hodges, "Token Binding over HTTP", RFC 8473, DOI 10.17487/RFC8473, October 2018, <https://www.rfc-editor.org/info/rfc8473>.

[SHS] National Institute of Standards and Technology, "Secure Hash Standard (SHS)", FIPS PUB 180-4, March 2012, <http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf>.

11.2. Informative References

[BCP212] Denniss, W. and J. Bradley, "OAuth 2.0 for Native Apps", BCP 212, RFC 8252, DOI 10.17487/RFC8252, October 2017, <https://www.rfc-editor.org/info/rfc8252>.

[OpenID.Core] Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0", August 2015, <http://openid.net/specs/openid-connect-core-1_0.html>.

[OpenID.Discovery] Sakimura, N., Bradley, J., Jones, M., and E. Jay, "OpenID Connect Discovery 1.0", August 2015, <http://openid.net/specs/openid-connect-discovery-1_0.html>.

[RFC6755] Campbell, B. and H. Tschofenig, "An IETF URN Sub-Namespace for OAuth", RFC 6755, DOI 10.17487/RFC6755, October 2012, <https://www.rfc-editor.org/info/rfc6755>.

[RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <https://www.rfc-editor.org/info/rfc7515>.

Appendix A. Acknowledgements

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

Appendix B. Document History

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

-08

- o Update reference to -03 of openid-connect-token-bound-authentication.
- o Update the references to the core token binding specs, which are now RFCs 8471, 8472, and 8473.
- o Update reference to AS metadata, which is now RFC 8414.
- o Add chairs and ADs to the Acknowledgements.

-07

- o Explicitly state that the base64url encoding of the tbh value doesn’t include any trailing pad characters, line breaks, whitespace, etc.
- o Update to latest references for tokbind drafts and draft-ietf-oauth-discovery.
- o Update reference to Implementation Considerations in draft-ietf-tokbind-https, which is section 6 rather than 5.
- o Try to tweak text that references specific sections in other documents so that the HTML generated by the ietf tools doesn’t link to the current document (based on old suggestion from Barry https://www.ietf.org/mail-archive/web/jose/current/msg04571.html).

-06

- Use the boilerplate from RFC 8174.

- Update reference for draft-ietf-tokbind-https to -12 and draft-ietf-oauth-discovery to -09.

- Minor editorial fixes.

-05

- 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.

- Add Token Binding for JWT Authorization Grants and JWT Client Authentication.

- Adjust the language around aborting authorizations in Phasing in Token Binding to be somewhat more general and not only about downgrades.

- Remove reference to, and usage of, 'OAuth 2.0 Protected Resource Metadata', which is no longer a going concern.

- 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.

- 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

- 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).

- Minor editorial fixes.

- Added an open issue about needing to allow for web server clients to opt-out of having refresh tokens bound while still allowing for binding of access tokens (following from mention of the problem on slide 16 of the presentation from Chicago https://www.ietf.org/proceedings/98/slides/slides-98-oauth-sessb-token-binding-00.pdf).

-03

- Fix a few mistakes in and around the examples that were noticed preparing the slides for IETF 98 Chicago.

-02

- Added a section on Token Binding for authorization codes with one variation for native clients and one for web server clients.
- 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.
- Added a bunch of examples.
- Added a few Open Issues so they are tracked in the document.
- Updated the Token Binding and OAuth Metadata references.
- Added William Denniss as an author.

-01

- 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.
- Defined Protected Resource Metadata value.
- Changed to use the more specific term "protected resource" instead of "resource server".

-00

- Created the initial working group version from draft-jones-oauth-token-binding-00.

Authors' Addresses

Michael B. Jones
Microsoft

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

Brian Campbell
Ping Identity

Email: brian.d.campbell@gmail.com

John Bradley
Yubico

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

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

Email: wdenniss@google.com
URI: http://wdenniss.com/

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: January 21, 2020

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

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

Abstract

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

Table of Contents

1. Introduction

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

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

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

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

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

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

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

1.1. Delegation vs. Impersonation Semantics

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

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

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

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

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

The specifics of representing a composite token and even whether or not such a token will be issued depend on the details of the implementation and the kind of token. The representations of composite tokens that are not JWTs are beyond the scope of this specification. The "actor_token" request parameter, however, does provide a means for providing information about the desired actor and the JWT "act" claim can provide a representation of a chain of delegation.

