Web Authorization Protocol
Internet-Draft
Intended status: Standards Track
Expires: 23 September 2022

V. Bertocci
Auth0/Okta
B. Campbell
Ping Identity

22 March 2022

OAuth 2.0 Step-up Authentication Challenge Protocol
draft-bertocci-oauth-step-up-authn-challenge-01

Abstract

It is not uncommon for resource servers to require different authentication strengths or freshness according to the characteristics of a request. This document introduces a mechanism for a resource server to signal to a client that the authentication event associated with the access token of the current request doesn't meet its authentication requirements and specify how to meet them. This document also codifies a mechanism for a client to request that an authorization server achieve a specific authentication strength or freshness when processing an authorization request.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

In simple API authorization scenarios, an authorization server will statically determine what authentication technique to use to handle a given request on the basis of aspects such as the scopes requested, the resource, the identity of the client and other characteristics known at provisioning time. Although the approach is viable in many situations, it falls short in several important circumstances. Consider, for instance, an eCommerce API requiring different authentication strengths depending on whether the item being purchased exceeds a certain threshold, dynamically estimated by the API itself using a logic that is opaque to the authorization server. An API might also determine that a more recent user authentication is required based on its own risk evaluation of the API request.

This document extends the error codes collection defined by [RFC6750] with a new value, insufficient_user_authentication, which can be used by resource servers to signal to the client that the authentication event associated with the access token presented with the request doesn't meet the authentication requirements of the resource server. This document also introduces acr_values and max_age parameters for the WWW-Authenticate response header defined by [RFC6750], which the resource server can use to explicitly communicate to the client the required authentication strength or recentness.

The client can use that information to reach back to the authorization server with an authorization request specifying the authentication requirements indicated by protected resource, by including the acr_values or max_age parameter as defined in [OIDC].

Those extensions will make it possible to implement interoperable step up authentication with minimal work from resource servers, clients and authorization servers.

1.1. Conventions and Definitions

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

2. Protocol Overview

Following is an end-to-end sequence of a typical step-up authentication scenario implemented according to this specification. The scenario assumes that, before the sequence described below takes place, the client already obtained an access token for the protected resource.

```
 +----------+                                +--------------+
 |          |                                |              |
 |          |-----(1) resource request------>|              |
 |          |                                |              |
 |          |<-------(2) challenge ----------|   Resource   |
 |          |                                |    Server    |
 |          |                                |              |
 |          |-----(5) resource request ----->|              |
 |          |                                |              |
 |          |<---(6) protected resource -----|              |
 |          |                                +--------------+
 |  Client  |
 |          |
 |          |                                +---------------+
 |          |                                |               |
 |          |---(3) authorization request--->|               |
 |          |                                |               |
 |          |<-------------...-------------->| Authorization |
 |          |                                |    Server     |
 |          |<------ (4) access token -------|               |
 |          |                                |               |
 +----------+                                +---------------+
```

Figure 1: Abstract protocol flow

1. The client requests a protected resource, presenting an access token.

2. The resource server determines that the circumstances in which the presented access token was obtained offer insufficient authentication strength and/or freshness, hence it denies the request and returns a challenge describing (using a combination of acr_values and max_age) what authentication requirements must be met for the resource server to authorize a request.

3. The client directs the user agent to the authorization server with an authorization request that includes the acr_values and/or max_age indicated by the resource server in the previous step.

4. After whatever sequence required by the grant of choice plays out, which will include the necessary steps to authenticate the user in accordance with the acr_values and/or max_age values of the authorization request, the authorization server returns a new access token to the client. The access token contains or references information about the authentication event.

5. The client repeats the request from step 1, presenting the newly obtained access token.

6. The resource server finds that the user authentication performed during the acquisition of the new access token complies with its requirements, and returns the requested protected resource.

The validation operations mentioned in step 2 and 6 imply that the resource server has a way of evaluating the authentication level by which the access token was obtained. This document will describe how the resource server can perform that determination when the access token is a JWT Access token [RFC9068] or is validated via introspection [RFC7662]. Other methods of determining the authentication level by which the access token was obtained are possible, per agreement by the authorization server and the protected resource, but are beyond the scope of this specification.

3. Authentication Requirements Challenge

This specification introduces a new error code value for the error parameter of [RFC6750] or authentication schemes, such as [I-D.ietf-oauth-dpop], which use the error parameter:

insufficient_user_authentication The authentication event associated

with the access token presented with the request doesn't meet the authentication requirements of the protected resource.

Note: the logic through which the resource server determines that the current request doesn't meet the authentication requirements of the protected resource, and associated functionality (such as expressing, deploying and publishing such requirements) is out of scope for this document.

Furthermore, this specification defines additional WWW-Authenticate auth-param values to convey the authentication requirements back to the client.

acr_values A space-separated string indicating, in order of preference, the authentication context class reference values that the protected resource requires the authentication event associated with the access token.

max_age Indicates the allowable elapsed time in seconds since the last active authentication event associated with the access token.

Below you can find an example of WWW-Authenticate header using the insufficient_user_authentication error code value to inform the client that the access token presented isn't sufficient to gain access to the protected resource, and the acr_values parameter to let the client know that the expected authentication level corresponds to the authentication context class reference identified by myACR.

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer error="insufficient_user_authentication",
  error_description="A different authentication level is required",
  acr_values="myACR"
```

Figure 2

If the resource server determines that the request is also lacking the scopes required by the requested resource, it MAY include the scope attribute with the scope necessary to access the protected resource, as described in section 3.1 of [RFC6750].

4. Authorization Request

A client receiving an authorization error from the resource server carrying the error code insufficient_user_authentication MAY parse the WWW-Authenticate header for acr_values and max_age and use them, if present, in a request to the authorization server to obtain a new access token complying with the corresponding requirements. Both acr_values and max_age authorization request parameters are OPTIONAL

parameters defined in Section 3.1.2.1. of [OIDC]. This document does not introduce any changes in the authorization server behavior defined in [OIDC] for precessing those parameters, hence any authorization server implementing OpenID Connect will be able to participate in the flow described here with little or no changes. See Section Section 5 for more details.

The example request below indicates to the authorization server that the client would like the authentication to occur according to the authentication context class reference identified by myACR.

```
GET https://as.example.net/authorize?client_id=s6BhdRkqt3
&response_type=code&scope=purchase&acr_values=myACR
```

Figure 3

5. Authorization Response

Section 5.5.1.1 of [OIDC] establishes that an authorization server receiving a request containing the acr_values parameter MAY attempt to authenticate the user in a manner that satisfies the requested Authentication Context Class Reference, and include the corresponding value in the acr claim in the resulting ID Token. The same section also establishes that in case the desired authentication level cannot be met, the authorization server SHOULD include in the acr claim a value reflecting the authentication level of the current session (if any). The same section also states that if a request includes thee max_age parameter, the authorization server MUST include the auth_time claim in the issued ID Token. An authorization server complying with this specification will react to the presence of the acr_values and max_age parameters by including acr and auth_time in the access token (see Section 6 for details). Although [OIDC] leaves the authorization server free to decide how to handle the inclusion of acr in ID Token when requested via acr_values, when it comes to access tokens in this specification it is RECOMMENDED that the requested acr value is treated as required for successfully fulfilling the request. That is, the requested acr value is included in the access token if the authentication operation successfully met its requirements, or that the authorization request fails in all other cases, returning unmet_authentication_requirements as defined in [OIDCUAR]. The recommended behavior will help prevent clients getting stuck in a loop where the authorization server keeps returning tokens that the resource server already identified as not meeting its requirements hence known to be rejected as well.

6. Authentication Information Conveyed via Access Token

To evaluate whether an access token meets the protected resource's requirements, the resource servers needs a way of accessing information about the authentication event by which that access token was obtained. This specification provides guidance on how to convey that information in conjunction with two common access token validation methods: the one described in [RFC9068], where the access token is encoded in JWT format and verified via a set of validation rules, and the one described in [RFC7662], where the token is validated and decoded by sending it to an introspection endpoint. Authorization servers and resource servers MAY elect to use other encoding and validation methods, however those are out of scope for this document.

6.1. JWT Access Tokens

When access tokens are represented as JSON Web Tokens (JWT) [RFC7519], the auth_time and acr claims (per Section 2.2.1 of [RFC9068]) are used to convey the time and context of the user authentication event that the authentication server performed during the course of obtaining the access token. It is useful to bear in mind that the values of those two parameters are established at user authentication time and won't change in the event of access token renewals. See the aforementioned Section 2.2.1 of [RFC9068] for details. The following is a conceptual example showing the decoded content of such a JWT access token.

Header:

{"typ":"at+JWT","alg":"RS256","kid":"LTacESbw"}

Claims:

{
 "iss": "https://as.example.net",
 "sub": "someone@example.net",
 "aud": "https://rs.example.com",
 "exp": 1646343000,
 "iat": 1646340200,
 "jti" : "e1j3V_bKic8-LAEB_lccD0G",
 "client_id": "s6BhdRkqt3",
 "scope": "purchase",
 "auth_time": 1646340198,
 "acr": "myACR"
}

Figure 4

6.2. OAuth 2.0 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 metainformation about the token. The following two top-level introspection response members are defined to convey information about the user authentication event that the authentication server performed during the course of obtaining the access token.

acr Authentication Context Class Reference. String specifying an Authentication Context Class Reference value that identifies the Authentication Context Class that the user authentication performed satisfied.

auth_time Time when the user authentication occurred. A JSON numeric value representing the number of seconds from 1970-01-01T00:00:00Z UTC until the time of date/time of the authentication event.

The following example shows an introspection response with information about the user authentication event by which the access token was obtained.

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

{
  "active": true,
  "client_id": "s6BhdRkqt3",
  "scope": "purchase",
  "sub": "someone@example.net",
  "aud": "https://rs.example.com",
  "iss": "https://as.example.net",
  "exp": 1639528912,
  "iat": 1618354090,
  "auth_time": 1646340198,
  "acr": "myACR"
}
```

Figure 5

7. Security Considerations

[[TBD]]

Remember that oauth is not authN, you need a layer like OIDC to handle that part. This is not an encouragement to abuse oauth. This is about the authentication event of the user to the AS by which the access token was obtained.

8. IANA Considerations

[[TBD]]

The insufficient_user_authentication error code in the "OAuth Extensions Error" registry [IANA.OAuth.Params].

Section 6.2 for acr and auth_time as top-level members of the introspection response in the "OAuth Token Introspection Response" registry [IANA.OAuth.Params].

The acr_values and max_age WWW-Authenticate auth-params are "new" but doesn't seem like any registration is needed or possible.

9. Normative References

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

[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>.

[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>.

[RFC9068] Bertocci, V., "JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens", RFC 9068, DOI 10.17487/RFC9068, October 2021, <https://www.rfc-editor.org/info/rfc9068>.

10. Informative References

[I-D.abr-twitter-reply]
Roach, A., "A reply to a specific tweet", Work in Progress, Internet-Draft, draft-abr-twitter-reply-00, 7 September 2018, <https://datatracker.ietf.org/doc/html/draft-abr-twitter-reply-00>.

[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-06, 1 March 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-oauth-dpop-06>.

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

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

[OIDCUAR] Lodderstedt, T., "OpenID Connect Core Error Code unmet_authentication_requirements", 8 May 2019, <https://openid.net/specs/openid-connect-unmet-authentication-requirements-1_0.html>.

[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>.

Appendix A. Acknowledgements

I wanted to thank the Academy, the viewers at home, the shampoo manufacturers, etc..

Initially (kinda) discussed at the OAuth Security Workshop 2021

A number of others already but haven’t kept track...

Appendix B. Document History

[[To be removed from the final specification]]

-01

* Fixed example

* Clarified/noted that scope can also be in the WWW-Authenticate/401

-00

* Initial Individual Draft (with all the authority thereby bestowed [I-D.abr-twitter-reply]).

Authors' Addresses

Vittorio Bertocci
Auth0/Okta
Email: vittorio@auth0.com

Brian Campbell
Ping Identity
Email: bcampbell@pingidentity.com

Web Authorization Protocol
Internet-Draft
Intended status: Standards Track
Expires: 15 October 2023

D. Fett
Authlete
B. Campbell
Ping Identity
J. Bradley
Yubico
T. Lodderstedt
yes.com
M. Jones
independent
D. Waite
Ping Identity
13 April 2023

OAuth 2.0 Demonstrating Proof-of-Possession at the Application Layer (DPoP)
draft-ietf-oauth-dpop-16

Abstract

This document describes a mechanism for sender-constraining OAuth 2.0 tokens via a proof-of-possession mechanism on the application level. This mechanism allows for the detection of replay attacks with access and refresh tokens.

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 15 October 2023.

Table of Contents

1. Introduction

DPoP (for Demonstrating Proof-of-Possession at the Application Layer) is an application-level mechanism for sender-constraining OAuth [RFC6749] access and refresh tokens. It enables a client to prove the possession of a public/private key pair by including a DPoP header in an HTTP request. The value of the header is a JSON Web Token (JWT) [RFC7519] that enables the authorization server to bind issued tokens to the public part of a client's key pair. Recipients of such tokens are then able to verify the binding of the token to the key pair that the client has demonstrated that it holds via the DPoP header, thereby providing some assurance that the client presenting the token also possesses the private key. In other words, the legitimate presenter of the token is constrained to be the sender that holds and can prove possession of the private part of the key pair.

The mechanism specified herein can be used in cases where other methods of sender-constraining tokens that utilize elements of the underlying secure transport layer, such as [RFC8705] or [I-D.ietf-oauth-token-binding], are not available or desirable. For example, due to a sub-par user experience of TLS client authentication in user agents and a lack of support for HTTP token binding, neither mechanism can be used if an OAuth client is an application that is dynamically downloaded and executed in a web browser (sometimes referred to as a "single-page application"). Applications installed and run directly on a user's device are another example well positioned to benefit from DPoP-bound tokens to guard against misuse of tokens by a compromised or malicious resource. Such applications often have dedicated protected storage for cryptographic keys.

DPoP can be used to sender-constrain access tokens regardless of the client authentication method employed, but DPoP itself is not used for client authentication. DPoP can also be used to sender-constrain refresh tokens issued to public clients (those without authentication credentials associated with the client_id).

1.1. 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 Augmented Backus-Naur Form (ABNF) notation of [RFC5234].

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", "client", "public client", and "confidential client" defined by The OAuth 2.0 Authorization Framework [RFC6749].

The terms "request", "response", "header field", and "target URI" are imported from [RFC9110].

The terms "JOSE" and "JOSE header" are imported from [RFC7515].

This document contains non-normative examples of partial and complete HTTP messages. Some examples use a single trailing backslash to indicate line wrapping for long values, as per [RFC8792]. The character and leading spaces on wrapped lines are not part of the value.

2. Objectives

The primary aim of DPoP is to prevent unauthorized or illegitimate parties from using leaked or stolen access tokens, by binding a token to a public key upon issuance and requiring that the client proves possession of the corresponding private key when using the token. This constrains the legitimate sender of the token to only the party with access to the private key and gives the server receiving the token added assurances that the sender is legitimately authorized to use it.

Access tokens that are sender-constrained via DPoP thus stand in contrast to the typical bearer token, which can be used by any party in possession of such a token. Although protections generally exist to prevent unintended disclosure of bearer tokens, unforeseen vectors for leakage have occurred due to vulnerabilities and implementation issues in other layers in the protocol or software stack (CRIME [CRIME], BREACH [BREACH], Heartbleed [Heartbleed], and the Cloudflare parser bug [Cloudbleed] are some examples). There have also been numerous published token theft attacks on OAuth implementations themselves ([GitHub.Tokens] as just one high profile example). DPoP provides a general defense in depth against the impact of unanticipated token leakage. DPoP is not, however, a substitute for a secure transport and MUST always be used in conjunction with HTTPS.

The very nature of the typical OAuth protocol interaction necessitates that the client discloses the access token to the protected resources that it accesses. The attacker model in [I-D.ietf-oauth-security-topics] describes cases where a protected resource might be counterfeit, malicious or compromised and plays received tokens against other protected resources to gain unauthorized access. Audience restricted access tokens (e.g., using the JWT [RFC7519] aud claim) can prevent such misuse, however, doing so in practice has proven to be prohibitively cumbersome for many deployments (even despite extensions such as [RFC8707]). Sender-constraining access tokens is a more robust and straightforward mechanism to prevent such token replay at a different endpoint and DPoP is an accessible application layer means of doing so.

Due to the potential for cross-site scripting (XSS), browser-based OAuth clients bring to bear added considerations with respect to protecting tokens. The most straightforward XSS-based attack is for

an attacker to exfiltrate a token and use it themselves completely independent of the legitimate client. A stolen access token is used for protected resource access and a stolen refresh token for obtaining new access tokens. If the private key is non-extractable (as is possible with [W3C.WebCryptoAPI]), DPoP renders exfiltrated tokens alone unusable.

XSS vulnerabilities also allow an attacker to execute code in the context of the browser-based client application and maliciously use a token indirectly through the client. That execution context has access to utilize the signing key and thus can produce DPoP proofs to use in conjunction with the token. At this application layer there is most likely no feasible defense against this threat except generally preventing XSS, therefore it is considered out of scope for DPoP.

Malicious XSS code executed in the context of the browser-based client application is also in a position to create DPoP proofs with timestamp values in the future and exfiltrate them in conjunction with a token. These stolen artifacts can later be used independent of the client application to access protected resources. To prevent this, servers can optionally require clients to include a server-chosen value into the proof that cannot be predicted by an attacker (nonce). In the absence of the optional nonce, the impact of pre-computed DPoP proofs is limited somewhat by the proof being bound to an access token on protected resource access. Because a proof covering an access token that does not yet exist cannot feasibly be created, access tokens obtained with an exfiltrated refresh token and pre-computed proofs will be unusable.

Additional security considerations are discussed in Section 11.

3. Concept

The main data structure introduced by this specification is a DPoP proof JWT, described in detail below, which is sent as a header in an HTTP request. A client uses a DPoP proof JWT to prove the possession of a private key corresponding to a certain public key.

Roughly speaking, a DPoP proof is a signature over some data of the HTTP request to which it is attached, a timestamp, a unique identifier, an optional server-provided nonce, and a hash of the associated access token when an access token is present within the request.

```
+--------+                                           +---------------+
|        |--(A)-- Token Request -------------------->|               |
| Client |        (DPoP Proof)                       | Authorization |
|        |                                           |     Server    |
|        |<-(B)-- DPoP-bound Access Token ----------|               |
|        |        (token_type=DPoP)                  +---------------+
|        |
|        |
|        |                                           +---------------+
|        |--(C)-- DPoP-bound Access Token --------->|               |
|        |        (DPoP Proof)                       |    Resource   |
|        |                                           |     Server    |
|        |<-(D)-- Protected Resource ---------------|               |
|        |                                           +---------------+
+--------+
```

Figure 1: Basic DPoP Flow

The basic steps of an OAuth flow with DPoP (without the optional nonce) are shown in Figure 1:

* (A) In the Token Request, the client sends an authorization grant (e.g., an authorization code, refresh token, etc.) to the authorization server in order to obtain an access token (and potentially a refresh token). The client attaches a DPoP proof to the request in an HTTP header.
* (B) The authorization server binds (sender-constrains) the access token to the public key claimed by the client in the DPoP proof; that is, the access token cannot be used without proving possession of the respective private key. If a refresh token is issued to a public client, it too is bound to the public key of the DPoP proof.
* (C) To use the access token, the client has to prove possession of the private key by, again, adding a header to the request that carries a DPoP proof for that request. The resource server needs to receive information about the public key to which the access token is bound. This information may be encoded directly into the access token (for JWT structured access tokens) or provided via token introspection endpoint (not shown). The resource server verifies that the public key to which the access token is bound matches the public key of the DPoP proof. It also verifies that the access token hash in the DPoP proof matches the access token presented in the request.
* (D) The resource server refuses to serve the request if the signature check fails or the data in the DPoP proof is wrong, e.g., the target URI does not match the URI claim in the DPoP proof JWT. The access token itself, of course, must also be valid in all other respects.

The DPoP mechanism presented herein is not a client authentication method. In fact, a primary use case of DPoP is for public clients (e.g., single page applications and applications on a user's device) that do not use client authentication. Nonetheless, DPoP is designed such that it is compatible with private_key_jwt and all other client authentication methods.

DPoP does not directly ensure message integrity but relies on the TLS layer for that purpose. See Section 11 for details.

4. DPoP Proof JWTs

DPoP introduces the concept of a DPoP proof, which is a JWT created by the client and sent with an HTTP request using the DPoP header field. Each HTTP request requires a unique DPoP proof.

A valid DPoP proof demonstrates to the server that the client holds the private key that was used to sign the DPoP proof JWT. This enables authorization servers to bind issued tokens to the corresponding public key (as described in Section 5) and for resource servers to verify the key-binding of tokens that it receives (see Section 7.1), which prevents said tokens from being used by any entity that does not have access to the private key.

The DPoP proof demonstrates possession of a key and, by itself, is not an authentication or access control mechanism. When presented in conjunction with a key-bound access token as described in Section 7.1, the DPoP proof provides additional assurance about the legitimacy of the client to present the access token. However, a valid DPoP proof JWT is not sufficient alone to make access control decisions.

