

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 25, 2019

V. Bertocci
Auth0
March 24, 2019

JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens
draft-bertocci-oauth-access-token-jwt-00

Abstract

This specification defines a profile for issuing OAuth2 access tokens in JSON web token (JWT) format. Authorization servers and resource servers from different vendors can leverage this profile to issue and consume access tokens in interoperable manner.

Status of This Memo

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

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

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

This Internet-Draft will expire on September 25, 2019.

Copyright Notice

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

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

Table of Contents

1. Introduction	2
1.1. Requirements Notation and Conventions	3
1.2. Terminology	3
2. JWT Access Token Header and Data Structure	3
2.1. Header	4
2.2. Data Structure	4
2.2.1. Identity Claims	5
2.2.2. Authorization Claims	5
2.2.2.1. Claims for Authorization Outside of Delegation Scenarios	5
3. Requesting a JWT Access Token	6
4. Validating JWT Access Tokens	7
5. Security Considerations	9
6. Privacy Considerations	9
7. IANA Considerations	10
8. References	10
8.1. Normative References	10
8.2. Informative References	11
Appendix A. Acknowledgements	11
Appendix B. Document History	12
Author's Address	12

1. Introduction

The original OAuth 2.0 Authorization Framework [RFC6749] specification does not mandate any specific format for access tokens. While that remains perfectly appropriate for many important scenario, in-market use has shown that many commercial OAuth2 implementations elected to issue access tokens using a format that can be parsed and validated by resource servers directly, without further authorization server involvement. The approach is particularly common in topologies where the authorization server and resource server are not co-located, are not ran by the same entity, or are otherwise separated by some boundary. All of the known commercial implementations known at this time leverage the JSON Web Tokens (JWT) [RFC7519] format.

Most vendor specific JWT access tokens share the same functional layout, including information in forms of claims meant to support the same scenarios: token validation, transporting authorization information in forms of scopes and entitlements, carrying identity information about the subject, and so on. The differences are mostly confined to the claim names and syntax used to represent the same entities, suggesting that interoperability could be easily achieved by standardizing on a common set of claims and validation rules.

The assumption that access tokens are associated to specific information doesn't appear only in commercial implementations. Various specifications in the OAuth2 family (such as resource indicators [ResourceIndicators], bearer token usage [RFC6750] and others) postulate the presence in access tokens of scoping mechanisms, such as an audience. The family of specifications associated to introspection also indirectly suggest a fundamental set of information access tokens are expected to carry or at least be associated with.

This specification aims to provide a standardized and interoperable profile as an alternative to the proprietary JWT access tokens layouts going forward. Besides defining a common set of mandatory and optional claims, the profile provides clear indications on how authorization requests parameters determine the content of the issued JWT access token, how an authorization server can publish metadata relevant to the JWT access tokens it issues, and how a resource server should validate incoming JWT access tokens.

Finally, this specification provides security and privacy considerations meant to prevent common mistakes and anti patterns that are likely to occur in naive use of the JWT format to represent access tokens.

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

JWT access token An OAuth 2.0 access token encoded in JWT format and complying with the requirements described in this specification.

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. JWT Access Token Header and Data Structure

JWT access tokens are regular JWT tokens complying with the requirements described in this section.

2.1. Header

Although JWT access tokens can use any signing algorithm, use of asymmetric algorithms is RECOMMENDED as it simplifies the process of acquiring validation information for resource servers (see Section 4).

The typ header parameter for a JWT access token MUST be at+jwt. See the security considerations section for details on the importance of preventing JWT access tokens to be interpreted as id_tokens.

2.2. Data Structure

The following claims are used in the JWT access token data structure.

iss REQUIRED - as defined in section 2 of [OpenID.Core].

exp REQUIRED - as defined in section 2 of [OpenID.Core].

aud REQUIRED - as defined in section 2 of [OpenID.Core]. See Section 3 for indications on how an authorization server should determine the value of aud depending on the request. [Note: some vendors seem to rely on resource aliases. If we believe this to be a valuable feature, here's some proposed language: The aud claim MAY include a list of individual resource indicators if they are all aliases referring to the same requested resource known by the authorization server.]

sub REQUIRED - as defined in section 2 of [OpenID.Core]. . In case of access tokens obtained through grants where no resource owner is involved, such as the client credentials grant, the value of sub SHOULD correspond to an identifier the authorization server uses to indicate the client application (such as the client_id).

client_id REQUIRED - as defined in section 4.3 of [TokenExchange].

iat OPTIONAL - as defined in section 2 of [OpenID.Core].

auth_time OPTIONAL - as defined in section 2 of [OpenID.Core].
Important: as this claim represents the time at which the end user authenticated, its value will remain the same for all the JWT access tokens issued within that session. For example: all the JWT access tokens obtained with a given refresh token will all have the same value of auth_time, corresponding to the instant in which the user first authenticated to obtain the refresh token.

jti OPTIONAL - as defined in section 4.1.7 of [RFC7519].

acr, amr OPTIONAL - as defined in section 2 of [OpenID.Core]. The same considerations presented for auth_time apply to acr and amr: those values reflect the authentication context and method used when the end user originally authenticated, and will remain unchanged for the JWT access tokens issued within the context of that session.

2.2.1. Identity Claims

Commercial authorization servers will often include resource owner attributes directly in access tokens, so that resource servers can consume them directly for authorization or other purposes without any further roundtrips to introspection ([RFC7662]) or userinfo ([OpenID.Core]) endpoints.

This profile does not introduce any mechanism for a client to directly request the presence of specific claims in JWT access tokens, as the authorization server can determine what additional claims are required by a particular resource server by taking in consideration the client_id of the client, the scope and the resource parameters included in the request.

Any additional attributes whose semantic is well described by the attributes description found in section 5.1 of [OpenID.Core] SHOULD be codified in JWT access tokens via the corresponding claim names in that section of the OpenID Connect specification.

Authorization servers including resource owner attributes in JWT access tokens should exercise care and verify that all privacy requirements are met, as discussed in Section 6.

2.2.2. Authorization Claims

If an authorization request includes a scope parameter, the corresponding issued JWT access token MUST include a scope claim as defined in section 4.2 of [TokenExchange].

All the individual scopes strings in the scope claim MUST have meaning for the resource indicated in the aud claim.

2.2.2.1. Claims for Authorization Outside of Delegation Scenarios

Many authorization servers embed in the access tokens they issue authorization attributes that go beyond the delegated scenarios described by [RFC7519]. Typical examples include resource owner memberships in roles and groups that are relevant to the resource being accessed, entitlements assigned to the resource owner for the

targeted resource that the authorization server knows about, and so on.

An authorization server wanting to include such attributes in a JWT access token SHOULD use as claim types the attributes described by section 4.1.2 of SCIM Core ([RFC7643]) and in particular roles, groups and entitlements. As in their original definition in [RFC7643] , this profile does not provide a specific vocabulary for those entities.

[[note 1 some commercial authorization server include claims indicating whether the client authenticated with the authorization server as a confidential client, for the purpose of determining whether the client_id can be used as a reliable indicator of the identity of the caller (and take that into account for authorization decisions). Discussions at OSW2019 on how to achieve this were inconclusive hence this was punted for further discussion]]

[[note 2 some commercial authorization server include claims indicating whether the resource owner authenticated with a federated identity provider rather than directly with the authorization server. During discussions at OSW2019 there were lukewarm reactions. One proposed line of investigation was to examine what <https://tools.ietf.org/html/draft-ietf-secevent-token-13#page-10> does with the sub structure and see whether some mechanisms can be applicable here; however for this early draft no further investigation was made and no info is provided beyond this note]]

3. Requesting a JWT Access Token

An authorization server can issue a JWT access token in response to any authorization grant defined by [RFC6749] and subsequent extensions meant to result in an access token.

Every JWT access token MUST include an aud claim (see Section 2.2).

If the request includes a resource parameter (as defined in [ResourceIndicators]), the resulting JWT access token aud claim MUST have the same value as the resource parameter in the request.

Example request below:

```
GET /as/authorization.oauth2?response_type=token
    &client_id=s6BhdRkqt3&state=laeb
    &scope=openid%20profile%20reademail
    &redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
    &resource=https%3A%2F%2Frs.example.com%2F HTTP/1.1
Host: authorization-server.example.com
```

Figure 1: Authorization Request with Resource and Scope Parameters

Once redeemed, the code obtained from the request above will result in a JWT access token in the form shown below:

```
{"typ": "at+JWT", "alg": "RS256", "kid": "RjEwOwOA"}
{
  "iss": "https://authorization-server.example.com/",
  "sub": " 5ba552d67",
  "aud": "https://rs.example.com/",
  "exp": 1544645174,
  "client_id": "s6BhdRkqt3_",
  "scope": "openid profile reademail"
}
```

Figure 2: A JWT Access Token

If it receives a request for an access token containing more than one resource parameter, an authorization server issuing JWT access tokens MUST reject the request and fail with `[[TODO: select appropriate error code]]`. See Section 2.2 and Section 5 for more details on how this measure ensures there's no confusion on to what resource the access token granted scopes apply.

If the request does not include a resource parameter, the authorization server MUST use in the aud claim a default resource indicator. If a scope parameter is present in the request, the authorization server SHOULD use it to infer the value of the default resource indicator to be used in the aud claim. The mechanism through which scopes are associated to default resource indicator values is outside the scope of this specification. If the values in the scope parameter refer to different default resource indicator values, the authorization server SHOULD reject the request with `[[TODO: select appropriate error code]]`.

4. Validating JWT Access Tokens

For the purpose of facilitating validation data retrieval, it is RECOMMENDED that authorization servers sign JWT access tokens with an asymmetric algorithm.

Authorization servers SHOULD implement OAuth 2.0 Authorization Server Metadata [RFC8414] to advertise to resource servers its signing keys via `jwt_keys_uri` and what `iss` claim value to expect via the issuer metadata value. Alternatively, authorization servers implementing OpenID connect MAY use the OpenID connect discovery document for the same purpose. If an authorization server supports both AS metadata and OpenID discovery, the values provided MUST be consistent across the two publication methods.

An authorization server MAY elect to use different keys to sign `id_tokens` and JWT access tokens.

When invoked as described in OAuth2 bearer token usage, resource servers receiving a JWT access token MUST validate it in the following manner.

1. The resource server MUST verify that the `typ` header value is `at+jwt` and reject tokens carrying any other value.
2. If the JWT access token is encrypted, decrypt it using the keys and algorithms that the resource server specified during registration. If encryption was negotiated with the authorization server at registration time and the incoming JWT access token is not encrypted, the resource server SHOULD reject it.
3. The Issuer Identifier for the authorization server (which is typically obtained during discovery) MUST exactly match the value of the `iss` claim.
4. The resource server MUST validate that the `aud` claim contains the resource indicator value corresponding to the identifier the resource server expects for itself. The `aud` claim MAY contain an array with more than one element. The JWT access token MUST be rejected if `aud` does not list the resource indicator of the current resource server as a valid audience, or if it contains additional audiences that are not known aliases of the resource indicator of the current resource server.
5. The resource server MUST validate the signature of all incoming JWT access token according to [RFC7515] using the algorithm specified in the JWT `alg` Header Parameter. The resource server MUST use the keys provided by the authorization server.
6. The current time MUST be before the time represented by the `exp` Claim.

7. If the `auth_time` claim is present, the resource server SHOULD check the `auth_time` value and request re-authentication if it determines too much time has elapsed since the last resource owner authentication.

[[Note: I would like to express the requirement that the resource server should not ignore authorization information when present in the JWT access token. I don't know if this belongs here or elsewhere. Here's some possible language: If the JWT access token includes authorization claims as described in the authorization claims section, the resource server SHOULD use them in combination with any other contextual information available to determine whether the current call should be authorized or rejected. Details about how a resource server performs those checks is beyond the scope of this profile specification.]]

5. Security Considerations

The JWT access token data layout described here is very similar to the one of the `id_token` as defined by [OpenID.Core]. Without the explicit typing required in this profile, in line with the recommendations in [JWT.BestPractices] there would be the risk of attackers using JWT access tokens in lieu of `id_tokens`.

This profile explicitly forbids the use of multi value aud claim when the individual values refer to different resources, as that would introduce confusion about what scopes apply to which resource-possibly opening up avenues for elevation of delegated privileges attacks. Alternative techniques to prevent scope confusion include "scope stuffing", imposing to every individual scope string to include a reference to the resource they are meant to be applied to, but its application is problematic (scope opacity violations, size inflation, more error conditions become possible when the combination of requested scopes and resource indicators is invalid) and the observed frequency of the scenario doesn't warrant complicating the more common cases.

[[todo: expand on Audience, issuer and expiration validation checks serve the usual purposes. What else?]]

6. Privacy Considerations

As JWT access tokens carry information by value, it now becomes possible for requestors and receivers to directly peek inside the token claims collection.

In scenarios in which JWT access tokens are accessible to the end user, it should be evaluated whether the information can be accessed

without privacy violations (for example, if an end user would simply access his or her own personal information) or if the token should be encrypted.

In every scenario, the content of the JWT access token will eventually be accessible to the resource server. It's important to evaluate whether the resource server gained the proper entitlement to have access to any content received in form of claims, for example through user consent in some form, policies and agreements with the organization running the authorization servers, and so on.

7. IANA Considerations

[[TODO: MIME type registration for at+jwt]]

8. References

8.1. Normative References

- [IANA.OAuth.Parameters]
IANA, "OAuth Parameters",
<<http://www.iana.org/assignments/oauth-parameters>>.
- [JWT.BestPractices]
Sheffer, Y., Hardt, D., and M. Jones, "JSON Web Token Best Current Practices", November 2018.
- [OpenID.Core]
Sakimura, N., Bradley, J., Jones, M., Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0", November 2014.
- [ResourceIndicators]
Campbell, B., Bradley, J., and H. Tschofenig, "OAuth 2.0 Token Exchange", November 2016.
- [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>>.

- [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>>.
- [RFC7643] Hunt, P., Ed., Grizzle, K., Wahlstroem, E., and C. Mortimore, "System for Cross-domain Identity Management: Core Schema", RFC 7643, DOI 10.17487/RFC7643, September 2015, <<https://www.rfc-editor.org/info/rfc7643>>.
- [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>>.
- [TokenExchange]
Nadalin, A., Bradley, J., Jones, M., Campbell, B., and C. Mortimore, "OAuth 2.0 Token Exchange", October 2018.

8.2. Informative References

- [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

The initial set of requirements informing this specification was extracted by numerous examples of access tokens issued in JWT format by production systems. Thanks to Dominick Bauer (IdentityServer), Brian Campbell (PingIdentity), Daniel Dobalian (Microsoft), Karl

Guinness (Okta) for providing sample tokens issued by their products and services. Brian Campbell and Filip Skokan provided early feedback that shaped the direction of the specification. This profile was discussed at length during the OAuth Security Workshop 2019, with several individuals contributing ideas and feedback. The author would like to acknowledge the contributions of:

John Bradley, Brian Campbell Vladimir Dzhuvinov, Torsten Lodderstedt, Nat Sakimura, Hannes Tschofenig and everyone who actively participated in the unconference discussions.

Appendix B. Document History

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

draft-bertocci-oauth-access-token-jwt-00

- o Initial draft to define a JWTt profile for OAuth 2.0 access tokens.

Author's Address

Vittorio Bertocci
Auth0

Email: vittorio@auth0.com

Network Working Group
Internet-Draft
Intended status: Best Current Practice
Expires: 8 September 2022

A. Parecki
Okta
D. Waite
Ping Identity
7 March 2022

OAuth 2.0 for Browser-Based Apps
draft-ietf-oauth-browser-based-apps-09

Abstract

This specification details the security considerations and best practices that must be taken into account when developing browser-based applications that use OAuth 2.0.

Status of This Memo

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

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

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

This Internet-Draft will expire on 8 September 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction	2
2. Notational Conventions	3
3. Terminology	3
4. Overview	4
5. First-Party Applications	5
6. Application Architecture Patterns	6
6.1. Browser-Based Apps that Can Share Data with the Resource Server	6
6.2. JavaScript Applications with a Backend	7
6.3. JavaScript Applications without a Backend	8
7. Authorization Code Flow	10
7.1. Initiating the Authorization Request from a Browser-Based Application	10
7.2. Handling the Authorization Code Redirect	10
8. Refresh Tokens	10
9. Security Considerations	12
9.1. Registration of Browser-Based Apps	12
9.2. Client Authentication	12
9.3. Client Impersonation	13
9.4. Cross-Site Request Forgery Protections	13
9.5. Authorization Server Mix-Up Mitigation	13
9.6. Cross-Domain Requests	14
9.7. Content Security Policy	14
9.8. OAuth Implicit Flow	14
9.8.1. Attacks on the Implicit Flow	15
9.8.2. Countermeasures	16
9.8.3. Disadvantages of the Implicit Flow	16
9.8.4. Historic Note	17
9.9. Additional Security Considerations	17
10. IANA Considerations	17
11. References	17
11.1. Normative References	17
11.2. Informative References	18
Appendix A. Server Support Checklist	19
Appendix B. Document History	19
Appendix C. Acknowledgements	22
Authors' Addresses	22

1. Introduction

This specification describes the current best practices for implementing OAuth 2.0 authorization flows in applications executing in a browser.

For native application developers using OAuth 2.0 and OpenID Connect, an IETF BCP (best current practice) was published that guides integration of these technologies. This document is formally known as [RFC8252] or BCP 212, but nicknamed "AppAuth" after the OpenID Foundation-sponsored set of libraries that assist developers in adopting these practices. [RFC8252] makes specific recommendations for how to securely implement OAuth in native applications, including incorporating additional OAuth extensions where needed.

OAuth 2.0 for Browser-Based Apps addresses the similarities between implementing OAuth for native apps and browser-based apps, and includes additional considerations when running in a browser. This is primarily focused on OAuth, except where OpenID Connect provides additional considerations.

Many of these recommendations are derived from the OAuth 2.0 Security Best Current Practice [oauth-security-topics] and browser-based apps are expected to follow those recommendations as well. This draft expands on and further restricts various recommendations in [oauth-security-topics].

2. Notational Conventions

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

3. Terminology

In addition to the terms defined in referenced specifications, this document uses the following terms:

"OAuth": In this document, "OAuth" refers to OAuth 2.0, [RFC6749] and [RFC6750].

"Browser-based application": An application that is dynamically downloaded and executed in a web browser, usually written in JavaScript. Also sometimes referred to as a "single-page application", or "SPA".

4. Overview

At the time that OAuth 2.0 [RFC6749] and [RFC6750] were created, browser-based JavaScript applications needed a solution that strictly complied with the same-origin policy. Common deployments of OAuth 2.0 involved an application running on a different domain than the authorization server, so it was historically not possible to use the Authorization Code flow which would require a cross-origin POST request. This was one of the motivations for the definition of the Implicit flow, which returns the access token in the front channel via the fragment part of the URL, bypassing the need for a cross-origin POST request.

However, there are several drawbacks to the Implicit flow, generally involving vulnerabilities associated with the exposure of the access token in the URL. See Section 9.8 for an analysis of these attacks and the drawbacks of using the Implicit flow in browsers. Additional attacks and security considerations can be found in [oauth-security-topics].

In recent years, widespread adoption of Cross-Origin Resource Sharing (CORS), which enables exceptions to the same-origin policy, allows browser-based apps to use the OAuth 2.0 Authorization Code flow and make a POST request to exchange the authorization code for an access token at the token endpoint. In this flow, the access token is never exposed in the less secure front channel. Furthermore, adding PKCE to the flow ensures that even if an authorization code is intercepted, it is unusable by an attacker.

For this reason, and from other lessons learned, the current best practice for browser-based applications is to use the OAuth 2.0 Authorization Code flow with PKCE.

Browser-based applications:

- * MUST use the OAuth 2.0 Authorization Code flow with the PKCE extension when obtaining an access token
- * MUST Protect themselves against CSRF attacks by either:
 - ensuring the authorization server supports PKCE, or
 - by using the OAuth 2.0 "state" parameter or the OpenID Connect "nonce" parameter to carry one-time use CSRF tokens
- * MUST Register one or more redirect URIs, and use only exact registered redirect URIs in authorization requests

OAuth 2.0 authorization servers supporting browser-based applications:

- * MUST Require exact matching of registered redirect URIs
- * MUST Support the PKCE extension
- * MUST NOT issue access tokens in the authorization response
- * If issuing refresh tokens to browser-based applications, then:
 - MUST rotate refresh tokens on each use or use sender-constrained refresh tokens, and
 - MUST set a maximum lifetime on refresh tokens or expire if they are not used in some amount of time

5. First-Party Applications

While OAuth was initially created to allow third-party applications to access an API on behalf of a user, it has proven to be useful in a first-party scenario as well. First-party apps are applications where the same organization provides both the API and the application.

Examples of first-party applications are a web email client provided by the operator of the email account, or a mobile banking application created by bank itself. (Note that there is no requirement that the application actually be developed by the same company; a mobile banking application developed by a contractor that is branded as the bank's application is still considered a first-party application.) The first-party app consideration is about the user's relationship to the application and the service.

To conform to this best practice, first-party applications using OAuth or OpenID Connect MUST use a redirect-based flow (such as the OAuth Authorization Code flow) as described later in this document.

The resource owner password credentials grant MUST NOT be used, as described in [oauth-security-topics] Section 2.4. Instead, by using the Authorization Code flow and redirecting the user to the authorization server, this provides the authorization server the opportunity to prompt the user for multi-factor authentication options, take advantage of single sign-on sessions, or use third-party identity providers. In contrast, the resource owner password credentials grant does not provide any built-in mechanism for these, and would instead be extended with custom code.

6. Application Architecture Patterns

There are three primary architectural patterns available when building browser-based applications.

- * a JavaScript application that has methods of sharing data with resource servers, such as using common-domain cookies
- * a JavaScript application with a backend component
- * a JavaScript application with no backend, accessing resource servers directly

These three architectures have different use cases and considerations.

6.1. Browser-Based Apps that Can Share Data with the Resource Server

For simple system architectures, such as when the JavaScript application is served from a domain that can share cookies with the domain of the API (resource server), OAuth adds additional attack vectors that could be avoided with a different solution.

In particular, using any redirect-based mechanism of obtaining an access token enables the redirect-based attacks described in [oauth-security-topics] Section 4, but if the application, authorization server and resource server share a domain, then it is unnecessary to use a redirect mechanism to communicate between them.

An additional concern with handling access tokens in a browser is that as of the date of this publication, there is no secure storage mechanism where JavaScript code can keep the access token to be later used in an API request. Using an OAuth flow results in the JavaScript code getting an access token, needing to store it somewhere, and then retrieve it to make an API request.

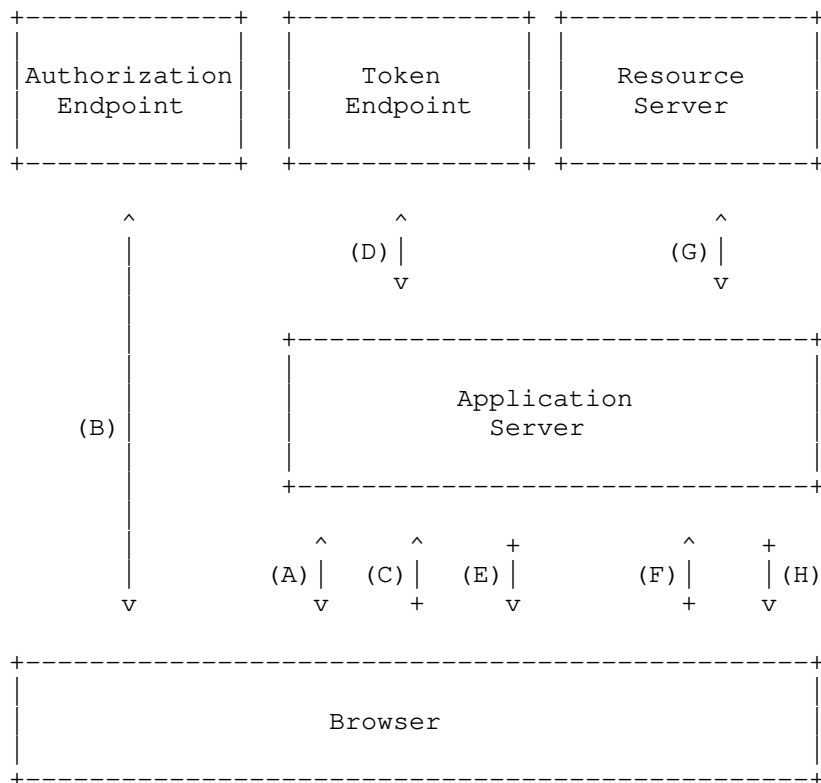
Instead, a more secure design is to use an HTTP-only cookie between the JavaScript application and API so that the JavaScript code can't access the cookie value itself. The Secure cookie attribute should be used to ensure the cookie is not included in unencrypted HTTP requests. Additionally, the SameSite cookie attribute can be used to counter CSRF attacks, but should not be considered the extent of the CSRF protection, as described in [draft-ietf-httpbis-rfc6265bis]

OAuth was originally created for third-party or federated access to APIs, so it may not be the best solution in a common-domain deployment. That said, there are still some advantages in using OAuth even in a common-domain architecture:

- * Allows more flexibility in the future, such as if you were to later add a new domain to the system. With OAuth already in place, adding a new domain wouldn't require any additional rearchitecting.
- * Being able to take advantage of existing library support rather than writing bespoke code for the integration.
- * Centralizing login and multifactor support, account management, and recovery at the OAuth server, rather than making it part of the application logic.