1.2. Requirements Notation and Conventions

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

1.3. Terminology

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

2. Token Exchange Request and Response

2.1. Request

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

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

The client makes a token exchange request to the token endpoint with an extension grant type using the HTTP "POST" method. The following parameters are included in the HTTP request entity-body using the "application/x-www-form-urlencoded" format with a character encoding of UTF-8 as described in Appendix B of RFC6749 [RFC6749].

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

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

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

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

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

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

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

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

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

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

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

2.1.1. Relationship Between Resource, Audience and Scope

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

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

2.2. Response

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

2.2.1. Successful Response

If the request is valid and meets all policy and other criteria of the authorization server, a successful token response is constructed by adding the following parameters to the entity-body of the HTTP response using the "application/json" media type, as specified by [RFC8259], and an HTTP 200 status code. The parameters are serialized into a JavaScript Object Notation (JSON) structure by adding each parameter at the top level. Parameter names and string values are included as JSON strings. Numerical values are included as JSON numbers. The order of parameters does not matter and can vary.

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

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

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

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

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

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

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

2.2.2. Error Response

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

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

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

Other error codes may also be used, as appropriate.

2.3. Example Token Exchange

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

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

```
GET /resource HTTP/1.1
Host: frontend.example.com
Authorization: Bearer accVkjcJyb4BWCxGsndESCJQbdFMogUC5PbRDqceLTC
```

Figure 1: Protected Resource Request

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

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

grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Atoken-exchange
&resource=https%3A%2F%2Fbackend.example.com%2Fapi
&subject_token=accVkjcJyb4BWCxGsndESCJQbdFMogUC5PbRDqceLTC
&subject_token_type=
 urn%3Aietf%3Aparams%3Aoauth%3Atoken-type%3Aaccess_token
```

Figure 2: Token Exchange Request

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

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

{
 "access_token":"eyJhbGciOiJFUzI1NiIsImtpZCI6IjllciJ9.eyJhdWQiOiJo
   dHRwczovL2JhY2tlbmQuZXhhbXBsZS5jb20iLCJpc3MiOiJodHRwczovL2FzLmV
   4YW1wbGUuY29tIiwiZXhwIjoxNDQxOTE3NTkzLCJpYXQiOjE0NDE5MTc1MzMsIn
   N1YiI6ImJkY0BleGFtcGxlLmNvbSIsInNjb3BlIjoiYXBpIn0.40y3ZgQedw6rx
   f59WlwHDD9jryFOr0_Wh3CGozQBihNBhnXEQgU85AI9x3KmsPottVMLPIWvmDCM
   y5-kdXjwhw",
 "issued_token_type":
     "urn:ietf:params:oauth:token-type:access_token",
 "token_type":"Bearer",
 "expires_in":60
}
```

Figure 3: Token Exchange Response

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

```
GET /api HTTP/1.1
Host: backend.example.com
Authorization: Bearer eyJhbGciOiJFUzI1NiIsImtpZCI6IjllciJ9.eyJhdWQ
   iOiJodHRwczovL2JhY2tlbmQuZXhhbXBsZS5jb20iLCJpc3MiOiJodHRwczovL2
   FzLmV4YW1wbGUuY29tIiwiZXhwIjoxNDQxOTE3NTkzLCJpYXQiOjE0NDE5MTc1M
   zMsInN1YiI6ImJkY0BleGFtcGxlLmNvbSIsInNjb3BlIjoiYXBpIn0.40y3ZgQe
   dw6rxf59WlwHDD9jryFOr0_Wh3CGozQBihNBhnXEQgU85AI9x3KmsPottVMLPIW
   vmDCMy5-kdXjwhw
```

Figure 4: Backend Protected Resource Request

Additional examples can be found in Appendix A.