4.1. The DPoP HTTP Header

A DPoP proof is included in an HTTP request using the following request header field.

DPoP A JWT that adheres to the structure and syntax of Section 4.2.

Figure 2 shows an example DPoP HTTP header field (with '\' line wrapping per RFC 8792).

```
DPoP: eyJ0eXAiOiJkcG9wK2p3dCIsImFsZyI6IkVTMjU2IiwiandrIjp7Imt0eSI6Ik\
 VDIiwieCI6Imw4dEZyaHgtMzR0VjNoUklDUkRZOXpDa0RscEJoRjQyVVFVZldWQVdCR\
 nMiLCJ5IjoiOVZFNGpmX09rX282NHpiVFRsY3VOSmFqSG10NnY5VERWclUwQ2R2R1JE\
 QSIsImNydiI6IlAtMjU2In19.eyJqdGkiOiItQndDM0VTYzZhY2MybFRjIiwiaHRtIj\
 oiUE9TVCIsImh0dSI6Imh0dHBzOi8vc2VydmVyLmV4YW1wbGUuY29tL3Rva2VuIiwia\
 WF0IjoxNTYyMjYyNjE2fQ.2-GxA6T8lP4vfrg8v-FdWP0A0zdrj8igiMLvqRMUvwnQg\
 4PtFLbdLXiOSsX0x7NVY-FNyJK70nfbV37xRZT3Lg
```


Figure 2: Example DPoP header

Note that per [RFC9110] header field names are case-insensitive; so DPoP, DPOP, dpop, etc., are all valid and equivalent header field names. Case is significant in the header field value, however.

The DPoP HTTP header field value uses the token68 syntax defined in Section 11.2 of [RFC9110] (repeated below in Figure 3 for ease of reference).

```
DPoP       = token68
token68    = 1*( ALPHA / DIGIT /
                 "-" / "." / "_" / "~" / "+" / "/" ) *"="
```

Figure 3: DPoP header field ABNF

4.2. DPoP Proof JWT Syntax

A DPoP proof is a JWT ([RFC7519]) that is signed (using JSON Web Signature (JWS) [RFC7515]) with a private key chosen by the client (see below). The JOSE header of a DPoP JWT MUST contain at least the following parameters:

* typ: with value dpop+jwt, which explicitly types the DPoP proof JWT as recommended in Section 3.11 of [RFC8725].
* alg: an identifier for a JWS asymmetric digital signature algorithm from [IANA.JOSE.ALGS]. MUST NOT be none or an identifier for a symmetric algorithm (MAC).
* jwk: representing the public key chosen by the client, in JSON Web Key (JWK) [RFC7517] format, as defined in Section 4.1.3 of [RFC7515]. MUST NOT contain a private key.

The payload of a DPoP proof MUST contain at least the following claims:

* jti: Unique identifier for the DPoP proof JWT. The value MUST be assigned such that there is a negligible probability that the same value will be assigned to any other DPoP proof used in the same context during the time window of validity. Such uniqueness can be accomplished by encoding (base64url or any other suitable

encoding) at least 96 bits of pseudorandom data or by using a version 4 UUID string according to [RFC4122]. The jti can be used by the server for replay detection and prevention, see Section 11.1.

* htm: The value of the HTTP method (Section 9.1 of [RFC9110]) of the request to which the JWT is attached.
* htu: The HTTP target URI (Section 7.1 of [RFC9110]), without query and fragment parts, of the request to which the JWT is attached.
* iat: Creation timestamp of the JWT ([RFC7519], section 4.1.6]).

When the DPoP proof is used in conjunction with the presentation of an access token in protected resource access, see Section 7, the DPoP proof MUST also contain the following claim:

* ath: hash of the access token. The value MUST be the result of a base64url encoding (as defined in Section 2 of [RFC7515]) the SHA-256 [SHS] hash of the ASCII encoding of the associated access token's value.

When the authentication server or resource server provides a DPoP-Nonce HTTP header in a response (see Section 8, Section 9), the DPoP proof MUST also contain the following claim:

* nonce: A recent nonce provided via the DPoP-Nonce HTTP header.

A DPoP proof MAY contain other JOSE header parameters or claims as defined by extension, profile, or deployment specific requirements.

Figure 4 is a conceptual example showing the decoded content of the DPoP proof in Figure 2. The JSON of the JWT header and payload are shown, but the signature part is omitted. As usual, line breaks and extra spaces are included for formatting and readability.

```
{
  "typ":"dpop+jwt",
  "alg":"ES256",
  "jwk": {
    "kty":"EC",
    "x":"l8tFrhx-34tV3hRICRDY9zCkDlpBhF42UQUfWVAWBFs",
    "y":"9VE4jf_Ok_o64zbTTlcuNJajHmt6v9TDVrU0CdvGRDA",
    "crv":"P-256"
  }
}
.
{
  "jti":"-BwC3ESc6acc2lTc",
  "htm":"POST",
  "htu":"https://server.example.com/token",
  "iat":1562262616
}
```


Figure 4: Example JWT content of a DPoP proof

Of the HTTP request, only the HTTP method and URI are included in the DPoP JWT, and therefore only these two message parts are covered by the DPoP proof. The idea is sign just enough of the HTTP data to provide reasonable proof-of-possession with respect to the HTTP request. This design approach of using only a minimal subset of the HTTP header data is to avoid the substantial difficulties inherent in attempting to normalize HTTP messages. Nonetheless, DPoP proofs can be extended to contain other information of the HTTP request (see also Section 11.7).

4.3. Checking DPoP Proofs

To validate a DPoP proof, the receiving server MUST ensure that

1. there is not more than one DPoP HTTP request header field,
2. the DPoP HTTP request header field value is a single well-formed JWT,
3. all required claims per Section 4.2 are contained in the JWT,
4. the typ JOSE header parameter has the value dpop+jwt,
5. the alg JOSE header parameter indicates a registered asymmetric digital signature algorithm [IANA.JOSE.ALGS], is not none, is supported by the application, and is acceptable per local policy,
6. the JWT signature verifies with the public key contained in the jwk JOSE header parameter,
7. the jwk JOSE header parameter does not contain a private key,
8. the htm claim matches the HTTP method of the current request,

9. the htu claim matches the HTTP URI value for the HTTP request in which the JWT was received, ignoring any query and fragment parts,
10. if the server provided a nonce value to the client, the nonce claim matches the server-provided nonce value,
11. the creation time of the JWT, as determined by either the iat claim or a server managed timestamp via the nonce claim, is within an acceptable window (see Section 11.1),
12. if presented to a protected resource in conjunction with an access token,
    * ensure that the value of the ath claim equals the hash of that access token,
    * confirm that the public key to which the access token is bound matches the public key from the DPoP proof.

To reduce the likelihood of false negatives, servers SHOULD employ Syntax-Based Normalization (Section 6.2.2 of [RFC3986]) and Scheme-Based Normalization (Section 6.2.3 of [RFC3986]) before comparing the htu claim.

These checks may be performed in any order.

5. DPoP Access Token Request

To request an access token that is bound to a public key using DPoP, the client MUST provide a valid DPoP proof JWT in a DPoP header when making an access token request to the authorization server's token endpoint. This is applicable for all access token requests regardless of grant type (including, for example, the common authorization_code and refresh_token grant types but also extension grants such as the JWT authorization grant [RFC7523]). The HTTP request shown in Figure 5 illustrates such an access token request using an authorization code grant with a DPoP proof JWT in the DPoP header (with '\' line wrapping per RFC 8792).

```
POST /token HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded
DPoP: eyJ0eXAiOiJkcG9wK2p3dCIsImFsZyI6IkVTMjU2IiwiandrIjp7Imt0eSI6Ik\
 VDIiwieCI6Imw4dEZyaHgtMzR0VjNoUklDUkRZOXpDa0RscEJoRjQyVVFVZldWQVdCR\
 nMiLCJ5IjoiOVZFNGpmX09rX282NHpiVFRsY3VOSmFqSG10NnY5VERWclUwQ2R2R1JE\
 QSIsImNydiI6IlAtMjU2In19.eyJqdGkiOiItQndDM0VTYzZhY2MybFRjIiwiaHRtIj\
 oiUE9TVCIsImh0dSI6Imh0dHBzOi8vc2VydmVyLmV4YW1wbGUuY29tL3Rva2VuIiwia\
 WF0IjoxNTYyMjYyNjE2fQ.2-GxA6T8lP4vfrg8v-FdWP0A0zdrj8igiMLvqRMUvwnQg\
 4PtFLbdLXiOSsX0x7NVY-FNyJK70nfbV37xRZT3Lg

grant_type=authorization_code\
&client_id=s6BhdRkqt\
&code=SplxlOBeZQQYbYS6WxSbIA
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb\
&code_verifier=bEaL42izcC-o-xBk0K2vuJ6U-y1p9r_wW2dFWIWgjz-
```

Figure 5: Token Request for a DPoP sender-constrained token using an authorization code

The DPoP HTTP header field MUST contain a valid DPoP proof JWT. If the DPoP proof is invalid, the authorization server issues an error response per Section 5.2 of [RFC6749] with invalid_dpop_proof as the value of the error parameter.

To sender-constrain the access token, after checking the validity of the DPoP proof, the authorization server associates the issued access token with the public key from the DPoP proof, which can be accomplished as described in Section 6. A token_type of DPoP MUST be included in the access token response to signal to the client that the access token was bound to its DPoP key and can be used as described in Section 7.1. The example response shown in Figure 6 illustrates such a response.

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

{
 "access_token": "Kz~8mXK1EalYznwH-LC-1fBAo.4Ljp~zsPE_NeO.gxU",
 "token_type": "DPoP",
 "expires_in": 2677,
 "refresh_token": "Q..Zkm29lexi8VnWg2zPW1x-tgGad0Ibc3s3EwM_Ni4-g"
}
```

Figure 6: Access Token Response

The example response in Figure 6 includes a refresh token which the client can use to obtain a new access token when the previous one expires. Refreshing an access token is a token request using the refresh_token grant type made to the authorization server's token endpoint. As with all access token requests, the client makes it a DPoP request by including a DPoP proof, as shown in the Figure 7 example (with '\' line wrapping per RFC 8792).

```
POST /token HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded
DPoP: eyJ0eXAiOiJkcG9wK2p3dCIsImFsZyI6IkVTMjU2IiwiandrIjp7Imt0eSI6Ik\
 VDIiwieCI6Imw4dEZyaHgtMzR0VjNoUklDUkRZOXpDa0RscEJoRjQyVVFVZldWQVdCR\
 nMiLCJ5IjoiOVZFNGpmX09rX282NHpiVFRsY3VOSmFqSG10NnY5VERWclUwQ2R2R1JE\
 QSIsImNydiI6IlAtMjU2In19.eyJqdGkiOiItQndDM0VTYzZhY2MybFRjIiwiaHRtIj\
 oiUE9TVCIsImh0dSI6Imh0dHBzOi8vc2VydmVyLmV4YW1wbGUuY29tL3Rva2VuIiwia\
 WF0IjoxNTYyMjY1Mjk2fQ.pAqut2IRDm_De6PR93SYmGBPXpwrAk90e8cP2hjiaG5Qs\
 GSuKDYW7_X620BxqhvYC8ynrrvZLTk41mSRroapUA

grant_type=refresh_token\
&client_id=s6BhdRkqt\
&refresh_token=Q..Zkm29lexi8VnWg2zPW1x-tgGad0Ibc3s3EwM_Ni4-g
```

Figure 7: Token Request for a DPoP-bound Token using a Refresh Token

When an authorization server supporting DPoP issues a refresh token to a public client that presents a valid DPoP proof at the token endpoint, the refresh token MUST be bound to the respective public key. The binding MUST be validated when the refresh token is later presented to get new access tokens. As a result, such a client MUST present a DPoP proof for the same key that was used to obtain the refresh token each time that refresh token is used to obtain a new access token. The implementation details of the binding of the refresh token are at the discretion of the authorization server. Since the authorization server both produces and validates its refresh tokens, there is no interoperability consideration in the specific details of the binding.

An authorization server MAY elect to issue access tokens which are not DPoP bound, which is signaled to the client with a value of Bearer in the token_type parameter of the access token response per [RFC6750]. For a public client that is also issued a refresh token, this has the effect of DPoP-binding the refresh token alone, which can improve the security posture even when protected resources are not updated to support DPoP.

If the access token response contains a different token_type value than DPoP, the access token protection provided by DPoP is not given. The client MUST discard the response in this case, if this protection is deemed important for the security of the application; otherwise, it may continue as in a regular OAuth interaction.

Refresh tokens issued to confidential clients (those having established authentication credentials with the authorization server) are not bound to the DPoP proof public key because they are already sender-constrained with a different existing mechanism. The OAuth 2.0 Authorization Framework [RFC6749] already requires that an authorization server bind refresh tokens to the client to which they were issued and that confidential clients authenticate to the authorization server when presenting a refresh token. As a result, such refresh tokens are sender-constrained by way of the client identifier and the associated authentication requirement. This existing sender-constraining mechanism is more flexible (e.g., it allows credential rotation for the client without invalidating refresh tokens) than binding directly to a particular public key.

5.1. Authorization Server Metadata

This document introduces the following authorization server metadata [RFC8414] parameter to signal support for DPoP in general and the specific JWS alg values the authorization server supports for DPoP proof JWTs.

dpop_signing_alg_values_supported A JSON array containing a list of the JWS alg values (from the [IANA.JOSE.ALGS] registry) supported by the authorization server for DPoP proof JWTs.

5.2. Client Registration Metadata

The Dynamic Client Registration Protocol [RFC7591] defines an API for dynamically registering OAuth 2.0 client metadata with authorization servers. The metadata defined by [RFC7591], and registered extensions to it, also 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.

This document introduces the following client registration metadata [RFC7591] parameter to indicate that the client always uses DPoP when requesting tokens from the authorization server.

dpop_bound_access_tokens Boolean value specifying whether the client

always uses DPoP for token requests. If omitted, the default value is false.

If true, the authorization server MUST reject token requests from this client that do not contain the DPoP header.

6. Public Key Confirmation

Resource servers MUST be able to reliably identify whether an access token is DPoP-bound and ascertain sufficient information to verify the binding to the public key of the DPoP proof (see Section 7.1). Such a binding is accomplished by associating the public key with the token in a way that can be accessed by the protected resource, such as embedding the JWK hash in the issued access token directly, using the syntax described in Section 6.1, or through token introspection as described in Section 6.2. Other methods of associating a public key with an access token are possible, per agreement by the authorization server and the protected resource, but are beyond the scope of this specification.

Resource servers supporting DPoP MUST ensure that the public key from the DPoP proof matches the one bound to the access token.

6.1. JWK Thumbprint Confirmation Method

When access tokens are represented as JWTs [RFC7519], the public key information is represented using the jkt confirmation method member defined herein. To convey the hash of a public key in a JWT, this specification introduces the following JWT Confirmation Method [RFC7800] member for use under the cnf claim.

jkt JWK SHA-256 Thumbprint Confirmation Method. The value of the jkt member MUST be the base64url encoding (as defined in [RFC7515]) of the JWK SHA-256 Thumbprint (according to [RFC7638]) of the DPoP public key (in JWK format) to which the access token is bound.

The following example JWT in Figure 8 with decoded JWT payload shown in Figure 9 contains a cnf claim with the jkt JWK Thumbprint confirmation method member. The jkt value in these examples is the hash of the public key from the DPoP proofs in the examples in Section 5. (The example uses '\' line wrapping per RFC 8792.)

eyJhbGciOiJFUzI1NiIsImtpZCI6IkJlQUxrYiJ9.eyJzdWIiOiJzb21lb25lQGV4YW1\
wbGUuY29tIiwiaXNzIjoiaHR0cHM6Ly9zZXJ2ZXIuZXhhbXBsZS5jb20iLCJuYmYiOjE\
1NjIyNjI2MTEsImV4cCI6MTU2MjI2NjIxNiwiY25mIjp7ImprdCI6IjBaY09DT1JaTll\
5LURXcHFxMzBqWnlKR0hUTjBkMkhnbEJWM3VpZ3VBNEkifX0.3Tyo8VTcn6u_PboUmAO\
YUY1kfAavomW_YwYMkmRNizLJoQzWy2fCo79Zi5yObpIzjWb5xW4OGld7ESZrh0fsrA

Figure 8: JWT containing a JWK SHA-256 Thumbprint Confirmation


```
{
  "sub":"someone@example.com",
  "iss":"https://server.example.com",
  "nbf":1562262611,
  "exp":1562266216,
  "cnf":
  {
    "jkt":"0ZcOCORZNYy-DWpqq30jZyJGHTN0d2HglBV3uiguA4I"
  }
}
```


Figure 9: JWT Claims Set with a JWK SHA-256 Thumbprint Confirmation

6.2. JWK Thumbprint Confirmation Method in 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 metainformation about the token.

For a DPoP-bound access token, the hash of the public key to which the token is bound is conveyed to the protected resource as metainformation in a token introspection response. The hash is conveyed using the same cnf content with jkt member structure as the JWK Thumbprint confirmation method, described in Section 6.1, as a top-level member of the introspection response JSON. Note that the resource server does not send a DPoP proof with the introspection request and the authorization server does not validate an access token’s DPoP binding at the introspection endpoint. Rather the resource server uses the data of the introspection response to validate the access token binding itself locally.

If the token_type member is included in the introspection response, it MUST contain the value DPoP.

The example introspection request in Figure 10 and corresponding response in Figure 11 illustrate an introspection exchange for the example DPoP-bound access token that was issued in Figure 6.

```
POST /as/introspect.oauth2 HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded
Authorization: Basic cnM6cnM6TWt1LTZnX2xDektJZHo0ZnNON2tZY3lhK1Rp

token=Kz~8mXK1EalYznwH-LC-1fBAo.4Ljp~zsPE_NeO.gxU
```

Figure 10: Example Introspection Request

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

{
  "active": true,
  "sub": "someone@example.com",
  "iss": "https://server.example.com",
  "nbf": 1562262611,
  "exp": 1562266216,
  "cnf":
  {
    "jkt": "0ZcOCORZNYy-DWpqq30jZyJGHTN0d2HglBV3uiguA4I"
  }
}
```

Figure 11: Example Introspection Response for a DPoP-Bound Access Token

7. Protected Resource Access

Requests to DPoP protected resources MUST include both a DPoP proof as per Section 4 and the access token as described in Section 7.1. The DPoP proof MUST include the ath claim with a valid hash of the associated access token.

Binding the token value to the proof in this way prevents a proof to be used with multiple different access token values across different requests. For example, if a client holds tokens bound to two different resource owners, AT1 and AT2, and uses the same key when talking to the AS, it’s possible that these tokens could be swapped. Without the ath field to bind it, a captured signature applied to AT1 could be replayed with AT2 instead, changing the rights and access of the intended request. This same substitution prevention remains for rotated access tokens within the same combination of client and resource owner -- a rotated token value would require the calculation of a new proof. This binding additionally ensures that a proof intended for use with the access token is not usable without an access token, or vice-versa.

The resource server is required to calculate the hash of the token value presented and verify that it is the same as the hash value in the ath field as described in Section 4.3. Since the ath field value is covered by the DPoP proof’s signature, its inclusion binds the access token value to the holder of the key used to generate the signature.

Note that the ath field alone does not prevent replay of the DPoP proof or provide binding to the request in which the proof is presented, and it is still important to check the time window of the proof as well as the included message parameters such as htm and htu.

7.1. The DPoP Authentication Scheme

A DPoP-bound access token is sent using the Authorization request header field per Section 11.6.2 of [RFC9110] using an authentication scheme of DPoP. The syntax of the Authorization header field for the DPoP scheme uses the token68 syntax defined in Section 11.2 of [RFC9110] (repeated below for ease of reference) for credentials. The ABNF notation syntax for DPoP authentication scheme credentials is as follows:

```
token68    = 1*( ALPHA / DIGIT /
                 "-" / "." / "_" / "~" / "+" / "/" ) *"="

credentials = "DPoP" 1*SP token68
```

Figure 12: DPoP Authentication Scheme ABNF

For such an access token, a resource server MUST check that a DPoP proof was also received in the DPoP header field of the HTTP request, check the DPoP proof according to the rules in Section 4.3, and check that the public key of the DPoP proof matches the public key to which the access token is bound per Section 6.

The resource server MUST NOT grant access to the resource unless all checks are successful.