Using OAuth for browser-based apps in a first-party same-domain scenario provides these advantages, and can be accomplished by either of the two architectural patterns described below.

6.2. JavaScript Applications with a Backend



In this architecture, commonly referred to as "backend for frontend" or "BFF", the JavaScript code is loaded from a dynamic Application Server (A) that also has the ability to execute code itself. This enables the ability to keep all of the steps involved in obtaining an access token outside of the JavaScript application.

Note that this application backend is not the Resource Server, it is still considered part of the OAuth client and would be accessing data at a separate resource server.

In this case, the Application Server initiates the OAuth flow itself, by redirecting the browser to the authorization endpoint (B). When the user is redirected back, the browser delivers the authorization code to the application server (C), where it can then exchange it for an access token at the token endpoint (D) using its client secret. The application server then keeps the access token and refresh token stored internally, and creates a separate session with the browser-based app via a traditional browser cookie (E).

When the JavaScript application in the browser wants to make a request to the Resource Server, it instead makes the request to the Application Server (F), and the Application Server will make the request with the access token to the Resource Server (G), and forward the response (H) back to the browser.

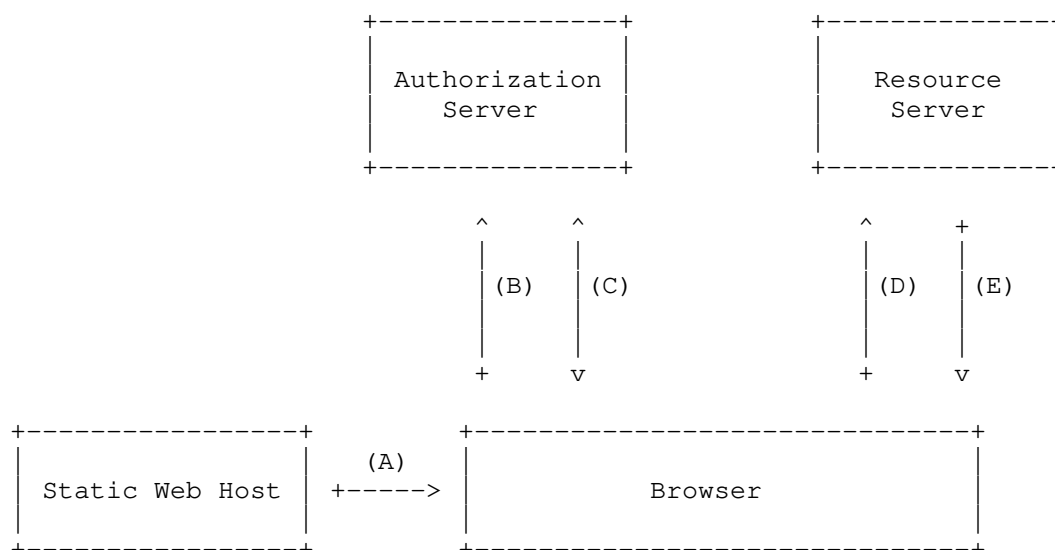
(Common examples of this architecture are an Angular front-end with a .NET backend, or a React front-end with a Spring Boot backend.)

The Application Server SHOULD be considered a confidential client, and issued its own client secret. The Application Server SHOULD use the OAuth 2.0 Authorization Code grant with PKCE to initiate a request for an access token. Detailed recommendations for confidential clients can be found in [oauth-security-topics] Section 2.1.1.

In this scenario, the connection between the browser and Application Server SHOULD be a session cookie provided by the Application Server.

Security of the connection between code running in the browser and this Application Server is assumed to utilize browser-level protection mechanisms. Details are out of scope of this document, but many recommendations can be found in the OWASP Cheat Sheet series (<https://cheatsheetseries.owasp.org/>), such as setting an HTTP-only and Secure cookie to authenticate the session between the browser and Application Server.

6.3. JavaScript Applications without a Backend



In this architecture, the JavaScript code is first loaded from a static web host into the browser (A), and the application then runs in the browser. This application is considered a public client, since there is no way to issue it a client secret and there is no other secure client authentication mechanism available in the browser.

The code in the browser initiates the Authorization Code flow with the PKCE extension (described in Section 7) (B) above, and obtains an access token via a POST request (C). The JavaScript application is then responsible for storing the access token (and optional refresh token) as securely as possible using appropriate browser APIs. As of the date of this publication there is no browser API that allows to store tokens in a completely secure way.

When the JavaScript application in the browser wants to make a request to the Resource Server, it can interact with the Resource Server directly. It includes the access token in the request (D) and receives the Resource Server's response (E).

In this scenario, the Authorization Server and Resource Server MUST support the necessary CORS headers to enable the JavaScript code to make this POST request from the domain on which the script is executing. (See Section 9.6 for additional details.)

7. Authorization Code Flow

Browser-based applications that are public clients and use the Authorization Code grant type described in Section 4.1 of OAuth 2.0 [RFC6749] MUST also follow these additional requirements described in this section.

7.1. Initiating the Authorization Request from a Browser-Based Application

Browser-based applications that are public clients MUST implement the Proof Key for Code Exchange (PKCE [RFC7636]) extension when obtaining an access token, and authorization servers MUST support and enforce PKCE for such clients.

The PKCE extension prevents an attack where the authorization code is intercepted and exchanged for an access token by a malicious client, by providing the authorization server with a way to verify the client instance that exchanges the authorization code is the same one that initiated the flow.

Browser-based applications MUST prevent CSRF attacks against their redirect URI. This can be accomplished by any of the below:

- * using PKCE, and confirming that the authorization server supports PKCE
- * using a unique value for the OAuth 2.0 "state" parameter
- * if the application is using OpenID Connect, by using the OpenID Connect "nonce" parameter

7.2. Handling the Authorization Code Redirect

Authorization servers MUST require an exact match of a registered redirect URI. As described in [oauth-security-topics] Section 4.1.1. this helps to prevent attacks targeting the authorization code.

8. Refresh Tokens

Refresh tokens provide a way for applications to obtain a new access token when the initial access token expires. With public clients, the risk of a leaked refresh token is greater than leaked access tokens, since an attacker may be able to continue using the stolen refresh token to obtain new access tokens potentially without being detectable by the authorization server.

Browser-based applications provide an attacker with several opportunities by which a refresh token can be leaked, just as with access tokens. As such, these applications are considered a higher risk for handling refresh tokens.

Authorization servers may choose whether or not to issue refresh tokens to browser-based applications. [oauth-security-topics] describes some additional requirements around refresh tokens on top of the recommendations of [RFC6749]. Applications and authorization servers conforming to this BCP MUST also follow the recommendations in [oauth-security-topics] around refresh tokens if refresh tokens are issued to browser-based applications.

In particular, authorization servers:

- * MUST either rotate refresh tokens on each use OR use sender-constrained refresh tokens as described in [oauth-security-topics] Section 4.13.2
- * MUST either set a maximum lifetime on refresh tokens OR expire if the refresh token has not been used within some amount of time
- * MUST NOT extend the lifetime of the new refresh token beyond the lifetime of the initial refresh token
- * upon issuing a rotated refresh token, MUST NOT extend the lifetime of the new refresh token beyond the lifetime of the initial refresh token if the refresh token has a preestablished expiration time

For example:

- * A user authorizes an application, issuing an access token that lasts 1 hour, and a refresh token that lasts 24 hours
- * After 1 hour, the initial access token expires, so the application uses the refresh token to get a new access token
- * The authorization server returns a new access token that lasts 1 hour, and a new refresh token that lasts 23 hours
- * This continues until 24 hours pass from the initial authorization
- * At this point, when the application attempts to use the refresh token after 24 hours, the request will fail and the application will have to involve the user in a new authorization request

By limiting the overall refresh token lifetime to the lifetime of the initial refresh token, this ensures a stolen refresh token cannot be used indefinitely.

Authorization servers MAY set different policies around refresh token issuance, lifetime and expiration for browser-based applications compared to other public clients.

9. Security Considerations

9.1. Registration of Browser-Based Apps

Browser-based applications are considered public clients as defined by Section 2.1 of OAuth 2.0 [RFC6749], and MUST be registered with the authorization server as such. Authorization servers MUST record the client type in the client registration details in order to identify and process requests accordingly.

Authorization servers MUST require that browser-based applications register one or more redirect URIs.

9.2. Client Authentication

Since a browser-based application's source code is delivered to the end-user's browser, it cannot contain provisioned secrets. As such, a browser-based app with native OAuth support is considered a public client as defined by Section 2.1 of OAuth 2.0 [RFC6749].

Secrets that are statically included as part of an app distributed to multiple users should not be treated as confidential secrets, as one user may inspect their copy and learn the shared secret. For this reason, and those stated in Section 5.3.1 of [RFC6819], it is NOT RECOMMENDED for authorization servers to require client authentication of browser-based applications using a shared secret, as this serves little value beyond client identification which is already provided by the `client_id` request parameter.

Authorization servers that still require a statically included shared secret for SPA clients MUST treat the client as a public client, and not accept the secret as proof of the client's identity. Without additional measures, such clients are subject to client impersonation (see Section 9.3 below).

9.3. Client Impersonation

As stated in Section 10.2 of OAuth 2.0 [RFC6749], the authorization server SHOULD NOT process authorization requests automatically without user consent or interaction, except when the identity of the client can be assured.

If authorization servers restrict redirect URIs to a fixed set of absolute HTTPS URIs, preventing the use of wildcard domains, wildcard paths, or wildcard query string components, this exact match of registered absolute HTTPS URIs MAY be accepted by authorization servers as proof of identity of the client for the purpose of deciding whether to automatically process an authorization request when a previous request for the client_id has already been approved.

9.4. Cross-Site Request Forgery Protections

Clients MUST prevent Cross-Site Request Forgery (CSRF) attacks against their redirect URI. Clients can accomplish this by either ensuring the authorization server supports PKCE and relying on the CSRF protection that PKCE provides, or if the client is also an OpenID Connect client, using the OpenID Connect "nonce" parameter, or by using the "state" parameter to carry one-time-use CSRF tokens as described in Section 7.1.

See Section 2.1 of [oauth-security-topics] for additional details.

9.5. Authorization Server Mix-Up Mitigation

Authorization server mix-up attacks mark a severe threat to every client that supports at least two authorization servers. To conform to this BCP such clients MUST apply countermeasures to defend against mix-up attacks.

It is RECOMMENDED to defend against mix-up attacks by identifying and validating the issuer of the authorization response. This can be achieved either by using the "iss" response parameter, as defined in [oauth-iss-auth-resp], or by using the "iss" Claim of the ID token when OpenID Connect is used.

Alternative countermeasures, such as using distinct redirect URIs for each issuer, SHOULD only be used if identifying the issuer as described is not possible.

Section 4.4 of [oauth-security-topics] provides additional details about mix-up attacks and the countermeasures mentioned above.

9.6. Cross-Domain Requests

To complete the Authorization Code flow, the browser-based application will need to exchange the authorization code for an access token at the token endpoint. If the authorization server provides additional endpoints to the application, such as metadata URLs, dynamic client registration, revocation, introspection, discovery or user info endpoints, these endpoints may also be accessed by the browser-based app. Since these requests will be made from a browser, authorization servers **MUST** support the necessary CORS headers (defined in [Fetch]) to allow the browser to make the request.

This specification does not include guidelines for deciding whether a CORS policy for the token endpoint should be a wildcard origin or more restrictive. Note, however, that the browser will attempt to GET or POST to the API endpoint before knowing any CORS policy; it simply hides the succeeding or failing result from JavaScript if the policy does not allow sharing.

9.7. Content Security Policy

A browser-based application that wishes to use either long-lived refresh tokens or privileged scopes **SHOULD** restrict its JavaScript execution to a set of statically hosted scripts via a Content Security Policy ([CSP2]) or similar mechanism. A strong Content Security Policy can limit the potential attack vectors for malicious JavaScript to be executed on the page.

9.8. OAuth Implicit Flow

The OAuth 2.0 Implicit flow (defined in Section 4.2 of OAuth 2.0 [RFC6749]) works by the authorization server issuing an access token in the authorization response (front channel) without the code exchange step. In this case, the access token is returned in the fragment part of the redirect URI, providing an attacker with several opportunities to intercept and steal the access token.

Authorization servers **MUST NOT** issue access tokens in the authorization response, and **MUST** issue access tokens only from the token endpoint.

9.8.1. Attacks on the Implicit Flow

Many attacks on the Implicit flow described by [RFC6819] and Section 4.1.2 of [oauth-security-topics] do not have sufficient mitigation strategies. The following sections describe the specific attacks that cannot be mitigated while continuing to use the Implicit flow.

9.8.1.1. Threat: Manipulation of the Redirect URI

If an attacker is able to cause the authorization response to be sent to a URI under their control, they will directly get access to the authorization response including the access token. Several methods of performing this attack are described in detail in [oauth-security-topics].

9.8.1.2. Threat: Access Token Leak in Browser History

An attacker could obtain the access token from the browser's history. The countermeasures recommended by [RFC6819] are limited to using short expiration times for tokens, and indicating that browsers should not cache the response. Neither of these fully prevent this attack, they only reduce the potential damage.

Additionally, many browsers now also sync browser history to cloud services and to multiple devices, providing an even wider attack surface to extract access tokens out of the URL.

This is discussed in more detail in Section 4.3.2 of [oauth-security-topics].

9.8.1.3. Threat: Manipulation of Scripts

An attacker could modify the page or inject scripts into the browser through various means, including when the browser's HTTPS connection is being intercepted by, for example, a corporate network. While man-in-the-middle attacks are typically out of scope of basic security recommendations to prevent, in the case of browser-based apps they are much easier to perform. An injected script can enable an attacker to have access to everything on the page.

The risk of a malicious script running on the page may be amplified when the application uses a known standard way of obtaining access tokens, namely that the attacker can always look at the `window.location` variable to find an access token. This threat profile is different from an attacker specifically targeting an individual application by knowing where or how an access token obtained via the Authorization Code flow may end up being stored.

9.8.1.4. Threat: Access Token Leak to Third-Party Scripts

It is relatively common to use third-party scripts in browser-based apps, such as analytics tools, crash reporting, and even things like a Facebook or Twitter "like" button. In these situations, the author of the application may not be able to be fully aware of the entirety of the code running in the application. When an access token is returned in the fragment, it is visible to any third-party scripts on the page.

9.8.2. Countermeasures

In addition to the countermeasures described by [RFC6819] and [oauth-security-topics], using the Authorization Code flow with PKCE extension prevents the attacks described above by avoiding returning the access token in the redirect response at all.

When PKCE is used, if an authorization code is stolen in transport, the attacker is unable to do anything with the authorization code.

9.8.3. Disadvantages of the Implicit Flow

There are several additional reasons the Implicit flow is disadvantageous compared to using the standard Authorization Code flow.

- * OAuth 2.0 provides no mechanism for a client to verify that a particular access token was intended for that client, which could lead to misuse and possible impersonation attacks if a malicious party hands off an access token it retrieved through some other means to the client.
- * Returning an access token in the front-channel redirect gives the authorization server no assurance that the access token will actually end up at the application, since there are many ways this redirect may fail or be intercepted.
- * Supporting the Implicit flow requires additional code, more upkeep and understanding of the related security considerations, while limiting the authorization server to just the Authorization Code flow reduces the attack surface of the implementation.
- * If the JavaScript application gets wrapped into a native app, then [RFC8252] also requires the use of the Authorization Code flow with PKCE anyway.

In OpenID Connect, the ID Token is sent in a known format (as a JWT), and digitally signed. Returning an ID token using the Implicit flow (`response_type=id_token`) requires the client validate the JWT signature, as malicious parties could otherwise craft and supply fraudulent ID tokens. Performing OpenID Connect using the Authorization Code flow provides the benefit of the client not needing to verify the JWT signature, as the ID token will have been fetched over an HTTPS connection directly from the authorization server. Additionally, in many cases an application will request both an ID token and an access token, so it is simpler and provides fewer attack vectors to obtain both via the Authorization Code flow.

9.8.4. Historic Note

Historically, the Implicit flow provided an advantage to browser-based apps since JavaScript could always arbitrarily read and manipulate the fragment portion of the URL without triggering a page reload. This was necessary in order to remove the access token from the URL after it was obtained by the app.

Modern browsers now have the Session History API (described in "Session history and navigation" of [HTML]), which provides a mechanism to modify the path and query string component of the URL without triggering a page reload. This means modern browser-based apps can use the unmodified OAuth 2.0 Authorization Code flow, since they have the ability to remove the authorization code from the query string without triggering a page reload thanks to the Session History API.

9.9. Additional Security Considerations

The OWASP Foundation (<https://www.owasp.org/>) maintains a set of security recommendations and best practices for web applications, and it is RECOMMENDED to follow these best practices when creating an OAuth 2.0 Browser-Based application.

10. IANA Considerations

This document does not require any IANA actions.

11. References

11.1. Normative References

[CSP2] West, M., "Content Security Policy", October 2018.

- [draft-ietf-httpbis-rfc6265bis]
Chen, L., Englehardt, S., West, M., and J. Wilander,
"Cookies: HTTP State Management Mechanism", October 2021.
- [Fetch] whatwg, ., "Fetch", 2018.
- [oauth-iss-auth-resp]
Meyer zu Selhausen, K. and D. Fett, "OAuth 2.0
Authorization Server Issuer Identifier in Authorization
Response", January 2021.
- [oauth-security-topics]
Lodderstedt, T., Bradley, J., Labunets, A., and D. Fett,
"OAuth 2.0 Security Best Current Practice", April 2021.
- [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>>.
- [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>>.
- [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>>.
- [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>>.

11.2. Informative References

- [HTML] whatwg, ., "HTML", 2020.

Appendix A. Server Support Checklist

OAuth authorization servers that support browser-based apps MUST:

1. Require "https" scheme redirect URIs.
2. Require exact matching of registered redirect URIs.
3. Support PKCE [RFC7636]. Required to protect authorization code grants sent to public clients. See Section 7.1
4. Support cross-domain requests at the token endpoint in order to allow browsers to make the authorization code exchange request. See Section 9.6
5. Not assume that browser-based clients can keep a secret, and SHOULD NOT issue secrets to applications of this type.
6. Not support the Resource Owner Password grant for browser-based clients.
7. Follow the [oauth-security-topics] recommendations on refresh tokens, as well as the additional requirements described in Section 8.

Appendix B. Document History

[[To be removed from the final specification]]

-09

- * Provide additional context for the same-domain architecture pattern
- * Added reference to draft-ietf-httpbis-rfc6265bis to clarify that SameSite is not the only CSRF protection measure needed
- * Editorial improvements

-08

- * Added a note to use the "Secure" cookie attribute in addition to SameSite etc
- * Updates to bring this draft in sync with the latest Security BCP
- * Updated text for mix-up countermeasures to reference the new "iss" extension

- * Changed "SHOULD" for refresh token rotation to MUST either use rotation or sender-constraining to match the Security BCP
- * Fixed references to other specs and extensions
- * Editorial improvements in descriptions of the different architectures

-07

- * Clarify PKCE requirements apply only to issuing access tokens
- * Change "MUST" to "SHOULD" for refresh token rotation
- * Editorial clarifications

-06

- * Added refresh token requirements to AS summary
- * Editorial clarifications

-05

- * Incorporated editorial and substantive feedback from Mike Jones
- * Added references to "nonce" as another way to prevent CSRF attacks
- * Updated headers in the Implicit Flow section to better represent the relationship between the paragraphs

-04

- * Disallow the use of the Password Grant
- * Add PKCE support to summary list for authorization server requirements
- * Rewrote refresh token section to allow refresh tokens if they are time-limited, rotated on each use, and requiring that the rotated refresh token lifetimes do not extend past the lifetime of the initial refresh token, and to bring it in line with the Security BCP
- * Updated recommendations on using state to reflect the Security BCP
- * Updated server support checklist to reflect latest changes

- * Updated the same-domain JS architecture section to emphasize the architecture rather than domain
- * Editorial clarifications in the section that talks about OpenID Connect ID tokens

-03

- * Updated the historic note about the fragment URL clarifying that the Session History API means browsers can use the unmodified authorization code flow
- * Rephrased "Authorization Code Flow" intro paragraph to better lead into the next two sections
- * Softened "is likely a better decision to avoid using OAuth entirely" to "it may be..." for common-domain deployments
- * Updated abstract to not be limited to public clients, since the later sections talk about confidential clients
- * Removed references to avoiding OpenID Connect for same-domain architectures
- * Updated headers to better describe architectures (Apps Served from a Static Web Server -> JavaScript Applications without a Backend)
- * Expanded "same-domain architecture" section to better explain the problems that OAuth has in this scenario
- * Referenced Security BCP in implicit flow attacks where possible
- * Minor typo corrections

-02

- * Rewrote overview section incorporating feedback from Leo Tohill
- * Updated summary recommendation bullet points to split out application and server requirements
- * Removed the allowance on hostname-only redirect URI matching, now requiring exact redirect URI matching
- * Updated Section 6.2 to drop reference of SPA with a backend component being a public client

- * Expanded the architecture section to explicitly mention three architectural patterns available to JS apps

-01

- * Incorporated feedback from Torsten Lodderstedt
- * Updated abstract
- * Clarified the definition of browser-based apps to not exclude applications cached in the browser, e.g. via Service Workers
- * Clarified use of the state parameter for CSRF protection
- * Added background information about the original reason the implicit flow was created due to lack of CORS support
- * Clarified the same-domain use case where the SPA and API share a cookie domain
- * Moved historic note about the fragment URL into the Overview

Appendix C. Acknowledgements

The authors would like to acknowledge the work of William Denniss and John Bradley, whose recommendation for native apps informed many of the best practices for browser-based applications. The authors would also like to thank Hannes Tschofenig and Torsten Lodderstedt, the attendees of the Internet Identity Workshop 27 session at which this BCP was originally proposed, and the following individuals who contributed ideas, feedback, and wording that shaped and formed the final specification:

Annabelle Backman, Brian Campbell, Brock Allen, Christian Mainka, Daniel Fett, George Fletcher, Hannes Tschofenig, Janak Amarasena, John Bradley, Joseph Heenan, Justin Richer, Karl McGuinness, Karsten Meyer zu Selhausen, Leo Tohill, Mike Jones, Tomek Stojekci, Torsten Lodderstedt, and Vittorio Bertocci.

Authors' Addresses

Aaron Parecki
Okta
Email: aaron@parecki.com
URI: <https://aaronparecki.com>

David Waite
Ping Identity
Email: david@alkaline-solutions.com

Open Authentication Protocol
Internet-Draft
Intended status: Standards Track
Expires: 8 March 2022

T. Lodderstedt, Ed.
yes.com AG
V. Dzhuvinov
Connect2id Ltd.
4 September 2021

JWT Response for OAuth Token Introspection
draft-ietf-oauth-jwt-introspection-response-12

Abstract

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

Status of This Memo

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

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

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

This Internet-Draft will expire on 8 March 2022.

Copyright Notice

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

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

Table of Contents

1. Introduction	2
2. Requirements Notation and Conventions	3
3. Resource Server Management	3
4. Requesting a JWT Response	4
5. JWT Response	4
6. Client Metadata	7
7. Authorization Server Metadata	8
8. Security Considerations	9
8.1. Cross-JWT Confusion	9
8.2. Token Data Leakage	9
9. Privacy Considerations	9
10. Acknowledgements	10
11. IANA Considerations	10
11.1. OAuth Dynamic Client Registration Metadata Registration	10
11.1.1. Registry Contents	10
11.2. OAuth Authorization Server Metadata Registration	11
11.2.1. Registry Contents	11
11.3. Media Type Registration	12
11.3.1. Registry Contents	12
11.4. JWT Claim Registration	13
11.4.1. Registry Contents	13
12. References	13
12.1. Normative References	13
12.2. Informative References	15
Appendix A. Document History	15
Authors' Addresses	18

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.

The authorization server MAY additionally encrypt the token introspection response JWTs. If encryption is used the authorization server is provisioned with encryption keys and algorithms for the RS.

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 scope 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 a "client_id" and a "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 sending an introspection request with an "Accept" HTTP header field set to "application/token-introspection+jwt".

The AS MUST authenticate the caller at the token introspection endpoint. Authentication can utilize client authentication methods or a separate access token issued to the resource server and identifying it as subject.

The following is a non-normative example request, with the resource server authenticating with a private key JWT:

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

```
token=2YotnFZFEjrlzCsicMWpAA&
client_assertion_type=
  urn%3Aietf%3Aparams%3Aoauth%3Aclient-assertion-type%3Ajwt-bearer&
client_assertion=PHNhbWxwOl[...omitted for brevity...]ZT
```

5. JWT Response

The introspection endpoint responds with a JWT, setting the "Content-Type" HTTP header field to "application/token-introspection+jwt" and the JWT "typ" ("type") header parameter 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 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 MUST NOT include other members. Otherwise, the "active" member is set to "true".