3. Token Type Identifiers

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

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

urn:ietf:params:oauth:token-type:access_token
: Indicates that the token is an OAuth 2.0 access token issued by the given authorization server.

urn:ietf:params:oauth:token-type:refresh_token
: Indicates that the token is an OAuth 2.0 refresh token issued by the given authorization server.

urn:ietf:params:oauth:token-type:id_token
: Indicates that the token is an ID Token, as defined in Section 2 of [OpenID.Core].

urn:ietf:params:oauth:token-type:saml1
: Indicates that the token is a base64url-encoded SAML 1.1 [OASIS.saml-core-1.1] assertion.

urn:ietf:params:oauth:token-type:saml2
: Indicates that the token is a base64url-encoded SAML 2.0 [OASIS.saml-core-2.0-os] assertion.

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

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

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

4. JSON Web Token Claims and Introspection Response Parameters

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

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

4.1. "act" (Actor) Claim

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

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

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

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

Figure 5: Actor Claim

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

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

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

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

Figure 6: Nested Actor Claim

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

4.2. "scope" (Scopes) Claim

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

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

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

Figure 7: Scopes Claim

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

4.3. "client_id" (Client Identifier) Claim

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

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

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

Figure 8: Client Identifier Claim

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

4.4. "may_act" (Authorized Actor) Claim

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

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

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

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

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

Figure 9: Authorized Actor Claim

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

5. Security Considerations

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

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

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

6. Privacy Considerations

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

7. IANA Considerations

7.1. OAuth URI Registration

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

7.1.1. Registry Contents

- 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]]

- URN: urn:ietf:params:oauth:token-type:refresh_token
- 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]]

- URN: urn:ietf:params:oauth:token-type:saml2
- Common Name: Token type URI for a base64url-encoded SAML 2.0 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

- o Specification document(s): Section 2.1 of [[this specification]]

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

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

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

7.3. OAuth Access Token Type Registration

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

7.3.1. Registry Contents

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

7.4. JSON Web Token Claims Registration

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

7.4.1. Registry Contents

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

- o Claim Name: "scope"
- o Claim Description: Scope Values
- o Change Controller: IESG

- o Specification Document(s): Section 4.2 of [[this specification]]

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

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

7.5. OAuth Token Introspection Response Registration

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

7.5.1. Registry Contents

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

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

7.6. OAuth Extensions Error Registration

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

7.6.1. Registry Contents

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

8. References

8.1. Normative References

[IANA.JWT.Claims]
IANA, "JSON Web Token Claims", <http://www.iana.org/assignments/jwt>.

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

[JWT] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <http://tools.ietf.org/html/rfc7519>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

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

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

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

8.2. Informative References

[I-D.ietf-oauth-resource-indicators]
Campbell, B., Bradley, J., and H. Tschofenig, "Resource Indicators for OAuth 2.0", draft-ietf-oauth-resource-indicators-02 (work in progress), January 2019.

[I-D.ietf-oauth-security-topics]
Lodderstedt, T., Bradley, J., Labunets, A., and D. Fett, "OAuth 2.0 Security Best Current Practice", draft-ietf-oauth-security-topics-13 (work in progress), July 2019.

[OASIS.saml-core-1.1]
Maler, E., Mishra, P., and R. Philpott, "Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V1.1", OASIS Standard oasis-sstc-saml-core-1.1, September 2003.

[OASIS.saml-core-2.0-os]
Cantor, S., Kemp, J., Philpott, R., and E. Maler, "Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V2.0", OASIS Standard saml-core-2.0-os, March 2005.

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

[RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, <https://www.rfc-editor.org/info/rfc6750>.

[RFC6755] Campbell, B. and H. Tschofenig, "An IETF URN Sub-Namespace for OAuth", RFC 6755, DOI 10.17487/RFC6755, October 2012, <https://www.rfc-editor.org/info/rfc6755>.

[RFC6819] Lodderstedt, T., Ed., McGloin, M., and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations", RFC 6819, DOI 10.17487/RFC6819, January 2013, <https://www.rfc-editor.org/info/rfc6819>.