Figure 13 shows an example request to a protected resource with a DPoP-bound access token in the Authorization header and the DPoP proof in the DPoP header (with '\' line wrapping per RFC 8792). Following that is Figure 14, which shows the decoded content of that DPoP proof. The JSON of the JWT header and payload are shown but the signature part is omitted. As usual, line breaks and indentation are included for formatting and readability.

```
GET /protectedresource HTTP/1.1
Host: resource.example.org
Authorization: DPoP Kz~8mXK1EalYznwH-LC-1fBAo.4Ljp~zsPE_NeO.gxU
DPoP: eyJ0eXAiOiJkcG9wK2p3dCIsImFsZyI6IkVTMjU2IiwiandrIjp7Imt0eSI6Ik\
 VDIiwieCI6Imw4dEZyaHgtMzR0VjNoUklDUkRZOXpDa0RscEJoRjQyVVFVZldWQVdCR\
 nMiLCJ5IjoiOVZFNGpmX09rX282NHpiVFRsY3VOSmFqSG10NnY5VERWclUwQ2R2R1JE\
 QSIsImNydiI6IlAtMjU2In19.eyJqdGkiOiJlMWozVl9iS2ljOC1MQUVCIiwiaHRtIj\
 oiR0VUIiwiaHR1IjoiaHR0cHM6Ly9yZXNvdXJjZS5leGFtcGxlLm9yZy9wcm90ZWN0Z\
 WRyZXNvdXJjZSIsImlhdCI6MTU2MjI2MjYxOCwiYXRoIjoiZlVIeU8ycjJaM0RaNTNF\
 c05yV0JiMHhXWG9hTnk1OUlpS0NBcWtzbVFFbyJ9.2oW9RP35yRqzhrtNP86L-Ey71E\
 OptxRimPPToA1plemAgR6pxHF8y6-yqyVnmcw6Fy1dqd-jfxSYoMxhAJpLjA
```

Figure 13: DPoP Protected Resource Request

```
{
  "typ":"dpop+jwt",
  "alg":"ES256",
  "jwk": {
    "kty":"EC",
    "x":"l8tFrhx-34tV3hRICRDY9zCkDlpBhF42UQUfWVAWBFs",
    "y":"9VE4jf_Ok_o64zbTTlcuNJajHmt6v9TDVrU0CdvGRDA",
    "crv":"P-256"
  }
}
.
{
  "jti":"e1j3V_bKic8-LAEB",
  "htm":"GET",
  "htu":"https://resource.example.org/protectedresource",
  "iat":1562262618,
  "ath":"fUHyO2r2Z3DZ53EsNrWBb0xWXoaNy59IiKCAqksmQEo"
}
```

Figure 14: Decoded Content of the DPoP Proof JWT in Figure 13

Upon receipt of a request to a protected resource within the protection space requiring DPoP authentication, if the request does not include valid credentials or does not contain an access token sufficient for access, the server can respond with a challenge to the client to provide DPoP authentication information. Such a challenge is made using the 401 (Unauthorized) response status code ([RFC9110], Section 15.5.2) and the WWW-Authenticate header field ([RFC9110], Section 11.6.1). The server MAY include the WWW-Authenticate header in response to other conditions as well.

In such challenges:

* The scheme name is DPoP.

* The authentication parameter realm MAY be included to indicate the scope of protection in the manner described in [RFC9110], Section 11.5.
* A scope authentication parameter MAY be included as defined in [RFC6750], Section 3.
* An error parameter ([RFC6750], Section 3) SHOULD be included to indicate the reason why the request was declined, if the request included an access token but failed authentication. The error parameter values described in Section 3.1 of [RFC6750] are suitable as are any appropriate values defined by extension. The value use_dpop_nonce can be used as described in Section 9 to signal that a nonce is needed in the DPoP proof of subsequent request(s). And invalid_dpop_proof is used to indicate that the DPoP proof itself was deemed invalid based on the criteria of Section 4.3.
* An error_description parameter ([RFC6750], Section 3) MAY be included along with the error parameter to provide developers a human-readable explanation that is not meant to be displayed to end-users.
* An algs parameter SHOULD be included to signal to the client the JWS algorithms that are acceptable for the DPoP proof JWT. The value of the parameter is a space-delimited list of JWS alg (Algorithm) header values ([RFC7515], Section 4.1.1).
* Additional authentication parameters MAY be used and unknown parameters MUST be ignored by recipients.

For example, in response to a protected resource request without authentication:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: DPoP algs="ES256 PS256"
```

Figure 15: HTTP 401 Response to a Protected Resource Request without Authentication

And in response to a protected resource request that was rejected because the confirmation of the DPoP binding in the access token failed (with '\' line wrapping per RFC 8792):

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: DPoP error="invalid_token", \
  error_description="Invalid DPoP key binding", algs="ES256"
```

Figure 16: HTTP 401 Response to a Protected Resource Request with an Invalid Token

Note that browser-based client applications using CORS [WHATWG.Fetch] only have access to CORS-safelisted response HTTP headers by default. In order for the application to obtain and use the WWW-Authenticate HTTP response header value, the server needs to make it available to the application by including WWW-Authenticate in the Access-Control-Expose-Headers response header list value.

This authentication scheme is for origin-server authentication only. Therefore, this authentication scheme MUST NOT be used with the Proxy-Authenticate or Proxy-Authorization header fields.

Note that the syntax of the Authorization header field for this authentication scheme follows the usage of the Bearer scheme defined in Section 2.1 of [RFC6750]. While not the preferred credential syntax of [RFC9110], it is compatible with the general authentication framework therein and was used for consistency and familiarity with the Bearer scheme.

7.2. Compatibility with the Bearer Authentication Scheme

Protected resources simultaneously supporting both the DPoP and Bearer schemes need to update how evaluation of bearer tokens is performed to prevent downgraded usage of a DPoP-bound access token. Specifically, such a protected resource MUST reject a DPoP-bound access token received as a bearer token per [RFC6750].

Section 11.6.1 of [RFC9110] allows a protected resource to indicate support for multiple authentication schemes (i.e., Bearer and DPoP) with the WWW-Authenticate header field of a 401 (Unauthorized) response.

A protected resource that supports only [RFC6750] and is unaware of DPoP would most presumably accept a DPoP-bound access token as a bearer token (JWT [RFC7519] says to ignore unrecognized claims, Introspection [RFC7662] says that other parameters might be present while placing no functional requirements on their presence, and [RFC6750] is effectively silent on the content of the access token as it relates to validity). As such, a client can send a DPoP-bound access token using the Bearer scheme upon receipt of a WWW-Authenticate: Bearer challenge from a protected resource (or if it has prior such knowledge about the capabilities of the protected resource). The effect of this likely simplifies the logistics of phased upgrades to protected resources in their support DPoP or even prolonged deployments of protected resources with mixed token type support.

If a protected resource supporting both Bearer and DPoP schemes elects to respond with multiple WWW-Authenticate challenges, attention should be paid to which challenge(s) should deliver the actual error information. It is RECOMMENDED that the following rules be adhered to:

* If no authentication information has been included with the request, then the challenges SHOULD NOT include an error code or other error information, as per [RFC6750], Section 3.1 (Figure 17).

* If the mechanism used to attempt authentication could be established unambiguously, then the corresponding challenge SHOULD be used to deliver error information (Figure 18).

* Otherwise, both Bearer and DPoP challenged MAY be used to deliver error information (Figure 19).

(Where needed, the following examples use '\' line wrapping per RFC 8792.)

```
GET /protectedresource HTTP/1.1
Host: resource.example.org

HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer, DPoP algs="ES256 PS256"
```

Figure 17: HTTP 401 Response to a Protected Resource Request without Authentication

```
GET /protectedresource HTTP/1.1
Host: resource.example.org
Authorization: Bearer INVALID_TOKEN

HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer error="invalid_token", \
    error_description="Invalid token", DPoP algs="ES256 PS256"
```

Figure 18: HTTP 401 Response to a Protected Resource Request with Invalid Authentication

```
GET /protectedresource HTTP/1.1
Host: resource.example.org
Authorization: Bearer Kz~8mXK1EalYznwH-LC-1fBAo.4Ljp~zsPE_NeO.gxU
Authorization: DPoP Kz~8mXK1EalYznwH-LC-1fBAo.4Ljp~zsPE_NeO.gxU

HTTP/1.1 400 Bad Request
WWW-Authenticate: Bearer error="invalid_request", \
 error_description="Multiple methods used to include access token", \
 DPoP algs="ES256 PS256", error="invalid_request", \
 error_description="Multiple methods used to include access token"
```

Figure 19: HTTP 400 Response to a Protected Resource Request with Ambiguous Authentication

7.3. Client Considerations

Authorization including a DPoP proof may not be idempotent (depending on server enforcement of jti, iat and nonce claims). Consequently, all previously idempotent requests for protected resources that were previously idempotent may no longer be idempotent. It is RECOMMENDED that clients generate a unique DPoP proof even when retrying idempotent requests in response to HTTP errors generally understood as transient.

Clients that encounter frequent network errors may experience additional challenges when interacting with servers with more strict nonce validation implementations.

8. Authorization Server-Provided Nonce

This section specifies a mechanism using opaque nonces provided by the server that can be used to limit the lifetime of DPoP proofs. Without employing such a mechanism, a malicious party controlling the client (including potentially the end-user) can create DPoP proofs for use arbitrarily far in the future.

Including a nonce value contributed by the authorization server in the DPoP proof MAY be used by authorization servers to limit the lifetime of DPoP proofs. The server determines when and if to issue a new DPoP nonce challenge thereby requiring the use of the nonce value in subsequent DPoP proofs. The logic through which the server makes that determination is out of scope of this document.

An authorization server MAY supply a nonce value to be included by the client in DPoP proofs sent. In this case, the authorization server responds to requests not including a nonce with an HTTP 400 (Bad Request) error response per Section 5.2 of [RFC6749] using use_dpop_nonce as the error code value. The authorization server

includes a DPoP-Nonce HTTP header in the response supplying a nonce value to be used when sending the subsequent request. Nonce values MUST be unpredictable. This same error code is used when supplying a new nonce value when there was a nonce mismatch. The client will typically retry the request with the new nonce value supplied upon receiving a use_dpop_nonce error with an accompanying nonce value.

For example, in response to a token request without a nonce when the authorization server requires one, the authorization server can respond with a DPoP-Nonce value such as the following to provide a nonce value to include in the DPoP proof:

```
HTTP/1.1 400 Bad Request
DPoP-Nonce: eyJ7S_zG.eyJH0-Z.HX4w-7v

{
 "error": "use_dpop_nonce",
 "error_description":
   "Authorization server requires nonce in DPoP proof"
}
```

Figure 20: HTTP 400 Response to a Token Request without a Nonce

Other HTTP headers and JSON fields MAY also be included in the error response, but there MUST NOT be more than one DPoP-Nonce header.

Upon receiving the nonce, the client is expected to retry its token request using a DPoP proof including the supplied nonce value in the nonce claim of the DPoP proof. An example unencoded JWT Payload of such a DPoP proof including a nonce is:

```
{
 "jti": "-BwC3ESc6acc2lTc",
 "htm": "POST",
 "htu": "https://server.example.com/token",
 "iat": 1562262616,
 "nonce": "eyJ7S_zG.eyJH0-Z.HX4w-7v"
}
```

Figure 21: DPoP Proof Payload Including a Nonce Value

The nonce is opaque to the client.

If the nonce claim in the DPoP proof does not exactly match a nonce recently supplied by the authorization server to the client, the authorization server MUST reject the request. The rejection response MAY include a DPoP-Nonce HTTP header providing a new nonce value to use for subsequent requests.

The intent is that clients need to keep only one nonce value and servers keep a window of recent nonces. That said, transient circumstances may arise in which the server's and client's stored nonce values differ. However, this situation is self-correcting; with any rejection message, the server can send the client the nonce value that the server wants it to use and the client can store that nonce value and retry the request with it. Even if the client and/or server discard their stored nonce values, that situation is also self-correcting because new nonce values can be communicated when responding to or retrying failed requests.

Note that browser-based client applications using CORS [WHATWG.Fetch] only have access to CORS-safelisted response HTTP headers by default. In order for the application to obtain and use the DPoP-Nonce HTTP response header value, the server needs to make it available to the application by including DPoP-Nonce in the Access-Control-Expose-Headers response header list value.

8.1. Nonce Syntax

The nonce syntax in ABNF as used by [RFC6749] (which is the same as the scope-token syntax) is:

```
nonce = 1*NQCHAR
```

Figure 22: Nonce ABNF

8.2. Providing a New Nonce Value

It is up to the authorization server when to supply a new nonce value for the client to use. The client is expected to use the existing supplied nonce in DPoP proofs until the server supplies a new nonce value.

The authorization server MAY supply the new nonce in the same way that the initial one was supplied: by using a DPoP-Nonce HTTP header in the response. The DPoP-Nonce HTTP header field uses the nonce syntax defined in Section 8.1. Of course, each time this happens it requires an extra protocol round trip.

A more efficient manner of supplying a new nonce value is also defined -- by including a DPoP-Nonce HTTP header in the HTTP 200 (OK) response from the previous request. The client MUST use the new nonce value supplied for the next token request, and for all subsequent token requests until the authorization server supplies a new nonce.

Responses that include the DPoP-Nonce HTTP header should be uncacheable (e.g., using Cache-Control: no-store in response to a GET request) to prevent the response being used to serve a subsequent request and a stale nonce value being used as a result.

An example 200 OK response providing a new nonce value is:

```
HTTP/1.1 200 OK
Cache-Control: no-store
DPoP-Nonce: eyJ7S_zG.eyJbYu3.xQmBj-1
```

Figure 23: HTTP 200 Response Providing the Next Nonce Value

9. Resource Server-Provided Nonce

Resource servers can also choose to provide a nonce value to be included in DPoP proofs sent to them. They provide the nonce using the DPoP-Nonce header in the same way that authorization servers do as described in Section 8 and Section 8.2. The error signaling is performed as described in Section 7.1. Resource servers use an HTTP 401 (Unauthorized) error code with an accompanying WWW-Authenticate: DPoP value and DPoP-Nonce value to accomplish this.

For example, in response to a resource request without a nonce when the resource server requires one, the resource server can respond with a DPoP-Nonce value such as the following to provide a nonce value to include in the DPoP proof (with '\' line wrapping per RFC 8792):

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: DPoP error="use_dpop_nonce", \
  error_description="Resource server requires nonce in DPoP proof"
DPoP-Nonce: eyJ7S_zG.eyJH0-Z.HX4w-7v
```

Figure 24: HTTP 401 Response to a Resource Request without a Nonce

Note that the nonces provided by an authorization server and a resource server are different and should not be confused with one another, since nonces will be only accepted by the server that issued them. Likewise, should a client use multiple authorization servers and/or resource servers, a nonce issued by any of them should be used only at the issuing server. Developers should also take care to not confuse DPoP nonces with the OpenID Connect [OpenID.Core] ID Token nonce.

10. Authorization Code Binding to DPoP Key

Binding the authorization code issued to the client's proof-of-possession key can enable end-to-end binding of the entire authorization flow. This specification defines the dpop_jkt authorization request parameter for this purpose. The value of the dpop_jkt authorization request parameter is the JSON Web Key (JWK) Thumbprint [RFC7638] of the proof-of-possession public key using the SHA-256 hash function - the same value as used for the jkt confirmation method defined in Section 6.1.

When a token request is received, the authorization server computes the JWK thumbprint of the proof-of-possession public key in the DPoP proof and verifies that it matches the dpop_jkt parameter value in the authorization request. If they do not match, it MUST reject the request.

An example authorization request using the dpop_jkt authorization request parameter follows (with '\' line wrapping per RFC 8792):