The AS SHOULD narrow down the "scope" value to the scopes relevant to the particular RS.

As specified in section 2.2 of [RFC7662], 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.

Token introspection response parameter names intended to be used across domains MUST be registered in the OAuth Token Introspection Response registry [IANA.OAuth.Token.Introspection] defined by [RFC7662].

When the AS acts as a provider of resource owner identity claims to the RS, the AS determines based on its RS-specific policy what identity claims 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 [RFC7519].

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: An AS compliant with this specification MUST refuse to serve introspection requests that don't authenticate the caller, and return an HTTP status code 400. This is done to ensure token data is released to legitimate recipients only and prevent downgrading to [RFC7662] behavior (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

```
eyJraWQioiJ3RzZEI1widHlwIjojdG9rZW4taW50cm9zcGVjdGlvbitqd3QiLCJhbGciOiJSUzI1NiJ9.eyJpc3MiOiJodHRwczovL2FzLmV4YW1wbGUuY29tLyIsImF1ZCI6Imh0dHBzOi8vcnMuZXhhbXBsZS5jb20vcmlvZ3VyY2UiLCJpYXQiOiJlMTQ3OTc0OTIsInRva2VuX2ludHJvc3BlY3Rpb24iOnsiYWN0aXZlIjp0cnVlLCJpc3MiOiJodHRwczovL2FzLmV4YW1wbGUuY29tLyIsImF1ZCI6Imh0dHBzOi8vcnMuZXhhbXBsZS5jb20vcmlvZ3VyY2UiLCJpYXQiOiJlMTQ3OTc0MjIsImV4cCI6MTUxNDc5Nzk0MiwiaWY2xpZW50X2lkIjoicGFpQjJnb28wYSIsInNjb3BlIjoicmVhZCB3cm10ZSBkb2xwaGluIiwic3ViIjoiwjVPM3VwUEM4OEFyQWp4MDBkaXMiLCJiaXJ0aGRhdGUiOiIxOTgyLTAYLTAXIiwia2Z12ZW5fcmFtZSI6IkpvaG4iLCJmYW1pbHlfbmFtZSI6IkRvZSIsImp0aSI6InQxRm9DQ2FaZDRYdJRPUpkpVV1ZVZVRaZnNLaFczMENRQ3JXRERqd1h5NncifX0.przJMU5GhmNzvwt1t1sr-xa9xTkpiAg5IshbQScRiRVP_7eGR1GHYrNwQh84kxOoHCYjq2g5WSRcYoSGEVIIiC-eoPJJ-qBwqWslgxJEEcdW2W5Djrb1OI_N0Jvsq_dUEoyOwVMq1OydbHKNY0smBrI4N2VEXeXcm9MuYJXMuJtvq1gBes-0go5j4TEV9sOP9uu81gqWTr_L0O6pdtT0tFFyZfWC4kbXPXiQ2YT6mxCiQRRNM-19cBdF6Jx6IOrsfFhBuYdYQ_mlL19HgDDOFaleymru6lKlASOsaE8dmLSeKcX91FbG79FKN8un24iwiDCbKT9xlUF154xWVShNDFA
```

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": "t1FoCCaZd4Xv40RJUVUeTZfsKhW30CQCrWDDjwXy6w"
  }
}
```

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.

Section 2.1 of [RFC7662] permits requests to the introspection endpoint to be authorized with an access token which doesn't identify the caller. To prevent introspection of tokens by parties that are not the intended consumer the authorization server 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 (PII) from the AS to the RS. The AS MUST conform to legal and jurisdictional constraints for the data transfer before any data is released to a particular RS. The details and determining of these constraints varies by jurisdiction and is outside the scope of this document.

A commonly found way to establish the legal basis for releasing PII 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, for example 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 authenticate 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 implication is not acceptable, implementers MUST use other means to relay access token data, for example by 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, Takahiko Kawasaki, Benjamin Kaduk, Robert Wilton and Roman Danyliw 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

- * Type name: application
- * Subtype name: token-introspection+jwt
- * Required parameters: N/A
- * Optional parameters: N/A
- * 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.
- * Security considerations: See Section 7 of this specification
- * Interoperability considerations: N/A
- * Published specification: Section 4 of this specification
- * Applications that use this media type: Applications that produce and consume OAuth Token Introspection Responses in JWT format
- * Fragment identifier considerations: N/A
- * Additional information:
 - Magic number(s): N/A

- File extension(s): N/A
- Macintosh file type code(s): N/A
- * Person & email address to contact for further information: Torsten Lodderstedt, torsten@lodderstedt.net
- * Intended usage: COMMON
- * Restrictions on usage: none
- * Author: Torsten Lodderstedt, torsten@lodderstedt.net
- * Change controller: IESG
- * 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", Work in Progress, Internet-Draft, draft-ietf-oauth-jwt-bcp-06, 7 June 2019, <<http://www.ietf.org/internet-drafts/draft-ietf-oauth-jwt-bcp-06.txt>>.

- [I-D.ietf-oauth-security-topics]
Lodderstedt, T., Bradley, J., Labunets, A., and D. Fett,
"OAuth 2.0 Security Best Current Practice", Work in
Progress, Internet-Draft, draft-ietf-oauth-security-
topics-13, 8 July 2019, <<http://www.ietf.org/internet-drafts/draft-ietf-oauth-security-topics-13.txt>>.
- [IANA.JWT] IANA, "JSON Web Token (JWT) claims registry",
<<https://www.iana.org/assignments/jwt/jwt.xhtml#claims>>.
- [IANA.MediaTypees]
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", 8 November 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]]

-12

- * made registration of response parameters intended for cross domain use a MUST (in RFC 7662)

-11

- * consistent normative language that the AS must authenticate all callers to the token introspection endpoint when complying with this specification

- * removes text that claims from the JSON Web Token Claims registry may be included in the token_introspection claim
- * updates the privacy considerations section
- * fixes the example BASE64URL encoded JWT payload

-10

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

-09

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

-08

- * made difference between introspected access token and introspection response clearer
- * defined semantics of JWT claims overlapping between introspected access token and introspection response as JWT
- * added section about RS management
- * added text about user claims including a privacy considerations section

- * removed registration of OpenID Connect claims to "Token Introspection Response" registry and refer to "JWT Claims" registry instead
- * added registration of "application/token-introspection+jwt" media type as type identifier of token introspection responses in JWT format
- * more changed to incorporate IESG review feedback

-07

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

-06

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

-05

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

-04

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

-03

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

-02

- * updated references

-01

- * adapted wording to preclude any accept header except "application/jwt" if encrypted responses are required
- * use registered alg value RS256 for default signing algorithm
- * 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

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

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

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

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

Copyright Notice

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

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

Table of Contents

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

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

1. Introduction

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

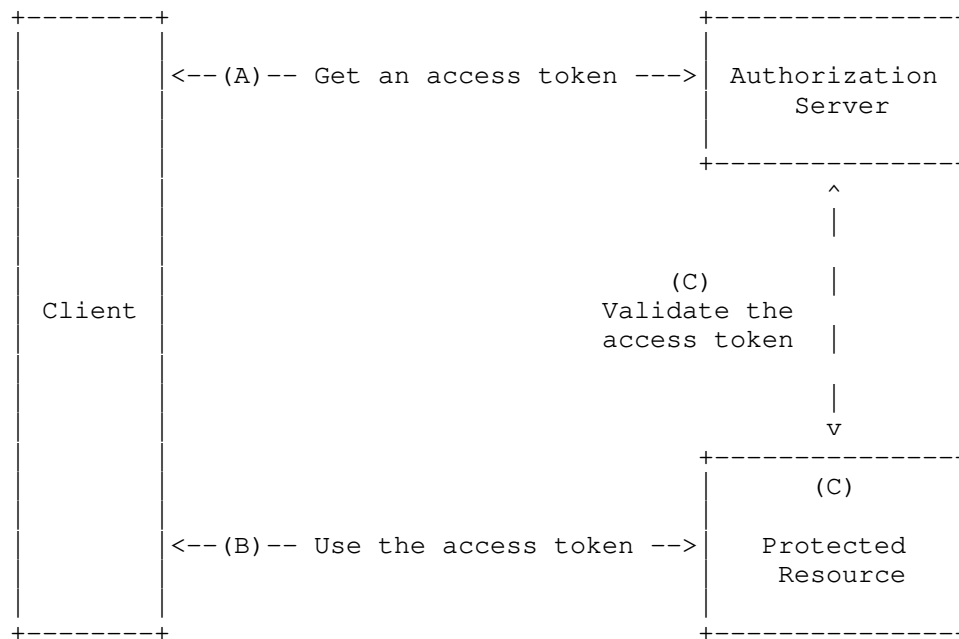


Figure 1: Abstract OAuth 2.0 Protocol Flow

The flow illustrated in Figure 1 includes the following steps:

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

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

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

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

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

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

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

1.1. Requirements Notation and Conventions

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

1.2. Terminology

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

2. Mutual TLS for OAuth Client Authentication

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

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

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

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

2.1. PKI Mutual-TLS Method

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

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

2.1.1. PKI Method Metadata Value

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

`tls_client_auth`

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

2.1.2. Client Registration Metadata

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

`tls_client_auth_subject_dn`

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

`tls_client_auth_san_dns`

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

`tls_client_auth_san_uri`

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

`tls_client_auth_san_ip`

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

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

`tls_client_auth_san_email`

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

2.2. Self-Signed Certificate Mutual-TLS Method

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

2.2.1. Self-Signed Method Metadata Value

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

`self_signed_tls_client_auth`

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

2.2.2. Client Registration Metadata

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

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

3. Mutual-TLS Client Certificate-Bound Access Tokens

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

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

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

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

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

3.1. JWT Certificate Thumbprint Confirmation Method

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

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

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


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

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

3.2. Confirmation Method for Token Introspection

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

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

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

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

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

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

3.3. Authorization Server Metadata

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

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

3.4. Client Registration Metadata

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

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

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

4. Public Clients and Certificate-Bound Tokens

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

5. Metadata for Mutual-TLS Endpoint Aliases

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

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

`mtls_endpoint_aliases`

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

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

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

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

6. Implementation Considerations

6.1. Authorization Server

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

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

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

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

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

6.2. Resource Server

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

6.3. Certificate Expiration and Bound Access Tokens

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

6.4. Implicit Grant Unsupported

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

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

6.5. TLS Termination

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

7. Security Considerations

7.1. Certificate-Bound Refresh Tokens

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

7.2. Certificate Thumbprint Binding

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

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

7.3. TLS Versions and Best Practices

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

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

7.4. X.509 Certificate Spoofing

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

7.5. X.509 Certificate Parsing and Validation Complexity

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

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

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

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

8. Privacy Considerations

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

9. IANA Considerations

9.1. JWT Confirmation Methods Registration

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

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

9.2. Authorization Server Metadata Registration

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

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

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

9.3. Token Endpoint Authentication Method Registration

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

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

9.4. Token Introspection Response Registration

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

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

- o Claim Name: "cnf"

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

9.5. Dynamic Client Registration Metadata Registration

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

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

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

10. References

10.1. Normative References

- [BCP195] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 7525, DOI 10.17487/RFC7525, May 2015, <<http://www.rfc-editor.org/info/bcp195>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC4514] Zeilenga, K., Ed., "Lightweight Directory Access Protocol (LDAP): String Representation of Distinguished Names", RFC 4514, DOI 10.17487/RFC4514, June 2006, <<https://www.rfc-editor.org/info/rfc4514>>.
- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <<https://www.rfc-editor.org/info/rfc4648>>.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <<https://www.rfc-editor.org/info/rfc5246>>.
- [RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <<https://www.rfc-editor.org/info/rfc5280>>.
- [RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <<https://www.rfc-editor.org/info/rfc6749>>.
- [RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, <<https://www.rfc-editor.org/info/rfc6750>>.

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

10.2. Informative References

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

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

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

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

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

Figure 5: x5t#S256 Confirmation Claim

```
-----BEGIN CERTIFICATE-----
MIIBBjCBRAIBAJAKBggqhkJOPQQDAjAPMQ0wCwYDVQQDDARTdGxzMB4XDTE4MTAx
ODEyMzcwOVoxDTIyMDUwMjEyMzcwOVowDzENMAsGA1UEAwEbXRsczBZMBMBGByqG
SM49AgEGCCqGSM49AwEHA0IABNcnxwqV6hY8QnhxxzFQ03C7HKW9OylMbnQZjjJ
/Au08/coZwxS7LFA4vOLS9WuneIXhbGGWvsDSb0tH6IxLm8wCgYIKoZIZj0EAWID
SQAwwRgIhAP0RC1E+vwJD/D1AGHGzuri+hlV/PpQEKTWUveORWz83AiEA5x2eXZOV
bUlJSGQgjd5vaUaK1LR50Q2DmFfQj1L+SY=
-----END CERTIFICATE-----
```

Figure 6: PEM Encoded Self-Signed Certificate

```
{
  "kty": "EC",
  "x": "1yfLHCpXqFjxCeHHMVDtCLscpb07KUxudBmOMn8C7Q",
  "y": "8_coZwxS7LFA4vOLS9WuneIXhbGGWvsDSb0tH6IxLm8",
  "crv": "P-256",
  "x5c": [
    "MIIBBjCBRAIBAJAKBggqhkJOPQQDAjAPMQ0wCwYDVQQDDARTdGxzMB4XDTE4MTAx
    xODEyMzcwOVoxDTIyMDUwMjEyMzcwOVowDzENMAsGA1UEAwEbXRsczBZMBMBGByqG
    qSM49AgEGCCqGSM49AwEHA0IABNcnxwqV6hY8QnhxxzFQ03C7HKW9OylMbnQZjjJ
    /Au08/coZwxS7LFA4vOLS9WuneIXhbGGWvsDSb0tH6IxLm8wCgYIKoZIZj0EAWID
    SQAwwRgIhAP0RC1E+vwJD/D1AGHGzuri+hlV/PpQEKTWUveORWz83AiEA5x2eXZOV
    bUlJSGQgjd5vaUaK1LR50Q2DmFfQj1L+SY="
  ]
}
```

Figure 7: JSON Web Key

Appendix B. Relationship to Token Binding

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

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

Appendix C. Acknowledgements

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

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

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

Appendix D. Document(s) History

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

draft-ietf-oauth-mtls-17

- o Updates from IESG ballot position comments.

draft-ietf-oauth-mtls-16

- o Editorial updates from last call review.

draft-ietf-oauth-mtls-15

- o Editorial updates from second AD review.

draft-ietf-oauth-mtls-14

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

draft-ietf-oauth-mtls-13

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

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

draft-ietf-oauth-mtls-12

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

draft-ietf-oauth-mtls-11

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

draft-ietf-oauth-mtls-10

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

draft-ietf-oauth-mtls-09

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

draft-ietf-oauth-mtls-08

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

draft-ietf-oauth-mtls-07

- o Update to use the boilerplate from RFC 8174

draft-ietf-oauth-mtls-06

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

draft-ietf-oauth-mtls-05

- o Editorial fixes

draft-ietf-oauth-mtls-04

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

draft-ietf-oauth-mtls-03

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

draft-ietf-oauth-mtls-02

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

draft-ietf-oauth-mtls-01

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

draft-ietf-oauth-mtls-00

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

draft-campbell-oauth-mtls-01

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

draft-campbell-oauth-mtls-00

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

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

- o Initial draft.

Authors' Addresses

Brian Campbell
Ping Identity

Email: brian.d.campbell@gmail.com

John Bradley
Yubico

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

Nat Sakimura
Nomura Research Institute

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

Torsten Lodderstedt
YES.com AG

Email: torsten@lodderstedt.net

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

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

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

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

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

Copyright Notice

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

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

Table of Contents

1. Introduction	2
2. Terminology	4
3. Processing Instructions	4
4. Examples	5
4.1. Symmetric Key Transport	5
4.1.1. Client-to-AS Request	5
4.1.2. Client-to-AS Response	6
4.2. Asymmetric Key Transport	9
4.2.1. Client-to-AS Request	9
4.2.2. Client-to-AS Response	10
5. Security Considerations	11
6. IANA Considerations	13
6.1. OAuth Access Token Types	13
6.2. OAuth Parameters Registration	13
6.3. OAuth Extensions Error Registration	13
7. Acknowledgements	13
8. References	14
8.1. Normative References	14
8.2. Informative References	15
Authors' Addresses	16

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.

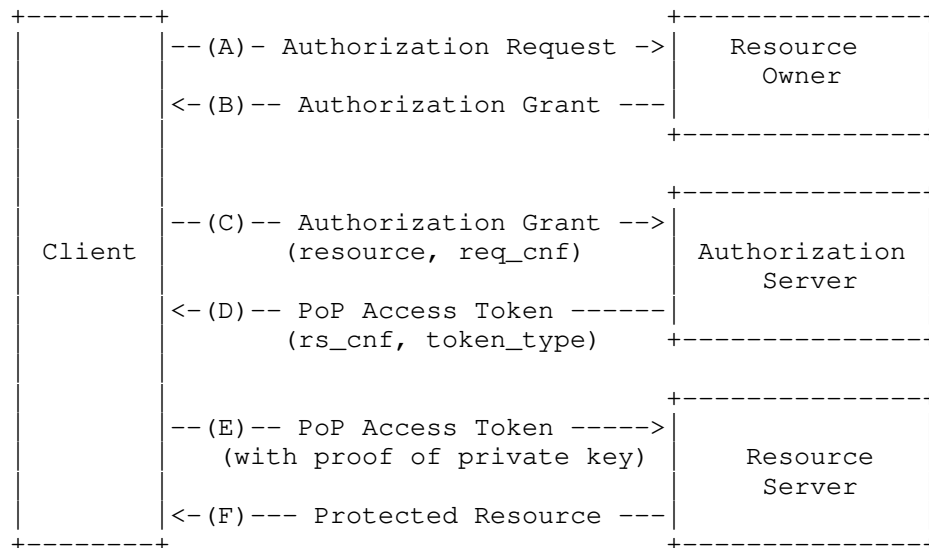


Figure 1: Augmented OAuth 2.0 Protocol Flow

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 identifier 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=Sp1xl0BeZQQYbYS6WxSbIA
&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":
      "eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkJExMjhdQkMtSFMyNTYifQ.
      (remainder of JWE omitted for brevity)"
    }
  }
}

```

Figure 3: Example: Encrypted Symmetric 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":
      "eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkJExMjhdQkMtSFMyNTYifQ.
      (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=SpIxl0BeZQQYbYS6WxSbIA
&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....jrlzCsicMWpAA",
  "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" : "usWxHK2PmfHnHKwXPS54m0kTcGJ90UiglWiGahtagnv8",
      "y" : "IBOL+C3BttVivg+1SreASjpkttcsz+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.independentid.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>

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

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

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

Abstract

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

Status of This Memo

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

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

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

This Internet-Draft will expire on 19 June 2022.

Copyright Notice

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

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

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

Table of Contents

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

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

1. Introduction

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

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

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

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

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

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

1.1. Structure

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

1.2. Conventions and Terminology

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

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

2. Recommendations

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

2.1. Protecting Redirect-Based Flows

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

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

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

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

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

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

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

2.1.1. Authorization Code Grant

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

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

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

Authorization servers MUST support PKCE [RFC7636].

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

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

2.1.2. Implicit Grant

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

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

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

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

2.2. Token Replay Prevention

2.2.1. Access Tokens

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

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

2.2.2. Refresh Tokens

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

2.3. Access Token Privilege Restriction

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

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

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

2.4. Resource Owner Password Credentials Grant

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

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

2.5. Client Authentication

Authorization servers SHOULD use client authentication if possible.

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

2.6. Other Recommendations

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

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

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

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

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

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

3. The Updated OAuth 2.0 Attacker Model

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4. Attacks and Mitigations

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

4.1. Insufficient Redirect URI Validation

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

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

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

These attacks are shown in detail in the following subsections.

4.1.1. Redirect URI Validation Attacks on Authorization Code Grant

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

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

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

The attack can then be conducted as follows:

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

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

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

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

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

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

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

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

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

4.1.2. Redirect URI Validation Attacks on Implicit Grant

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

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

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

The attack can now be conducted as follows:

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

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

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

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

HTTP/1.1 303 See Other

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

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

HTTP/1.1 303 See Other

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

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

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

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

4.1.3. Countermeasures

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

Additional recommendations:

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

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

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

4.2. Credential Leakage via Referer Headers

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

4.2.1. Leakage from the OAuth Client

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

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

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

4.2.2. Leakage from the Authorization Server

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

4.2.3. Consequences

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

4.2.4. Countermeasures

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

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

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

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

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

4.3. Credential Leakage via Browser History

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

4.3.1. Authorization Code in Browser History

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

Countermeasures:

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

4.3.2. Access Token in Browser History

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

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

Countermeasures:

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

4.4. Mix-Up Attacks

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

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

4.4.1. Attack Description

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

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

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

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

Attack on the authorization code grant:

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

Variants:

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

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

4.4.2. Countermeasures

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

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

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

4.4.2.1. Mix-Up Defense via Issuer Identification

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

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

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

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

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

4.4.2.2. Mix-Up Defense via Distinct Redirect URIs

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

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

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

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

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

4.5. Authorization Code Injection

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

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

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

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

4.5.1. Attack Description

The attack works as follows:

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

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

4.5.2. Discussion

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

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

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

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

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

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

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

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

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

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

4.5.3. Countermeasures

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

4.5.3.1. PKCE

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

4.5.3.2. Nonce

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

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

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

4.5.3.3. Other Solutions

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

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

4.5.4. Limitations

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

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

4.6. Access Token Injection

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

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

4.6.1. Countermeasures

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

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

4.7. Cross Site Request Forgery

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

4.7.1. Countermeasures

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

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

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

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

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

4.8. PKCE Downgrade Attack

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

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

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

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

4.8.1. Attack Description

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

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

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

4.8.2. Countermeasures

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

Therefore, AS MUST take precautions against this threat.

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

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

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

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

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

4.9. Access Token Leakage at the Resource Server

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

4.9.1. Access Token Phishing by Counterfeit Resource Server

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

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

4.9.1.1. Countermeasures

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

4.9.1.1.1. Metadata

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

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

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

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

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

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

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

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

4.9.1.1.2. Sender-Constrained Access Tokens

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

A typical flow looks like this:

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

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

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

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

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

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

4.9.1.1.3. Audience Restricted Access Tokens

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

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

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

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

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

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

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

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

4.9.2. Compromised Resource Server

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

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

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

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

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

- * Audience restriction as described in Section 4.9.1.1.3 SHOULD be used to prevent replay of captured access tokens on other resource servers.
- * The resource server MUST treat access tokens like any other credentials. It is considered good practice to not log them and not store them in plain text.

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

4.10. Open Redirection

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

4.10.1. Client as Open Redirector

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

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

4.10.2. Authorization Server as Open Redirector

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

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

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

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

4.11. 307 Redirect

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

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

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

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

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

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

4.12. TLS Terminating Reverse Proxies

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

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

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

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

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

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

4.13. Refresh Token Protection

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

4.13.1. Discussion

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

[RFC6749] already provides a robust baseline protection by requiring

- * confidentiality of the refresh tokens in transit and storage,
- * the transmission of refresh tokens over TLS-protected connections between authorization server and client,
- * the authorization server to maintain and check the binding of a refresh token to a certain client and authentication of this client during token refresh, if possible, and
- * that refresh tokens cannot be generated, modified, or guessed.

[RFC6749] also lays the foundation for further (implementation specific) security measures, such as refresh token expiration and revocation as well as refresh token rotation by defining respective error codes and response behavior.

This specification gives recommendations beyond the scope of [RFC6749] and clarifications.

4.13.2. Recommendations

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

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

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

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

- * *Sender-constrained refresh tokens:* the authorization server cryptographically binds the refresh token to a certain client instance by utilizing [RFC8705] or [I-D.ietf-oauth-token-binding].
- * *Refresh token rotation:* the authorization server issues a new refresh token with every access token refresh response. The previous refresh token is invalidated but information about the relationship is retained by the authorization server. If a refresh token is compromised and subsequently used by both the attacker and the legitimate client, one of them will present an invalidated refresh token, which will inform the authorization server of the breach. The authorization server cannot determine which party submitted the invalid refresh token, but it will revoke the active refresh token. This stops the attack at the cost of forcing the legitimate client to obtain a fresh authorization grant.

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

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

- * password change
- * logout at the authorization server

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

4.14. Client Impersonating Resource Owner

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

4.14.1. Countermeasures

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

4.15. Clickjacking

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

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

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

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

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

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

5. Acknowledgements

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

6. IANA Considerations

This draft includes no request to IANA.