[RFC7521] Campbell, B., Mortimore, C., Jones, M., and Y. Goland, "Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants", RFC 7521, DOI 10.17487/RFC7521, May 2015, <https://www.rfc-editor.org/info/rfc7521>.

[RFC7523] Jones, M., Campbell, B., and C. Mortimore, "JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants", RFC 7523, DOI 10.17487/RFC7523, May 2015, <https://www.rfc-editor.org/info/rfc7523>.

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

Appendix A. Additional Token Exchange Examples

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

A.1. Impersonation Token Exchange Example

A.1.1. Token Exchange Request

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

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

grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Atoken-exchange
&audience=urn%3Aexample%3Acooperation-context
&subject_token=eyJhbGciOiJFUzI1NiIsImtpZCI6IjE2In0.eyJhdWQiOiJodHRwc
  zovL2FzLmV4YW1wbGUuY29tIiwiaXNzIjoiaHR0cHM6Ly9vcmlnaW5hbC1pc3N1ZXI
  uZXhhbXBsZS5uZXQiLCJleHAiOjE0NDE5MTA2MDAsIm5iZiI6MTQ0MTkwOTAwMCwic
  3ViIjoiYmRjQGV4YW1wbGUubmV0Iiwic2NvcGUiOiJvcmRlcnMgcHJvZmlsZSBoaXN
  0b3J5In0.PRBg-jXn4cJuj1gmYXFiGkZzRuzbXZ_sDxdE98ddW44ufsbWLKd3JJ1VZ
  hF64pbTtfjy4VXFVBDaQpKjn5JzAw
&subject_token_type=urn%3Aietf%3Aparams%3Aoauth%3Atoken-type%3Ajwt
```

Figure 10: Token Exchange Request

A.1.2. Subject Token Claims

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

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

Figure 11: Subject Token Claims

A.1.3. Token Exchange Response

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

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

{
 "access_token":"eyJhbGciOiJFUzI1NiIsImtpZCI6IjcyIn0.eyJhdWQiOiJ1cm4
   6ZXhhbXBsZTpjb29wZXJhdGlvbi1jb250ZXh0IiwiaXNzIjoiaHR0cHM6Ly9hcy5l
   eGFtcGxlLmNvbSIsImV4cCI6MTQ0MTkxMzYxMCwic3ViIjoiYmRjQGV4YW1wbGUub
   mV0Iiwic2NvcGUiOiJvcmRlcnMgcHJvZmlsZSBoaXN0b3J5In0.rMdWpSGNACTvnF
   uOL74sYZ6MVuld2Z2WkGLmQeR9ztj6w2OXraQlkJmGjyiCq24kcB7AI2VqVxl3wSW
   nVKh85A",
 "issued_token_type":
   "urn:ietf:params:oauth:token-type:access_token",
 "token_type":"Bearer",
 "expires_in":3600
}
```