```
GET /authorize?response_type=code&client_id=s6BhdRkqt3&state=xyz\
    &redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb\
    &code_challenge=E9Melhoa2OwvFrEMTJguCHaoeK1t8URWbuGJSstw-cM\
    &code_challenge_method=S256\
    &dpop_jkt=NzbLsXh8uDCcd-6MNwXF4W_7noWXFZAfHkxZsRGC9Xs HTTP/1.1
Host: server.example.com
```

Figure 25: Authorization Request using the dpop_jkt Parameter

Use of the dpop_jkt authorization request parameter is OPTIONAL. Note that the dpop_jkt authorization request parameter MAY also be used in combination with PKCE [RFC7636], which is recommended by [I-D.ietf-oauth-security-topics] as a countermeasure to authorization code injection. The dpop_jkt authorization request parameter only provides similar protections when a unique DPoP key is used for each authorization request.

10.1. DPoP with Pushed Authorization Requests

When Pushed Authorization Requests (PAR, [RFC9126]) are used in conjunction with DPoP, there are two ways in which the DPoP key can be communicated in the PAR request:

* The dpop_jkt parameter can be used as described in Section 10 to bind the issued authorization code to a specific key. In this case, dpop_jkt MUST be included alongside other authorization request parameters in the POST body of the PAR request.

* Alternatively, the DPoP header can be added to the PAR request. In this case, the authorization server MUST check the provided DPoP proof JWT as defined in Section 4.3. It MUST further behave as if the contained public key's thumbprint was provided using dpop_jkt, i.e., reject the subsequent token request unless a DPoP proof for the same key is provided. This can help to simplify the implementation of the client, as it can "blindly" attach the DPoP header to all requests to the authorization server regardless of the type of request. Additionally, it provides a stronger binding, as the DPoP header contains a proof of possession of the private key.

Both mechanisms MUST be supported by an authorization server that supports PAR and DPoP. If both mechanisms are used at the same time, the authorization server MUST reject the request if the JWK Thumbprint in dpop_jkt does not match the public key in the DPoP header.

Allowing both mechanisms ensures that clients that use dpop_jkt do not need to distinguish between front-channel and pushed authorization requests, and at the same time, clients that only have one code path for protecting all calls to authorization server endpoints do not need to distinguish between requests to the PAR endpoint and the token endpoint.

11. Security Considerations

In DPoP, the prevention of token replay at a different endpoint (see Section 2) is achieved through authentication of the server per [RFC6125] and binding of the DPoP proof to a certain URI and HTTP method. DPoP, however, has a somewhat different nature of protection than TLS-based methods such as OAuth Mutual TLS [RFC8705] or OAuth Token Binding [I-D.ietf-oauth-token-binding] (see also Section 11.1 and Section 11.7). TLS-based mechanisms can leverage a tight integration between the TLS layer and the application layer to achieve strong message integrity, authenticity, and replay protection.

11.1. DPoP Proof Replay

If an adversary is able to get hold of a DPoP proof JWT, the adversary could replay that token at the same endpoint (the HTTP endpoint and method are enforced via the respective claims in the JWTs). To limit this, servers MUST only accept DPoP proofs for a limited time after their creation (preferably only for a relatively brief period on the order of seconds or minutes).

To prevent multiple uses of the same DPoP proof, servers can store, in the context of the target URI, the jti value of each DPoP proof for the time window in which the respective DPoP proof JWT would be accepted. HTTP requests to the same URI for which the jti value has been seen before would be declined. Such a single-use check, when strictly enforced, provides a very strong protection against DPoP proof replay, but may not always be feasible in practice, e.g., when multiple servers behind a single endpoint have no shared state.

In order to guard against memory exhaustion attacks, a server that is tracking jti values should reject DPoP proof JWTs with unnecessarily large jti values or store only a hash thereof.

Note: To accommodate for clock offsets, the server MAY accept DPoP proofs that carry an iat time in the reasonably near future (on the order of seconds or minutes). Because clock skews between servers and clients may be large, servers MAY limit DPoP proof lifetimes by using server-provided nonce values containing the time at the server rather than comparing the client-supplied iat time to the time at the server. Nonces created in this way yield the same result even in the face of arbitrarily large clock skews.

Server-provided nonces are an effective means for further reducing the chances for successful DPoP proof replay. Unlike cryptographic nonces, it is acceptable for clients to use the same nonce multiple times, and for the server to accept the same nonce multiple times. As long as the jti value is tracked and duplicates rejected for the lifetime of the nonce, there is no additional risk of token replay.

11.2. DPoP Proof Pre-Generation

An attacker in control of the client can pre-generate DPoP proofs for specific endpoints arbitrarily far into the future by choosing the iat value in the DPoP proof to be signed by the proof-of-possession key. Note that one such attacker is the person who is the legitimate user of the client. The user may pre-generate DPoP proofs to exfiltrate from the machine possessing the proof-of-possession key upon which they were generated and copy them to another machine that does not possess the key. For instance, a bank employee might pre-generate DPoP proofs on a bank computer and then copy them to another machine for use in the future, thereby bypassing bank audit controls. When DPoP proofs can be pre-generated and exfiltrated, all that is actually being proved in DPoP protocol interactions is possession of a DPoP proof -- not of the proof-of-possession key.

Use of server-provided nonce values that are not predictable by attackers can prevent this attack. By providing new nonce values at times of its choosing, the server can limit the lifetime of DPoP

proofs, preventing pre-generated DPoP proofs from being used. When server-provided nonces are used, possession of the proof-of-possession key is being demonstrated -- not just possession of a DPoP proof.

The ath claim limits the use of pre-generated DPoP proofs to the lifetime of the access token. Deployments that do not utilize the nonce mechanism SHOULD NOT issue long-lived DPoP constrained access tokens, preferring instead to use short-lived access tokens and refresh tokens. Whilst an attacker could pre-generate DPoP proofs to use the refresh token to obtain a new access token, they would be unable to realistically pre-generate DPoP proofs to use a newly issued access token.

11.3. DPoP Nonce Downgrade

A server MUST NOT accept any DPoP proofs without the nonce claim when a DPoP nonce has been provided to the client.

11.4. Untrusted Code in the Client Context

If an adversary is able to run code in the client's execution context, the security of DPoP is no longer guaranteed. Common issues in web applications leading to the execution of untrusted code are cross-site scripting and remote code inclusion attacks.

If the private key used for DPoP is stored in such a way that it cannot be exported, e.g., in a hardware or software security module, the adversary cannot exfiltrate the key and use it to create arbitrary DPoP proofs. The adversary can, however, create new DPoP proofs as long as the client is online, and use these proofs (together with the respective tokens) either on the victim's device or on a device under the attacker's control to send arbitrary requests that will be accepted by servers.

To send requests even when the client is offline, an adversary can try to pre-compute DPoP proofs using timestamps in the future and exfiltrate these together with the access or refresh token.

An adversary might further try to associate tokens issued from the token endpoint with a key pair under the adversary's control. One way to achieve this is to modify existing code, e.g., by replacing cryptographic APIs. Another way is to launch a new authorization grant between the client and the authorization server in an iframe. This grant needs to be "silent", i.e., not require interaction with the user. With code running in the client's origin, the adversary has access to the resulting authorization code and can use it to associate their own DPoP keys with the tokens returned from the token endpoint. The adversary is then able to use the resulting tokens on their own device even if the client is offline.

Therefore, protecting clients against the execution of untrusted code is extremely important even if DPoP is used. Besides secure coding practices, Content Security Policy [W3C.CSP] can be used as a second layer of defense against cross-site scripting.

11.5. Signed JWT Swapping

Servers accepting signed DPoP proof JWTs MUST verify that the typ field is dpop+jwt in the headers of the JWTs to ensure that adversaries cannot use JWTs created for other purposes.

11.6. Signature Algorithms

Implementers MUST ensure that only asymmetric digital signature algorithms (such as ES256) that are deemed secure can be used for signing DPoP proofs. In particular, the algorithm none MUST NOT be allowed.

11.7. Request Integrity

DPoP does not ensure the integrity of the payload or headers of requests. The DPoP proof only contains claims for the HTTP URI and method, but not, for example, the message body or general request headers.

This is an intentional design decision intended to keep DPoP simple to use, but as described, makes DPoP potentially susceptible to replay attacks where an attacker is able to modify message contents and headers. In many setups, the message integrity and confidentiality provided by TLS is sufficient to provide a good level of protection.

Note: While signatures covering other parts of requests are out of the scope of this specification, additional information to be signed can be added into DPoP proofs.

11.8. Access Token and Public Key Binding

The binding of the access token to the DPoP public key, which is specified in Section 6, uses a cryptographic hash of the JWK representation of the public key. It relies on the hash function having sufficient second-preimage resistance so as to make it computationally infeasible to find or create another key 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.

Similarly, the binding of the DPoP proof to the access token uses a hash of that access token as the value of the ath claim in the DPoP proof (see Section 4.2). This relies on the value of the hash being sufficiently unique so as to reliably identify the access token. The collision resistance of SHA-256 meets that requirement.

11.9. Authorization Code and Public Key Binding

Cryptographic binding of the authorization code to the DPoP public key, is specified in Section 10. This binding prevents attacks in which the attacker captures the authorization code and creates a DPoP proof using a proof-of-possession key other than that held by the client and redeems the authorization code using that DPoP proof. By ensuring end-to-end that only the client's DPoP key can be used, this prevents captured authorization codes from being exfiltrated and used at locations other than the one to which the authorization code was issued.

Authorization codes can, for instance, be harvested by attackers from places that the HTTP messages containing them are logged. Even when efforts are made to make authorization codes one-time-use, in practice, there is often a time window during which attackers can replay them. For instance, when authorization servers are implemented as scalable replicated services, some replicas may temporarily not yet have the information needed to prevent replay. DPoP binding of the authorization code solves these problems.

If an authorization server does not (or cannot) strictly enforce the single-use limitation for authorization codes and an attacker can access the authorization code (and if PKCE is used, the code_verifier), the attacker can create a forged token request, binding the resulting token to an attacker-controlled key. For example, using cross-site scripting, attackers might obtain access to the authorization code and PKCE parameters. Use of the dpop_jkt parameter prevents this attack.

The binding of the authorization code to the DPoP public key uses a JWK Thumbprint of the public key, just as the access token binding does. The same JWK Thumbprint considerations apply.

11.10. Hash Algorithm Agility

The jkt confirmation method member, the ath JWT claim, and the dpop_jkt authorization request parameter defined herein all use the output of the SHA-256 hash function as their value. The use of a single hash function by this specification was intentional and aimed at simplicity and avoidance of potential security and interoperability issues arising from common mistakes implementing and deploying parameterized algorithm agility schemes. The use of a different hash function is not precluded, however, if future circumstances change making SHA-256 insufficient for the requirements of this specification. Should that need arise, it is expected that a short specification be produced that updates this one. That specification will likely define, using the output of a then appropriate hash function as the value, a new confirmation method member, a new JWT claim, and a new authorization request parameter. These items will be used in place of, or alongside, their respective counterparts in the same message structures and flows of the larger protocol defined by this specification.

11.11. Binding to Client Identity

In cases where DPoP is used with client authentication, it is only bound to authentication by being coincident in the same TLS tunnel. Since the DPoP proof is not directly cryptographically bound to the authentication, it’s possible that the authentication or the DPoP messages were copied into the tunnel. While including the URI in the DPoP can partially mitigate some of this risk, modifying the authentication mechanism to provide cryptographic binding between authentication and DPoP could provide better protection. However, providing additional binding with authentication through the modification of authentication mechanisms or other means is beyond the scope of this specification.

12. IANA Considerations

12.1. OAuth Access Token Type Registration

This specification requests registration of the following access token type in the "OAuth Access Token Types" registry [IANA.OAuth.Params] established by [RFC6749].

* Type name: DPoP
* Additional Token Endpoint Response Parameters: (none)

* HTTP Authentication Scheme(s): DPoP
* Change controller: IETF
* Specification document(s): [[this specification]]

12.2. OAuth Extensions Error Registration

This specification requests registration of the following error values in the "OAuth Extensions Error" registry [IANA.OAuth.Params] established by [RFC6749].

Invalid DPoP proof:

* Name: invalid_dpop_proof
* Usage Location: token error response, resource access error response
* Protocol Extension: Demonstrating Proof of Possession (DPoP)
* Change controller: IETF
* Specification document(s): [[this specification]]

Use DPoP nonce:

* Name: use_dpop_nonce
* Usage Location: token error response, resource access error response
* Protocol Extension: Demonstrating Proof of Possession (DPoP)
* Change controller: IETF
* Specification document(s): [[this specification]]

12.3. OAuth Parameters Registration

This specification requests registration of the following authorization request parameter in the "OAuth Parameters" registry [IANA.OAuth.Params] established by [RFC6749].

* Name: dpop_jkt
* Parameter Usage Location: authorization request
* Change Controller: IETF
* Reference: [[Section 10 of this specification]]

12.4. HTTP Authentication Scheme Registration

This specification requests registration of the following scheme in the "Hypertext Transfer Protocol (HTTP) Authentication Scheme Registry" [RFC9110][IANA.HTTP.AuthSchemes]:

* Authentication Scheme Name: DPoP
* Reference: [[Section 7.1 of this specification]]

12.5. Media Type Registration

This section registers the application/dpop+jwt media type [RFC2046] in the IANA "Media Types" registry [IANA.MediaTypes] in the manner described in [RFC6838], which is used to indicate that the content is a DPoP JWT.

* Type name: application
* Subtype name: dpop+jwt
* Required parameters: n/a
* Optional parameters: n/a
* Encoding considerations: binary; A DPoP JWT is a JWT; JWT values are encoded as a series of base64url-encoded values (some of which may be the empty string) separated by period ('.') characters.
* Security considerations: See Section 11 of [[this specification]]
* Interoperability considerations: n/a
* Published specification: [[this specification]]
* Applications that use this media type: Applications using [[this specification]] for application-level proof of possession
* Fragment identifier considerations: n/a
* Additional information:
  - File extension(s): n/a
  - Macintosh file type code(s): n/a
* Person & email address to contact for further information: Michael B. Jones, mbj@microsoft.com
* Intended usage: COMMON
* Restrictions on usage: none
* Author: Michael B. Jones, mbj@microsoft.com
* Change controller: IETF
* Provisional registration? No

12.6. JWT Confirmation Methods Registration

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

* Confirmation Method Value: jkt
* Confirmation Method Description: JWK SHA-256 Thumbprint
* Change Controller: IETF
* Specification Document(s): [[Section 6 of this specification]]

12.7. JSON Web Token Claims Registration

This specification requests registration of the following Claims in the IANA "JSON Web Token Claims" registry [IANA.JWT] established by [RFC7519].

HTTP method:

* Claim Name: htm
* Claim Description: The HTTP method of the request
* Change Controller: IETF
* Specification Document(s): [[Section 4.2 of this specification]]

HTTP URI:

* Claim Name: htu
* Claim Description: The HTTP URI of the request (without query and fragment parts)
* Change Controller: IETF
* Specification Document(s): [[Section 4.2 of this specification]]

Access token hash:

* Claim Name: ath
* Claim Description: The base64url encoded SHA-256 hash of the ASCII encoding of the associated access token's value
* Change Controller: IETF
* Specification Document(s): [[Section 4.2 of this specification]]

12.7.1. "nonce" Registry Update

The Internet Security Glossary [RFC4949] provides a useful definition of nonce as a random or non-repeating value that is included in data exchanged by a protocol, usually for the purpose of guaranteeing liveness and thus detecting and protecting against replay attacks.

However, the initial registration of the nonce claim by [OpenID.Core] used language that was contextually specific to that application, which was potentially limiting to its general applicability.

This specification therefore requests that the entry for nonce in the IANA "JSON Web Token Claims" registry [IANA.JWT] be updated as follows to reflect that the claim can be used appropriately in other contexts.

* Claim Name: nonce
* Claim Description: Value used to associate a Client session with an ID Token (MAY also be used for nonce values in other applications of JWTs)
* Change Controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
* Specification Document(s): Section 2 of [OpenID.Core] and [[this specification]]

12.8. HTTP Message Header Field Names Registration

This document specifies the following HTTP header fields, registration of which is requested in the "Hypertext Transfer Protocol (HTTP) Field Name Registry" registry [RFC9110][IANA.HTTP.Fields]:

* Field name: DPoP

* Status: permanent

* Specification document: [[this specification]]

* Field name: DPoP-Nonce

* Status: permanent

* Specification document: [[this specification]]

12.9. OAuth Authorization Server Metadata Registration

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

* Metadata Name: dpop_signing_alg_values_supported
* Metadata Description: JSON array containing a list of the JWS algorithms supported for DPoP proof JWTs
* Change Controller: IETF
* Specification Document(s): [[Section 5.1 of this specification]]

12.10. OAuth Dynamic Client Registration Metadata

This specification requests registration of the following value in the IANA "OAuth Dynamic Client Registration Metadata" registry [IANA.OAuth.Params] established by [RFC7591].

* Metadata Name: dpop_bound_access_tokens
* Metadata Description: Boolean value specifying whether the client always uses DPoP for token requests
* Change Controller: IETF
* Specification Document(s): [[Section 5.2 of this specification]]

13. Normative References

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

[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>.

[RFC5234] 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>.

[RFC6125] Saint-Andre, P. and J. Hodges, "Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)", RFC 6125, DOI 10.17487/RFC6125, March 2011, <https://www.rfc-editor.org/info/rfc6125>.

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

[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>.

[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>.

[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>.

[RFC7638] 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>.

[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>.

[SHS] National Institute of Standards and Technology, "Secure Hash Standard (SHS)", FIPS PUB 180-4, August 2015, <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf>.

14. Informative References

[BREACH] "CVE-2013-3587", <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-3587>.

[CRIME] "CVE-2012-4929", <https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2012-4929>.

[Cloudbleed] "Incident report on memory leak caused by Cloudflare parser bug", <https://blog.cloudflare.com/incident-report-on-memory-leak-caused-by-cloudflare-parser-bug/>.

[GitHub.Tokens] "Security alert: Attack campaign involving stolen OAuth user tokens issued to two third-party integrators", <https://github.blog/2022-04-15-security-alert-stolen-oauth-user-tokens/>.

[Heartbleed] "CVE-2014-0160", <https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160>.

[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-22, 13 March 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-oauth-security-topics-22>.

[I-D.ietf-oauth-token-binding] Jones, M., Campbell, B., Bradley, J., and W. Denniss, "OAuth 2.0 Token Binding", Work in Progress, Internet-

Draft, draft-ietf-oauth-token-binding-08, 19 October 2018, <https://datatracker.ietf.org/doc/html/draft-ietf-oauth-token-binding-08>.

[IANA.HTTP.AuthSchemes]
IANA, "Hypertext Transfer Protocol (HTTP) Authentication Scheme Registry", <https://www.iana.org/assignments/http-authschemes>.

[IANA.HTTP.Fields]
IANA, "Hypertext Transfer Protocol (HTTP) Field Name Registry", <https://www.iana.org/assignments/http-fields/http-fields.xhtml>.

[IANA.JOSE.ALGS]
IANA, "JSON Web Signature and Encryption Algorithms", <https://www.iana.org/assignments/jose/jose.xhtml#web-signature-encryption-algorithms>.

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

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

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

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

[RFC2046] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types", RFC 2046, DOI 10.17487/RFC2046, November 1996, <https://www.rfc-editor.org/info/rfc2046>.

[RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, DOI 10.17487/RFC4122, July 2005, <https://www.rfc-editor.org/info/rfc4122>.

[RFC4949] Shirey, R., "Internet Security Glossary, Version 2", FYI 36, RFC 4949, DOI 10.17487/RFC4949, August 2007, <https://www.rfc-editor.org/info/rfc4949>.

[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>.

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

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

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

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

[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>.

[RFC8705] Campbell, B., Bradley, J., Sakimura, N., and T. Lodderstedt, "OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens", RFC 8705, DOI 10.17487/RFC8705, February 2020, <https://www.rfc-editor.org/info/rfc8705>.

[RFC8707] Campbell, B., Bradley, J., and H. Tschofenig, "Resource Indicators for OAuth 2.0", RFC 8707, DOI 10.17487/RFC8707, February 2020, <https://www.rfc-editor.org/info/rfc8707>.

[RFC8725] Sheffer, Y., Hardt, D., and M. Jones, "JSON Web Token Best Current Practices", BCP 225, RFC 8725, DOI 10.17487/RFC8725, February 2020, <https://www.rfc-editor.org/info/rfc8725>.

[RFC8792] Watsen, K., Auerswald, E., Farrel, A., and Q. Wu, "Handling Long Lines in Content of Internet-Drafts and RFCs", RFC 8792, DOI 10.17487/RFC8792, June 2020, <https://www.rfc-editor.org/info/rfc8792>.

[RFC9110] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022, <https://www.rfc-editor.org/info/rfc9110>.

[RFC9126] Lodderstedt, T., Campbell, B., Sakimura, N., Tonge, D., and F. Skokan, "OAuth 2.0 Pushed Authorization Requests", RFC 9126, DOI 10.17487/RFC9126, September 2021, <https://www.rfc-editor.org/info/rfc9126>.

[W3C.CSP] West, M., "Content Security Policy Level 3", World Wide Web Consortium Working Draft WD-CSP3-20181015, 15 October 2018, <https://www.w3.org/TR/2018/WD-CSP3-20181015/>.

[W3C.WebCryptoAPI] Watson, M., "Web Cryptography API", World Wide Web Consortium Recommendation REC-WebCryptoAPI-20170126, 26 January 2017, <https://www.w3.org/TR/2017/REC-WebCryptoAPI-20170126>.

[WHATWG.Fetch] WHATWG, "Fetch Living Standard", May 2022, <https://fetch.spec.whatwg.org/>.

Appendix A. Acknowledgements

We would like to thank Brock Allen, Annabelle Backman, Spencer Balogh, Dominick Baier, Vittorio Bertocci, Jeff Corrigan, Domingos Creado, Andrii Deinega, William Denniss, Vladimir Dzhuvinov, Mike Engan, Nikos Fotiou, Mark Haine, Dick Hardt, Joseph Heenan, Bjorn Hjelm, Jacob Ideskog, Jared Jennings, Benjamin Kaduk, Pieter Kasselman, Neil Madden, Rohan Mahy, Karsten Meyer zu Selhausen, Nicolas Mora, Steinar Noem, Mark Nottingham, Rob Otto, Aaron Parecki, Michael Peck, Roberto Polli, Paul Querna, Justin Richer, Joseph Salowey, Rifaat Shekh-Yusef, Filip Skokan, Dmitry Telegin, Dave Tonge, Jim Willeke, Philippe De Ryck, and others (please let us know, if you've been mistakenly omitted) for their valuable input, feedback and general support of this work.

This document originated from discussions at the 4th OAuth Security Workshop in Stuttgart, Germany. We thank the organizers of this workshop (Ralf Kusters, Guido Schmitz).

Appendix B. Document History

[[To be removed from the final specification]]

-16

* Per suggestion of the registry's designated expert, change "resource error response" to "resource access error response" for location of the two items in the "OAuth Extensions Error Registration" section

-15

* Editorial updates from IESG review/ballot
* Mike Jones and Daniel Fett with new email/organization info

-14

* Add sec considerations sub-section about binding to client identity
* Explicitly say that nonces must be unpredictable
* Change to a numbered list in 'Checking DPoP Proofs'
* Editorial adjustments
* Incorporated HTTP header field definition and RFC 8792 '\' line wrapping suggestions by Mark Nottingham

-13

* Editorial updates/fixes
* Make sure RFC7519 is a normative reference

-12

* Updates from Roman Danyliw's AD review
* DPoP-Nonce now included in HTTP header field registration request
* Fixed section reference to URI Scheme-Based Normalization
* Attempt to better describe the rationale for SHA-256 only and expectations for how hash algorithm agility would be achieved if needed in the future
* Elaborate on the use of multiple WWW-Authenticate challenges by protected resources
* Fix access token request examples that were missing a client_id

-11

* Updates addressing outstanding shepherd review comments per side meeting discussions at IETF 114
* Added more explanation of the PAR considerations

* Added parenthetical remark "(such as ES256)" to Signature Algorithms subsection
* Added more explanation for ath
* Added a reference to RFC8725 in mention of explicit JWT typing

-10

* Updates addressing some shepherd review comments
* Update HTTP references as RFCs 723x have been superseded by RFC 9110
* Editorial fixes
* Added some clarifications, etc. around nonce
* Added client considerations subsection
* Use bullets rather than numbers in Checking DPoP Proofs so as not to imply specific order
* Added notes/reminders about browser-based client applications using CORS needing access to response headers
* Added a JWT claims registry update request for "nonce" to (better) allow for more general use in other contexts

-09

* Add note/reminder about browser-based client applications using CORS needing access to response headers.
* Fixed typo

-08

* Lots of editorial updates from WGLC feedback
* Further clarify that either iat or nonce can be used alone in validating the timeliness of the proof and somewhat de-emphasize jti tracking

-07

* Registered the application/dpop+jwt media type.
* Editorial updates/clarifications based on review feedback.
* Added "(on the order of seconds or minutes)" to somewhat qualify "relatively brief period" and "reasonably near future" and give a general idea of expected timeframe without being overly prescriptive.
* Added a step to Section 4.3 to reiterate that the jwk header cannot have a private key.

-06

* Editorial updates and fixes

* Changed name of client metadata parameter to dpop_bound_access_tokens

-05

* Added Authorization Code binding via the dpop_jkt parameter.
* Described the authorization code reuse attack and how dpop_jkt mitigates it.
* Enhanced description of DPoP proof expiration checking.
* Described nonce storage requirements and how nonce mismatches and missing nonces are self-correcting.
* Specified the use of the use_dpop_nonce error for missing and mismatched nonce values.
* Specified that authorization servers use 400 (Bad Request) errors to supply nonces and resource servers use 401 (Unauthorized) errors to do so.
* Added a bit more about ath and pre-generated proofs to the security considerations.
* Mentioned confirming the DPoP binding of the access token in the list in Section 4.3.
* Added the always_uses_dpop client registration metadata parameter.
* Described the relationship between DPoP and Pushed Authorization Requests (PAR).
* Updated references for drafts that are now RFCs.

-04

* Added the option for a server-provided nonce in the DPoP proof.
* Registered the invalid_dpop_proof and use_dpop_nonce error codes.
* Removed fictitious uses of realm from the examples, as they added no value.
* State that if the introspection response has a token_type, it has to be DPoP.
* Mention that RFC7235 allows multiple authentication schemes in WWW-Authenticate with a 401.
* Editorial fixes.

-03

* Add an access token hash (ath) claim to the DPoP proof when used in conjunction with the presentation of an access token for protected resource access
* add Untrusted Code in the Client Context section to security considerations
* Editorial updates and fixes

-02

* Lots of editorial updates and additions including expanding on the objectives, better defining the key confirmation representations, example updates and additions, better describing mixed bearer/dpop token type deployments, clarify RT binding only being done for public clients and why, more clearly allow for a bound RT but with bearer AT, explain/justify the choice of SHA-256 for key binding, and more
* Require that a protected resource supporting bearer and DPoP at the same time must reject an access token received as bearer, if that token is DPoP-bound
* Remove the case-insensitive qualification on the htm claim check
* Relax the jti tracking requirements a bit and qualify it by URI

-01

* Editorial updates
* Attempt to more formally define the DPoP Authorization header scheme
* Define the 401/WWW-Authenticate challenge
* Added invalid_dpop_proof error code for DPoP errors in token request
* Fixed up and added to the IANA section
* Added dpop_signing_alg_values_supported authorization server metadata
* Moved the Acknowledgements into an Appendix and added a bunch of names (best effort)

-00 [[Working Group Draft]]

* Working group draft

-04

* Update OAuth MTLS reference to RFC 8705
* Use the newish RFC v3 XML and HTML format

-03

* rework the text around uniqueness requirements on the jti claim in the DPoP proof JWT
* make tokens a bit smaller by using htm, htu, and jkt rather than http_method, http_uri, and jkt#S256 respectively
* more explicit recommendation to use mTLS if that is available
* added David Waite as co-author
* editorial updates

-02

* added normalization rules for URIs
* removed distinction between proof and binding
* "jwk" header again used instead of "cnf" claim in DPoP proof
* renamed "Bearer-DPoP" token type to "DPoP"
* removed ability for key rotation
* added security considerations on request integrity
* explicit advice on extending DPoP proofs to sign other parts of the HTTP messages
* only use the jkt#S256 in ATs
* iat instead of exp in DPoP proof JWTs
* updated guidance on token_type evaluation

-01

* fixed inconsistencies
* moved binding and proof messages to headers instead of parameters
* extracted and unified definition of DPoP JWTs
* improved description

-00

* first draft

Authors' Addresses

Daniel Fett
Authlete
Email: mail@danielfett.de

Brian Campbell
Ping Identity
Email: bcampbell@pingidentity.com

John Bradley
Yubico
Email: ve7jtb@ve7jtb.com

Torsten Lodderstedt
yes.com
Email: torsten@lodderstedt.net

Michael Jones
independent
Email: michael_b_jones@hotmail.com

URI: https://self-issued.info/

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

OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: 12 July 2024

D. Hardt
Hell
A. Parecki
Okta
T. Lodderstedt
yes.com

9 January 2024

The OAuth 2.1 Authorization Framework
draft-ietf-oauth-v2-1-10

Abstract

The OAuth 2.1 authorization framework enables an application to obtain limited access to a protected resource, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and an authorization service, or by allowing the application to obtain access on its own behalf. This specification replaces and obsoletes the OAuth 2.0 Authorization Framework described in RFC 6749 and the Bearer Token Usage in RFC 6750.

Discussion Venues

This note is to be removed before publishing as an RFC.

Discussion of this document takes place on the OAuth Working Group mailing list (oauth@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/oauth/.

Source for this draft and an issue tracker can be found at https://github.com/oauth-wg/oauth-v2-1.

Status of This Memo

This Internet-Draft will expire on 12 July 2024.

Copyright Notice

Table of Contents

1. Introduction

OAuth introduces an authorization layer to the client-server authentication model by separating the role of the client from that of the resource owner. In OAuth, the client requests access to resources controlled by the resource owner and hosted by the resource server. Instead of using the resource owner's credentials to access protected resources, the client obtains an access token - a credential representing a specific set of access attributes such as scope and lifetime. Access tokens are issued to clients by an authorization server with the approval of the resource owner. The client uses the access token to access the protected resources hosted by the resource server.

In the older, more limited client-server authentication model, the client requests an access-restricted resource (protected resource) on the server by authenticating to the server using the resource owner's credentials. In order to provide applications access to restricted resources, the resource owner shares their credentials with the application. This creates several problems and limitations:

* Applications are required to store the resource owner's credentials for future use, typically a password in clear-text.

* Servers are required to support password authentication, despite the security weaknesses inherent in passwords.

* Applications gain overly broad access to the resource owner's protected resources, leaving resource owners without any ability to restrict duration or access to a limited subset of resources.

* Resource owners often reuse passwords with other unrelated services, despite best security practices. This password reuse means a vulnerability or exposure in one service may have security implications in completely unrelated services.

* Resource owners cannot revoke access to an individual application without revoking access to all third parties, and must do so by changing their password.

* Compromise of any application results in compromise of the end-user's password and all of the data protected by that password.

With OAuth, an end-user (resource owner) can grant a printing service (client) access to their protected photos stored at a photo- sharing service (resource server), without sharing their username and password with the printing service. Instead, they authenticate directly with a server trusted by the photo-sharing service (authorization server), which issues the printing service delegation-specific credentials (access token).

This separation of concerns also provides the ability to use more advanced user authentication methods such as multi-factor authentication and even passwordless authentication, without any modification to the applications. With all user authentication logic handled by the authorization server, applications don't need to be concerned with the specifics of implementing any particular authentication mechanism. This provides the ability for the authorization server to manage the user authentication policies and even change them in the future without coordinating the changes with applications.

The authorization layer can also simplify how a resource server determines if a request is authorized. Traditionally, after authenticating the client, each resource server would evaluate policies to compute if the client is authorized on each API call. In a distributed system, the policies need to be synchronized to all the resource servers, or the resource server must call a central policy server to process each request. In OAuth, evaluation of the policies is performed only when a new access token is created by the authorization server. If the authorized access is represented in the access token, the resource server no longer needs to evaluate the policies, and only needs to validate the access token. This simplification applies when the application is acting on behalf of a resource owner, or on behalf of itself.

OAuth is an authorization protocol, and is not an authentication protocol. The access token represents the authorization granted to the client. It is a common practice for the client to present the

access token to a proprietary API which returns a user identifier for the resource owner, and then using the result of the API as a proxy for authenticating the user. This practice is not part of the OAuth standard or security considerations, and may not have been considered by the resource owner. Implementors should carefully consult the documentation of the resource server before adopting this practice.

This specification is designed for use with HTTP ([RFC9110]). The use of OAuth over any protocol other than HTTP is out of scope.

Since the publication of the OAuth 2.0 Authorization Framework ([RFC6749]) in October 2012, it has been updated by OAuth 2.0 for Native Apps ([RFC8252]), OAuth Security Best Current Practice ([I-D.ietf-oauth-security-topics]), and OAuth 2.0 for Browser-Based Apps ([I-D.ietf-oauth-browser-based-apps]). The OAuth 2.0 Authorization Framework: Bearer Token Usage ([RFC6750]) has also been updated with ([I-D.ietf-oauth-security-topics]). This Standards Track specification consolidates the information in all of these documents and removes features that have been found to be insecure in [I-D.ietf-oauth-security-topics].

1.1. Roles

OAuth defines four roles:

"resource owner": An entity capable of granting access to a protected resource. When the resource owner is a person, it is referred to as an end-user. This is sometimes abbreviated as "RO".

"resource server": The server hosting the protected resources, capable of accepting and responding to protected resource requests using access tokens. The resource server is often accessible via an API. This is sometimes abbreviated as "RS".

"client": An application making protected resource requests on behalf of the resource owner and with its authorization. The term "client" does not imply any particular implementation characteristics (e.g., whether the application executes on a server, a desktop, or other devices).

"authorization server": The server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization. This is sometimes abbreviated as "AS".

The interaction between the authorization server and resource server is beyond the scope of this specification, however several extensions have been defined to provide an option for interoperability between

resource servers and authorization servers. The authorization server may be the same server as the resource server or a separate entity. A single authorization server may issue access tokens accepted by multiple resource servers.

1.2. Protocol Flow