7. Security Considerations

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

8. Normative References

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

- [oauth-v2-form-post-response-mode]
Jones, M. and B. Campbell, "OAuth 2.0 Form Post Response Mode", 27 April 2015, <http://openid.net/specs/oauth-v2-form-post-response-mode-1_0.html>.
- [RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <<https://www.rfc-editor.org/info/rfc7231>>.
- [OpenIDDisc]
Sakimura, N., Bradley, J., Jones, M., and E. Jay, "OpenID Connect Discovery 1.0 incorporating errata set 1", 8 November 2014, <https://openid.net/specs/openid-connect-discovery-1_0.html>.
- [RFC7662] Richer, J., Ed., "OAuth 2.0 Token Introspection", RFC 7662, DOI 10.17487/RFC7662, October 2015, <<https://www.rfc-editor.org/info/rfc7662>>.
- [RFC8252] Denniss, W. and J. Bradley, "OAuth 2.0 for Native Apps", BCP 212, RFC 8252, DOI 10.17487/RFC8252, October 2017, <<https://www.rfc-editor.org/info/rfc8252>>.
- [RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, <<https://www.rfc-editor.org/info/rfc6750>>.
- [RFC6819] Lodderstedt, T., Ed., McGloin, M., and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations", RFC 6819, DOI 10.17487/RFC6819, January 2013, <<https://www.rfc-editor.org/info/rfc6819>>.
- [RFC8705] Campbell, B., Bradley, J., Sakimura, N., and T. Lodderstedt, "OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens", February 2020, <<https://www.rfc-editor.org/info/rfc8705>>.
- [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <<https://www.rfc-editor.org/info/rfc3986>>.
- [OpenID] Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0 incorporating errata set 1", 8 November 2014, <https://openid.net/specs/openid-connect-core-1_0.html>.

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

9. Informative References

- [arXiv.1601.01229]
Fett, D., Küsters, R., and G. Schmitz, "A Comprehensive Formal Security Analysis of OAuth 2.0", 6 January 2016, <<http://arxiv.org/abs/1601.01229/>>.
- [I-D.ietf-oauth-dpop]
Fett, D., Campbell, B., Bradley, J., Lodderstedt, T., Jones, M., and D. Waite, "OAuth 2.0 Demonstrating Proof-of-Possession at the Application Layer (DPoP)", Work in Progress, Internet-Draft, draft-ietf-oauth-dpop-04, 4 October 2021, <<https://datatracker.ietf.org/doc/html/draft-ietf-oauth-dpop-04>>.
- [CSP-2] West, M., Barth, A., and D. Veditz, "Content Security Policy Level 2", July 2015, <<https://www.w3.org/TR/CSP2>>.
- [I-D.ietf-oauth-rar]
Lodderstedt, T., Richer, J., and B. Campbell, "OAuth 2.0 Rich Authorization Requests", Work in Progress, Internet-Draft, draft-ietf-oauth-rar-08, 18 October 2021, <<https://datatracker.ietf.org/doc/html/draft-ietf-oauth-rar-08>>.
- [I-D.ietf-oauth-signed-http-request]
Richer, J., Bradley, J., and H. Tschofenig, "A Method for Signing HTTP Requests for OAuth", Work in Progress, Internet-Draft, draft-ietf-oauth-signed-http-request-03, 8 August 2016, <<https://datatracker.ietf.org/doc/html/draft-ietf-oauth-signed-http-request-03>>.
- [bug.chromium]
"Referer header includes URL fragment when opening link using New Tab", <<https://bugs.chromium.org/p/chromium/issues/detail?id=168213>>.
- [webappsec-referrer-policy]
Eisinger, J. and E. Stark, "Referrer Policy", 20 April 2017, <<https://w3c.github.io/webappsec-referrer-policy>>.

- [I-D.bradley-oauth-jwt-encoded-state]
Bradley, J., Lodderstedt, D. T., and H. Zandbelt,
"Encoding claims in the OAuth 2 state parameter using a
JWT", Work in Progress, Internet-Draft, draft-bradley-
oauth-jwt-encoded-state-09, 4 November 2018,
<[https://datatracker.ietf.org/doc/html/draft-bradley-
oauth-jwt-encoded-state-09](https://datatracker.ietf.org/doc/html/draft-bradley-oauth-jwt-encoded-state-09)>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
Requirement Levels", BCP 14, RFC 2119,
DOI 10.17487/RFC2119, March 1997,
<<https://www.rfc-editor.org/info/rfc2119>>.
- [webcrypto]
Watson, M., "Web Cryptography API", 26 January 2017,
<<https://www.w3.org/TR/2017/REC-WebCryptoAPI-20170126/>>.
- [oauth_security_jcs_14]
Bansal, C., Bhargavan, K., Delignat-Lavaud, A., and S.
Maffeis, "Discovering concrete attacks on website
authorization by formal analysis", 23 April 2014,
<<https://www.doc.ic.ac.uk/~maffeis/papers/jcs14.pdf>>.
- [owasp_redir]
"OWASP Cheat Sheet Series - Unvalidated Redirects and
Forwards",
<[https://cheatsheetseries.owasp.org/cheatsheets/
Unvalidated_Redirects_and_Forwards_Cheat_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Unvalidated_Redirects_and_Forwards_Cheat_Sheet.html)>.
- [RFC7591] Richer, J., Ed., Jones, M., Bradley, J., Machulak, M., and
P. Hunt, "OAuth 2.0 Dynamic Client Registration Protocol",
RFC 7591, DOI 10.17487/RFC7591, July 2015,
<<https://www.rfc-editor.org/info/rfc7591>>.
- [arXiv.1901.11520]
Fett, D., Hosseini, P., and R. Küsters, "An Extensive
Formal Security Analysis of the OpenID Financial-grade
API", 31 January 2019, <<http://arxiv.org/abs/1901.11520/>>.
- [I-D.ietf-oauth-par]
Lodderstedt, T., Campbell, B., Sakimura, N., Tonge, D.,
and F. Skokan, "OAuth 2.0 Pushed Authorization Requests",
Work in Progress, Internet-Draft, draft-ietf-oauth-par-10,
29 July 2021, <[https://datatracker.ietf.org/doc/html/
draft-ietf-oauth-par-10](https://datatracker.ietf.org/doc/html/draft-ietf-oauth-par-10)>.

- [arXiv.1704.08539]
Fett, D., Küsters, R., and G. Schmitz, "The Web SSO Standard OpenID Connect: In-Depth Formal Security Analysis and Security Guidelines", 27 April 2017,
<<http://arxiv.org/abs/1704.08539/>>.
- [arXiv.1508.04324v2]
Mladenov, V., Mainka, C., and J. Schwenk, "On the security of modern Single Sign-On Protocols: Second-Order Vulnerabilities in OpenID Connect", 7 January 2016,
<<http://arxiv.org/abs/1508.04324v2/>>.
- [webauthn] Balfanz, D., Czeskis, A., Hodges, J., Jones, J.C., Jones, M.B., Kumar, A., Liao, A., Lindemann, R., and E. Lundberg, "Web Authentication: An API for accessing Public Key Credentials Level 1", 4 March 2019,
<<https://www.w3.org/TR/2019/REC-webauthn-1-20190304/>>.
- [RFC8473] Popov, A., Nystroem, M., Balfanz, D., Ed., Harper, N., and J. Hodges, "Token Binding over HTTP", RFC 8473, DOI 10.17487/RFC8473, October 2018,
<<https://www.rfc-editor.org/info/rfc8473>>.
- [I-D.ietf-oauth-pop-key-distribution]
Bradley, J., Hunt, P., Jones, M. B., Tschofenig, H., and M. Meszaros, "OAuth 2.0 Proof-of-Possession: Authorization Server to Client Key Distribution", Work in Progress, Internet-Draft, draft-ietf-oauth-pop-key-distribution-07, 27 March 2019, <<https://datatracker.ietf.org/doc/html/draft-ietf-oauth-pop-key-distribution-07>>.
- [I-D.ietf-oauth-jwsreq]
Sakimura, N., Bradley, J., and M. B. Jones, "The OAuth 2.0 Authorization Framework: JWT-Secured Authorization Request (JAR)", Work in Progress, Internet-Draft, draft-ietf-oauth-jwsreq-34, 8 April 2021,
<<https://datatracker.ietf.org/doc/html/draft-ietf-oauth-jwsreq-34>>.
- [oauth_security_ubic]
Sun, S.-T. and K. Beznosov, "The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems", October 2012,
<<http://passwordresearch.com/papers/paper267.html>>.
- [oauth_security_cmu]
Chen, E., Pei, Y., Chen, S., Tian, Y., Kotcher, R., and P. Tague, "OAuth Demystified for Mobile Application

Developers", November 2014,
<<http://css.csail.mit.edu/6.858/2012/readings/oauth-sso.pdf>>.

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

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

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

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

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

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

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

[I-D.ietf-oauth-iss-auth-resp]
Selhausen, K. M. Z. and D. Fett, "OAuth 2.0 Authorization Server Issuer Identification", Work in Progress, Internet-

Draft, draft-ietf-oauth-iss-auth-resp-04, 2 December 2021,
<<https://datatracker.ietf.org/doc/html/draft-ietf-oauth-iss-auth-resp-04>>.

[JARM] Lodderstedt, T. and B. Campbell, "Financial-grade API: JWT Secured Authorization Response Mode for OAuth 2.0 (JARM)", 17 October 2018,
<<https://openid.net/specs/openid-financial-api-jarm.html>>.

Appendix A. Document History

[[To be removed from the final specification]]

-19

- * Changed affiliation of Andrey Labunets
- * Editorial change to clarify the new recommendations for refresh tokens

-18

- * Fix editorial and spelling issues.
- * Change wording for disallowing HTTP redirect URIs.

-17

- * Make the use of metadata RECOMMENDED for both servers and clients
- * Make announcing PKCE support in metadata the RECOMMENDED way (before: either metadata or deployment-specific way)
- * AS also MUST NOT expose open redirectors.
- * Mention that attackers can collaborate.
- * Update recommendations regarding mix-up defense, building upon [I-D.ietf-oauth-iss-auth-resp].
- * Improve description of mix-up attack.
- * Make HTTPS mandatory for most redirect URIs.

-16

- * Make MTLS a suggestion, not RECOMMENDED.

- * Add important requirements when using nonce for code injection protection.
- * Highlight requirements for refresh token sender-constraining.
- * Make PKCE a MUST for public clients.
- * Describe PKCE Downgrade Attacks and countermeasures.
- * Allow variable port numbers in localhost redirect URIs as in RFC8252, Section 7.3.

-15

- * Update reference to DPoP
- * Fix reference to RFC8414
- * Move to xml2rfcv3

-14

- * Added info about using CSP to prevent clickjacking
- * Changes from WGLC feedback
- * Editorial changes
- * AS MUST announce PKCE support either in metadata or using deployment-specific ways (before: SHOULD)

-13

- * Discourage use of Resource Owner Password Credentials Grant
- * Added text on client impersonating resource owner
- * Recommend asymmetric methods for client authentication
- * Encourage use of PKCE mode "S256"
- * PKCE may replace state for CSRF protection
- * AS SHOULD publish PKCE support
- * Cleaned up discussion on auth code injection
- * AS MUST support PKCE

-12

- * Added updated attacker model

-11

- * Adapted section 2.1.2 to outcome of consensus call
- * more text on refresh token inactivity and implementation note on refresh token replay detection via refresh token rotation

-10

- * incorporated feedback by Joseph Heenan
- * changed occurrences of SHALL to MUST
- * added text on lack of token/cert binding support tokens issued in the authorization response as justification to not recommend issuing tokens there at all
- * added requirement to authenticate clients during code exchange (PKCE or client credential) to 2.1.1.
- * added section on refresh tokens
- * editorial enhancements to 2.1.2 based on feedback

-09

- * changed text to recommend not to use implicit but code
- * added section on access token injection
- * reworked sections 3.1 through 3.3 to be more specific on implicit grant issues

-08

- * added recommendations re implicit and token injection
- * uppercased key words in Section 2 according to RFC 2119

-07

- * incorporated findings of Doug McDorman
- * added section on HTTP status codes for redirects

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

-06

- * reworked section 3.8.1
- * incorporated Phil Hunt's feedback
- * reworked section on mix-up
- * extended section on code leakage via referrer header to also cover state leakage
- * added Daniel Fett as author
- * replaced text intended to inform WG discussion by recommendations to implementors
- * modified example URLs to conform to RFC 2606

-05

- * Completed sections on code leakage via referrer header, attacks in browser, mix-up, and CSRF
- * Reworked Code Injection Section
- * Added reference to OpenID Connect spec
- * removed refresh token leakage as respective considerations have been given in section 10.4 of RFC 6749
- * first version on open redirection
- * incorporated Christian Mainka's review feedback

-04

- * Restructured document for better readability
- * Added best practices on Token Leakage prevention

-03

- * Added section on Access Token Leakage at Resource Server
- * incorporated Brian Campbell's findings

-02

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

-01

- * Added references to mitigation methods for token leakage
- * Added reference to Token Binding for Authorization Code
- * incorporated feedback of Phil Hunt
- * fixed numbering issue in attack descriptions in section 2

-00 (WG document)

- * turned the ID into a WG document and a BCP
- * Added federated app login as topic in Other Topics

Authors' Addresses

Torsten Lodderstedt
yes.com

Email: torsten@lodderstedt.net

John Bradley
Yubico

Email: ve7jtb@ve7jtb.com

Andrey Labunets
Independent Researcher

Email: isciurus@gmail.com

Daniel Fett
yes.com

Email: mail@danielfett.de

Network Working Group
Internet-Draft
Intended status: Informational
Expires: August 17, 2019

E. Maler, Ed.
ForgeRock
M. Machulak
HSBC
J. Richer
Bespoke Engineering
T. Hardjono
MIT
February 13, 2019

Federated Authorization for User-Managed Access (UMA) 2.0
draft-maler-oauth-umafedauthz-00

Abstract

This specification defines a means for an UMA-enabled authorization server and resource server to be loosely coupled, or federated, in a secure and authorized resource owner context.

Status of This Memo

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

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

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

This Internet-Draft will expire on August 17, 2019.

Copyright Notice

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

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

include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
1.1. Notational Conventions	3
1.2. Abstract Flow	4
1.3. HTTP Usage, API Security, and Identity Context	5
1.4. Separation of Responsibility and Authority	6
1.5. Protection API Summary	7
1.5.1. Permissions	8
2. Authorization Server Metadata	8
3. Resource Registration Endpoint	9
3.1. Resource Description	11
3.1.1. Scope Description	12
3.2. Resource Registration API	13
3.2.1. Create Resource Description	14
3.2.2. Read Resource Description	15
3.2.3. Update Resource Description	16
3.2.4. Delete Resource Description	17
3.2.5. List Resource Descriptions	17
4. Permission Endpoint	18
4.1. Resource Server Request to Permission Endpoint	20
4.2. Authorization Server Response to Resource Server on Permission Request Success	22
4.3. Authorization Server Response to Resource Server on Permission Request Failure	23
5. Token Introspection Endpoint	23
5.1. Resource Server Request to Token Introspection Endpoint	24
5.1.1. Authorization Server Response to Resource Server on Token Introspection Success	25
6. Error Messages	26
7. Security Considerations	27
8. Privacy Considerations	27
9. IANA Considerations	28
9.1. OAuth 2.0 Authorization Server Metadata Registry	28
9.1.1. Registry Contents	28
9.2. OAuth Token Introspection Response Registration	28
9.2.1. Registry Contents	28
10. Acknowledgments	29
11. References	29
11.1. Normative References	29
11.2. Informative References	31
Authors' Addresses	31

1. Introduction

This specification extends and complements [UMAGrant] to loosely couple, or federate, its authorization process. This enables multiple resource servers operating in different domains to communicate with a single authorization server operating in yet another domain that acts on behalf of a resource owner. A service ecosystem can thus automate resource protection, and the resource owner can monitor and control authorization grant rules through the authorization server over time. Further, authorization grants can increase and decrease at the level of individual resources and scopes.

Building on the example provided in the introduction in [UMAGrant], bank customer (resource owner) Alice has a bank account service (resource server), a cloud file system (different resource server hosted elsewhere), and a dedicated sharing management service (authorization server) hosted by the bank. She can manage access to her various protected resources by spouse Bob, accounting professional Charline, financial information aggregation company DecideAccount, and neighbor Erik (requesting parties), all using different client applications. Her bank accounts and her various files and folders are protected resources, and she can use the same sharing management service to monitor and control different scopes of access to them by these different parties, such as viewing, editing, or printing files and viewing account data or accessing payment functions.

This specification, together with [UMAGrant], constitutes UMA 2.0. This specification is OPTIONAL to use with the UMA grant.

1.1. Notational Conventions

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

Unless otherwise noted, all parameter names and values are case sensitive. JSON [RFC7159] data structures defined in this specification MAY contain extension parameters that are not defined in this specification. Any entity receiving or retrieving a JSON data structure SHOULD ignore extension parameters it is unable to understand. Extension names that are unprotected from collisions are outside the scope of this specification.

1.2. Abstract Flow

The UMA grant defined in [UMAGrant] enhances the abstract protocol flow of OAuth. This specification enhances the UMA grant by defining formal communications between the UMA-enabled authorization server and resource server as they act on behalf of the resource owner, responding to authorization and resource requests, respectively, by a client that is acting on behalf of a requesting party.

A summary of UMA 2.0 communications, combining the UMA grant with federated authorization, is shown in Figure 1.

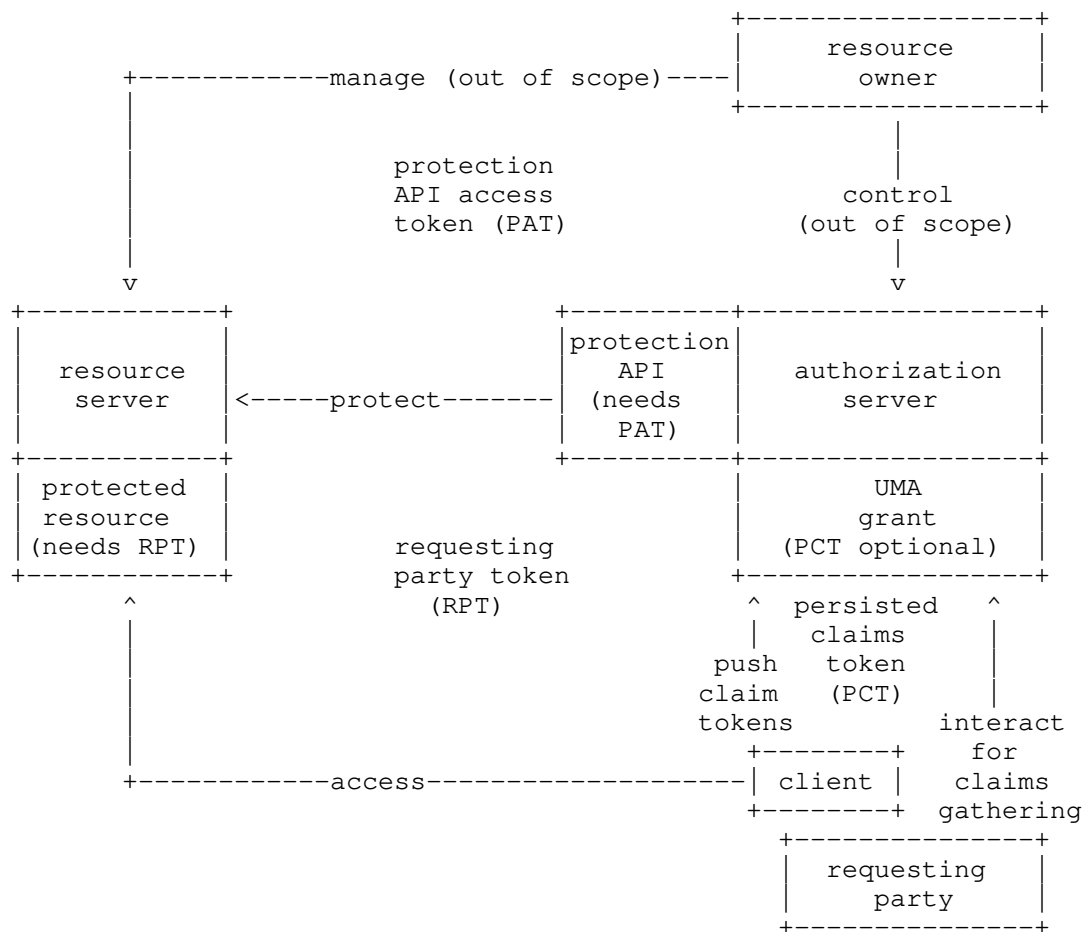


Figure 1: Federated Authorization Enhancements to UMA Grant Flow

This specification uses all of the terms and concepts in [UMAGrant]. This figure introduces the following new concepts:

protection API The API presented by the authorization server to the resource server, defined in this specification. This API is OAuth-protected.

protection API access token (PAT) An [RFC6749] access token with the scope "uma_protection", used by the resource server as a client of the authorization server's protection API. The resource owner involved in the UMA grant is the same entity taking on the role of the resource owner authorizing issuance of the PAT.

1.3. HTTP Usage, API Security, and Identity Context

This specification is designed for use with HTTP [RFC2616], and for interoperability and security in the context of loosely coupled services and applications operated by independent parties in independent domains. The use of UMA over any protocol other than HTTP is undefined. In such circumstances, it is RECOMMENDED to define profiles or extensions to achieve interoperability among independent implementations (see Section 4 of [UMAGrant]).

The authorization server MUST use TLS protection over its protection API endpoints, as governed by [BCP195], which discusses deployment and adoption characteristics of different TLS versions.

The authorization server MUST use OAuth and require a valid PAT to secure its protection API endpoints. The authorization server and the resource server (as an OAuth client) MUST support bearer usage of the PAT, as defined in [RFC6750]. All examples in this specification show the use of bearer-style PATs in this format.

As defined in [UMAGrant], the resource owner -- the entity here authorizing PAT issuance -- MAY be an end-user (natural person) or a non-human entity treated as a person for limited legal purposes (legal person), such as a corporation. A PAT is unique to a resource owner, resource server used for resource management, and authorization server used for protection of those resources. The issuance of the PAT represents the authorization of the resource owner for the resource server to use the authorization server for protecting those resources.

Different grant types for PAT issuance might be appropriate for different types of resource owners; for example, the client credentials grant is useful in the case of an organization acting as a resource owner, whereas an interactive grant type is typically more appropriate for capturing the approval of an end-user resource owner.

Where an identity token is desired in addition to an access token, it is RECOMMENDED to use [OIDCCore] in addition.

1.4. Separation of Responsibility and Authority

Federation of authorization for the UMA grant delivers a conceptual separation of responsibility and authority:

- o The resource owner can control access to resources residing at multiple resource servers from a single authorization server, by virtue of authorizing PAT issuance for each resource server. Any one resource server MAY be operated by a party different from the one operating the authorization server.
- o The resource server defines the boundaries of resources and the scopes available to each resource, and interprets how clients' resource requests map to permission requests, by virtue of being the publisher of the API being protected and using the protection API to communicate to the authorization server.
- o The resource owner works with the authorization server to configure policy conditions (authorization grant rules), which the authorization server executes in the process of issuing access tokens. The authorization process makes use of claims gathered from the requesting party and client in order to satisfy all operative operative policy conditions.

The separation of authorization decision making and authorization enforcement is similar to the architectural separation often used in enterprises between policy decision points and policy enforcement points. However, the resource server MAY apply additional authorization controls beyond those imposed by the authorization server. For example, even if an RPT provides sufficient permissions for a particular case, the resource server can choose to bar access based on its own criteria.

Practical control of access among loosely coupled parties typically requires more than just messaging protocols. It is outside the scope of this specification to define more than the technical contract between UMA-conforming entities. Laws may govern authorization-granting relationships. It is RECOMMENDED for the resource owner, authorization server, and resource server to establish agreements about which parties are responsible for establishing and maintaining authorization grant rules and other authorization rules on a legal or contractual level, and parties operating entities claiming to be UMA-conforming should provide documentation of rights and obligations between and among them. See Section 4 of [UMAGrant] for more information.

Except for PAT issuance, the resource owner-resource server and resource owner-authorization server interfaces -- including the setting of policy conditions -- are outside the scope of this specification (see Section 8 and Section 6.1 of [UMAGrant] for privacy considerations). Some elements of the protection API enable the building of user interfaces for policy condition setting (for example, see Section 3.2, which can be used in concert with user interaction for resource protection and sharing and offers an end-user redirection mechanism for policy interactions).

Note: The resource server typically requires access to at least the permission and token introspection endpoints when an end-user resource owner is not available ("offline" access). Thus, the authorization server needs to manage the PAT in a way that ensures this outcome. [UMA-Impl] discusses ways the resource server can enhance its error handling when the PAT is invalid.

1.5. Protection API Summary

The protection API defines the following endpoints:

- o Resource registration endpoint as defined in Section 3. The API available at this endpoint provides a means for the resource server to put resources under the protection of an authorization server on behalf of the resource owner and manage them over time.
- o Permission endpoint as defined in Section 4. This endpoint provides a means for the resource server to request a set of one or more permissions on behalf of the client based on the client's resource request when that request is unaccompanied by an access token or is accompanied by an RPT that is insufficient for access to that resource.
- o OPTIONAL token introspection endpoint as defined in [RFC7662] and as extended in Section 5. This endpoint provides a means for the resource server to introspect the RPT.