Figure 12: Token Exchange Response

A.1.4. Issued Token Claims

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

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

Figure 13: Issued Token Claims

A.2. Delegation Token Exchange Example

A.2.1. Token Exchange Request

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

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

grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Atoken-exchange
&audience=urn%3Aexample%3Acooperation-context
&subject_token=eyJhbGciOiJFUzI1NiIsImtpZCI6IjE2In0.eyJhdWQiOiJodHRwc
  zovL2FzLmV4YW1wbGUuY29tIiwiaXNzIjoiaHR0cHM6Ly9vcmlnaW5hbC1pc3N1ZXI
  uZXhhbXBsZS5uZXQiLCJleHAiOjE0NDE5MTAwNjAsInNjb3BlIjoic3RhdHVzIGZlZ
  WQiLCJzdWIiOiJ1c2VyQGV4YW1wbGUubmV0IiwibWF5X2FjdCI6eyJzdWIiOiJhZG1
  pbkBleGFtcGxlLm5ldCJ9fQ.4rPRSWihQbpMIgAmAoqaJojAxj-p2X8_fAtAGTXrvM
  xU-eEZHnXqY0_AOZgLdxw5DyLzua8H_I10MCcckF-Q_g
&subject_token_type=urn%3Aietf%3Aparams%3Aoauth%3Atoken-type%3Ajwt
&actor_token=eyJhbGciOiJFUzI1NiIsImtpZCI6IjE2In0.eyJhdWQiOiJodHRwczo
  vL2FzLmV4YW1wbGUuY29tIiwiaXNzIjoiaHR0cHM6Ly9vcmlnaW5hbC1pc3N1ZXIuZ
  XhhbXBsZS5uZXQiLCJleHAiOjE0NDE5MTAwNjAsInN1YiI6ImFkbWluQGV4YW1wbGU
  ubmV0In0.7YQ-3zPfhUvzje5oqw8COCvN5uP6NsKik9CVV6cAOf4QKgM-tKfiOwcgZ
  oUuDL2tEs6tqPlcBlMjiSzEjm3yBg
&actor_token_type=urn%3Aietf%3Aparams%3Aoauth%3Atoken-type%3Ajwt
```

Figure 14: Token Exchange Request

A.2.2. Subject Token Claims

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

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

Figure 15: Subject Token Claims

A.2.3. Actor Token Claims

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

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

Figure 16: Actor Token Claims

A.2.4. Token Exchange Response

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

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

{
 "access_token":"eyJhbGciOiJFUzI1NiIsImtpZCI6IjcyIn0.eyJhdWQiOiJ1cm4
   6ZXhhbXBsZTpjb29wZXJhdGlvbi1jb250ZXh0IiwiaXNzIjoiaHR0cHM6Ly9hcy5l
   eGFtcGxlLmNvbSIsImV4cCI6MTQ0MTkxMzYxMCwic2NvcGUiOiJzdGF0dXMgZmVlZ
   CIsInN1YiI6InVzZXJAZXhhbXBsZS5uZXQiLCJhY3QiOnsic3ViIjoiYWRtaW5AZX
   hhbXBsZS5uZXQifX0.3paKl9UySKYB5ng6_cUtQ2qlO8Rc_y7Mea7IwEXTcYbNdwG
   9-G1EKCFe5fW3H0hwX-MSZ49Wpcb1SiAZaOQBtw",
 "issued_token_type":"urn:ietf:params:oauth:token-type:jwt",
 "token_type":"N_A",
 "expires_in":3600
}
```

Figure 17: Token Exchange Response

A.2.5. Issued Token Claims

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

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

Figure 18: Issued Token Claims

Appendix B. Acknowledgements

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

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

Appendix C. Document History

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

-19

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

-18

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

-17

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

-16

- Fixed typo and added an AD to Acknowledgements.

-15

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

-14

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

-13

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

-12

- o Updated to use the boilerplate from RFC 8174.

-11

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

-10

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

-09

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

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

-08

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

-07

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

-06

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

-05

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

-04

- 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).
- 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).
- 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).
- 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

- Updated the document editors (adding Campbell, Bradley, and Mortimore).
- Added to the title.
- Added to the abstract and introduction.
- 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.
- Changed the grant type identifier to urn:ietf:params:oauth:grant-type:token-exchange.
- Added RFC 6755 registration requests for urn:ietf:params:oauth:token-type:refresh_token, urn:ietf:params:oauth:token-type:access_token, and urn:ietf:params:oauth:grant-type:token-exchange.
- Added RFC 6749 registration requests for request/response parameters.
- Removed the Implementation Considerations and the requirement to support JWTs.
- Clarified many aspects of the text.
- 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".
- 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.
- Added a "want_composite" request parameter used to indicate the desire for a composite token rather than trying to infer it from the presence/absence of token(s) in the request.

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

-02

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

-01

- o Updated references.

-00

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

Authors' Addresses

Michael B. Jones
Microsoft

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

Anthony Nadalin
Microsoft

Email: tonynad@microsoft.com

Brian Campbell (editor)
Ping Identity

Email: brian.d.campbell@gmail.com

John Bradley
Yubico

Email: ve7jtb@ve7jtb.com

Chuck Mortimore
Salesforce

Email: cmortimore@salesforce.com