```
     +--------+                               +---------------+
     |        |--(1)- Authorization Request ->|   Resource    |
     |        |                               |     Owner     |
     |        |<-(2)-- Authorization Grant ---|               |
     |        |                               +---------------+
     |        |
     |        |                               +---------------+
     |        |--(3)-- Authorization Grant -->| Authorization |
     | Client |                               |     Server    |
     |        |<-(4)----- Access Token -------|               |
     |        |                               +---------------+
     |        |
     |        |                               +---------------+
     |        |--(5)----- Access Token ------>|    Resource   |
     |        |                               |     Server    |
     |        |<-(6)--- Protected Resource ---|               |
     +--------+                               +---------------+
```

Figure 1: Abstract Protocol Flow

The abstract OAuth 2.1 flow illustrated in Figure 1 describes the interaction between the four roles and includes the following steps:

1. The client requests authorization from the resource owner. The authorization request can be made directly to the resource owner (as shown), or preferably indirectly via the authorization server as an intermediary.

2. The client receives an authorization grant, which is a credential representing the resource owner's authorization, expressed using one of the authorization grant types defined in this specification or using an extension grant type. The authorization grant type depends on the method used by the client to request authorization and the types supported by the authorization server.

3. The client requests an access token by authenticating with the authorization server and presenting the authorization grant.

4. The authorization server authenticates the client and validates the authorization grant, and if valid, issues an access token.

5. The client requests the protected resource from the resource server and authenticates by presenting the access token.

6. The resource server validates the access token, and if valid, serves the request.

The preferred method for the client to obtain an authorization grant from the resource owner (depicted in steps (1) and (2)) is to use the authorization server as an intermediary, which is illustrated in Figure 3 in Section 4.1.

1.3. Authorization Grant

An authorization grant represents the resource owner's authorization (to access its protected resources) used by the client to obtain an access token. This specification defines three grant types -- authorization code, refresh token, and client credentials -- as well as an extensibility mechanism for defining additional types.

1.3.1. Authorization Code

An authorization code is a temporary credential used to obtain an access token. Instead of the client requesting authorization directly from the resource owner, the client directs the resource owner to an authorization server (via its user agent) which in turn directs the resource owner back to the client with the authorization code. The client can then exchange the authorization code for an access token.

Before directing the resource owner back to the client with the authorization code, the authorization server authenticates the resource owner, and may request the resource owner's consent or otherwise inform them of the client's request. Because the resource owner only authenticates with the authorization server, the resource owner's credentials are never shared with the client, and the client does not need to have knowledge of any additional authentication steps such as multi-factor authentication or delegated accounts.

The authorization code provides a few important security benefits, such as the ability to authenticate the client, as well as the transmission of the access token directly to the client without passing it through the resource owner's user agent and potentially exposing it to others, including the resource owner.

1.3.2. Refresh Token

Refresh tokens are credentials used to obtain access tokens. Refresh tokens may be issued to the client by the authorization server and are used to obtain a new access token when the current access token becomes invalid or expires, or to obtain additional access tokens with identical or narrower scope (access tokens may have a shorter lifetime and fewer privileges than authorized by the resource owner). Issuing a refresh token is optional at the discretion of the authorization server, and may be issued based on properties of the client, properties of the request, policies within the authorization server, or any other criteria. If the authorization server issues a refresh token, it is included when issuing an access token (i.e., step (2) in Figure 2).

A refresh token is a string representing the authorization granted to the client by the resource owner. The string is considered opaque to the client. The refresh token may be an identifier used to retrieve the authorization information or may encode this information into the string itself. Unlike access tokens, refresh tokens are intended for use only with authorization servers and are never sent to resource servers.

```
+--------+                                           +---------------+
|        |--(1)------- Authorization Grant --------->|               |
|        |                                           |               |
|        |<-(2)----------- Access Token -------------|               |
|        |               & Refresh Token             |               |
|        |                                           |               |
|        |                             +----------+  |               |
|        |--(3)---- Access Token ---->|          |  |               |
|        |                            |          |  |               |
|        |<-(4)- Protected Resource --| Resource |  | Authorization |
| Client |                            |  Server  |  |     Server    |
|        |--(5)---- Access Token ---->|          |  |               |
|        |                            |          |  |               |
|        |<-(6)- Invalid Token Error -|          |  |               |
|        |                             +----------+  |               |
|        |                                           |               |
|        |--(7)----------- Refresh Token ----------->|               |
|        |                                           |               |
|        |<-(8)----------- Access Token -------------|               |
+--------+           & Optional Refresh Token        +---------------+
```

Figure 2: Refreshing an Expired Access Token

The flow illustrated in Figure 2 includes the following steps:

1. The client requests an access token by authenticating with the authorization server and presenting an authorization grant.

2. The authorization server authenticates the client and validates the authorization grant, and if valid, issues an access token and optionally a refresh token.

3. The client makes a protected resource request to the resource server by presenting the access token.

4. The resource server validates the access token, and if valid, serves the request.

5. Steps (3) and (4) repeat until the access token expires. If the client knows the access token expired, it skips to step (7); otherwise, it makes another protected resource request.

6. Since the access token is invalid, the resource server returns an invalid token error.

7. The client requests a new access token by presenting the refresh token and providing client authentication if it has been issued credentials. The client authentication requirements are based on the client type and on the authorization server policies.

8. The authorization server authenticates the client and validates the refresh token, and if valid, issues a new access token (and, optionally, a new refresh token).

1.3.3. Client Credentials

The client credentials or other forms of client authentication (e.g. a private key used to sign a JWT, as described in [RFC7523]) can be used as an authorization grant when the authorization scope is limited to the protected resources under the control of the client, or to protected resources previously arranged with the authorization server. Client credentials are used when the client is requesting access to protected resources based on an authorization previously arranged with the authorization server.

1.4. Access Token

Access tokens are credentials used to access protected resources. An access token is a string representing an authorization issued to the client.

The string is considered opaque to the client, even if it has a structure. The client MUST NOT expect to be able to parse the access token value. The authorization server is not required to use a consistent access token encoding or format other than what is expected by the resource server.

Access tokens represent specific scopes and durations of access, granted by the resource owner, and enforced by the resource server and authorization server.

Depending on the authorization server implementation, the token string may be used by the resource server to retrieve the authorization information, or the token may self-contain the authorization information in a verifiable manner (i.e., a token string consisting of a signed data payload). One example of a token retrieval mechanism is Token Introspection [RFC7662], in which the RS calls an endpoint on the AS to validate the token presented by the client. One example of a structured token format is JWT Profile for Access Tokens [RFC9068], a method of encoding and signing access token data as a JSON Web Token [RFC7519].

Additional authentication credentials, which are beyond the scope of this specification, may be required in order for the client to use an access token. This is typically referred to as a sender-constrained access token, such as DPoP [RFC9449] and Mutual TLS Certificate-Bound Access Tokens [RFC8705].

The access token provides an abstraction layer, replacing different authorization constructs (e.g., username and password) with a single token understood by the resource server. This abstraction enables issuing access tokens more restrictive than the authorization grant used to obtain them, as well as removing the resource server’s need to understand a wide range of authentication methods.

Access tokens can have different formats, structures, and methods of utilization (e.g., cryptographic properties) based on the resource server security requirements. Access token attributes and the methods used to access protected resources may be extended beyond what is described in this specification.

Access tokens (as well as any confidential access token attributes) MUST be kept confidential in transit and storage, and only shared among the authorization server, the resource servers the access token is valid for, and the client to which the access token is issued.

The authorization server MUST ensure that access tokens cannot be generated, modified, or guessed to produce valid access tokens by unauthorized parties.

1.4.1. Access Token Scope

Access tokens are intended to be issued to clients with less privileges than the user granting the access has. This is known as a limited "scope" access token. The authorization server and resource server can use this scope mechanism to limit what types of resources or level of access a particular client can have. For example, a client may only need "read" access to a user's resources, but doesn't need to update resources, so the client can request the read-only scope defined by the authorization server, and obtain an access token that cannot be used to update resources. This requires coordination between the authorization server and resource server. The authorization server provides the client the ability to request specific scopes, and associates those scopes with the access token issued to the client. The resource server is then responsible for enforcing scopes when presented with a limited-scope access token.

To request a limited-scope access token, the client uses the scope request parameter at the authorization or token endpoints, depending on the grant type used. In turn, the authorization server uses the scope response parameter to inform the client of the scope of the access token issued.

The value of the scope parameter is expressed as a list of space-delimited, case-sensitive strings. The strings are defined by the authorization server. If the value contains multiple space-delimited strings, their order does not matter, and each string adds an additional access range to the requested scope.