Use of these endpoints assumes that the resource server has acquired OAuth client credentials from the authorization server by static or dynamic means, and has a valid PAT. Note: Although the resource identifiers that appear in permission and token introspection request messages could sufficiently identify the resource owner, the PAT is still required because it represents the resource owner's authorization to use the protection API, as noted in Section 1.3.

The authorization server MUST declare its protection API endpoints in the discovery document (see Section 2).

1.5.1. Permissions

A permission is (requested or granted) authorized access to a particular resource with some number of scopes bound to that resource. The concept of permissions is used in authorization assessment, results calculation, and RPT issuance in [UMAGrant]. This concept takes on greater significance in relation to the protection API.

The resource server's resource registration operations at the authorization server result in a set of resource owner-specific resource identifiers. When the client makes a resource request that is unaccompanied by an access token or its resource request fails, the resource server is responsible for interpreting that request and mapping it to a choice of authorization server, resource owner, resource identifier(s), and set of scopes for each identifier, in order to request one or more permissions -- resource identifiers and a set of scopes -- and obtain a permission ticket on the client's behalf. Finally, when the client has made a resource request accompanied by an RPT and token introspection is in use, the returned token introspection object reveals the structure of permissions, potentially including expiration of individual permissions.

2. Authorization Server Metadata

This specification makes use of the authorization server discovery document structure and endpoint defined in [UMAGrant]. The resource server uses this discovery document to discover the endpoints it needs.

In addition to the metadata defined in that specification and [OAuthMeta], this specification defines the following metadata for inclusion in the discovery document:

permission_endpoint

REQUIRED. The endpoint URI at which the resource server requests permissions on the client's behalf.

resource_registration_endpoint

REQUIRED. The endpoint URI at which the resource server registers resources to put them under authorization manager protection.

Following are additional requirements related to metadata:

introspection_endpoint

If the authorization server supports token introspection as defined in this specification, it MUST supply this metadata value (defined in [OAuthMeta]).

The authorization server SHOULD document any profiled or extended features it supports explicitly, ideally by supplying the URI identifying each UMA profile and extension as an "uma_profiles_supported" metadata array value (defined in [UMAGrant]), and by using extension metadata to indicate specific usage details as necessary.

3. Resource Registration Endpoint

The API available at the resource registration endpoint enables the resource server to put resources under the protection of an authorization server on behalf of the resource owner and manage them over time. Protection of a resource at the authorization server begins on successful registration and ends on successful deregistration.

The resource server uses a RESTful API at the authorization server's resource registration endpoint to create, read, update, and delete resource descriptions, along with retrieving lists of such descriptions. The descriptions consist of JSON documents that are maintained as web resources at the authorization server. (Note carefully the similar but distinct senses in which the word "resource" is used in this section.)

Figure 2 illustrates the resource registration API operations, with requests and success responses shown.

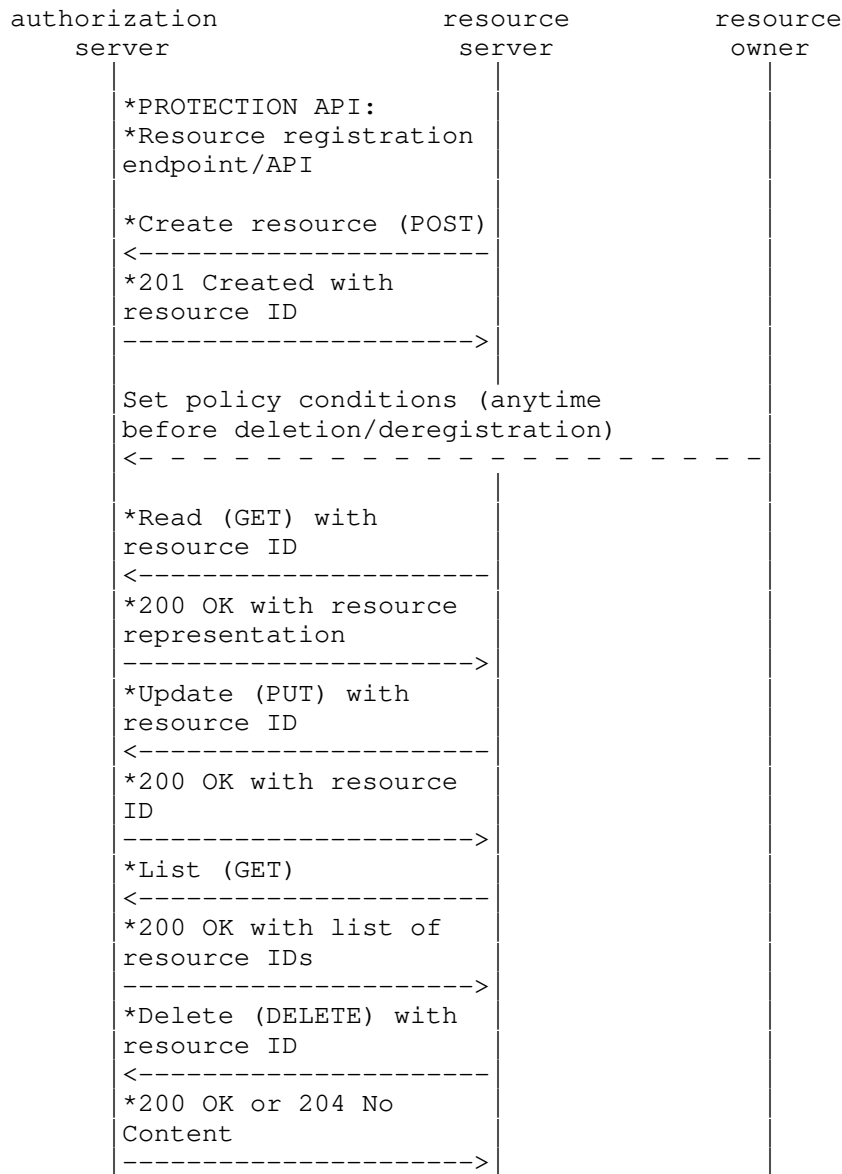


Figure 2: Resource Registration Endpoint and API: Requests and Success Responses

The resource server MAY protect any subset of the resource owner's resources using different authorization servers or other means entirely, or to protect some resources and not others. Additionally, the choice of protection regimes MAY be made explicitly by the

resource owner or implicitly by the resource server. Any such partitioning by the resource server or owner is outside the scope of this specification.

The resource server MAY register a single resource for protection that, from its perspective, has multiple parts, or has dynamic elements such as the capacity for querying or filtering, or otherwise has internal complexity. The resource server alone is responsible for maintaining any required mappings between internal representations and the resource identifiers and scopes known to the authorization server.

Note: The resource server is responsible for managing the process and timing of registering resources, maintaining the registration of resources, and deregistering resources at the authorization server. Motivations for updating a resource might include, for example, new scopes added to a new API version or resource owner actions at a resource server that result in new resource description text. See [UMA-Impl] for a discussion of initial resource registration timing options.

3.1. Resource Description

A resource description is a JSON document that describes the characteristics of a resource sufficiently for an authorization server to protect it. A resource description has the following parameters:

resource_scopes REQUIRED. An array of strings, serving as scope identifiers, indicating the available scopes for this resource. Any of the strings MAY be either a plain string or a URI.

description OPTIONAL. A human-readable string describing the resource at length. The authorization server MAY use this description in any user interface it presents to a resource owner, for example, for resource protection monitoring or policy setting. The value of this parameter MAY be internationalized, as described in Section 2.2 of [RFC7591].

icon_uri OPTIONAL. A URI for a graphic icon representing the resource. The authorization server MAY use the referenced icon in any user interface it presents to a resource owner, for example, for resource protection monitoring or policy setting.

name OPTIONAL. A human-readable string naming the resource. The authorization server MAY use this name in any user interface it presents to a resource owner, for example, for resource protection

monitoring or policy setting. The value of this parameter MAY be internationalized, as described in Section 2.2 of [RFC7591].

type OPTIONAL. A string identifying the semantics of the resource. For example, if the resource is an identity claim that leverages standardized claim semantics for "verified email address", the value of this parameter could be an identifying URI for this claim. The authorization server MAY use this information in processing information about the resource or displaying information about it in any user interface it presents to a resource owner.

For example, this description characterizes a resource (a photo album) that can potentially be viewed or printed; the scope URI points to a scope description as defined in Section 3.1.1:

```
{
  "resource_scopes":[
    "view",
    "http://photoz.example.com/dev/scopes/print"
  ],
  "description":"Collection of digital photographs",
  "icon_uri":"http://www.example.com/icons/flower.png",
  "name":"Photo Album",
  "type":"http://www.example.com/rsracs/photoalbum"
}
```

3.1.1.1. Scope Description

A scope description is a JSON document that describes the characteristics of a scope sufficiently for an authorization server to protect the resource with this available scope.

While a scope URI appearing in a resource description (see Section 3.1) MAY resolve to a scope description document, and thus scope description documents are possible to standardize and reference publicly, the authorization server is not expected to resolve scope description details at resource registration time or at any other run-time requirement. The resource server and authorization server are presumed to have negotiated any required interpretation of scope handling out of band.

A scope description has the following parameters:

description OPTIONAL. A human-readable string describing the resource at length. The authorization server MAY use this description in any user interface it presents to a resource owner, for example, for resource protection monitoring or policy setting.

The value of this parameter MAY be internationalized, as described in Section 2.2 of [RFC7591].

`icon_uri` OPTIONAL. A URI for a graphic icon representing the scope. The authorization server MAY use the referenced icon in any user interface it presents to a resource owner, for example, for resource protection monitoring or policy setting.

`name` OPTIONAL. A human-readable string naming the scope. The authorization server MAY use this name in any user interface it presents to a resource owner, for example, for resource protection monitoring or policy setting. The value of this parameter MAY be internationalized, as described in Section 2.2 of [RFC7591].

For example, this scope description characterizes a scope that involves printing (as opposed to, say, creating or editing in some fashion):

```
{
  "description": "Print out and produce PDF files of photos",
  "icon_uri": "http://www.example.com/icons/printer",
  "name": "Print"
}
```

3.2. Resource Registration API

The authorization server MUST support the following five registration options and MUST require a valid PAT for access to them; any other operations are undefined by this specification. Here, `_rreguri_` stands for the resource registration endpoint and `__id_` stands for the authorization server-assigned identifier for the web resource corresponding to the resource at the time it was created, included within the URL returned in the Location header. Each operation is defined in its own section below.

- o Create resource description: POST `_rreguri_`/
- o Read resource description: GET `_rreguri_`/`__id_`
- o Update resource description: PUT `_rreguri_`/`__id_`
- o Delete resource description: DELETE `_rreguri_`/`__id_`
- o List resource descriptions: GET `_rreguri_`/

Within the JSON body of a successful response, the authorization server includes common parameters, possibly in addition to method-specific parameters, as follows:

`_id` REQUIRED (except for the Delete and List methods). A string value repeating the authorization server-defined identifier for the web resource corresponding to the resource. Its appearance in the body makes it readily available as an identifier for various protected resource management tasks.

`user_access_policy_uri` OPTIONAL. A URI that allows the resource server to redirect an end-user resource owner to a specific user interface within the authorization server where the resource owner can immediately set or modify access policies subsequent to the resource registration action just completed. The authorization server is free to choose the targeted user interface, for example, in the case of a deletion action, enabling the resource server to direct the end-user to a policy-setting interface for an overall "folder" resource formerly "containing" the deleted resource (a relationship the authorization server is not aware of), to enable adjustment of related policies.

If the request to the resource registration endpoint is incorrect, then the authorization server instead responds as follows (see Section 6 for information about error messages):

- o If the referenced resource cannot be found, the authorization server MUST respond with an HTTP 404 (Not Found) status code and MAY respond with a "not_found" error code.
- o If the resource server request used an unsupported HTTP method, the authorization server MUST respond with the HTTP 405 (Method Not Allowed) status code and MAY respond with an "unsupported_method_type" error code.
- o If the request is missing a required parameter, includes an invalid parameter value, includes a parameter more than once, or is otherwise malformed, the authorization server MUST respond with the HTTP 400 (Bad Request) status code and MAY respond with an "invalid_request" error code.

3.2.1. Create Resource Description

Adds a new resource description to the authorization server using the POST method. If the request is successful, the resource is thereby registered and the authorization server MUST respond with an HTTP 201 status message that includes a "Location" header and an "_id" parameter.

Form of a create request, with a PAT in the header:

```
POST /rreg/ HTTP/1.1 Content-Type: application/json
Authorization: Bearer MHg3OUZEQkZBMjcx
...
{
  "resource_scopes":[
    "read-public",
    "post-updates",
    "read-private",
    "http://www.example.com/scopes/all"
  ],
  "icon_uri":"http://www.example.com/icons/sharesocial.png",
  "name":"Tweedl Social Service",
  "type":"http://www.example.com/rsrscs/socialstream/140-compatible"
}
```

Form of a successful response, also containing an optional "user_access_policy_uri" parameter:

```
HTTP/1.1 201 Created
Content-Type: application/json
Location: /rreg/KX3A-39WE
...
{
  "_id":"KX3A-39WE",
  "user_access_policy_uri":"http://as.example.com/rs/222/resource/KX3A-39WE/policy"
}
```

3.2.2. Read Resource Description

Reads a previously registered resource description using the GET method. If the request is successful, the authorization server MUST respond with an HTTP 200 status message that includes a body containing the referenced resource description, along with an "_id" parameter.

Form of a read request, with a PAT in the header:

```
GET /rreg/KX3A-39WE HTTP/1.1
Authorization: Bearer MHg3OUZEQkZBMjcx
...
```

Form of a successful response, containing all the parameters that were registered as part of the description:

```
HTTP/1.1 200 OK
Content-Type: application/json
...
{
  "_id": "KX3A-39WE",
  "resource_scopes": [
    "read-public",
    "post-updates",
    "read-private",
    "http://www.example.com/scopes/all"
  ],
  "icon_uri": "http://www.example.com/icons/sharesocial.png",
  "name": "Tweedl Social Service",
  "type": "http://www.example.com/rsrscs/socialstream/140-compatible"
}
```

3.2.3. Update Resource Description

Updates a previously registered resource description, by means of a complete replacement of the previous resource description, using the PUT method. If the request is successful, the authorization server MUST respond with an HTTP 200 status message that includes an "_id" parameter.

Form of an update request adding a "description" parameter to a resource description that previously had none, with a PAT in the header:

```
PUT /rreg/9UQU-DUWW HTTP/1.1
Content-Type: application/json
Authorization: Bearer 204c69636b6c69
...
{
  "resource_scopes": [
    "http://photoz.example.com/dev/scopes/view",
    "public-read"
  ],
  "description": "Collection of digital photographs",
  "icon_uri": "http://www.example.com/icons/sky.png",
  "name": "Photo Album",
  "type": "http://www.example.com/rsrscs/photoalbum"
}
```

Form of a successful response, not containing the optional "user_access_policy_uri" parameter:

```
HTTP/1.1 200 OK
...
{
  "_id": "9UQU-DUWW"
}
```

3.2.4. Delete Resource Description

Deletes a previously registered resource description using the DELETE method. If the request is successful, the resource is thereby deregistered and the authorization server MUST respond with an HTTP 200 or 204 status message.

Form of a delete request, with a PAT in the header:

```
DELETE /rreg/9UQU-DUWW
Authorization: Bearer 204c69636b6c69
...
```

Form of a successful response:

```
HTTP/1.1 204 No content
...
```

3.2.5. List Resource Descriptions

Lists all previously registered resource identifiers for this resource owner using the GET method. The authorization server MUST return the list in the form of a JSON array of "_id" string values.

The resource server can use this method as a first step in checking whether its understanding of protected resources is in full synchronization with the authorization server's understanding.

Form of a list request, with a PAT in the header:

```
GET /rreg/ HTTP/1.1
Authorization: Bearer 204c69636b6c69
...
```

Form of a successful response:

```
HTTP/1.1 200 OK
...
[
  "KX3A-39WE",
  "9UQU-DUWW"
]
```

4. Permission Endpoint

The permission endpoint defines a means for the resource server to request one or more permissions (resource identifiers and corresponding scopes) with the authorization server on the client's behalf, and to receive a permission ticket in return, in order to respond as indicated in Section 3.2 of [UMAGrant]. The resource server uses this endpoint on the following occasions:

- o After the client's initial resource request without an access token
- o After the client's resource request that was accompanied by an invalid RPT or a valid RPT that had insufficient permissions associated with it

The use of the permission endpoint is illustrated in Figure 3, with a request and a success response shown.

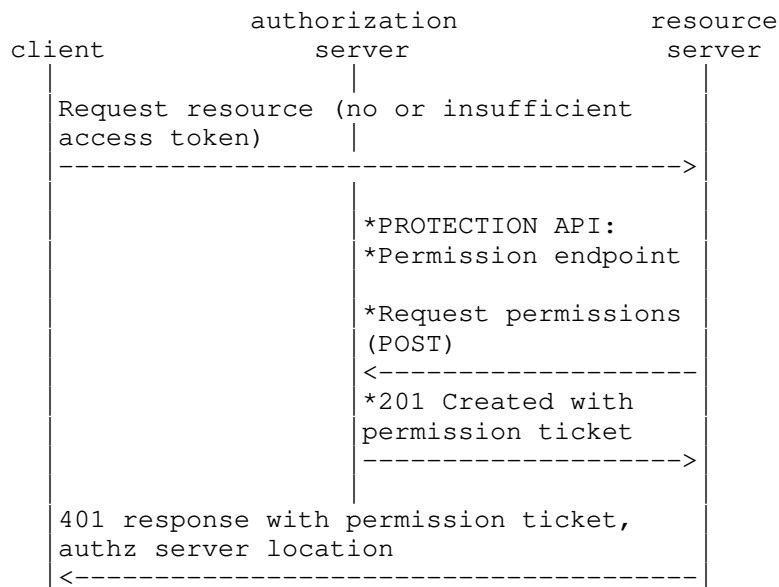


Figure 3: Permission Endpoint: Request and Success Response

The PAT provided in the API request enables the authorization server to map the resource server's request to the appropriate resource owner. It is only possible to request permissions for access to the resources of a single resource owner, protected by a single authorization server, at a time.

In its response, the authorization server returns a permission ticket for the resource server to give to the client that represents the same permissions that the resource server requested.

The process of choosing what permissions to request from the authorization server may require interpretation and mapping of the client's resource request. The resource server SHOULD request a set of permissions with scopes that is reasonable for the client's resource request. The resource server MAY request multiple permissions, and any permission MAY have zero scopes associated with it. Requesting multiple permissions might be appropriate, for example, in cases where the resource server expects the requesting party to need access to several related resources if they need access to any one of the resources (see Section 3.3.4 of [UMAGrant] for an example). Requesting a permission with no scopes might be appropriate, for example, in cases where an access attempt involves an API call that is ambiguous without further context (role-based scopes such as "user" and "admin" could have this ambiguous quality, and an explicit client request for a particular scope at the token

endpoint later can clarify the desired access). The resource server SHOULD document its intended pattern of permission requests in order to assist the client in pre-registering for and requesting appropriate scopes at the authorization server. See [UMA-Impl] for a discussion of permission request patterns.

Note: In order for the resource server to know which authorization server to approach for the permission ticket and on which resource owner's behalf (enabling a choice of permission endpoint and PAT), it needs to derive the necessary information using cues provided by the structure of the API where the resource request was made, rather than by an access token. Commonly, this information can be passed through the URI, headers, or body of the client's request. Alternatively, the entire interface could be dedicated to the use of a single resource owner and protected by a single authorization server.

4.1. Resource Server Request to Permission Endpoint

The resource server uses the POST method at the permission endpoint. The body of the HTTP request message contains a JSON object for requesting a permission for single resource identifier, or an array of one or more objects for requesting permissions for a corresponding number of resource identifiers. The object format in both cases is derived from the resource description format specified in Section 3.1; it has the following parameters:

`resource_id` REQUIRED. The identifier for a resource to which the resource server is requesting a permission on behalf of the client. The identifier MUST correspond to a resource that was previously registered.

`resource_scopes` REQUIRED. An array referencing zero or more identifiers of scopes to which the resource server is requesting access for this resource on behalf of the client. Each scope identifier MUST correspond to a scope that was previously registered by this resource server for the referenced resource.

Example of an HTTP request for a single permission at the authorization server's permission endpoint, with a PAT in the header:

```
POST /perm HTTP/1.1
Content-Type: application/json
Host: as.example.com
Authorization: Bearer 204c69636b6c69
...

{
  "resource_id": "112210f47de98100",
  "resource_scopes": [
    "view",
    "http://photoz.example.com/dev/actions/print"
  ]
}
```


Example of an HTTP request for multiple permissions at the authorization server's permission endpoint, with a PAT in the header:

```
POST /perm HTTP/1.1
Content-Type: application/json
Host: as.example.com
Authorization: Bearer 204c69636b6c69
...

[
  {
    "resource_id": "7b727369647d",
    "resource_scopes": [
      "view",
      "crop",
      "lightbox"
    ]
  },
  {
    "resource_id": "7b72736964327d",
    "resource_scopes": [
      "view",
      "layout",
      "print"
    ]
  },
  {
    "resource_id": "7b72736964337d",
    "resource_scopes": [
      "http://www.example.com/scopes/all"
    ]
  }
]
```

4.2. Authorization Server Response to Resource Server on Permission Request Success

If the authorization server is successful in creating a permission ticket in response to the resource server's request, it responds with an HTTP 201 (Created) status code and includes the "ticket" parameter in the JSON-formatted body. Regardless of whether the request contained one or multiple permissions, only a single permission ticket is returned.

For example:

```
HTTP/1.1 201 Created
Content-Type: application/json
...

{
  "ticket": "016f84e8-f9b9-11e0-bd6f-0021cc6004de"
}
```

4.3. Authorization Server Response to Resource Server on Permission Request Failure

If the resource server's permission registration request is authenticated properly but fails due to other reasons, the authorization server responds with an HTTP 400 (Bad Request) status code and includes one of the following error codes (see Section 6 for more information about error codes and responses):

`invalid_resource_id` At least one of the provided resource identifiers was not found at the authorization server.

`invalid_scope` At least one of the scopes included in the request was not registered previously by this resource server for the referenced resource.

5. Token Introspection Endpoint

When the client makes a resource request accompanied by an RPT, the resource server needs to determine whether the RPT is active and, if so, its associated permissions. Depending on the nature of the RPT and operative caching parameters, the resource server MAY take any of the following actions as appropriate to determine the RPT's status:

- o Introspect the RPT at the authorization server using the OAuth token introspection endpoint (defined in [RFC7662] and this section) that is part of the protection API. The authorization server's response contains an extended version of the introspection response. If the authorization server supports this specification's version of the token introspection endpoint, it MUST declare the endpoint in its discovery document (see Section 2) and support this extended version of the response.
- o Use a cached copy of the token introspection response if allowed (see Section 4 of [RFC7662]).
- o Validate the RPT locally if it is self-contained.

The use of the token introspection endpoint is illustrated in Figure 4, with a request and a success response shown.

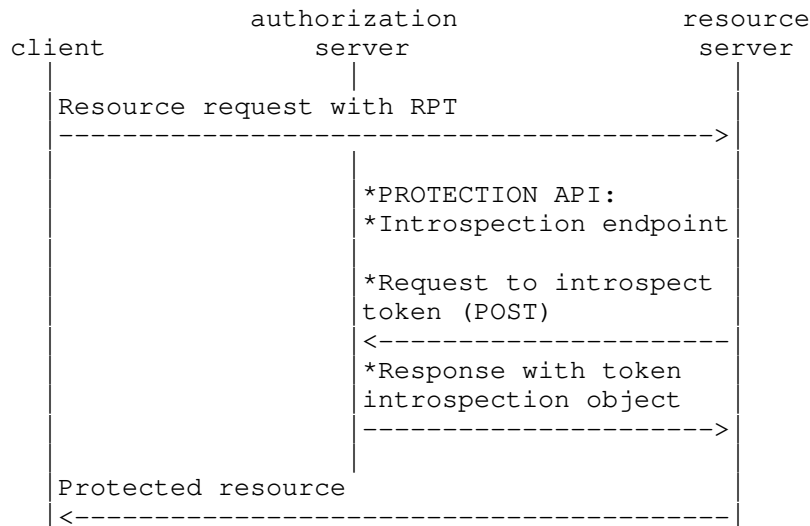


Figure 4: Token Introspection Endpoint: Request and Success Response

The authorization server MAY support both UMA-extended and non-UMA introspection requests and responses.

5.1. Resource Server Request to Token Introspection Endpoint

Note: In order for the resource server to know which authorization server, PAT (representing a resource owner), and endpoint to use in making the token introspection API call, it may need to interpret the client's resource request.