```
scope       = scope-token *( SP scope-token )
scope-token = 1*( %x21 / %x23-5B / %x5D-7E )
```

The authorization server MAY fully or partially ignore the scope requested by the client, based on the authorization server policy or the resource owner's instructions. If the issued access token scope is different from the one requested by the client, the authorization server MUST include the scope response parameter in the token response (Section 3.2.3) to inform the client of the actual scope granted.

If the client omits the scope parameter when requesting authorization, the authorization server MUST either process the request using a pre-defined default value or fail the request indicating an invalid scope. The authorization server SHOULD document its scope requirements and default value (if defined).

1.4.2. Bearer Tokens

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

Bearer Tokens may be enhanced with proof-of-possession specifications such as DPoP [RFC9449] and mTLS [RFC8705] to provide proof-of-possession characteristics.

To protect against access token disclosure, the communication interaction between the client and the resource server MUST utilize confidentiality and integrity protection as described in Section 1.5.

There is no requirement on the particular structure or format of a bearer token. If a bearer token is a reference to authorization information, such references MUST be infeasible for an attacker to guess, such as using a sufficiently long cryptographically random string. If a bearer token uses an encoding mechanism to contain the authorization information in the token itself, the access token MUST use integrity protection sufficient to prevent the token from being modified. One example of an encoding and signing mechanism for access tokens is described in JSON Web Token Profile for Access Tokens [RFC9068].

1.4.3. Sender-Constrained Access Tokens

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

Authorization and resource servers SHOULD use mechanisms for sender-constraining access tokens, such as OAuth Demonstration of Proof of Possession (DPoP) [RFC9449] or Mutual TLS for OAuth 2.0 [RFC8705]. See [I-D.ietf-oauth-security-topics] Section 4.10.1, to prevent misuse of stolen and leaked access tokens.

It is RECOMMENDED to use end-to-end TLS between the client and the resource server. If TLS traffic needs to be terminated at an intermediary, refer to Section 4.13 of [I-D.ietf-oauth-security-topics] for further security advice.

1.5. Communication security

Implementations MUST use a mechanism to provide communication authentication, integrity and confidentiality such as Transport-Layer Security [RFC8446], to protect the exchange of clear-text credentials and tokens either in the content or in header fields from eavesdropping, tampering, and message forgery (eg. see Section 2.4.1, Section 7.5.1, Section 3.2, and Section 1.4.2).

OAuth URLs MUST use the https scheme except for loopback interface redirect URIs, which MAY use the http scheme. When using https, TLS certificates MUST be checked according to [RFC9110]. At the time of this writing, TLS version 1.3 [RFC8446] is the most recent version.

Implementations MAY also support additional transport-layer security mechanisms that meet their security requirements.

The identification of the TLS versions and algorithms is outside the scope of this specification. Refer to [BCP195] for up to date recommendations on transport layer security, and to the relevant specifications for certificate validation and other security considerations.

1.6. HTTP Redirections

This specification makes extensive use of HTTP redirections, in which the client or the authorization server directs the resource owner's user agent to another destination. While the examples in this specification show the use of the HTTP 302 status code, any other method available via the user agent to accomplish this redirection, with the exception of HTTP 307, is allowed and is considered to be an implementation detail. See Section 7.5.4 for details.

1.7. Interoperability

OAuth 2.1 provides a rich authorization framework with well-defined security properties.

This specification leaves a few required components partially or fully undefined (e.g., client registration, authorization server capabilities, endpoint discovery). Some of these behaviors are defined in optional extensions which implementations can choose to use, such as:

* [RFC8414]: Authorization Server Metadata, defining an endpoint clients can use to look up the information needed to interact with a particular OAuth server

* [RFC7591]: Dynamic Client Registration, providing a mechanism for programmatically registering clients with an authorization server

* [RFC7592]: Dynamic Client Management, providing a mechanism for updating dynamically registered client information

* [RFC7662]: Token Introspection, defining a mechanism for resource servers to obtain information about access tokens

Please refer to Appendix C for a list of current known extensions at the time of this publication.

1.8. Compatibility with OAuth 2.0

OAuth 2.1 is compatible with OAuth 2.0 with the extensions and restrictions from known best current practices applied. Specifically, features not specified in OAuth 2.0 core, such as PKCE, are required in OAuth 2.1. Additionally, some features available in OAuth 2.0, such as the Implicit or Resource Owner Credentials grant types, are not specified in OAuth 2.1. Furthermore, some behaviors allowed in OAuth 2.0 are restricted in OAuth 2.1, such as the strict string matching of redirect URIs required by OAuth 2.1.

See Section 10 for more details on the differences from OAuth 2.0.

1.9. 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 BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

This specification uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234]. Additionally, the rule URI-reference is included from "Uniform Resource Identifier (URI): Generic Syntax" [RFC3986].

Certain security-related terms are to be understood in the sense defined in [RFC4949]. These terms include, but are not limited to, "attack", "authentication", "authorization", "certificate", "confidentiality", "credential", "encryption", "identity", "sign", "signature", "trust", "validate", and "verify".

The term "content" is to be interpreted as described in Section 6.4 of [RFC9110].

The term "user agent" is to be interpreted as described in Section 3.5 of [RFC9110].

Unless otherwise noted, all the protocol parameter names and values are case sensitive.

2. Client Registration

Before initiating the protocol, the client must establish its registration with the authorization server. The means through which the client registers with the authorization server are beyond the scope of this specification but typically involve the client developer manually registering the client at the authorization server’s website after creating an account and agreeing to the service’s Terms of Service, or by using Dynamic Client Registration ([RFC7591]).

Client registration does not require a direct interaction between the client and the authorization server. When supported by the authorization server, registration can rely on other means for establishing trust and obtaining the required client properties (e.g., redirect URI, client type). For example, registration can be accomplished using a self-issued or third-party-issued assertion, or by the authorization server performing client discovery using a trusted channel.

When registering a client, the client developer SHALL:

* specify the client type as described in Section 2.1,

* provide client details needed by the grant type in use, such as redirect URIs as described in Section 2.3, and

* include any other information required by the authorization server (e.g., application name, website, description, logo image, the acceptance of legal terms).

Dynamic Client Registration ([RFC7591]) defines a common general data model for clients that may be used even with manual client registration.

2.1. Client Types

OAuth 2.1 defines two client types based on their ability to authenticate securely with the authorization server.

"confidential": Clients that have credentials with the AS are designated as "confidential clients"

"public": Clients without credentials are called "public clients"

Any clients with credentials MUST take precautions to prevent leakage and abuse of their credentials.

Client authentication allows an Authorization Server to ensure it is interacting with a certain client (identified by its client_id) in an OAuth flow. The Authorization Server might make policy decisions about things such as whether to prompt the user for consent on every authorization or only the first based on the confidence that the Authorization Server is actually communicating with the legitimate client.

Whether and how an Authorization Server validates the identity of a client or the party providing/operating this client is out of scope of this specification. Authorization servers SHOULD consider the level of confidence in a client's identity when deciding whether they allow a client access to more sensitive resources and operations such as the Client Credentials grant type and how often to prompt the user for consent.

A single client_id SHOULD NOT be treated as more than one type of client.

This specification has been designed around the following client profiles:

"web application": A web application is a client running on a web server. Resource owners access the client via an HTML user interface rendered in a user agent on the device used by the resource owner. The client credentials as well as any access tokens issued to the client are stored on the web server and are not exposed to or accessible by the resource owner.

"browser-based application": A browser-based application is a client in which the client code is downloaded from a web server and executes within a user agent (e.g., web browser) on the device used by the resource owner. Protocol data and credentials are easily accessible (and often visible) to the resource owner. If such applications wish to use client credentials, it is recommended to utilize the backend for frontend pattern. Since such applications reside within the user agent, they can make seamless use of the user agent capabilities when requesting authorization.

"native application": A native application is a client installed and

executed on the device used by the resource owner. Protocol data and credentials are accessible to the resource owner. It is assumed that any client authentication credentials included in the application can be extracted. Dynamically issued access tokens and refresh tokens can receive an acceptable level of protection. On some platforms, these credentials are protected from other applications residing on the same device. If such applications wish to use client credentials, it is recommended to utilize the backend for frontend pattern, or issue the credentials at runtime using Dynamic Client Registration ([RFC7591]).

2.2. Client Identifier

Every client is identified in the context of an authorization server by a client identifier -- a unique string representing the registration information provided by the client. While the Authorization Server typically issues the client identifier itself, it may also serve clients whose client identifier was created by a party other than the Authorization Server. The client identifier is not a secret; it is exposed to the resource owner and MUST NOT be used alone for client authentication. The client identifier is unique in the context of an authorization server.

The client identifier is an opaque string whose size is left undefined by this specification. The client should avoid making assumptions about the identifier size. The authorization server SHOULD document the size of any identifier it issues.

If the authorization server supports clients with client identifiers issued by parties other than the authorization server, the authorization server SHOULD take precautions to avoid clients impersonating resource owners as described in Section 7.4.

2.3. Client Redirection Endpoint

The client redirection endpoint (also referred to as "redirect endpoint") is the URI of the client that the authorization server redirects the user agent back to after completing its interaction with the resource owner.

The authorization server redirects the user agent to one of the client’s redirection endpoints previously established with the authorization server during the client registration process.

The redirect URI MUST be an absolute URI as defined by [RFC3986] Section 4.3. The redirect URI MAY include an "application/x-www-form-urlencoded" formatted query component ([WHATWG.URL]), which MUST be retained when adding additional query parameters. The redirect URI MUST NOT include a fragment component.

2.3.1. Registration Requirements

Authorization servers MUST require clients to register their complete redirect URI (including the path component). Authorization servers MUST reject authorization requests that specify a redirect URI that doesn't exactly match one that was registered, with an exception for loopback redirects, where an exact match is required except for the port URI component, see Section 4.1.1 for details.

The authorization server MAY allow the client to register multiple redirect URIs.

Registration may happen out of band, such as a manual step of configuring the client information at the authorization server, or may happen at runtime, such as in the initial POST in Pushed Authorization Requests [RFC9126].

For private-use URI scheme-based redirect URIs, authorization servers SHOULD enforce the requirement in Section 8.4.3 that clients use schemes that are reverse domain name based. At a minimum, any private-use URI scheme that doesn't contain a period character (.) SHOULD be rejected.

In addition to the collision-resistant properties, this can help to prove ownership in the event of a dispute where two apps claim the same private-use URI scheme (where one app is acting maliciously). For example, if two apps claimed com.example.app, the owner of example.com could petition the app store operator to remove the counterfeit app. Such a petition is harder to prove if a generic URI scheme was used.

Clients MUST NOT expose URLs that forward the user's browser to arbitrary URIs obtained from a query parameter ("open redirector"), as described in Section 7.12. Open redirectors can enable exfiltration of authorization codes and access tokens.

The client MAY use the state request parameter to achieve per-request customization if needed rather than varying the redirect URI per request.

Without requiring registration of redirect URIs, attackers can use the authorization endpoint as an open redirector as described in Section 7.12.

2.3.2. Multiple Redirect URIs

If multiple redirect URIs have been registered to a client, the client MUST include a redirect URI with the authorization request using the redirect_uri request parameter (Section 4.1.1). If only a single redirect URI has been registered to a client, the redirect_uri request parameter is optional.

2.3.3. Preventing CSRF Attacks

Clients MUST prevent Cross-Site Request Forgery (CSRF) attacks. 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 the code_challenge parameter MAY rely on the CSRF protection provided by that mechanism. In OpenID Connect flows, validating 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 7.9).

2.3.4. Preventing Mix-Up Attacks

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. In order to prevent mix-up attacks, clients MUST only process redirect responses of the issuer they sent the respective request to and from the same user agent this authorization request was initiated with.

See Section 7.13 for a detailed description of two different defenses against mix-up attacks.

2.3.5. Invalid Endpoint

If an authorization request fails validation due to a missing, invalid, or mismatching redirect URI, the authorization server SHOULD inform the resource owner of the error and MUST NOT automatically redirect the user agent to the invalid redirect URI.

2.3.6. Endpoint Content

The redirection request to the client's endpoint typically results in an HTML document response, processed by the user agent. If the HTML response is served directly as the result of the redirection request, any script included in the HTML document will execute with full access to the redirect URI and the artifacts (e.g. authorization code) it contains. Additionally, the request URL containing the authorization code may be sent in the HTTP Referer header to any embedded images, stylesheets and other elements loaded in the page.

The client SHOULD NOT include any third-party scripts (e.g., third-party analytics, social plug-ins, ad networks) in the redirect URI endpoint response. Instead, it SHOULD extract the artifacts from the URI and redirect the user agent again to another endpoint without exposing the artifacts (in the URI or elsewhere). If third-party scripts are included, the client MUST ensure that its own scripts (used to extract and remove the credentials from the URI) will execute first.

2.4. Client Authentication

The authorization server MUST only rely on client authentication if the process of issuance/registration and distribution of the underlying credentials ensures their confidentiality.

If the client is confidential, the authorization server MAY accept any form of client authentication meeting its security requirements (e.g., password, public/private key pair).

It is RECOMMENDED to use asymmetric (public-key based) methods for client authentication such as mTLS [RFC8705] or using signed JWTs ("Private Key JWT") in accordance with [RFC7521] and [RFC7523] (in [OpenID] defined as the client authentication method private_key_jwt). When such 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.

When client authentication is not possible, the authorization server SHOULD employ other means to validate the client's identity -- for example, by requiring the registration of the client redirect URI or enlisting the resource owner to confirm identity. A valid redirect URI is not sufficient to verify the client's identity when asking for resource owner authorization but can be used to prevent delivering credentials to a counterfeit client after obtaining resource owner authorization.

The client MUST NOT use more than one authentication method in each request to prevent a conflict of which authentication mechanism is authoritative for the request.

The authorization server MUST consider the security implications of interacting with unauthenticated clients and take measures to limit the potential exposure of tokens issued to such clients, (e.g., limiting the lifetime of refresh tokens).

The privileges an authorization server associates with a certain client identity MUST depend on the assessment of the overall process for client identification and client credential lifecycle management. See Section 7.2 for additional details.

2.4.1. Client Secret

Clients in possession of a client secret, sometimes known as a client password, MAY use the HTTP Basic authentication scheme as defined in Section 11 of [RFC9110] to authenticate with the authorization server. The client identifier is encoded using the application/x-www-form-urlencoded encoding algorithm per Appendix B, and the encoded value is used as the username; the client secret is encoded using the same algorithm and used as the password. The authorization server MUST support the HTTP Basic authentication scheme for authenticating clients that were issued a client secret.

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

Authorization: Basic czZCaGRSa3F0Mzo3RmpmcDBaQnIxS3REUmJuZlZkbUl3

In addition to that, the authorization server MAY support including the client credentials in the request content using the following parameters:

"client_id": REQUIRED. The client identifier issued to the client during the registration process described by Section 2.2.

"client_secret": REQUIRED. The client secret.

Including the client credentials in the request content using the two parameters is NOT RECOMMENDED and SHOULD be limited to clients unable to directly utilize the HTTP Basic authentication scheme (or other password-based HTTP authentication schemes). The parameters can only be transmitted in the request content and MUST NOT be included in the request URI.

For example, a request to refresh an access token (Section 4.3) using the content parameters (with extra line breaks for display purposes only):

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

grant_type=refresh_token&refresh_token=tGzv3JOkF0XG5Qx2TlKWIA
&client_id=s6BhdRkqt3&client_secret=7Fjfp0ZBr1KtDRbnfVdmIw
```

Since this client authentication method involves a password, the authorization server MUST protect any endpoint utilizing it against brute force attacks.

2.4.2. Other Authentication Methods

The authorization server MAY support any suitable authentication scheme matching its security requirements. When using other authentication methods, the authorization server MUST define a mapping between the client identifier (registration record) and authentication scheme.

Some additional authentication methods such as mTLS [RFC8705] and Private Key JWT [RFC7523] are defined in the "OAuth Token Endpoint Authentication Methods (https://www.iana.org/assignments/oauth-parameters/oauth-parameters.xhtml#token-endpoint-auth-method)" registry, and may be useful as generic client authentication methods beyond the specific use of protecting the token endpoint.

2.5. Unregistered Clients

This specification does not require that clients be registered with the authorization server. However, the use of unregistered clients is beyond the scope of this specification and requires additional security analysis and review of its interoperability impact.

3. Protocol Endpoints

The authorization process utilizes two authorization server endpoints (HTTP resources):

* Authorization endpoint - used by the client to obtain authorization from the resource owner via user agent redirection.

* Token endpoint - used by the client to exchange an authorization grant for an access token, typically with client authentication.

As well as one client endpoint:

* Redirection endpoint - used by the authorization server to return responses containing authorization credentials to the client via the resource owner user agent.

Not every authorization grant type utilizes both endpoints. Extension grant types MAY define additional endpoints as needed.

3.1. Authorization Endpoint

The authorization endpoint is used to interact with the resource owner and obtain an authorization grant. The authorization server MUST first authenticate the resource owner. The way in which the authorization server authenticates the resource owner (e.g., username and password login, passkey, federated login, or by using an established session) is beyond the scope of this specification.

The means through which the client obtains the URL of the authorization endpoint are beyond the scope of this specification, but the URL is typically provided in the service documentation, or in the authorization server’s metadata document ([RFC8414]).

The authorization endpoint URL MUST NOT include a fragment component, and MAY include an "application/x-www-form-urlencoded" formatted query component [WHATWG.URL], which MUST be retained when adding additional query parameters.

The authorization server MUST support the use of the HTTP GET method Section 9.3.1 of [RFC9110] for the authorization endpoint and MAY support the POST method (Section 9.3.3 of [RFC9110]) as well.

The authorization server MUST ignore unrecognized request parameters sent to the authorization endpoint.

Request and response parameters defined by this specification MUST NOT be included more than once. Parameters sent without a value MUST be treated as if they were omitted from the request.

An authorization server that redirects a request potentially containing user credentials MUST avoid forwarding these user credentials accidentally (see Section 7.5.4 for details).

Cross-Origin Resource Sharing (also known as CORS) [WHATWG.CORS] MUST NOT be supported at the Authorization Endpoint as the client does not access this endpoint directly, instead the client redirects the user agent to it.

3.2. Token Endpoint

The token endpoint is used by the client to obtain an access token using a grant such as those described in Section 4 and Section 4.3.

The means through which the client obtains the URL of the token endpoint are beyond the scope of this specification, but the URL is typically provided in the service documentation and configured during development of the client, or provided in the authorization server's metadata document ([RFC8414]) and fetched programmatically at runtime.

The token endpoint URL MUST NOT include a fragment component, and MAY include an application/x-www-form-urlencoded formatted query component ([WHATWG.URL]).

The client MUST use the HTTP POST method when making requests to the token endpoint.

The authorization server MUST ignore unrecognized request parameters sent to the token endpoint.

Parameters sent without a value MUST be treated as if they were omitted from the request. Request and response parameters defined by this specification MUST NOT be included more than once.

Authorization servers that wish to support browser-based applications (applications running exclusively in client-side JavaScript without access to a supporting backend server) will need to ensure the token endpoint supports the necessary CORS ([WHATWG.CORS]) headers to allow the responses to be visible to the application. 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 application, and will also need to have the CORS headers defined to allow access. See [I-D.ietf-oauth-browser-based-apps] for further details.

3.2.1. Client Authentication

Confidential clients MUST authenticate with the authorization server as described in Section 2.4 when making requests to the token endpoint.

Client authentication is used for:

* Enforcing the binding of refresh tokens and authorization codes to the client they were issued to. Client authentication adds an additional layer of security when an authorization code is transmitted to the redirection endpoint over an insecure channel.

* Recovering from a compromised client by disabling the client or changing its credentials, thus preventing an attacker from abusing stolen refresh tokens. Changing a single set of client credentials is significantly faster than revoking an entire set of refresh tokens.

* Implementing authentication management best practices, which require periodic credential rotation. Rotation of an entire set of refresh tokens can be challenging, while rotation of a single set of client credentials is significantly easier.

3.2.2. Token Request

The client makes a request to the token endpoint by sending the following parameters using the application/x-www-form-urlencoded format per Appendix B with a character encoding of UTF-8 in the HTTP request content:

"client_id": REQUIRED, if the client is not authenticating with the authorization server as described in Section 3.2.1.

"grant_type": REQUIRED. Identifier of the grant type the client uses with the particular token request. This specification defines the values authorization_code, refresh_token, and client_credentials. The grant type determines the further parameters required or supported by the token request. The details of those grant types are defined below.

Confidential clients MUST authenticate with the authorization server as described in Section 3.2.1.

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

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

grant_type=authorization_code&code=SplxlOBeZQQYbYS6WxSbIA
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
&code_verifier=3641a2d12d66101249cdf7a79c000c1f8c05d2aafcf14bf146497bed
```

The authorization server MUST:

* require client authentication for confidential clients (or clients with other authentication requirements),

* authenticate the client if client authentication is included

Further grant type specific processing rules apply and are specified with the respective grant type.

3.2.3. Token Response

If the access token request is valid and authorized, the authorization server issues an access token and optional refresh token.

If the request client authentication failed or is invalid, the authorization server returns an error response as described in Section 3.2.4.

The authorization server issues an access token and optional refresh token by creating an HTTP response content using the application/json media type as defined by [RFC8259] with the following parameters and an HTTP 200 (OK) status code:

"access_token": REQUIRED. The access token issued by the authorization server.

"token_type": REQUIRED. The type of the access token issued as described in Section 1.4. Value is case insensitive.

"expires_in": RECOMMENDED. The lifetime in seconds of the access token. For example, the value 3600 denotes that the access token will expire in one hour from the time the response was generated. If omitted, the authorization server SHOULD provide the expiration time via other means or document the default value.

"scope": RECOMMENDED, if identical to the scope requested by the client; otherwise, REQUIRED. The scope of the access token as described by Section 1.4.1.

"refresh_token": OPTIONAL. The refresh token, which can be used to obtain new access tokens based on the grant passed in the corresponding token request.

Authorization servers SHOULD determine, based on a risk assessment and their own policies, whether to issue refresh tokens to a certain client. If the authorization server decides not to issue refresh

tokens, the client MAY obtain new access tokens by starting the OAuth flow over, for example initiating a new authorization code request. 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.

The parameters are serialized into a JavaScript Object Notation (JSON) structure by adding each parameter at the highest structure level. Parameter names and string values are included as JSON strings. Numerical values are included as JSON numbers. The order of parameters does not matter and can vary.

The authorization server MUST include the HTTP Cache-Control response header field (see Section 5.2 of [RFC9111]) with a value of no-store in any response containing tokens, credentials, or other sensitive information.

For example:

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

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

The client MUST ignore unrecognized value names in the response. The sizes of tokens and other values received from the authorization server are left undefined. The client should avoid making assumptions about value sizes. The authorization server SHOULD document the size of any value it issues.

3.2.4. Error Response

The authorization server responds with an HTTP 400 (Bad Request) status code (unless specified otherwise) and includes the following parameters with the response:

"error": REQUIRED. A single ASCII [USASCII] error code from the

following:

"invalid_request": The request is missing a required parameter, includes an unsupported parameter value (other than grant type), repeats a parameter, includes multiple credentials, utilizes more than one mechanism for authenticating the client, contains a code_verifier although no code_challenge was sent in the authorization request, or is otherwise malformed.

"invalid_client": Client authentication failed (e.g., unknown client, no client authentication included, or unsupported authentication method). The authorization server MAY return an HTTP 401 (Unauthorized) status code to indicate which HTTP authentication schemes are supported. If the client attempted to authenticate via the Authorization request header field, the authorization server MUST respond with an HTTP 401 (Unauthorized) status code and include the WWW-Authenticate response header field matching the authentication scheme used by the client.

"invalid_grant": The provided authorization grant (e.g., authorization code, resource owner credentials) or refresh token is invalid, expired, revoked, does not match the redirect URI used in the authorization request, or was issued to another client.

"unauthorized_client": The authenticated client is not authorized to use this authorization grant type.

"unsupported_grant_type": The authorization grant type is not supported by the authorization server.

"invalid_scope": The requested scope is invalid, unknown, malformed, or exceeds the scope granted by the resource owner.

Values for the error parameter MUST NOT include characters outside the set %x20-21 / %x23-5B / %x5D-7E.

"error_description": OPTIONAL. Human-readable ASCII [USASCII] text providing additional information, used to assist the client developer in understanding the error that occurred. Values for the error_description parameter MUST NOT include characters outside the set %x20-21 / %x23-5B / %x5D-7E.

"error_uri": OPTIONAL. A URI identifying a human-readable web page

with information about the error, used to provide the client developer with additional information about the error. Values for the error_uri parameter MUST conform to the URI-reference syntax and thus MUST NOT include characters outside the set %x21 / %x23-5B / %x5D-7E.

The parameters are included in the content of the HTTP response using the application/json media type as defined by [RFC7159]. The parameters are serialized into a JSON structure by adding each parameter at the highest structure level. Parameter names and string values are included as JSON strings. Numerical values are included as JSON numbers. The order of parameters does not matter and can vary.

For example:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Cache-Control: no-store

{
 "error":"invalid_request"
}
```

4. Grant Types

To request an access token, the client obtains authorization from the resource owner. This specification defines the following authorization grant types:

* authorization code

* client credentials, and

* refresh token

It also provides an extension mechanism for defining additional grant types.

4.1. Authorization Code Grant

The authorization code grant type is used to obtain both access tokens and refresh tokens.

The grant type uses the additional authorization endpoint to let the authorization server interact with the resource owner in order to get consent for resource access.

Since this is a redirect-based flow, the client must be capable of initiating the flow with the resource owner's user agent (typically a web browser) and capable of being redirected back to from the authorization server.

```
  +----------+
  | Resource |
  |   Owner  |
  +----------+
        ^
        |
        |
  +-----|----+          Client Identifier      +---------------+
  | .---+---------(1)-- & Redirect URI ------->|               |
  | |   |    |                                 |               |
  | |   '---------(2)-- User authenticates --->|               |
  | | User-  |                                 | Authorization |
  | | Agent  |                                 |     Server    |
  | |        |                                 |               |
  | |     .--------(3)-- Authorization Code ---<|               |
  +-|----|---+                                 +---------------+
    |    |                                         ^      v
    |    |                                         |      |
    ^    v                                         |      |
  +---------+                                      |      |
  |         |>---(4)-- Authorization Code ---------'      |
  |  Client |          & Redirect URI                    |
  |         |                                            |
  |         |<---(5)----- Access Token -------------------'
  +---------+       (w/ Optional Refresh Token)
```

Figure 3: Authorization Code Flow

The flow illustrated in Figure 3 includes the following steps:

(1) The client initiates the flow by directing the resource owner's user agent to the authorization endpoint. The client includes its client identifier, code challenge (derived from a generated code verifier), optional requested scope, optional local state, and a redirect URI to which the authorization server will send the user agent back once access is granted (or denied).

(2) The authorization server authenticates the resource owner (via the user agent) and establishes whether the resource owner grants or denies the client's access request.

(3) Assuming the resource owner grants access, the authorization server redirects the user agent back to the client using the redirect URI provided earlier (in the request or during client registration). The redirect URI includes an authorization code and any local state provided by the client earlier.

(4) The client requests an access token from the authorization server's token endpoint by including the authorization code received in the previous step, and including its code verifier. When making the request, the client authenticates with the authorization server if it can. The client includes the redirect URI used to obtain the authorization code for verification.

(5) The authorization server authenticates the client when possible, validates the authorization code, validates the code verifier, and ensures that the redirect URI received matches the URI used to redirect the client in step (3). If valid, the authorization server responds back with an access token and, optionally, a refresh token.

4.1.1. Authorization Request

To begin the authorization request, the client builds the authorization request URI by adding parameters to the authorization server's authorization endpoint URI. The client will eventually redirect the user agent to this URI to initiate the request.

Clients use a unique secret per authorization request to protect against authorization code injection and CSRF attacks. The client first generates this secret, which it can use at the time of redeeming the authorization code to prove that the client using the authorization code is the same client that requested it.

The client constructs the request URI by adding the following parameters to the query component of the authorization endpoint URI using the application/x-www-form-urlencoded format, per Appendix B:

"response_type": REQUIRED. The authorization endpoint supports different sets of request and response pameters. The client determines the type of flow by using a certain response_type value. This specification defines the value code, which must be used to signal that the client wants to use the authorization code flow.

Extension response types MAY contain a space-delimited (%x20) list of values, where the order of values does not matter (e.g., response type a b is the same as b a). The meaning of such composite response types is defined by their respective specifications.

Some extension response types are defined by ([OpenID]).

If an authorization request is missing the response_type parameter, or if the response type is not understood, the authorization server MUST return an error response as described in Section 4.1.2.1.

"client_id": REQUIRED. The client identifier as described in Section 2.2.

"code_challenge": REQUIRED or RECOMMENDED (see Section 7.5.1). Code challenge.

"code_challenge_method": OPTIONAL, defaults to plain if not present in the request. Code verifier transformation method is S256 or plain.

"redirect_uri": OPTIONAL if only one redirect URI is registered for this client. REQUIRED if multiple redirict URIs are registered for this client. See Section 2.3.2.

"scope": OPTIONAL. The scope of the access request as described by Section 1.4.1.

"state": OPTIONAL. 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 code_verifier is a unique high-entropy cryptographically random string generated for each authorization request, using the unreserved characters [A-Z] / [a-z] / [0-9] / "-" / "." / "_" / "~", with a minimum length of 43 characters and a maximum length of 128 characters.

The client stores the code_verifier temporarily, and calculates the code_challenge which it uses in the authorization request.

ABNF for code_verifier is as follows.

```
code-verifier = 43*128unreserved
unreserved = ALPHA / DIGIT / "-" / "." / "_" / "~"
ALPHA = %x41-5A / %x61-7A
DIGIT = %x30-39
```

Clients SHOULD use code challenge methods that do not expose the code_verifier in the authorization request. Otherwise, attackers that can read the authorization request (cf. Attacker A4 in [I-D.ietf-oauth-security-topics]) can break the security provided by this mechanism. Currently, S256 is the only such method.

NOTE: The code verifier SHOULD have enough entropy to make it impractical to guess the value. It is RECOMMENDED that the output of a suitable random number generator be used to create a 32-octet sequence. The octet sequence is then base64url-encoded to produce a 43-octet URL-safe string to use as the code verifier.

The client then creates a code_challenge derived from the code verifier by using one of the following transformations on the code verifier:

```
S256
  code_challenge = BASE64URL-ENCODE(SHA256(ASCII(code_verifier)))

plain
  code_challenge = code_verifier
```

If the client is capable of using S256, it MUST use S256, as S256 is Mandatory To Implement (MTI) on the server. Clients are permitted to use plain only if they cannot support S256 for some technical reason, for example constrained environments that do not have a hashing function available, and know via out-of-band configuration or via Authorization Server Metadata ([RFC8414]) that the server supports plain.

ABNF for code_challenge is as follows.

```
code-challenge = 43*128unreserved
unreserved = ALPHA / DIGIT / "-" / "." / "_" / "~"
ALPHA = %x41-5A / %x61-7A
DIGIT = %x30-39
```

The properties code_challenge and code_verifier are adopted from the OAuth 2.0 extension known as "Proof-Key for Code Exchange", or PKCE ([RFC7636]) where this technique was originally developed.

Authorization servers MUST support the code_challenge and code_verifier parameters.

Clients MUST use code_challenge and code_verifier and authorization servers MUST enforce their use except under the conditions described in Section 7.5.1. In this case, using and enforcing code_challenge and code_verifier as described in the following is still RECOMMENDED.

The state and scope parameters SHOULD NOT include sensitive client or resource owner information in plain text, as they can be transmitted over insecure channels or stored insecurely.

The client directs the resource owner to the constructed URI using an HTTP redirection, or by other means available to it via the user agent.

For example, the client directs the user agent to make the following HTTP request (with extra line breaks for display purposes only):

```
GET /authorize?response_type=code&client_id=s6BhdRkqt3&state=xyz
    &redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
    &code_challenge=6fdkQaPm51l13DSukcAH3Mdx7_ntecHYd1vi3n0hMZY
    &code_challenge_method=S256 HTTP/1.1
Host: server.example.com
```

The authorization server validates the request to ensure that all required parameters are present and valid.

In particular, the authorization server MUST validate the redirect_uri in the request if present, ensuring that it matches one of the registered redirect URIs previously established during client registration (Section 2). When comparing the two URIs the authorization server MUST ensure that the two URIs are equal, see [RFC3986], Section 6.2.1, Simple String Comparison, for details.

If the request is valid, the authorization server authenticates the resource owner and obtains an authorization decision (by asking the resource owner or by establishing approval via other means).

When a decision is established, the authorization server directs the user agent to the provided client redirect URI using an HTTP redirection response, or by other means available to it via the user agent.

4.1.2. Authorization Response

If the resource owner grants the access request, the authorization server issues an authorization code and delivers it to the client by adding the following parameters to the query component of the redirect URI using the application/x-www-form-urlencoded format, per Appendix B:

"code": REQUIRED. The authorization code is generated by the

authorization server and opaque to the client. The authorization code MUST expire shortly after it is issued to mitigate the risk of leaks. A maximum authorization code lifetime of 10 minutes is RECOMMENDED. The authorization code is bound to the client identifier, code challenge and redirect URI.

"state": REQUIRED if the state parameter was present in the client authorization request. The exact value received from the client.

"iss": OPTIONAL. The identifier of the authorization server which the client can use to prevent mix-up attacks, if the client interacts with more than one authorization server. See Section 7.13 and [RFC9207] for additional details on when this parameter is necessary, and how the client can use it to prevent mix-up attacks.

For example, the authorization server redirects the user agent by sending the following HTTP response:

```
HTTP/1.1 302 Found
Location: https://client.example.com/cb?code=SplxlOBeZQQYbYS6WxSbIA
          &state=xyz&iss=https%3A%2F%2Fauthorization-server.example.com
```

The client MUST ignore unrecognized response parameters. The authorization code string size is left undefined by this specification. The client should avoid making assumptions about code value sizes. The authorization server SHOULD document the size of any value it issues.

The authorization server MUST associate the code_challenge and code_challenge_method values with the issued authorization code so the code challenge can be verified later.

The exact method that the server uses to associate the code_challenge with the issued code is out of scope for this specification. The code challenge could be stored on the server and associated with the code there. The code_challenge and code_challenge_method values may be stored in encrypted form in the code itself, but the server MUST NOT include the code_challenge value in a response parameter in a form that entities other than the AS can extract.

Clients MUST prevent injection (replay) of authorization codes into the authorization response by attackers. Using code_challenge and code_verifier prevents injection of authorization codes since the authorization server will reject a token request with a mismatched code_verifier. See Section 7.5.1 for more details.

4.1.2.1. Error Response

If the request fails due to a missing, invalid, or mismatching redirect URI, or if the client identifier is missing or invalid, the authorization server SHOULD inform the resource owner of the error and MUST NOT automatically redirect the user agent to the invalid redirect URI.

An AS MUST reject requests without a code_challenge from public clients, and MUST reject such requests from other clients unless there is reasonable assurance that the client mitigates authorization code injection in other ways. See Section 7.5.1 for details.

If the server does not support the requested code_challenge_method transformation, the authorization endpoint MUST return the authorization error response with error value set to invalid_request. The error_description or the response of error_uri SHOULD explain the nature of error, e.g., transform algorithm not supported.

If the resource owner denies the access request or if the request fails for reasons other than a missing or invalid redirect URI, the authorization server informs the client by adding the following parameters to the query component of the redirect URI using the application/x-www-form-urlencoded format, per Appendix B:

"error": REQUIRED. A single ASCII [USASCII] error code from the following:

"invalid_request": The request is missing a required parameter, includes an invalid parameter value, includes a parameter more than once, or is otherwise malformed.

"unauthorized_client": The client is not authorized to request an authorization code using this method.

"access_denied": The resource owner or authorization server denied the request.

"unsupported_response_type": The authorization server does not support obtaining an authorization code using this method.

"invalid_scope": The requested scope is invalid, unknown, or malformed.

"server_error": The authorization server encountered an

unexpected condition that prevented it from fulfilling the request. (This error code is needed because a 500 Internal Server Error HTTP status code cannot be returned to the client via an HTTP redirect.)

"temporarily_unavailable": The authorization server is currently unable to handle the request due to a temporary overloading or maintenance of the server. (This error code is needed because a 503 Service Unavailable HTTP status code cannot be returned to the client via an HTTP redirect.)

Values for the error parameter MUST NOT include characters outside the set %x20-21 / %x23-5B / %x5D-7E.

"error_description": OPTIONAL. Human-readable ASCII [USASCII] text providing additional information, used to assist the client developer in understanding the error that occurred. Values for the error_description parameter MUST NOT include characters outside the set %x20-21 / %x23-5B / %x5D-7E.

"error_uri": OPTIONAL. A URI identifying a human-readable web page with information about the error, used to provide the client developer with additional information about the error. Values for the error_uri parameter MUST conform to the URI-reference syntax and thus MUST NOT include characters outside the set %x21 / %x23-5B / %x5D-7E.

"state": REQUIRED if a state parameter was present in the client authorization request. The exact value received from the client.

"iss": OPTIONAL. The identifier of the authorization server. See Section 4.1.2 above for details.

For example, the authorization server redirects the user agent by sending the following HTTP response:

```
HTTP/1.1 302 Found
Location: https://client.example.com/cb?error=access_denied
          &state=xyz&iss=https%3A%2F%2Fauthorization-server.example.com
```

4.1.3. Token Endpoint Extension

The authorization grant type is identified at the token endpoint with the grant_type value of authorization_code.

If this value is set, the following additional token request parameters beyond Section 3.2.2 are required:

"code": REQUIRED. The authorization code received from the authorization server.

"code_verifier": REQUIRED, if the code_challenge parameter was included in the authorization request. MUST NOT be used otherwise. The original code verifier string.

The authorization server MUST return an access token only once for a given authorization code.

If a second valid token request is made with the same authorization code as a previously successful token request, the authorization server MUST deny the request and SHOULD revoke (when possible) all access tokens and refresh tokens previously issued based on that authorization code. See Section 7.5.3 for further details.

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

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

grant_type=authorization_code
&code=SplxlOBeZQQYbYS6WxSbIA
&code_verifier=3641a2d12d66101249cdf7a79c000c1f8c05d2aafcf14bf146497bed
```

In addition to the processing rules in Section 3.2.2, the authorization server MUST:

* ensure that the authorization code was issued to the authenticated confidential client, or if the client is public, ensure that the code was issued to client_id in the request,

* verify that the authorization code is valid,

* verify that the code_verifier parameter is present if and only if a code_challenge parameter was present in the authorization request,

* if a code_verifier is present, verify the code_verifier by calculating the code challenge from the received code_verifier and comparing it with the previously associated code_challenge, after first transforming it according to the code_challenge_method method specified by the client, and

* If there was no code_challenge in the authorization request associated with the authorization code in the token request, the authorization server MUST reject the token request.

See Section 10.2 for details on backwards compatibility with OAuth 2.0 clients regarding the redirect_uri parameter in the token request.

4.2. Client Credentials Grant

The client can request an access token using only its client credentials (or other supported means of authentication) when the client is requesting access to the protected resources under its control, or those of another resource owner that have been previously arranged with the authorization server (the method of which is beyond the scope of this specification).

The client credentials grant type MUST only be used by confidential clients.

```
     +---------+                                  +---------------+
     |         |                                  |               |
     |         |>--(1)- Client Authentication --->| Authorization |
     | Client  |                                  |     Server    |
     |         |<--(2)---- Access Token ---------<|               |
     |         |                                  |               |
     +---------+                                  +---------------+
```

Figure 4: Client Credentials Grant

The use of the client credentials grant illustrated in Figure 4 includes the following steps:

(1) The client authenticates with the authorization server and requests an access token from the token endpoint.

(2) The authorization server authenticates the client, and if valid, issues an access token.

4.2.1. Token Endpoint Extension

The authorization grant type is identified at the token endpoint with the grant_type value of client_credentials.

If this value is set, the following additional token request parameters beyond Section 3.2.2 are supported:

"scope": OPTIONAL. The scope of the access request as described by

Section 1.4.1.

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

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

grant_type=client_credentials
```

The authorization server MUST authenticate the client.

4.3. Refresh Token Grant

The refresh token is a credential issued by the authorization server to a client, which can be used to obtain new (fresh) access tokens based on an existing grant. The client uses this option either because the previous access token has expired or the client previously obtained an access token with a scope more narrow than approved by the respective grant and later requires an access token with a different scope under the same grant.

Refresh tokens MUST be kept confidential in transit and storage, and shared only among the authorization server and the client to whom the refresh tokens were issued. The authorization server MUST maintain the binding between a refresh token and the client to whom it was issued.

The authorization server MUST verify the binding between the refresh token and client identity whenever the client identity can be authenticated. When client authentication is not possible, the authorization server SHOULD issue sender-constrained refresh tokens or use refresh token rotation as described in Section 4.3.1.

The authorization server MUST ensure that refresh tokens cannot be generated, modified, or guessed to produce valid refresh tokens by unauthorized parties.

4.3.1. Token Endpoint Extension

The authorization grant type is identified at the token endpoint with the grant_type value of refresh_token.

If this value is set, the following additional parameters beyond Section 3.2.2 are required/supported:

"refresh_token": REQUIRED. The refresh token issued to the client.

"scope": OPTIONAL. The scope of the access request as described by Section 1.4.1. The requested scope MUST NOT include any scope not originally granted by the resource owner, and if omitted is treated as equal to the scope originally granted by the resource owner.

Because refresh tokens are typically long-lasting credentials used to request additional access tokens, the refresh token is bound to the client to which it was issued. Confidential clients MUST authenticate with the authorization server as described in Section 3.2.1.

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

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

grant_type=refresh_token&refresh_token=tGzv3JOkF0XG5Qx2TlKWIA
```

In addition to the processing rules in Section 3.2.2, the authorization server MUST:

* if client authentication is included in the request, ensure that the refresh token was issued to the authenticated client, OR if a client_id is included in the request, ensure the refresh token was issued to the matching client

* validate that the grant corresponding to this refresh token is still active

* validate the refresh token

Authorization servers 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, e.g. by utilizing DPoP [RFC9449] or mTLS [RFC8705].

* _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 as well as the access authorization grant associated with it. 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.

4.3.2. Refresh Token Response

If valid and authorized, the authorization server issues an access token as described in Section 3.2.3.

The authorization server MAY issue a new refresh token, in which case the client MUST discard the old refresh token and replace it with the new refresh token.

4.3.3. Refresh Token Recommendations

The authorization server MAY revoke the old refresh token after issuing a new refresh token to the client. If a new refresh token is issued, the refresh token scope MUST be identical to that of the refresh token included by the client in the request.

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 new 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).

4.4. Extension Grants

The client uses an extension grant type by specifying the grant type using an absolute URI (defined by the authorization server) as the value of the grant_type parameter of the token endpoint, and by adding any additional parameters necessary.

For example, to request an access token using the Device Authorization Grant as defined by [RFC8628] after the user has authorized the client on a separate device, the client makes the following HTTP request (with extra line breaks for display purposes only):

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

grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Adevice_code
&device_code=GmRhmhcxhwEzkoEqiMEg_DnyEysNkuNhszIySk9eS
&client_id=C409020731
```

If the access token request is valid and authorized, the authorization server issues an access token and optional refresh token as described in Section 3.2.3. If the request failed client authentication or is invalid, the authorization server returns an error response as described in Section 3.2.4.

5. Resource Requests

The client accesses protected resources by presenting an access token to the resource server. The resource server MUST validate the access token and ensure that it has not expired and that its scope covers the requested resource. The methods used by the resource server to validate the access token are beyond the scope of this specification, but generally involve an interaction or coordination between the resource server and the authorization server. For example, when the resource server and authorization server are colocated or are part of the same system, they may share a database or other storage; when the two components are operated independently, they may use Token Introspection [RFC7662] or a structured access token format such as a JWT [RFC9068].

5.1. Bearer Token Requests

This section defines two methods of sending Bearer tokens in resource requests to resource servers. Clients MUST use one of the two methods defined below, and MUST NOT use more than one method to transmit the token in each request.

In particular, clients MUST NOT send the access token in a URI query parameter, and resource servers MUST ignore access tokens in a URI query parameter.

5.1.1. Authorization Request Header Field

When sending the access token in the Authorization request header field defined by HTTP/1.1 [RFC7235], the client uses the Bearer scheme to transmit the access token.

For example:

```
GET /resource HTTP/1.1
Host: server.example.com
Authorization: Bearer mF_9.B5f-4.1JqM
```

The syntax of the Authorization header field for this scheme follows the usage of the Basic scheme defined in Section 2 of [RFC2617]. Note that, as with Basic, it does not conform to the generic syntax defined in Section 1.2 of [RFC2617] but is compatible with the general authentication framework in HTTP 1.1 Authentication [RFC7235], although it does not follow the preferred practice outlined therein in order to reflect existing deployments. The syntax for Bearer credentials is as follows:

```
token68    = 1*( ALPHA / DIGIT /
                 "-" / "." / "_" / "~" / "+" / "/" ) *"="
credentials = "Bearer" 1*SP token68
```

Clients SHOULD make authenticated requests with a bearer token using the Authorization request header field with the Bearer HTTP authorization scheme. Resource servers MUST support this method.

5.1.2. Form-Encoded Content Parameter

When sending the access token in the HTTP request content, the client adds the access token to the request content using the access_token parameter. The client MUST NOT use this method unless all of the following conditions are met:

* The HTTP request includes the Content-Type header field set to application/x-www-form-urlencoded.

* The content follows the encoding requirements of the application/x-www-form-urlencoded content-type as defined by the URL Living Standard [WHATWG.URL].

* The HTTP request content is single-part.

* The content to be encoded in the request MUST consist entirely of ASCII [USASCII] characters.

* The HTTP request method is one for which the content has defined semantics. In particular, this means that the GET method MUST NOT be used.

The content MAY include other request-specific parameters, in which case the access_token parameter MUST be properly separated from the request-specific parameters using & character(s) (ASCII code 38).