Example of the resource server's request to the authorization server for introspection of an RPT, with a PAT in the header:

```

POST /introspect HTTP/1.1
Host: as.example.com
Authorization: Bearer 204c69636b6c69
...
token=sbjsbhs(/SSJHBSUSSJHVhjsghsgvshgsv
  
```

Because an RPT is an access token, if the resource server chooses to supply a token type hint, it would use a "token_type_hint" of "access_token".

5.1.1.1. Authorization Server Response to Resource Server on Token Introspection Success

The authorization server's response to the resource server MUST use [RFC7662], responding with a JSON object with the structure dictated by that specification, extended as follows.

If the introspection object's "active" parameter has a Boolean value of "true", then the object MUST NOT contain a "scope" parameter, and MUST contain an extension parameter named "permissions" that contains an array of objects, each one (representing a single permission) containing these parameters:

resource_id REQUIRED. A string that uniquely identifies the protected resource, access to which has been granted to this client on behalf of this requesting party. The identifier MUST correspond to a resource that was previously registered as protected.

resource_scopes REQUIRED. An array referencing zero or more strings representing scopes to which access was granted for this resource. Each string MUST correspond to a scope that was registered by this resource server for the referenced resource.

exp OPTIONAL. Integer timestamp, measured in the number of seconds since January 1 1970 UTC, indicating when this permission will expire. If the token-level "exp" value pre-dates a permission-level "exp" value, the token-level value takes precedence.

iat OPTIONAL. Integer timestamp, measured in the number of seconds since January 1 1970 UTC, indicating when this permission was originally issued. If the token-level "iat" value post-dates a permission-level "iat" value, the token-level value takes precedence.

nbf OPTIONAL. Integer timestamp, measured in the number of seconds since January 1 1970 UTC, indicating the time before which this permission is not valid. If the token-level "nbf" value post-dates a permission-level "nbf" value, the token-level value takes precedence.

Example of a response containing the introspection object with the "permissions" parameter containing a single permission:

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

```
{
  "active":true,
  "exp":1256953732,
  "iat":1256912345,
  "permissions":[
    {
      "resource_id":"112210f47de98100",
      "resource_scopes":[
        "view",
        "http://photoz.example.com/dev/actions/print"
      ],
      "exp":1256953732
    }
  ]
}
```

6. Error Messages

If a request is successfully authenticated, but is invalid for another reason, the authorization server produces an error response by supplying a JSON-encoded object with the following members in the body of the HTTP response:

`error` REQUIRED except as noted. A single error code. Values for this parameter are defined throughout this specification.

`error_description` OPTIONAL. Human-readable text providing additional information.

`error_uri` OPTIONAL. A URI identifying a human-readable web page with information about the error.

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Cache-Control: no-store
...
{
  "error": "invalid_resource_id",
  "error_description": "Permission request failed with bad resource ID.",
  "error_uri": "https://as.example.com/uma_errors/invalid_resource_id"
}
```

7. Security Considerations

This specification inherits the security considerations of [UMAGrant] and has the following additional security considerations.

In the context of federated authorization, more parties may be operating and using UMA software entities, and thus may need to establish agreements about the parties' rights and responsibilities on a legal or contractual level, as discussed in Section 5.8 of [UMAGrant].

The protection API is secured by means of OAuth (through the use of the PAT). Therefore, it is susceptible to OAuth threats.

8. Privacy Considerations

This specification inherits the privacy considerations of [UMAGrant] and has the following additional privacy considerations.

As noted in Section 6.1 of [UMAGrant], the authorization server should apply authorization, security, and time-to-live strategies in a way that favors resource owner needs and action so that removal of authorization grants is achieved in a timely fashion. PATs are another construct to which it can apply these strategies.

In the context of federated authorization, more parties may be operating and using UMA software entities, and thus may need to establish agreements about mutual rights, responsibilities, and common interpretations of UMA constructs for consistent and expected software behavior, as discussed in Section 6.4 of [UMAGrant].

The authorization server comes to be in possession of resource details that may reveal information about the resource owner, which the authorization server's trust relationship with the resource server is assumed to accommodate. The more information about a resource that is registered, the more risk of privacy compromise there is through a less-trusted authorization server. For example,

if resource owner Alice introduces her electronic health record resource server to an authorization server in the cloud, the authorization server may come to learn a great deal of detail about Alice's health information just so that she can control access by others to that information.

9. IANA Considerations

This document makes the following requests of IANA.

9.1. OAuth 2.0 Authorization Server Metadata Registry

This specification registers OAuth 2.0 authorization server metadata defined in Section 2, as required by Section 7.1 of [OAuthMeta].

9.1.1. Registry Contents

- o Metadata name: "permission_endpoint"
- o Metadata description: endpoint metadata
- o Change controller: Kantara Initiative User-Managed Access Work Group - staff@kantarainitiative.org
- o Specification document: Section 2 in this document
- o Metadata name: "resource_registration_endpoint"
- o Metadata description: endpoint metadata
- o Change controller: Kantara Initiative User-Managed Access Work Group - staff@kantarainitiative.org
- o Specification document: Section 2 in this document

9.2. OAuth Token Introspection Response Registration

This specification registers the name defined in Section 5.1.1, as required by Section 3.1 of [RFC7662].

9.2.1. Registry Contents

- o Name: "permissions"
- o Description: array of objects, each describing a scoped, time-limited permission for a resource

- o Change controller: Kantara Initiative User-Managed Access Work Group – staff@kantarainitiative.org
- o Specification document: Section 5.1.1 in this document

10. Acknowledgments

The following people made significant text contributions to the specification:

- o Paul C. Bryan, ForgeRock US, Inc. (former editor)
- o Domenico Catalano, Oracle (former author)
- o Mark Dobrinic, Cozmanova
- o George Fletcher, AOL
- o Thomas Hardjono, MIT (former editor)
- o Andrew Hindle, Hindle Consulting Limited
- o Lukasz Moren, Cloud Identity Ltd
- o James Phillpotts, ForgeRock
- o Christian Scholz, COMlounge GmbH (former editor)
- o Mike Schwartz, Gluu
- o Cigdem Sengul, Nominet UK
- o Jacek Szpot, Newcastle University

Additional contributors to this specification include the Kantara UMA Work Group participants, a list of whom can be found at [UMAnitarians].

11. References

11.1. Normative References

- [BCP195] Sheffer, Y., "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", May 2015, <<https://tools.ietf.org/html/bcp195>>.

- [OAuthMeta] Jones, M., "OAuth 2.0 Authorization Server Metadata", November 2017, <<https://tools.ietf.org/html/draft-ietf-oauth-discovery-08>>.
- [OIDCCore] Sakimura, N., "OpenID Connect Core 1.0 incorporating errata set 1", November 2014, <http://openid.net/specs/openid-connect-core-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>>.
- [RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, DOI 10.17487/RFC2616, June 1999, <<https://www.rfc-editor.org/info/rfc2616>>.
- [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>>.
- [RFC5785] Nottingham, M. and E. Hammer-Lahav, "Defining Well-Known Uniform Resource Identifiers (URIs)", RFC 5785, DOI 10.17487/RFC5785, April 2010, <<https://www.rfc-editor.org/info/rfc5785>>.
- [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>>.
- [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>>.
- [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>>.

- [RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, <<https://www.rfc-editor.org/info/rfc7159>>.
- [RFC7591] Richer, J., Ed., Jones, M., Bradley, J., Machulak, M., and P. Hunt, "OAuth 2.0 Dynamic Client Registration Protocol", RFC 7591, DOI 10.17487/RFC7591, July 2015, <<https://www.rfc-editor.org/info/rfc7591>>.
- [RFC7662] Richer, J., Ed., "OAuth 2.0 Token Introspection", RFC 7662, DOI 10.17487/RFC7662, October 2015, <<https://www.rfc-editor.org/info/rfc7662>>.
- [UMAGrant] Maler, E., "User-Managed Access (UMA) Grant for OAuth 2.0 Authorization", January 2019, <<https://docs.kantarainitiative.org/uma/rec-oauth-uma-grant-2.0.html>>.

11.2. Informative References

- [UMA-Impl] Maler, E., "UMA Implementer's Guide", 2017, <<https://kantarainitiative.org/confluence/display/uma/UMA+Implementer%27s+Guide>>.
- [UMAnitarians] Maler, E., "UMA Participant Roster", 2017, <<https://kantarainitiative.org/confluence/display/uma/Participant+Roster>>.

Authors' Addresses

Eve Maler (editor)
ForgeRock

Email: eve.maler@forgerock.com

Maciej Machulak
HSBC

Email: maciej.p.machulak@hsbc.com

Justin Richer
Bespoke Engineering

Email: justin@bspk.io

Thomas Hardjono
MIT

Email: hardjono@mit.edu

Network Working Group
Internet-Draft
Intended status: Informational
Expires: August 17, 2019

E. Maler, Ed.
ForgeRock
M. Machulak
HSBC
J. Richer
Bespoke Engineering
T. Hardjono
MIT
February 13, 2019

User-Managed Access (UMA) 2.0 Grant for OAuth 2.0 Authorization
draft-maler-oauth-umagrants-00

Abstract

This specification defines a means for a client, representing a requesting party, to use a permission ticket to request an OAuth 2.0 access token to gain access to a protected resource asynchronously from the time a resource owner authorizes access.

Status of This Memo

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

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

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

This Internet-Draft will expire on August 17, 2019.

Copyright Notice

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

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

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

Table of Contents

1. Introduction	3
1.1. Notational Conventions	4
1.2. Roles	4
1.3. Abstract Flow	5
1.3.1. Authorization Process	7
2. Authorization Server Metadata	8
3. Flow Details	9
3.1. Client Requests Resource Without Providing an Access Token	9
3.2. Resource Server Responds to Client's Tokenless Access Attempt	9
3.2.1. Resource Server Response to Client on Permission Request Success	10
3.2.2. Resource Server Response to Client on Permission Request Failure	10
3.3. Client Seeks RPT on Requesting Party's Behalf	11
3.3.1. Client Request to Authorization Server for RPT	11
3.3.2. Client Redirect of Requesting Party to Authorization Server for Interactive Claims-Gathering	13
3.3.3. Authorization Server Redirect of Requesting Party Back to Client After Interactive Claims-Gathering	15
3.3.4. Authorization Assessment and Results Determination	16
3.3.5. Authorization Server Response to Client on Authorization Success	18
3.3.6. Authorization Server Response to Client on Authorization Failure	20
3.4. Client Requests Resource and Provides an RPT	23
3.5. Resource Server Responds to Client's RPT-Accompanied Resource Request	23
3.6. Authorization Server Refreshes RPT	24
3.7. Client Requests Token Revocation	24
4. Profiles and Extensions	24
5. Security Considerations	25
5.1. Cross-Site Request Forgery	25
5.2. RPT and PCT Exposure	26
5.3. Strengthening RPT Protection Using Proof of Possession	27
5.4. Credentials-Guessing	28
5.5. Permission Ticket Management	28
5.6. Naive Implementations of Default-Deny Authorization	28
5.7. Requirements for Pre-Established Trust Regarding Claim Tokens	29

5.8. Profiles and Trust Establishment	29
6. Privacy Considerations	30
6.1. Policy Condition Setting, Time-to-Live Management, and Removal of Authorization Grants	30
6.2. Requesting Party Information at the Authorization Server	30
6.3. Resource Owner Information at the Resource Server	31
6.4. Profiles and Trust Establishment	31
7. IANA Considerations	31
7.1. Well-Known URI Registration	31
7.1.1. Registry Contents	31
7.2. OAuth 2.0 Authorization Server Metadata Registry	32
7.2.1. Registry Contents	32
7.3. OAuth 2.0 Dynamic Client Registration Metadata Registry	32
7.3.1. Registry Contents	32
7.4. OAuth 2.0 Extension Grant Parameters Registration	33
7.4.1. Registry Contents	33
7.5. OAuth 2.0 Extensions Error Registration	34
7.5.1. Registry Contents	34
7.6. OAuth Token Type Hints Registration	35
7.6.1. Registry Contents	35
8. Acknowledgments	35
9. References	36
9.1. Normative References	36
9.2. Informative References	37
Authors' Addresses	38

1. Introduction

This specification defines an extension OAuth 2.0 [RFC6749] grant. The grant enhances OAuth capabilities in the following ways:

- o The resource owner authorizes protected resource access to clients used by entities that are in a `_requesting party_` role. This enables party-to-party authorization, rather than authorization of application access alone.
- o The authorization server and resource server interact with the client and requesting party in a way that is `_asynchronous_` with respect to resource owner interactions. This lets a resource owner configure an authorization server with authorization grant rules (policy conditions) at will, rather than authorizing access token issuance synchronously just after authenticating.

For example, bank customer (resource owner) Alice with a bank account service (resource server) can use a sharing management service (authorization server) hosted by the bank to manage access to her various protected resources by spouse Bob, accounting professional Charline, and and financial information aggregation company Decide

Account, all using different client applications. Each of her bank accounts is a protected resource, and two different scopes of access she can control on them are viewing account data and accessing payment functions.

An OPTIONAL second specification, [UMAFedAuthz], defines a means for an UMA-enabled authorization server and resource server to be loosely coupled, or federated, in a resource owner context. This specification, together with [UMAFedAuthz], constitutes UMA 2.0.

1.1. Notational Conventions

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

Unless otherwise noted, all parameter names and values are case sensitive. JSON [RFC7159] data structures defined in this specification MAY contain extension parameters that are not defined in this specification. Any entity receiving or retrieving a JSON data structure SHOULD ignore extension parameters it is unable to understand. Extension names that are unprotected from collisions are outside the scope of this specification.

1.2. Roles

The UMA grant enhances the OAuth definitions of entities in order to accommodate the requesting party role.

resource owner

An entity capable of granting access to a protected resource, the "user" in User-Managed Access. The resource owner MAY be an end-user (natural person) or MAY be a non-human entity treated as a person for limited legal purposes (legal person), such as a corporation.

requesting party

A natural or legal person that uses a client to seek access to a protected resource. The requesting party may or may not be the same party as the resource owner.

client

An application that is capable of making requests for protected resources with the resource owner's authorization and on the requesting party's behalf.

resource server

A server that hosts resources on a resource owner's behalf and is capable of accepting and responding to requests for protected resources.

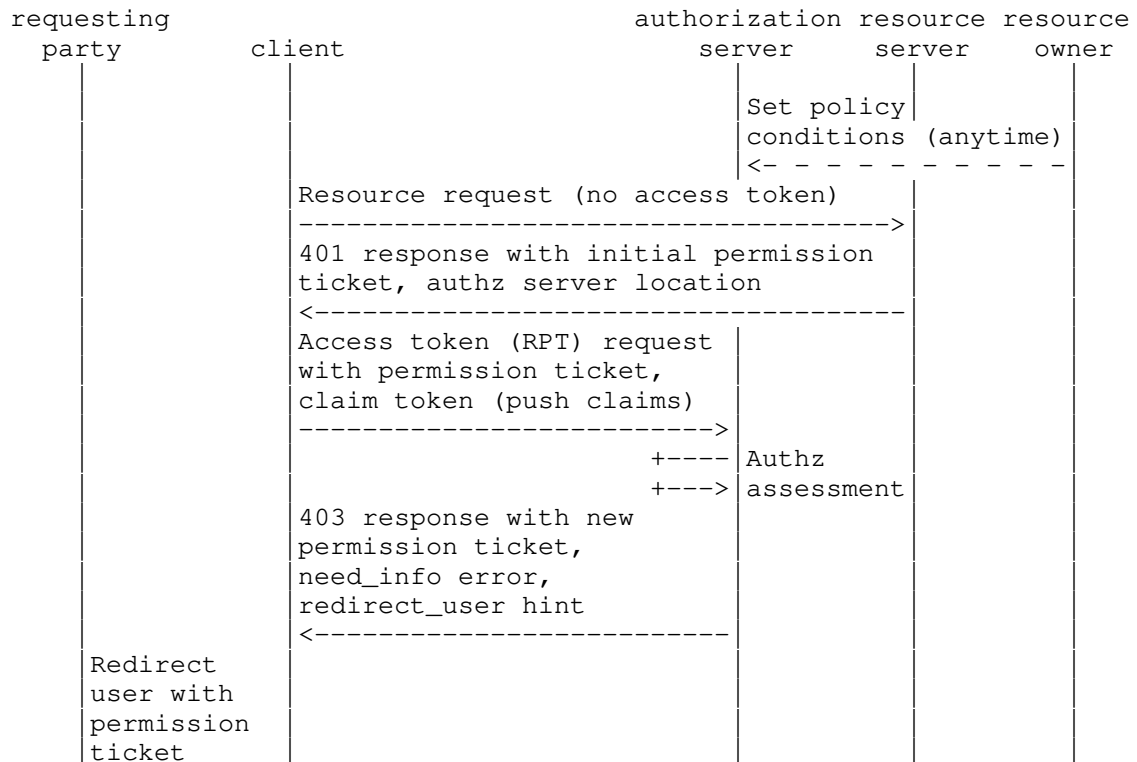
authorization server

A server that protects, on a resource owner's behalf, resources hosted at a resource server.

1.3. Abstract Flow

The UMA grant enhances the abstract protocol flow of OAuth.

Figure 1 shows an example flow illustrating a variety of messaging paths and artifacts. The resource owner entity and its communications with the authorization server are included for completeness, although policy condition setting is outside the scope of this specification and communications among the other four entities are asynchronous with respect to resource owner actions. Further, although both claims pushing and interactive claims gathering are shown, both might not typically be used in one scenario.



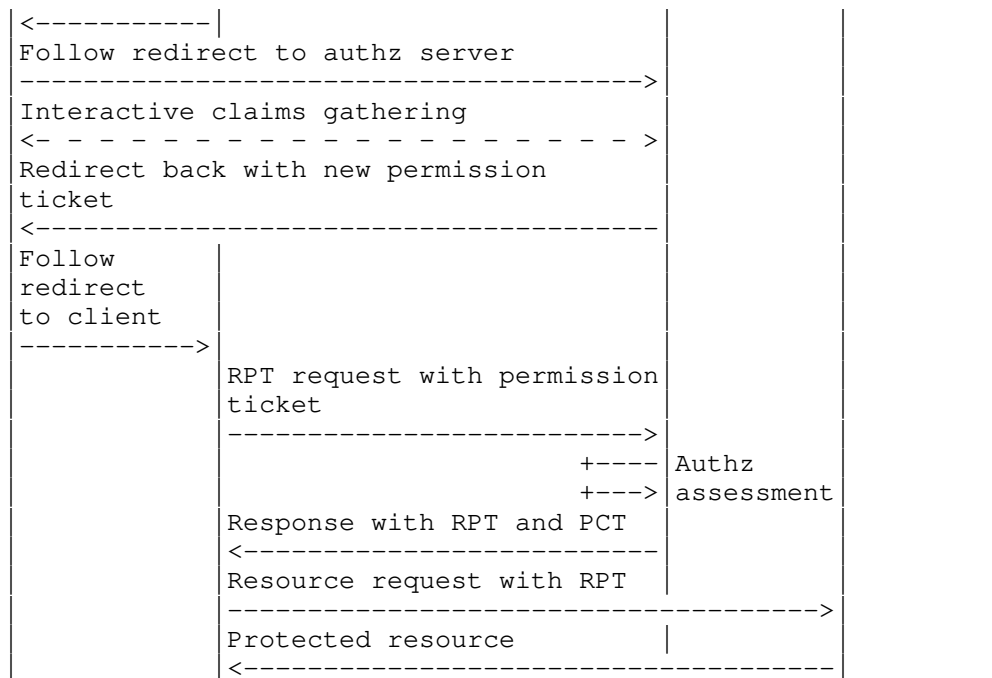


Figure 1: Example Flow

Following are key concepts relevant to this specification, as illustrated in the figure:

requesting party token (RPT) An OAuth access token associated with the UMA grant. An RPT is unique to a requesting party, client, authorization server, resource server, and resource owner.

permission Authorized access to a particular resource with some number of scopes bound to that resource. A permission ticket represents some number of requested permissions. An RPT represents some number of granted permissions. Permissions are part of the authorization server's process and are opaque to the client.

permission ticket A correlation handle representing requested permissions that is created and maintained by the authorization server, initially passed to the client by the resource server, and presented by the client at the token endpoint and during requesting party redirects.

authorization process The process through which the authorization server determines whether it should issue an RPT to the client on the requesting party's behalf, based on a variety of inputs. A key component of the process is authorization assessment. (See Section 1.3.1.)

claim A statement of the value or values of one or more attributes of an entity. The authorization server typically needs to collect and assess one or more claims of the requesting party or client against policy conditions as part of protecting a resource. The two methods available for UMA claims collection are claims pushing and interactive claims gathering. Note: Claims collection might involve authentication for unique user identification, but depending on policy conditions might additionally or instead involve the collection of non-uniquely identifying attributes, authorization for some action (for example, see Section 3.3.3), or other statements of agreement.

claim token A package of claims provided directly by the client to the authorization server through claims pushing.

persisted claims token (PCT) A correlation handle issued by an authorization server that represents a set of claims collected during one authorization process, available for a client to use in attempting to optimize a future authorization process.

Note: How the client acquired knowledge of the resource server's interface and the specific endpoint of the desired protected resource is outside the scope of this specification. For example, the resource server might have a programmatic API or it might serve up simple web pages, and the resource owner might have advertised the endpoint publicly on a blog or other website, listed it in a discovery service, or emailed a link to a particular intended requesting party.

1.3.1. Authorization Process

The authorization process involves the following activities:

- o Claims collection. Claims pushing by a client is defined in Section 3.3.1, and interactive claims gathering with an end-user requesting party is defined in Section 3.3.2.
- o Authorization assessment (as defined in Section 3.3.4). Authorization assessment involves the authorization server assembling and evaluating policy conditions, scopes, claims, and any other relevant information sourced outside of UMA claims collection flows, in order to mitigate access authorization risk.

- o Authorization results determination (as defined in Section 3.3.4). The authorization server either returns a success code (as defined in Section 3.3.5), an RPT, and an optional PCT, or an error code (as defined in Section 3.3.6). If the error code is "need_info" or "request_submitted", the authorization server provides a permission ticket, giving the client an opportunity to continue within the same authorization process (including engaging in further claims collection).

Different choices of claims collection methods, other inputs to authorization assessment, and error codes might be best suited for different deployment ecosystems. For example, where no pre-established relationship is expected between the resource owner's authorization server and the requesting party, initial requesting party redirection might be a useful pattern, at which point the authorization server might either authenticate the requesting party locally or serve as a relying party for a remote identity provider. Where a common authorization server functions as an identity provider for all resource owners and requesting parties, having the client push claim tokens sourced from that central server itself with a pre-negotiated format and contents might be a useful pattern.

2. Authorization Server Metadata

The authorization server supplies metadata in a discovery document to declare its endpoints. The client uses this discovery document to discover these endpoints for use in the flows defined in Section 3.

The authorization server **MUST** make a discovery document available. The structure of the discovery document **MUST** conform to that defined in [OAuthMeta]. The discovery document **MUST** be available at an endpoint formed by concatenating the string `"/.well-known/uma2-configuration"` to the "issuer" metadata value defined in [OAuthMeta], using the well-known URI syntax and semantics defined in [RFC5785]. In addition to the metadata defined in [OAuthMeta], this specification defines the following metadata for inclusion in the discovery document:

claims_interaction_endpoint

OPTIONAL. A static endpoint URI at which the authorization server declares that it interacts with end-user requesting parties to gather claims. If the authorization server also provides a claims interaction endpoint URI as part of its "redirect_user" hint in a "need_info" response to a client on authorization failure (see Section 3.3.6), that value overrides this metadata value. Providing the static endpoint URI is useful for enabling interactive claims gathering prior to any

pushed-claims flows taking place, for example, for gathering authorization for subsequent claim pushing (see Section 3.3.2).

`uma_profiles_supported`

OPTIONAL. UMA profiles and extensions supported by this authorization server. The value is an array of string values, where each string value is a URI identifying an UMA profile or extension. As discussed in Section 4, an authorization server supporting a profile or extension related to UMA SHOULD supply the specification's identifying URI (if any) here.

If the authorization server supports dynamic client registration, it MUST allow client applications to register "claims_redirect_uri" metadata, as defined in Section 3.3.2, using the following metadata field:

`claims_redirect_uris`

OPTIONAL. Array of one or more claims redirection URIs.

3. Flow Details

3.1. Client Requests Resource Without Providing an Access Token

The client requests a protected resource without providing any access token.

Note: This process does not assume that any relevant policy conditions have already been defined at the authorization server.

For an example of how the resource server can put resources under the protection of an authorization server, see [UMAFedAuthz].

Example of a client request at a protected resource without providing an access token:

```
GET /users/alice/album/photo.jpg HTTP/1.1 Host:
    photoz.example.com ...
```

3.2. Resource Server Responds to Client's Tokenless Access Attempt

The resource server responds to the client's tokenless resource request.

The resource server MUST obtain a permission ticket from the authorization server to provide in its response, but the means of doing so is outside the scope of this specification. For an example of how the resource server can obtain the permission ticket, see [UMAFedAuthz].

The process of choosing what permissions to request from the authorization server may require interpretation and mapping of the client's resource request. The resource server SHOULD request a set of permissions with scopes that is reasonable for the client's resource request.

Note: In order for the resource server to know which authorization server to approach for the permission ticket and on which resource owner's behalf, it needs to derive the necessary information using cues provided by the structure of the API where the resource request was made, rather than by an access token. Commonly, this information can be passed through the URI, headers, or body of the client's request. Alternatively, the entire interface could be dedicated to the use of a single resource owner and protected by a single authorization server.

See Section 5.5 for permission ticket security considerations.

3.2.1. Resource Server Response to Client on Permission Request Success

If the resource server is able to provide a permission ticket from the authorization server, it responds to the client by providing a "WWW-Authenticate" header with the authentication scheme "UMA", with the "issuer" URI from the authorization server's discovery document in an "as_uri" parameter and the permission ticket in a "ticket" parameter.

For example:

```
HTTP/1.1 401 Unauthorized WWW-Authenticate: UMA
      realm="example", as_uri="https://as.example.com",
      ticket="016f84e8-f9b9-11e0-bd6f-0021cc6004de" ...
```

3.2.2. Resource Server Response to Client on Permission Request Failure

If the resource server is unable to provide a permission ticket from the authorization server, then it includes a header of the following form in its response to the client: "Warning: 199 - "UMA Authorization Server Unreachable"".

For example:

```
HTTP/1.1 403 Forbidden Warning: 199 - "UMA Authorization
      Server Unreachable" ...
```

Without an authorization server location and permission ticket, the client is unable to continue.

3.3. Client Seeks RPT on Requesting Party's Behalf

The client seeks issuance of an RPT.

This process assumes that:

- o The client has obtained a permission ticket and an authorization server location from the resource server.
- o The client has retrieved the authorization server's discovery document as needed.
- o The client has obtained a client identifier or a full set of client credentials as appropriate, either statically or dynamically (for example, through [RFC7591] or [OIDCDynClientReg]). This grant works with clients of both confidential and public types.

Initiation of this process has two options. One option is for the client to request an RPT from the token endpoint immediately, as defined in Section 3.3.1. Claim pushing is available at this endpoint. The other option, if the authorization server's discovery document statically provided a claims interaction endpoint, is for the client to redirect the requesting party immediately to that endpoint for interactive claims gathering, as defined in Section 3.3.2.

3.3.1. Client Request to Authorization Server for RPT

The client makes a request to the token endpoint by sending the following parameters:

grant_type REQUIRED. MUST be the value
"urn:ietf:params:oauth:grant-type:uma-ticket".

ticket REQUIRED. The most recent permission ticket received by the client as part of this authorization process.

claim_token OPTIONAL. If this parameter is used, it MUST appear together with the "claim_token_format" parameter. A string containing directly pushed claim information in the indicated format. It MUST be base64url encoded unless specified otherwise by the claim token format. The client MAY provide this information on both first and subsequent requests to this endpoint. The client and authorization server together might need to establish proper audience restrictions for the claim token prior to claims pushing. See Section 5.7 and Section 6.2 for security and privacy considerations regarding pushing of claims.

`claim_token_format` OPTIONAL. If this parameter is used, it MUST appear together with the "claim_token" parameter. A string specifying the format of the claim token in which the client is directly pushing claims to the authorization server. The string MAY be a URI. Examples of potential types of claim token formats are [OIDCCore] ID Tokens and SAML assertions.

`pct` OPTIONAL. If the authorization server previously returned a PCT along with an RPT, the client MAY include the PCT in order to optimize the process of seeking a new RPT. Given that some claims represented by a PCT are likely to contain identity information about a requesting party, a client supplying a PCT in its RPT request MUST make a best effort to ensure that the requesting party using the client now is the same as the requesting party that was associated with the PCT when it was issued. See Section 5.7 and Section 6.2 for additional security and privacy considerations regarding persistence of claims. The client MAY use the PCT for the same requesting party when seeking an RPT for a resource different from the one sought when the PCT was issued, or a protected resource at a different resource server entirely. See Section 5.2 for additional PCT security considerations. See Section 3.3.5 for the form of the authorization server's response with a PCT.

`rpt` OPTIONAL. Supplying an existing RPT (which MAY be expired) gives the authorization server the option of upgrading that RPT instead of issuing a new one (see Section 3.3.5.1 for more about this option).

`scope` OPTIONAL. A string of space-separated values representing requested scopes. For the authorization server to consider any requested scope in its assessment, the client MUST have been pre-registered for the same scope with the authorization server. The client should consult the resource server's API documentation for details about which scopes it can expect the resource server's initial returned permission ticket to represent as part of the authorization assessment (see Section 3.3.4).

Example of a request message with no optional parameters (line breaks are shown only for display convenience):

```
POST /token HTTP/1.1 Host: as.example.com Authorization:
Basic jwflG53^sad$#f ...
grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Auma-ticket
&ticket=016f84e8-f9b9-11e0-bd6f-0021cc6004de
```

Example of a request message that includes an existing RPT for upgrading, a scope being sought that was previously registered with the authorization server, and a PCT and a claim token for consideration in the authorization process:

```
POST /token HTTP/1.1 Host: as.example.com Authorization:
    Basic jwflG53^sad$#f ...
    grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Auma-ticket
    &ticket=016f84e8-f9b9-11e0-bd6f-0021cc6004de
    &claim_token=eyJ0...
    &claim_token_format=http%3A%2F%2Fopenid.net%2Fspecs%2Fopenid-connect
-core-1_0.html%23IDToken
    &pct=c2F2ZWRjb25zZW50
    &rpt=sbjsbhs(/SSJHBSUSSJHVhjsgvshgsv
    &scope=read
```

This specification provides a means to define profiles of claim token formats for use with UMA (see Section 4). The authorization server SHOULD document the profiles it supports in its discovery document.

3.3.2. Client Redirect of Requesting Party to Authorization Server for Interactive Claims-Gathering

The client redirects an end-user requesting party to the authorization server's claims interaction endpoint for one or more interactive claims-gathering processes as the authorization server requires. These can include direct interactions, such as account registration and authentication local to the authorization server as an identity provider, filling out a questionnaire, or asking the user to authorize subsequent collection of claims by interaction or pushing, and persistent storage of such claims (for example, as associated with a PCT). Interactions could also involve further redirection, for example, for federated (such as social) authentication at a remote identity provider, and other federated claims gathering. See Section 5.7 and Section 6.2 for security and privacy considerations regarding pushing and persistence of claims.

The client might have initiated redirection immediately on receiving an initial permission ticket from the resource server, or, for example, in response to receiving a "redirect_user" hint in a "need_info" error (see Section 3.3.6).

In order for the client to redirect the requesting party immediately on receiving the initial permission ticket from the resource server, this process assumes that the authorization server has statically declared its claims interaction endpoint in its discovery document.

The client constructs the request URI by adding the following parameters to the query component of the claims interaction endpoint URI using the "application/x-www-form-urlencoded" format:

`client_id` REQUIRED. The client's identifier issued by the authorization server.

`ticket` REQUIRED. The most recent permission ticket received by the client as part of this authorization process.

`claims_redirect_uri` REQUIRED if the client has pre-registered multiple claims redirection URIs or has pre-registered no claims redirection URI; OPTIONAL only if the client has pre-registered a single claims redirection URI. The URI to which the client wishes the authorization server to direct the requesting party's user agent after completing its interaction. The URI MUST be absolute, MAY contain an "application/x-www-form-urlencoded"-formatted query parameter component that MUST be retained when adding additional parameters, and MUST NOT contain a fragment component. The client SHOULD pre-register its "claims_redirect_uri" with the authorization server, and the authorization server SHOULD require all clients, and MUST require public clients, to pre-register their claims redirection endpoints (see Section 2). Claims redirection URIs are different from the redirection URIs defined in [RFC6749] in that they are intended for the exclusive use of requesting parties and not resource owners. Therefore, authorization servers MUST NOT redirect requesting parties to pre-registered redirection URIs defined in [RFC6749] unless such URIs are also pre-registered specifically as claims redirection URIs. If the URI is pre-registered, this URI MUST exactly match one of the pre-registered claims redirection URIs, with the matching performed as described in Section 6.2.1 of [RFC3986] (Simple String Comparison).

`state` RECOMMENDED. An opaque value used by the client to maintain state between the request and callback. The authorization server includes this value when redirecting the user agent back to the client. The use of this parameter is for preventing cross-site request forgery (see Section 5.1 for further security information).

Example of a request issued by a client application (line breaks are shown only for display convenience):

```
GET /rqp_claims?client_id=some_client_id
    &ticket=016f84e8-f9b9-11e0-bd6f-0021cc6004de
    &claims_redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fredirect
_claims
    &state=abc HTTP/1.1 Host: as.example.com
```

3.3.3. Authorization Server Redirect of Requesting Party Back to Client After Interactive Claims-Gathering

At the conclusion of a successful interaction with the requesting party, the authorization server returns the requesting party to the client, adding the following parameters to the query component of the claims redirection URI using the "application/x-www-form-urlencoded" format:

`ticket` REQUIRED. A permission ticket that allows the client to make further requests to the authorization server during this authorization process. The value MUST NOT be the same as the one the client used to make its request.

`state` OPTIONAL. The same state value that the client provided in the request. It MUST be present if and only if the client provided it (see Section 5.1 for further security information).

Note: Interactive claims-gathering processes are outside the scope of this specification. The purpose of the interaction is for the authorization server to gather information for its own authorization assessment purposes. This redirection does not involve sending any of the information back to the client.

The authorization server MAY use interactive claims-gathering to request authorization from the requesting party for persisting claims across authorization processes. Such persisted claims will be represented by a PCT issued to the client in a subsequent step.

The client MUST ignore unrecognized response parameters. If the request fails due to a missing, invalid, or mismatching claims redirection URI, or if the client identifier is missing or invalid, the authorization server SHOULD inform the requesting party of the error and MUST NOT automatically redirect the user agent to the invalid redirection URI.

If the request fails for reasons other than a missing or invalid claims redirection URI, the authorization server informs the client by adding an "error" parameter to the query component of the claims redirection URI as defined in Section 4.1.2.1 of [RFC6749].

Example of a response issued by an authorization server (line breaks are shown only for display convenience):

```
HTTP/1.1 302 Found Location:
  https://client.example.com/redirect_claims?
  ticket=CHJpdmFjeSBpcyBjb250ZXh0LCBjb250cm9s&state=abc
```

3.3.4. Authorization Assessment and Results Determination

When the authorization server has received a request for an RPT from a client as defined in Section 3.3.1, it assesses whether the client is authorized to receive the requested RPT and determines the results.

The authorization server **MUST** apply the following conceptual authorization assessment calculation in determining authorization results. Note: As this calculation is internal to authorization server operations, its particulars are outside the scope of this specification.

1. Assemble a set called `_RegisteredScopes_` containing the scopes for which the client is pre-registered (either dynamically or through some static process) at the authorization server. Assemble a set called `_RequestedScopes_` containing the scopes the client most recently requested at the token endpoint. The permission ticket that was presented by the client at the token endpoint represents some number of resources, each with some number of scopes; for each of those resources, assemble a set called `_TicketScopes(resource)_` containing the scopes associated with that resource.
2. For each resource in the permission ticket, determine a final set of requested scopes as follows:
`_RequestedScopes(resource)={TicketScopes(resource) ∪ {RegisteredScopes ∩ RequestedScopes}}_`. Treat each scope in `_RegisteredScopes ∩ RequestedScopes_` as matching any available scope associated with a resource found in the permission ticket.
3. For each `_RequestedScopes(resource)_` set, determine all operative policy conditions, and claims and other relevant information serving as input to them, and evaluate its authorization status.
4. For each scope in `_RequestedScopes(resource)_` that passes the evaluation, add it to a set called `_CandidateGrantedScopes(resource)_`.

Note: Claims and other information gathered during one authorization process may become out of date in terms of their relevance for future authorization processes. The authorization server is responsible for managing such relevance wherever information associated with a PCT, or other persistently stored information, is used as input to authorization, including policy conditions themselves.

Note: Since the authorization server's policy expression and evaluation capabilities are outside the scope of this specification, any one implementation might take a simple or arbitrarily complex form, with varying abilities to combine or perform calculations over claims and their values. For example, logical operations such as accepting "either claim value A or claim value B" as correct are possible to implement.

In the authorization results phase, the authorization server examines each `_CandidateGrantedScopes(resource)_` set to determine whether to issue an RPT and what permissions should be associated with it. If all `_RequestedScopes(resource)_` sets can be granted, then the authorization server subsequently responds with a success code and issues an RPT containing `_CandidateGrantedScopes_` for each resource.

Otherwise, the authorization server subsequently issues either an RPT containing `_CandidateGrantedScopes_` for each resource, or one of the error codes, as appropriate. The reason for the two options is that granting only partial scopes might not be useful for the client's and requesting party's purposes in seeking authorization for access. The choice of error depends on policy conditions and the authorization server's implementation choices. The conditions for the "need_info", "request_denied", and "request_submitted" error codes are dependent on authorization assessment and thus these codes might be more likely than the others to be issued subsequent to such a calculation.

The following example illustrates authorization assessment and partial results.

- o The resource server has three of the resource owner's resources of interest to the client and requesting party, "photo1" and "photo2" with scopes "view", "resize", "print", and "download", and "album" with scopes "view", "edit", and "download". It considers "photo1" and "photo2" to be logically "inside" "album".
- o Though the exact contents of RPTs, permissions, and permission requests are opaque to the client, the resource server has documented its API, available scopes, and permission requesting practices. For example, if the client requests an album resource, it expects that the resource server will request a permission for the album with a scope that approximates the attempted client operation, but will also request permissions for all the photos "inside" the album, with "view" scope only.
- o The client has a pre-registered scope of "download" with the authorization server. This enables the client later to request this scope dynamically on behalf of its requesting party from the

token endpoint. The authorization server assembles the set `_RegisteredScopes_` with contents of scope "download".

- o The client requests the album resource in an attempt to edit it, so the resource server obtains a permission ticket with three permissions in it: for "album" with a scope of "edit", and for "photo1" and "photo2", each with a scope of "view". The authorization server assembles the following sets:
`_TicketScopes_("album")` containing "edit",
`_TicketScopes_("photo1")` containing "view", and
`_TicketScopes_("photo2")` containing "view".
- o While asking for an RPT at the token endpoint, the client requests "download" scope on the requesting party's behalf. The authorization server determines the contents of the following sets: `_RequestedScopes_("album")` containing "edit" and "download", `_RequestedScopes_("photo1")` containing "view" and "download", and `_RequestedScopes_("photo2")` containing "view" and "download".
- o The resource owner has set policy conditions that allow access by this particular requesting party only to "photo1" and only for "view" scope.
- o Based on the authorization server's authorization assessment calculation, it determines the contents of the following sets: `_CandidateGrantedScopes_("album")` containing no scopes, `_CandidateGrantedScopes_("photo1")` containing "view", and `_CandidateGrantedScopes_("photo2")` containing no scopes. This adds up to less than in the corresponding `_RequestedScopes_` sets. The authorization server therefore has a choice whether to issue an RPT (in this case, containing a permission for "photo1" with "view" scope) or an error (say, "request_denied", or "request_submitted" if has a way to notify the resource owner about the album editing resource request and seek an added policy covering it).

See Section 5.6 for a discussion of authorization implementation threats.

3.3.5. Authorization Server Response to Client on Authorization Success

If the authorization server's assessment process results in issuance of permissions, it issues the RPT with which it has associated the permissions by using the successful response form defined in Section 5.1 of [RFC6749].

The authorization server MAY return a refresh token. See Section 3.6 for more information about refreshing an RPT.

The authorization server MAY add the following parameters to its response:

pct OPTIONAL. A correlation handle representing claims and other information collected during this authorization process, which the client is able to present later in order to optimize future authorization processes on behalf of a requesting party. The PCT MUST be unguessable by an attacker. The PCT MUST NOT disclose claims from the requesting party directly to possessors of the PCT. Instead, such claims SHOULD be associated by reference to the PCT or expressed in an encrypted format that can be decrypted only by the authorization server that issued the PCT. See Section 3.3.2 for more information about the end-user requesting party interaction option. See Section 5.2 for additional PCT security considerations.

upgraded OPTIONAL. Boolean value. If the client submits an RPT in the request and the authorization server includes the permissions of the RPT from the request as part of the newly issued RPT, then it MUST set this value to "true". If it sets the value to "false" or the value is absent, the client MUST act as if the newly issued RPT does not include the permissions associated with the RPT from the request. (See Section 3.3.5.1.)

The authorization server MAY include any of the parameters defined in Section 5.1 of [RFC6749] on its response, except that it SHOULD NOT include the "scope" parameter. This is because for an RPT's permissions, each scope is associated with a specific resource, even though this association is opaque to the client. Note: The outcome of authorization assessment may result in expiration periods for RPTs, permissions, and refresh tokens that can affect the client's later requests for refreshing the RPT.

Example:

```
HTTP/1.1 200 OK Content-Type: application/json ... {
  "access_token":"sbjsbhs(/SSJHBSUSSJHVhjsgvhsgvshgsv",
  "token_type":"Bearer" }
```

Example with a PCT in the response:

```
HTTP/1.1 200 OK Content-Type: application/json ... {
  "access_token":"sbjsbhs(/SSJHBSUSSJHVhjsgvhsgvshgsv",
  "token_type":"Bearer", "pct":"c2F2ZWRjb25zZW50" }
```

3.3.5.1. Authorization Server Upgrades RPT

The authorization server MAY implement RPT upgrading. The authorization server SHOULD document its practices regarding RPT upgrades and to act consistently with respect to RPT upgrades so as to enable clients to manage received RPTs efficiently.

If the authorization server has implemented RPT upgrading, the client has submitted an RPT in its request, and the result is success, the authorization server adds the permissions from the client's previous RPT to the RPT it is about to issue, setting the value of "upgraded" in its response containing the upgraded RPT to "true".

If the authorization server is upgrading an RPT, and the RPT string is new rather than repeating the RPT provided by the client in the request, then the authorization server SHOULD revoke the existing RPT, if possible, and the client MUST discard its previous RPT. If the authorization server does not upgrade the RPT but issues a new RPT, the client MAY retain the existing RPT.

Example with "upgraded" in the response:

```
HTTP/1.1 200 OK Content-Type: application/json ... {  
    "access_token":"sbjsbhs(/SSJHBSUSSJHVhjsgvhsgvshgsv",  
    "token_type":"Bearer", "upgraded":true }  
}
```

3.3.6. Authorization Server Response to Client on Authorization Failure

If the client's request to the token endpoint results in failure, the authorization server responds with an error, as defined in Section 5.2 of [RFC6749] and as follows.

invalid_grant If the provided permission ticket was not found at the authorization server, or the provided permission ticket has expired, or any other original reasons to use this error code are found as defined in [RFC6749], the authorization server responds with the HTTP 400 (Bad Request) status code.

invalid_scope At least one of the scopes included in the request does not match an available scope for any of the resources associated with requested permissions for the permission ticket provided by the client. The authorization server MAY also return this error when at least one of the scopes included in the request does not match a scope for which the client is pre-registered with the authorization server. The authorization server responds with the HTTP 400 (Bad Request) status code.

need_info The authorization server needs additional information in order for a request to succeed, for example, a provided claim token was invalid or expired, or had an incorrect format, or additional claims are needed to complete the authorization assessment. The authorization server responds with the HTTP 403 (Forbidden) status code. It MUST include a "ticket" parameter, and it MUST also include either the "required_claims" parameter or the "redirect_user" parameter, or both, as hints about the information it needs.

ticket REQUIRED. A permission ticket that allows the client to make a further request to the authorization server's token endpoint as part of this same authorization process, potentially immediately. The value MUST NOT be the same as the one the client used to make its request.

required_claims An array of objects that describe the required claims, with the following subparameters:

claim_token_format OPTIONAL. An array of strings specifying a set of acceptable formats for a claim token pushed by the client containing this claim, as defined in Section 3.3.1. Any one of the referenced formats would satisfy the authorization server's requirements. Each string MAY be a URI.

claim_type OPTIONAL. A string, indicating the expected interpretation of the provided claim value. The string MAY be a URI.

friendly_name OPTIONAL. A string that provides a human-readable form of the claim's name. This can be useful as a "display name" for use in user interfaces in cases where the actual name is complex or opaque, such as an OID or a UUID.

issuer OPTIONAL. An array of strings specifying a set of acceptable issuing authorities for the claim. Any one of the referenced authorities would satisfy the authorization server's requirements. Each string MAY be a URI.

name OPTIONAL. A string (which MAY be a URI) representing the name of the claim; the "key" in a key-value pair.

redirect_user The claims interaction endpoint URI to which to redirect the end-user requesting party at the authorization server to continue the process of interactive claims gathering, as defined in Section 3.3.2. For example, the authorization server could require the requesting party to log in to an

account, or fill out a CAPTCHA to help prove humanness, or perform any number of other interactive tasks. If the requesting party is not an end-user, then no client action is possible on receiving the hint. If a static claims interaction endpoint was also provided in the authorization server's discovery document, then this value overrides the static value. Providing a value in this response might be appropriate, for example, if the URI needs to be customized per requesting party with a query parameter.

`request_denied` The client is not authorized to have these permissions. The authorization server responds with the HTTP 403 (Forbidden) status code.

`request_submitted` The authorization server requires intervention by the resource owner to determine whether the client is authorized to have these permissions. The authorization server responds with the HTTP 403 (Forbidden) status code. It MUST include a "ticket" parameter and MAY include an "interval" parameter.

`ticket` REQUIRED. A permission ticket that allows the client to make one or more later polling requests to the token endpoint as part of this same authorization process, when the resource owner might have completed some approval (or denial) action. The value MUST NOT be the same as the one the client used to make its request.

`interval` OPTIONAL. The minimum amount of time in seconds that the client SHOULD wait between polling requests to the token endpoint. See Section 5.5 for security considerations in scenarios involving polling and consequences for permission ticket lifetimes.

Example when the permission ticket was not found or has expired:

```
HTTP/1.1 400 Bad Request Content-Type: application/json
Cache-Control: no-store ... { "error":"invalid_grant" }
```

Example of a "need_info" response with hints about required claims:

```
HTTP/1.1 403 Forbidden Content-Type: application/json
Cache-Control: no-store ... { "error":"need_info",
  "ticket":"ZXJyb3JfZGV0YWlscw==", "required_claims":[ {
    "claim_token_format":[
      "http://openid.net/specs/openid-connect-core-1_0.html#IDToken" ],
    "claim_type":"urn:oid:0.9.2342.19200300.100.1.3",
    "friendly_name":"email", "issuer":[ "https://example.com/idp" ],
    "name":"email23423453ou453" } ] }
```

Example of a "need_info" response with a hint to redirect the requesting party to a claims interaction endpoint:

```
HTTP/1.1 403 Forbidden Content-Type: application/json
Cache-Control: no-store ... { "error":"need_info",
    "ticket":"ZXJyb3JfZGV0YWlscw==",
    "redirect_user":"https://as.example.com/rqp_claims?id=2346576421"
}
```

Example when the client was not authorized to have the permissions:

```
HTTP/1.1 403 Forbidden Content-Type: application/json
Cache-Control: no-store ... { "error":"request_denied" }
```

Example when the authorization server requires resource owner intervention, including the optional "interval" parameter:

```
HTTP/1.1 403 Forbidden Content-Type: application/json
Cache-Control: no-store ... { "error":"request_submitted",
    "ticket"?:ZXJyb3JfZGV0YWlscw==, "interval": 5 }
```

3.4. Client Requests Resource and Provides an RPT

The client requests the resource, now in possession of an RPT. The client uses [RFC6750] for a bearer token, and any other suitable presentation mechanism for an RPT of another access token type.

Example of a client request for the resource carrying an RPT:

```
GET /users/alice/album/photo.jpg HTTP/1.1 Authorization:
    Bearer sbjsbhs(/SSJHBSUSSJHVhjsghvshgsv Host: photoz.example.com
...
```

3.5. Resource Server Responds to Client's RPT-Accompanied Resource Request

The resource server responds to the client's RPT-accompanied resource request.

If the resource request fails, the resource server responds as if the request were unaccompanied by an access token, as defined in Section 3.2.

The resource server MUST NOT give access in the case of an invalid RPT or an RPT associated with insufficient authorization.

For an example of how the resource server can introspect the RPT and its permissions at the authorization server prior to responding to the client's request, see [UMAFedAuthz].

3.6. Authorization Server Refreshes RPT

As noted in Section 3.3.5, when issuing an RPT, the authorization server MAY also issue a refresh token.

Having previously received a refresh token from the authorization server, the client MAY use the refresh token grant as defined in [RFC6749] to attempt to refresh an expired RPT. If the client includes the "scope" parameter in its request, the authorization server MAY limit the scopes in the permissions associated with any resulting refreshed RPT to the scopes requested by the client.

The authorization server MUST NOT perform an authorization assessment calculation on receiving the client's request to refresh an RPT.

3.7. Client Requests Token Revocation

If the authorization server presents a token revocation endpoint as defined in [RFC7009], the client MAY use the endpoint to request revocation of an RPT (access token), refresh token, or PCT previously issued to it on behalf of a requesting party. This specification defines the following token type hint value:

pct Helps the authorization server optimize lookup of a PCT for revocation.

4. Profiles and Extensions

An UMA profile restricts UMA's available options. An UMA extension defines how to use UMA's extensibility points. The two can be combined. Some reasons for creating profiles and extensions include:

- o A profile restricting options in order to tighten security
- o A profile/extension restricting options and adding messaging parameters for use with a specific industry API
- o A profile that documents a specific URI, format, and interpretation for pushed claim tokens (see Section 3.3.1)
- o An extension that defines additional metadata for the authorization server discovery document to define machine-readable usage details

The following actions are RECOMMENDED regarding the creation and use of profiles and extensions:

- o The creator of a profile or extension related to UMA SHOULD assign it a uniquely identifying URI.
- o The authorization server supporting a profile or extension related to UMA with such a URI SHOULD supply the identifying URI in its "uma_profiles_supported" metadata (see Section 2).

5. Security Considerations

This specification relies mainly on OAuth 2.0 security mechanisms as well as transport-level security. Thus, implementers are strongly advised to read [BCP195] and the security considerations in [RFC6749] (Section 10) and [RFC6750] (Section 5) along with the security considerations of any other OAuth token-defining specifications in use, along with the entire [RFC6819] specification, and apply the countermeasures described therein. As well, implementers should take into account the security considerations in all other normatively referenced specifications.

The following sections describe additional security considerations.

5.1. Cross-Site Request Forgery

Redirection used for gathering claims interactively from an end-user requesting party (described in Section 3.3.2) creates the potential for cross-site request forgery (CSRF). This may be the result of an open redirect if the authorization server does not force the client to pre-register its claims redirection endpoint, and server-side artifact tampering if the client does not avail itself of the "state" parameter.

A CSRF attack against the authorization server's claims interaction endpoint can result in an attacker obtaining authorization for access through a malicious client without involving or alerting the end-user requesting party. The authorization server MUST implement CSRF protection for its claims interaction endpoint and ensure that a malicious client cannot obtain authorization without the awareness and involvement of the requesting party.

If the client uses the interactive claims gathering feature, it MUST implement CSRF protection for its claims redirection URI. It SHOULD use the "state" parameter when redirecting the requesting party to the claims interaction endpoint. The value of the "state" parameter MUST be unguessable by an attacker. Once the authorization server redirects the requesting party back, with the required binding value

contained in the "state" parameter, the client MUST check that the value of the "state" parameter received is equal to the value sent in the initial redirection request. Depending on the type of application, a client has several methods for storing and later verifying the value of the "state" parameter in between the initial redirect and the eventual resulting request to the claims redirection URI, including storage in a server-side session-bound variable, cryptographic derivation from a browser cookie, or secure application-level storage. The client MUST treat requests containing an invalid or unknown "state" parameter value as an error.

The "state" parameter SHOULD NOT include sensitive client or requesting party information in plain text, as it is transmitted through third-party components (the requesting party's user agent) and could be stored insecurely.

5.2. RPT and PCT Exposure

When a client redirects an end-user requesting party to the claims interaction endpoint, the client provides no a priori context to the authorization server about which user is appearing at the endpoint, other than implicitly through the permission ticket. Thus, a malicious client has the opportunity to switch end-users -- say, enabling malicious end-user Carlos to impersonate legitimate end-user Bob, who might be represented by a PCT already in that client's possession and might even have authorized the issuance of that PCT -- after the redirect completes and before it returns to the token endpoint to seek permissions.

To mitigate this threat, the authorization server, with the support of the resource owner, should consider the following strategies in combination.

- o Require that the requesting party legitimately represent the wielder of the RPT on a legal or contractual level. This solution alone does not reduce the risk from a technical perspective.
- o Gather claims interactively from an end-user requesting party that demonstrate that some sufficiently strong level of authentication was performed.
- o Require claims to have a high degree of freshness in order for them to satisfy policy conditions.
- o Tighten time-to-live strategies around RPTs and their associated permissions (see Section 6.1).

The client MUST only share the RPT (access token) with the resource server and authorization server, as explained in Section 10.3 of [RFC6749], and thus MUST keep it confidential from the requesting party. Because a malicious requesting party (the user of the client in the UMA grant) may have incentives to steal an RPT that the resource owner (the user of the client in other OAuth grants) does not, this security consideration takes on especial importance.

The PCT is similar to a refresh token in that it allows non-interactive issuance of access tokens. The authorization server and client MUST keep the PCT confidential in transit and storage, and MUST NOT share the PCT with any entity other than each other. The authorization server MUST maintain the binding between the PCT and the client to which it was issued.

Given that the PCT represents a set of requesting party claims, a client supplying a PCT in its RPT request MUST make a best effort to ensure that the requesting party using the client now is the same as the requesting party that was associated with the PCT when it was issued. Different clients will have different capabilities in this respect; for example, some applications are single-user and perform no local authentication, associating all PCTs with the "current user", while others might have more sophisticated authentication and user mapping capabilities.

If the authorization server has reason to believe that a PCT is compromised, for example, if the PCT has been supplied by a client that has "impossible geography" parameters, the authorization server should consider not using the claims based on that PCT in its authorization assessment.

5.3. Strengthening RPT Protection Using Proof of Possession

After the client's resource request with an RPT, assuming the client sent an RPT of the bearer style such as defined in [RFC6750], the resource server will have received from the client the entire secret portion of the token. This specification assumes only bearer-type tokens because they are the only type standardized as of this specification's publication. However, to strengthen protection for RPTs using a proof-of-possession approach, the resource server could receive an RPT that consists of only a cryptographically signed token identifier, and then to validate the signature, it could, for example, submit the token identifier to the token introspection endpoint to obtain the necessary key information. The details of this usage are outside the scope of this specification.

5.4. Credentials-Guessing

Permission tickets and PCTs are additional credentials that the authorization server **MUST** prevent attackers from guessing, as defined in Section 10.10 of [RFC6749].

5.5. Permission Ticket Management

Within the constraints of making permission ticket values unguessable, the authorization server **MAY** format the permission ticket however it chooses, for example, either as a random string that references data held on the server or by including data within the ticket itself.

Permission tickets **MUST** be single-use. This prevents susceptibility to a session fixation attack.

The authorization server **MUST** invalidate a permission ticket when the client presents the permission ticket to either the token endpoint or the claims interaction endpoint, or when the permission ticket expires, whichever occurs first.

The client **SHOULD** check that the value of the "ticket" parameter it receives back from the authorization server in each response and each redirect of the requesting party back to it differs from the one it sent to the server in the initial request or redirect.

If the authorization server has reason to believe that a permission ticket is compromised, for example, because it has seen the permission ticket before and it believes the first appearance was from a legitimate client and the second appearance is from an attacker, it should consider invalidating any access tokens based on this evidence.

Given that scenarios involving the "request_submitted" error code are likely to involve polling intervals, the permission ticket needs to last long enough to give the client a chance to attempt a polling request, which then needs to figure into other permission ticket security considerations.

5.6. Naive Implementations of Default-Deny Authorization

While a reasonable approach for most scenarios is to implement the classic stance of default-deny ("everything that is not expressly allowed is forbidden"), corner cases can inadvertently result in default-permit behavior. For example, it is insufficient to create default "empty" policy conditions stating "no claims are needed", and

then accept an empty set of supplied claims as sufficient for access during authorization assessment.

5.7. Requirements for Pre-Established Trust Regarding Claim Tokens

When a client makes an RPT request, it has the opportunity to push a claim token to attempt to satisfy policy conditions (see Section 3.3.1).

Claim tokens of any format typically contain audience restrictions, and an authorization server would not typically be in the primary audience for a claim token held or generated by a client. It is RECOMMENDED to document how the client, authorization server, requesting party, and any additional ecosystem entities and parties will establish a trust relationship and communicate any required keying material in a claim token profile, as described in Section 4. Authorization servers are RECOMMENDED not to accept claim tokens pushed by untrusted clients and not to ignore audience restrictions found in claim tokens pushed by clients.

A malicious client could push a claim token to the authorization server (revealing the claims therein; see Section 6.2) to seek resource access on its own behalf prior to any opportunity for an end-user requesting party to authorize claims collection. It is RECOMMENDED either for trust relationships established by the ecosystem parties to include prior requesting party authorization as required, or for end-user requesting party authorization to be gathered interactively prior to claims pushing, as described in Section 3.3.2.

Some deployments could have exceptional circumstances allowing the authorization server to validate claim tokens. For example, if the authorization server itself is also the identity provider for the requesting party, then it would be able to validate any ID token that the client pushes as a claim token and also validate the client to which it was issued.

5.8. Profiles and Trust Establishment

Parties that are operating and using UMA software entities may need to establish agreements about the parties' rights and responsibilities on a legal or contractual level, along with common interpretations of UMA constructs for consistent and expected software behavior. These agreements can be used to improve the parties' respective security postures. Written profiles are a key mechanism for conveying and enforcing these agreements. Section 4 discusses profiling. See [UMA-legal] to learn about frameworks and

tools to assist in the legal and contractual elements of deploying UMA-enabled services.

6. Privacy Considerations

UMA has the following privacy considerations.

6.1. Policy Condition Setting, Time-to-Live Management, and Removal of Authorization Grants

The setting of policy conditions, the resource owner-authorization server interface, and the resource owner-resource server interface are outside the scope of this specification. (For an example of how a secure and authorized resource owner context can be established between the resource server and authorization server, see [UMAFedAuthz].)

A variety of flows and user interfaces for policy condition setting involving user agents for both of these servers are possible, each with different privacy consequences for end-user resource owners. As well, various authorization, security, and time-to-live strategies could be applied on a per-resource owner basis or a per-authorization server basis, as the entities see fit. Validity periods of RPTs, refresh tokens, permissions, caching periods for responses, and even OAuth client credentials are all subject to management. Different time-to-live strategies might be suitable for different resources and scopes.

In order to account for modifications of policy conditions that result in the withdrawal of authorization grants (for example, fewer scopes, fewer resources, or resources available for a shorter time) in as timely a fashion as possible, the authorization server should align its strategies for management of these factors with resource owner needs and actions rather than those of clients and requesting parties. For example, the authorization server may want to invalidate a client's RPT and refresh token as soon as a resource owner changes policy conditions in such a way as to deny the client and its requesting party future access to a full set of previously held permissions.

6.2. Requesting Party Information at the Authorization Server

Claims are likely to contain personal, personally identifiable, and sensitive information, particularly in the case of requesting parties who are end-users.

If the authorization server supports persisting claims for any length of time (for example, to support issuance of PCTs), then it SHOULD

provide a secure and privacy-protected means of storing claim data. It is also RECOMMENDED for the authorization server to use an interactive claims-gathering flow to ask an end-user requesting party for authorization to collect any claims subsequently and to persist their claims (for example, before issuing a PCT), if no prior requesting party authorization has been established among the ecosystem parties (see Section 5.7).

6.3. Resource Owner Information at the Resource Server

Since the client's initial request for a protected resource is made in an unauthorized and unauthenticated context, such requests are by definition open to all users. The response to that request includes the authorization server's location to enable the client to request an access token and present claims. If it is known out of band that authorization server is owned and controlled by a single user, or visiting the authorization server contains other identifying information, then an unauthenticated and unauthorized client would be able to tell which resource owner is associated with a given resource. Other information about the resource owner, such as organizational affiliation or group membership, may be gained from this transaction as well.

6.4. Profiles and Trust Establishment

Parties that are operating and using UMA software entities may need to establish agreements about mutual rights, responsibilities, and common interpretations of UMA constructs for consistent and expected software behavior. These agreements can be used to improve the parties' respective privacy postures. See Section 5.8 for more information. Additional considerations related to Privacy by Design concepts are discussed in [UMA-PbD].

7. IANA Considerations

This document makes the following requests of IANA.

7.1. Well-Known URI Registration

This specification registers the well-known URI defined in Section 2, as required by Section 5.1 of [RFC5785].

7.1.1. Registry Contents

- o URI suffix: "uma2-configuration"
- o Change controller: Kantara Initiative User-Managed Access Work Group – staff@kantarainitiative.org

- o Specification document: Section 2 in this document

7.2. OAuth 2.0 Authorization Server Metadata Registry

This specification registers OAuth 2.0 authorization server metadata defined in Section 2, as required by Section 7.1 of [OAuthMeta].

7.2.1. Registry Contents

- o Metadata name: "claims_interaction_endpoint"
- o Metadata description: endpoint metadata
- o Change controller: Kantara Initiative User-Managed Access Work Group – staff@kantarainitiative.org
- o Specification document: Section 2 in this document
- o Metadata name: "uma_profiles_supported"
- o Metadata description: profile/extension feature metadata
- o Change controller: Kantara Initiative User-Managed Access Work Group – staff@kantarainitiative.org
- o Specification document: Section 2 in this document

7.3. OAuth 2.0 Dynamic Client Registration Metadata Registry

This specification registers OAuth 2.0 dynamic client registration metadata defined in Section 2, as required by Section 4.1 of [RFC7591].

7.3.1. Registry Contents

- o Metadata name: "claims_redirect_uris"
- o Metadata description: claims redirection endpoints
- o Change controller: Kantara Initiative User-Managed Access Work Group – staff@kantarainitiative.org
- o Specification document: Section 2 in this document

7.4. OAuth 2.0 Extension Grant Parameters Registration

This specification registers the parameters defined in Section 3.3.1, as required by Section 11.2 of [RFC6749].

7.4.1. Registry Contents

- o Parameter name: "claim_token"
- o Parameter usage location: client request, token endpoint
- o Change controller: Kantara Initiative User-Managed Access Work Group – staff@kantarainitiative.org
- o Specification document: Section 3.3.1 in this document
- o Parameter name: "pct"
- o Parameter usage location: client request, token endpoint
- o Change controller: Kantara Initiative User-Managed Access Work Group – staff@kantarainitiative.org
- o Specification document: Section 3.3.1 in this document
- o Parameter name: "pct"
- o Parameter usage location: authorization server response, token endpoint
- o Change controller: Kantara Initiative User-Managed Access Work Group – staff@kantarainitiative.org
- o Specification document: Section 3.3.5 in this document
- o Parameter name: "rpt"
- o Parameter usage location: client request, token endpoint
- o Change controller: Kantara Initiative User-Managed Access Work Group – staff@kantarainitiative.org
- o Specification document: Section 3.3.1 in this document
- o Parameter name: "ticket"
- o Parameter usage location: client request, token endpoint

- o Change controller: Kantara Initiative User-Managed Access Work Group – staff@kantarainitiative.org
- o Specification document: Section 3.3.1 in this document
- o Parameter name: "upgraded"
- o Parameter usage location: authorization server response, token endpoint
- o Change controller: Kantara Initiative User-Managed Access Work Group – staff@kantarainitiative.org
- o Specification document: Section 3.3.5 in this document

7.5. OAuth 2.0 Extensions Error Registration

This specification registers the errors defined in Section 3.3.6, as required by Section 11.4 of [RFC6749].

7.5.1. Registry Contents

- o Error name: "need_info" (and its subsidiary parameters)
- o Change controller: Kantara Initiative User-Managed Access Work Group – staff@kantarainitiative.org
- o Specification document: Section 3.3.6 in this document
- o Error usage location: authorization server response, token endpoint
- o Error name: "request_denied"
- o Change controller: Kantara Initiative User-Managed Access Work Group – staff@kantarainitiative.org
- o Specification document: Section 3.3.6 in this document
- o Error usage location: authorization server response, token endpoint
- o Error name: "request_submitted" (and its subsidiary parameters)
- o Change controller: Kantara Initiative User-Managed Access Work Group – staff@kantarainitiative.org
- o Specification document: Section 3.3.6 in this document

- o Error usage location: authorization server response, token endpoint

7.6. OAuth Token Type Hints Registration

This specification registers the errors defined in Section 3.7, as required by Section 4.1.2 of [RFC7009].

7.6.1. Registry Contents

- o Hint value: "pct"
- o Change controller: Kantara Initiative User-Managed Access Work Group – staff@kantarainitiative.org
- o Specification document: Section 3.7 in this document

8. Acknowledgments

The following people made significant text contributions to the specification:

- o Paul C. Bryan, ForgeRock US, Inc. (former editor)
- o Domenico Catalano, Oracle (former author)
- o Mark Dobrinic, Cozmanova
- o George Fletcher, AOL
- o Thomas Hardjono, MIT (former editor)
- o Andrew Hindle, Hindle Consulting Limited
- o Lukasz Moren, Cloud Identity Ltd
- o James Phillpotts, ForgeRock
- o Christian Scholz, COMlounge GmbH (former editor)
- o Mike Schwartz, Gluu
- o Cigdem Sengul, Nominet UK
- o Jacek Szpot, Newcastle University

Additional contributors to this specification include the Kantara UMA Work Group participants, a list of whom can be found at [UMAnitarians].

9. References

9.1. Normative References

- [BCP195] Sheffer, Y., "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", May 2015, <<https://tools.ietf.org/html/bcp195>>.
- [OAuthMeta] Jones, M., "OAuth 2.0 Authorization Server Metadata", November 2017, <<https://tools.ietf.org/html/draft-ietf-oauth-discovery-08>>.
- [OIDCCore] Sakimura, N., "OpenID Connect Core 1.0 incorporating errata set 1", November 2014, <http://openid.net/specs/openid-connect-core-1_0.html>.
- [OIDCDynClientReg] Sakimura, N., "OpenID Connect Dynamic Client Registration 1.0 incorporating errata set 1", November 2014, <http://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>>.
- [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>>.
- [RFC5785] Nottingham, M. and E. Hammer-Lahav, "Defining Well-Known Uniform Resource Identifiers (URIs)", RFC 5785, DOI 10.17487/RFC5785, April 2010, <<https://www.rfc-editor.org/info/rfc5785>>.
- [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>>.
- [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>>.
- [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>>.
- [RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, <<https://www.rfc-editor.org/info/rfc7159>>.
- [RFC7591] Richer, J., Ed., Jones, M., Bradley, J., Machulak, M., and P. Hunt, "OAuth 2.0 Dynamic Client Registration Protocol", RFC 7591, DOI 10.17487/RFC7591, July 2015, <<https://www.rfc-editor.org/info/rfc7591>>.
- [UMAFedAuthz]
Maler, E., "Federated Authorization for User-Managed Access (UMA) 2.0", January 2019, <<https://docs.kantarainitiative.org/uma/rec-oauth-uma-federated-authz-2.0.html>>.

9.2. Informative References

- [UMA-legal]
Maler, E., "UMA Legal", 2017, <<http://kantarainitiative.org/confluence/display/uma/UMA+Legal>>.
- [UMA-PbD] Maler, E., "Privacy by Design Implications of UMA", 2018, <<https://kantarainitiative.org/confluence/display/uma/Privacy+by+Design+Implications+of+UMA>>.
- [UMAnitarians]
Maler, E., "UMA Participant Roster", 2017, <<https://kantarainitiative.org/confluence/display/uma/Participant+Roster>>.

Authors' Addresses

Eve Maler (editor)
ForgeRock

Email: eve.maler@forgerock.com

Maciej Machulak
HSBC

Email: maciej.p.machulak@hsbc.com

Justin Richer
Bespoke Engineering

Email: justin@bspk.io

Thomas Hardjono
MIT

Email: hardjono@mit.edu

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 11, 2021

R. Shekh-Yusef
Auth0
March 10, 2021

Multi-Subject JSON Web Token (JWT)
draft-yusef-oauth-nested-jwt-04

Abstract

This specification defines a mechanism for including multiple subjects in a JWT. A primary subject in an enclosing JWT with its own claims, and a related subject in a nested JWT with its own claims.

Status of This Memo

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

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

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

This Internet-Draft will expire on September 11, 2021.

Copyright Notice

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

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

This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November 10, 2008. The person(s) controlling the copyright in some of this material may not have granted the IETF Trust the right to allow modifications of such material outside the IETF Standards Process. Without obtaining an adequate license from the person(s) controlling the copyright in such materials, this document may not be modified outside the IETF Standards Process, and derivative works of it may not be created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.

Table of Contents

1. Introduction	2
1.1. Terminology	3
2. Use Cases	3
2.1. Primary Subject with Secondary Authority Subject	3
2.2. Multiple Primary Subjects	3
2.3. Delegation of Authority	3
2.4. Replaced Primary Subjects	4
2.4.1. STIR	4
2.4.2. Network Service Mesh (NSM)	4
3. JWT Content	4
4. Example	5
5. Security Considerations	5
6. IANA Considerations	5
7. Acknowledgments	5
8. References	5
8.1. Normative References	5
8.2. Informative References	6
Author's Address	6

1. Introduction

JSON Web Token (JWT) [RFC7519] is a mechanism that is used to transfer claims between two parties across security domains. Nested JWT is a JWT in which the payload is another JWT. The current specification does not define a means by which the enclosing JWT could have its own Claims Set, only the enclosed JWT would have claims.

There are a number of use cases where there is a need to represent multiple related subjects in one JWT; a primary subject and a related secondary subject.

This specification defines a mechanism for including multiple subjects in a JWT. A primary subject in an enclosing JWT with its

own claims, and a related subject in a nested JWT with its own claims.

1.1. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC8174].

2. Use Cases

The following are few categories of use cases that might benefit from such a concept:

2.1. Primary Subject with Secondary Authority Subject

A primary subject with a related secondary subject that has authority over the primary subject, e.g. Child/Parent, Pet/Owner.

The secondary user (e.g., parent) logs in to an application (e.g., pharmacy application), gets redirected to the authorization server, authenticates, and asks for permission to access resources (e.g., medication) for the primary subject (e.g., child). The authorization server then issues a JWT with the primary subject in the enclosing JWT and the secondary subject in the nested JWT.

In this case, both JWTs are issued by the same issuer.

2.2. Multiple Primary Subjects

Two or more primary related subjects e.g. a married couple. The authorization server is setup to provide one of the subjects with permissions to access the other related subject resources.

One user (e.g., wife) logs in to a application (e.g., pharmacy application), gets redirected to the authorization server, authenticates, and asks for permission to access resources (e.g., medication) for the other primary subject (e.g., husband). The authorization server then issues a JWT with the primary subject in the enclosing JWT and the other primary subject in the nested JWT.

In this case, both JWTs are issued by the same issuer.

2.3. Delegation of Authority

A primary subject delegates authority over a resource to a secondary subject who acts on behalf of the primary subject, as defined in [RFC8693].

In this case, both JWTs are issued by the same issuer.

2.4. Replaced Primary Subjects

A primary subject is replaced with a new primary subject, and the original primary subject included in the new issued JWT as a nested JWT.

2.4.1. STIR

[RFC8225] defines a PASSporT, which is a JWT, that is used to verify the identity of a caller in an incoming call.

The PASSporT Extension for Diverted Calls draft [STIR] uses a nested PASSporT to deliver the details of an incoming call that get redirected. An authentication service acting for a retargeting entity generates new PASSporT and embeds the original PASSporT inside the new one. When the new target receives the nested PASSporT it will be able to validate the enclosing PASSporT and use the details of the enclosed PASSporT to identify the original target.

In this case, the original JWT is issued by the calling service, and the new enclosing JWT is issued by the retargeting service.

2.4.2. Network Service Mesh (NSM)

Network Service Mesh [NSM] is a mechanism that maps the concept of a service mesh in Kubernetes to L2/L3 payloads.

NSM GRPS messages may pass through multiple intermediaries, each of which may transform the message. Each intermediary is expected to create its own JWT token, and include a claim that contains the JWT it received with the message it has transformed.

In this case, the original JWT is issued by the entity sending the initial message, and the new enclosing JWT is issued by the intermediate entity.

3. JWT Content

The payload of the enclosing JWT is JSON object that contains the Claims Set of the primary subject, and one new claim that is used to hold the enclosed JWT and its relation to the primary subject.

This document defines a new claim, "rsub" (Related Subject) Claim, that is used to contain the enclosed JWT and its relation to the primary subject.

4. Example

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

{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022,
  "rsub": {
    "rel" : urn:ietf:params:oauth:subject-type:authority |
            urn:ietf:params:oauth:subject-type:primary |
            urn:ietf:params:oauth:subject-type:actor |
            urn:ietf:params:oauth:subject-type:original
    "jwt" : "<jwt>"
  }
}
```

5. Security Considerations

TODO

6. IANA Considerations

TODO

7. Acknowledgments

TODO

8. References

8.1. Normative References

- [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>>.
- [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>>.

8.2. Informative References

- [RFC8225] Wendt, C. and J. Peterson, "PASSporT: Personal Assertion Token", RFC 8225, DOI 10.17487/RFC8225, February 2018, <<https://www.rfc-editor.org/info/rfc8225>>.
- [RFC8693] Jpnes, M., Nadalin, A., Campbell, B., Bradley, J., and C. Mortimore, "OAuth 2.0 Token Exchange", October 2018.
- [STIR] Peterson, J., "PASSporT Extension for Diverted Calls", October 2018.

Author's Address

Rifaat Shekh-Yusef
Auth0
Ottawa, Ontario, Canada

Email: rifaat.s.ietf@gmail.com