For example, the client makes the following HTTP request using transport-layer security:

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

access_token=mF_9.B5f-4.1JqM
```

The application/x-www-form-urlencoded method SHOULD NOT be used except in application contexts where participating clients do not have access to the Authorization request header field. Resource servers MAY support this method.

5.2. Access Token Validation

After receiving the access token, the resource server MUST check that the access token is not yet expired, is authorized to access the requested resource, was issued with the appropriate scope, and meets other policy requirements of the resource server to access the protected resource.

Access tokens generally fall into two categories: reference tokens or self-encoded tokens. Reference tokens can be validated by querying the authorization server or looking up the token in a token database, whereas self-encoded tokens contain the authorization information in an encrypted and/or signed string which can be extracted by the resource server.

A standardized method to query the authorization server to check the validity of an access token is defined in Token Introspection ([RFC7662]).

A standardized method of encoding information in a token string is defined in JWT Profile for Access Tokens ([RFC9068]).

See Section 7.1 for additional considerations around creating and validating access tokens.

5.3. Error Response

If a resource access request fails, the resource server SHOULD inform the client of the error. The details of the error response is determined by the particular token type, such as the description of Bearer tokens in Section 5.3.2.

5.3.1. The WWW-Authenticate Response Header Field

If the protected resource request does not include authentication credentials or does not contain an access token that enables access to the protected resource, the resource server MUST include the HTTP WWW-Authenticate response header field; it MAY include it in response to other conditions as well. The WWW-Authenticate header field uses the framework defined by HTTP/1.1 [RFC7235].

All challenges for this token type MUST use the auth-scheme value Bearer. This scheme MUST be followed by one or more auth-param values. The auth-param attributes used or defined by this specification for this token type are as follows. Other auth-param attributes MAY be used as well.

"realm": A realm attribute MAY be included to indicate the scope of protection in the manner described in HTTP/1.1 [RFC7235]. The realm attribute MUST NOT appear more than once.

"scope": The scope attribute is defined in Section 1.4.1. The scope attribute is a space-delimited list of case-sensitive scope values indicating the required scope of the access token for accessing the requested resource. scope values are implementation defined; there is no centralized registry for them; allowed values are defined by the authorization server. The order of scope values is not significant. In some cases, the scope value will be used when requesting a new access token with sufficient scope of access to utilize the protected resource. Use of the scope attribute is OPTIONAL. The scope attribute MUST NOT appear more than once. The scope value is intended for programmatic use and is not meant to be displayed to end-users.

Two example scope values follow; these are taken from the OpenID Connect [OpenID.Messages] and the Open Authentication Technology Committee (OATC) Online Multimedia Authorization Protocol [OMAP] OAuth 2.0 use cases, respectively:

```
scope="openid profile email"
scope="urn:example:channel=HBO&urn:example:rating=G,PG-13"
```

"error": If the protected resource request included an access token and failed authentication, the resource server SHOULD include the error attribute to provide the client with the reason why the access request was declined. The parameter value is described in Section 5.3.2.

"error_description": The resource server MAY include the error_description attribute to provide developers a human-readable explanation that is not meant to be displayed to end-users.

"error_uri": The resource server MAY include the error_uri attribute with an absolute URI identifying a human-readable web page explaining the error.

The error, error_description, and error_uri attributes MUST NOT appear more than once.

Values for the scope attribute (specified in Appendix A.4) MUST NOT include characters outside the set %x21 / %x23-5B / %x5D-7E for representing scope values and %x20 for delimiters between scope values. Values for the error and error_description attributes (specified in Appendix A.7 and Appendix A.8) MUST NOT include characters outside the set %x20-21 / %x23-5B / %x5D-7E. Values for the error_uri attribute (specified in Appendix A.9 of) MUST conform to the URI-reference syntax and thus MUST NOT include characters outside the set %x21 / %x23-5B / %x5D-7E.

5.3.2. Error Codes

When a request fails, the resource server responds using the appropriate HTTP status code (typically, 400, 401, 403, or 405) and includes one of the following error codes in the response:

"invalid_request": The request is missing a required parameter, includes an unsupported parameter or parameter value, repeats the same parameter, uses more than one method for including an access token, or is otherwise malformed. The resource server SHOULD respond with the HTTP 400 (Bad Request) status code.

"invalid_token": The access token provided is expired, revoked, malformed, or invalid for other reasons. The resource SHOULD respond with the HTTP 401 (Unauthorized) status code. The client MAY request a new access token and retry the protected resource request.

"insufficient_scope": The request requires higher privileges (scopes) than provided by the scopes granted to the client and represented by the access token. The resource server SHOULD respond with the HTTP 403 (Forbidden) status code and MAY include the scope attribute with the scope necessary to access the protected resource.

Extensions may define additional error codes or specify additional circumstances in which the above error codes are retured.

If the request lacks any authentication information (e.g., the client was unaware that authentication is necessary or attempted using an unsupported authentication method), the resource server SHOULD NOT include an error code or other error information.

For example:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="example"
```

And in response to a protected resource request with an authentication attempt using an expired access token:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="example",
                  error="invalid_token",
                  error_description="The access token expired"
```

6. Extensibility

6.1. Defining Access Token Types

Access token types can be defined in one of two ways: registered in the Access Token Types registry (following the procedures in Section 11.1 of [RFC6749]), or by using a unique absolute URI as its name.

6.1.1. Registered Access Token Types

[RFC6750] establishes a common registry in Section 11.4 of [RFC6749] for error values to be shared among OAuth token authentication schemes.

New authentication schemes designed primarily for OAuth token authentication SHOULD define a mechanism for providing an error status code to the client, in which the error values allowed are registered in the error registry established by this specification.

Such schemes MAY limit the set of valid error codes to a subset of the registered values. If the error code is returned using a named parameter, the parameter name SHOULD be error.

Other schemes capable of being used for OAuth token authentication, but not primarily designed for that purpose, MAY bind their error values to the registry in the same manner.

New authentication schemes MAY choose to also specify the use of the error_description and error_uri parameters to return error information in a manner parallel to their usage in this specification.

Type names MUST conform to the type-name ABNF. If the type definition includes a new HTTP authentication scheme, the type name SHOULD be identical to the HTTP authentication scheme name (as defined by [RFC2617]). The token type example is reserved for use in examples.

```
type-name  = 1*name-char
name-char  = "-" / "." / "_" / DIGIT / ALPHA
```

6.1.2. Vendor-Specific Access Token Types

Types utilizing a URI name SHOULD be limited to vendor-specific implementations that are not commonly applicable, and are specific to the implementation details of the resource server where they are used.

All other types MUST be registered.

6.2. Defining New Endpoint Parameters

New request or response parameters for use with the authorization endpoint or the token endpoint are defined and registered in the OAuth Parameters registry following the procedure in Section 11.2 of [RFC6749].

Parameter names MUST conform to the param-name ABNF, and parameter values syntax MUST be well-defined (e.g., using ABNF, or a reference to the syntax of an existing parameter).

```
param-name  = 1*name-char
name-char   = "-" / "." / "_" / DIGIT / ALPHA
```

Unregistered vendor-specific parameter extensions that are not commonly applicable and that are specific to the implementation details of the authorization server where they are used SHOULD utilize a vendor-specific prefix that is not likely to conflict with other registered values (e.g., begin with 'companyname_').

6.3. Defining New Authorization Grant Types

New authorization grant types can be defined by assigning them a unique absolute URI for use with the grant_type parameter. If the extension grant type requires additional token endpoint parameters, they MUST be registered in the OAuth Parameters registry as described by Section 11.2 of [RFC6749].

6.4. Defining New Authorization Endpoint Response Types

New response types for use with the authorization endpoint are defined and registered in the Authorization Endpoint Response Types registry following the procedure in Section 11.3 of [RFC6749]. Response type names MUST conform to the response-type ABNF.

```
response-type  = response-name *( SP response-name )
response-name  = 1*response-char
response-char  = "_" / DIGIT / ALPHA
```

If a response type contains one or more space characters (%x20), it is compared as a space-delimited list of values in which the order of values does not matter. Only one order of values can be registered, which covers all other arrangements of the same set of values.

For example, an extension can define and register the code other_token response type. Once registered, the same combination cannot be registered as other_token code, but both values can be used to denote the same response type.

6.5. Defining Additional Error Codes

In cases where protocol extensions (i.e., access token types, extension parameters, or extension grant types) require additional error codes to be used with the authorization code grant error response (Section 4.1.2.1), the token error response (Section 3.2.4), or the resource access error response (Section 5.3), such error codes MAY be defined.

Extension error codes MUST be registered (following the procedures in Section 11.4 of [RFC6749]) if the extension they are used in conjunction with is a registered access token type, a registered endpoint parameter, or an extension grant type. Error codes used with unregistered extensions MAY be registered.

Error codes MUST conform to the error ABNF and SHOULD be prefixed by an identifying name when possible. For example, an error identifying an invalid value set to the extension parameter example SHOULD be named example_invalid.

```
error      = 1*error-char
error-char = %x20-21 / %x23-5B / %x5D-7E
```

7. Security Considerations

As a flexible and extensible framework, OAuth's security considerations depend on many factors. The following sections provide implementers with security guidelines focused on the three client profiles described in Section 2.1: web application, browser-based application, and native application.

A comprehensive OAuth security model and analysis, as well as background for the protocol design, is provided by [RFC6819] and [I-D.ietf-oauth-security-topics].

7.1. Access Token Security Considerations

7.1.1. Security Threats

The following list presents several common threats against protocols utilizing some form of tokens. This list of threats is based on NIST Special Publication 800-63 [NIST800-63].

7.1.1.1. Access token manufacture/modification

An attacker may generate a bogus access token or modify the token contents (such as the authentication or attribute statements) of an existing token, causing the resource server to grant inappropriate access to the client. For example, an attacker may modify the token to extend the validity period; a malicious client may modify the assertion to gain access to information that they should not be able to view.

7.1.1.2. Access token disclosure

Access tokens may contain authentication and attribute statements that include sensitive information.

7.1.1.3. Access token redirect

An attacker uses an access token generated for consumption by one resource server to gain access to a different resource server that mistakenly believes the token to be for it.

7.1.1.4. Access token replay

An attacker attempts to use an access token that has already been used with that resource server in the past.

7.1.2. Threat Mitigation

A large range of threats can be mitigated by protecting the contents of the access token by using a digital signature.

Alternatively, a bearer token can contain a reference to authorization information, rather than encoding the information directly. Using a reference may require an extra interaction between a resource server and authorization server to resolve the reference to the authorization information. The mechanics of such an interaction are not defined by this specification, but one such mechanism is defined in Token Introspection [RFC7662].

This document does not specify the encoding or the contents of the access token; hence, detailed recommendations about the means of guaranteeing access token integrity protection are outside the scope of this specification. One example of an encoding and signing mechanism for access tokens is described in JSON Web Token Profile for Access Tokens [RFC9068].

To deal with access token redirects, it is important for the authorization server to include the identity of the intended recipients (the audience), typically a single resource server (or a list of resource servers), in the token. Restricting the use of the token to a specific scope is also RECOMMENDED.

If cookies are transmitted without TLS protection, any information contained in them is at risk of disclosure. Therefore, Bearer tokens MUST NOT be stored in cookies that can be sent in the clear, as any information in them is at risk of disclosure. See "HTTP State Management Mechanism" [RFC6265] for security considerations about cookies.

In some deployments, including those utilizing load balancers, the TLS connection to the resource server terminates prior to the actual server that provides the resource. This could leave the token unprotected between the front-end server where the TLS connection

terminates and the back-end server that provides the resource. In such deployments, sufficient measures MUST be employed to ensure confidentiality of the access token between the front-end and back-end servers; encryption of the token is one such possible measure.

7.1.3. Summary of Recommendations

7.1.3.1. Safeguard bearer tokens

Client implementations MUST ensure that bearer tokens are not leaked to unintended parties, as they will be able to use them to gain access to protected resources. This is the primary security consideration when using bearer tokens and underlies all the more specific recommendations that follow.

7.1.3.2. Validate TLS certificate chains

The client MUST validate the TLS certificate chain when making requests to protected resources. Failing to do so may enable DNS hijacking attacks to steal the token and gain unintended access.

7.1.3.3. Always use TLS (https)

Clients MUST always use TLS (https) or equivalent transport security when making requests with bearer tokens. Failing to do so exposes the token to numerous attacks that could give attackers unintended access.

7.1.3.4. Don’t store bearer tokens in HTTP cookies

Implementations MUST NOT store bearer tokens within cookies that can be sent in the clear (which is the default transmission mode for cookies). Implementations that do store bearer tokens in cookies MUST take precautions against cross-site request forgery.

7.1.3.5. Issue short-lived bearer tokens

Authorization servers SHOULD issue short-lived bearer tokens, particularly when issuing tokens to clients that run within a web browser or other environments where information leakage may occur. Using short-lived bearer tokens can reduce the impact of them being leaked.

7.1.3.6. Issue scoped bearer tokens

Authorization servers SHOULD issue bearer tokens that contain an audience restriction, scoping their use to the intended relying party or set of relying parties.

7.1.3.7. Don’t pass bearer tokens in page URLs

Bearer tokens MUST NOT be passed in page URLs (for example, as query string parameters). Instead, bearer tokens SHOULD be passed in HTTP message headers or message bodies for which confidentiality measures are taken. Browsers, web servers, and other software may not adequately secure URLs in the browser history, web server logs, and other data structures. If bearer tokens are passed in page URLs, attackers might be able to steal them from the history data, logs, or other unsecured locations.

7.1.4. 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 this document and [RFC8707], 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 and authorization_details as specified in [RFC9396] to determine those resources and/or actions.

7.2. Client Authentication

Depending on the overall process of client registration and credential lifecycle management, this may affect the confidence an authorization server has in a particular client.

For example, authentication of a dynamically registered client does not prove the identity of the client, it only ensures that repeated requests to the authorization server were made from the same client instance. Such clients may be limited in terms of which scopes they are allowed to request, or may have other limitations such as shorter token lifetimes.

In contrast, if there is a registered application whose developer's identity was verified, who signed a contract and is issued a client secret that is only used in a secure backend service, the authorization server might allow this client to request more sensitive scopes or to be issued longer-lasting tokens.

7.3. Client Impersonation

If a confidential client has its credentials stolen, a malicious client can impersonate the client and obtain access to protected resources.

The authorization server SHOULD enforce explicit resource owner authentication and provide the resource owner with information about the client and the requested authorization scope and lifetime. It is up to the resource owner to review the information in the context of the current client and to authorize or deny the request.

The authorization server SHOULD NOT process repeated authorization requests automatically (without active resource owner interaction) without authenticating the client or relying on other measures to ensure that the repeated request comes from the original client and not an impersonator.

7.3.1. Impersonation of Native Apps

As stated above, the authorization server SHOULD NOT process authorization requests automatically without user consent or interaction, except when the identity of the client can be assured. This includes the case where the user has previously approved an authorization request for a given client ID -- unless the identity of the client can be proven, the request SHOULD be processed as if no previous request had been approved.

Measures such as claimed https scheme redirects MAY be accepted by authorization servers as identity proof. Some operating systems may offer alternative platform-specific identity features that MAY be accepted, as appropriate.

7.3.2. Access Token Privilege Restriction

The client SHOULD request access tokens with the minimal scope necessary. The authorization server SHOULD take the client identity into account when choosing how to honor the requested scope and MAY issue an access token with fewer scopes than requested.

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 [RFC8707], respectively, to determine the resource server they want to access.

7.4. Client Impersonating Resource Owner

Resource servers may make access control decisions based on the identity of a resource owner for which an access token was issued, or based on the identity of a client in the client credentials grant. If both options are possible, depending on the details of the implementation, a client’s identity may be mistaken for the identity of a resource owner. For example, if a client is able to choose its own client_id during registration with the authorization server, a malicious client may set it to a value identifying an end-user (e.g., a sub value if OpenID Connect is used). If the resource server cannot properly distinguish between access tokens issued to clients and access tokens issued to end-users, the client may then be able to access resource of the end-user.

If the authorization server has a common namespace for client IDs and user identifiers, causing the resource server to be unable to distinguish an access token authorized by a resource owner from an access token authorized by a client itself, authorization servers SHOULD NOT allow clients to influence their client_id 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 the two types of access tokens.

7.5. Authorization Code Security Considerations

7.5.1. Authorization Code Injection

Authorization code injection is an attack where the client receives an authorization code from the attacker in its redirect URI instead of the authorization code from the legitimate authorization server. Without protections in place, there is no mechanism by which the client can know that the attack has taken place. Authorization code injection can lead to both the attacker obtaining access to a victim's account, as well as a victim accidentally gaining access to the attacker's account.

7.5.2. Countermeasures

To prevent injection of authorization codes into the client, using code_challenge and code_verifier is REQUIRED for clients, and authorization servers MUST enforce their use, unless both of the following criteria are met:

* The client is a confidential client.

* In the specific deployment and the specific request, there is reasonable assurance by the authorization server that the client implements the OpenID Connect nonce mechanism properly.

In this case, using and enforcing code_challenge and code_verifier is still RECOMMENDED.

The code_challenge or OpenID Connect nonce value MUST be transaction-specific and securely bound to the client and the user agent in which the transaction was started. If a transaction leads to an error, fresh values for code_challenge or nonce MUST be chosen.

Relying on the client to validate the OpenID Connect nonce parameter means the authorization server has no way to confirm that the client has actually protected itself against authorization code injection attacks. If an attacker is able to inject an authorization code into

a client, the client would still exchange the injected authorization code and obtain tokens, and would only later reject the ID token after validating the nonce and seeing that it doesn't match. In contrast, the authorization server enforcing the code_challenge and code_verifier parameters provides a higher security outcome, since the authorization server is able to recognize the authorization code injection attack pre-emtpively and avoid issuing any tokens in the first place.

Historic note: Although PKCE [RFC7636] (where the code_challenge and code_verifier parameters were created) was originally designed as a mechanism to protect native apps from authorization code exfiltration attacks, all kinds of OAuth clients, including web applications and other confidential clients, are susceptible to authorziation code injection attacks, which are solved by the code_challenge and code_verifier mechanism.

7.5.3. Reuse of Authorization Codes

Several types of attacks are possible if authorization codes are able to be used more than once.

As described in Section 4.1.3, the authorization server must reject a token request and revoke any issued tokens when receiving a second valid request with an authorization code that has already been used to issue an access token. If an attacker is able to exfiltrate an authorization code and use it before the legitimate client, the attacker will obtain the access token and the legitimate client will not. Revoking any issued tokens means the attacker's tokens will then be revoked, stopping the attack from proceeding any further.

However, the authorization server should only revoke issued tokens if the request containing the authorization code is also valid, including any other parameters such as the code_verifier and client authentication. The authorization server SHOULD NOT revoke any issued tokens when receiving a replayed authorization code that contains invalid parameters. If it were to do so, this would create a denial of service opportunity for an attacker who is able to obtain an authorization code but unable to obtain the client authentication or code_verifier by sending an invalid authorization code request before the legitimate client and thereby revoking the legitimate client's tokens once it makes the valid request.

7.5.4. HTTP 307 Redirect

An authorization server which redirects a request that potentially contains user credentials MUST NOT use the 307 status code (Section 15.4.8 of [RFC9110]) for redirection. If an HTTP redirection (and not, for example, JavaScript) is used for such a request, AS SHOULD use the status code 303 ("See Other").

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

If the status code 307 were used for redirection, the user agent would send the user credentials via a POST request 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 Section 15.4.8 of [RFC9110]. 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 content.

In HTTP [RFC9110], only the status code 303 unambigiously enforces rewriting the HTTP POST request to an HTTP GET request. For all other status codes, including the popular 302, user agents can opt not to rewrite POST to GET requests and therefore reveal the user credentials to the client. (In practice, however, most user agents will only show this behaviour for 307 redirects.)

7.6. Ensuring Endpoint Authenticity

The risk related to man-in-the-middle attacks is mitigated by the mandatory use of channel security mechanisms such as [RFC8446] for communicating with the Authorization and Token Endpoints. See Section 1.5 for further details.

7.7. Credentials-Guessing Attacks

The authorization server MUST prevent attackers from guessing access tokens, authorization codes, refresh tokens, resource owner passwords, and client credentials.

The probability of an attacker guessing generated tokens (and other credentials not intended for handling by end-users) MUST be less than or equal to 2^(-128) and SHOULD be less than or equal to 2^(-160).

The authorization server MUST utilize other means to protect credentials intended for end-user usage.

7.8. Phishing Attacks

Wide deployment of this and similar protocols may cause end-users to become inured to the practice of being redirected to websites where they are asked to enter their passwords. If end-users are not careful to verify the authenticity of these websites before entering their credentials, it will be possible for attackers to exploit this practice to steal resource owners' passwords.

Service providers should attempt to educate end-users about the risks phishing attacks pose and should provide mechanisms that make it easy for end-users to confirm the authenticity of their sites. Client developers should consider the security implications of how they interact with the user agent (e.g., external, embedded), and the ability of the end-user to verify the authenticity of the authorization server.

See Section 1.5 for further details on mitigating the risk of phishing attacks.

7.9. 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).

The traditional countermeasure is that clients pass a random value, also known as a CSRF Token, in the state parameter that links the request to the redirect URI to the user agent session as described. This countermeasure is described in detail in [RFC6819], Section 5.3.5. The same protection is provided by the code_verifier parameter or the OpenID Connect nonce value.

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

* Clients MUST ensure that the AS supports the code_challenge_method intended to be used by the client. If an authorization server does not support the requested method, state or nonce MUST be used for CSRF protection instead.

* 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 supported code challenge methods either via AS metadata according to [RFC8414] or provide a deployment-specific way to ensure or determine support.

7.10. Clickjacking

As described in Section 4.4.1.9 of [RFC6819], the authorization request is susceptible to clickjacking attacks, also called user interface redressing. In such an attack, an attacker embeds the authorization endpoint user interface in an innocuous context. A user believing to interact with that context, for example, clicking on buttons, inadvertently interacts with the authorization endpoint user interface instead. The opposite can be achieved as well: A user believing to interact with the authorization endpoint might inadvertently type a password into an attacker-provided input field overlaid over the original user interface. Clickjacking attacks can be designed such that users can hardly notice the attack, for example using almost invisible iframes overlaid on top of other elements.

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.

7.11. Code Injection and Input Validation

A code injection attack occurs when an input or otherwise external variable is used by an application unsanitized and causes modification to the application logic. This may allow an attacker to gain access to the application device or its data, cause denial of service, or introduce a wide range of malicious side-effects.

The authorization server and client MUST sanitize (and validate when possible) any value received -- in particular, the value of the state and redirect_uri parameters.

7.12. Open Redirection

An open redirector is an endpoint that forwards a user's browser to an arbitrary URI obtained from a query parameter. Such endpoints are sometimes implemented, for example, to show a message before a user is then redirected to an external website, or to redirect users back to a URL they were intending to visit before being interrupted, e.g., by a login prompt.

The following attacks can occur when an AS or client has an open redirector.

7.12.1. 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, as described in Section 4.1.1 of [I-D.ietf-oauth-security-topics]. 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].

7.12.2. 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.

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 execute one of the following attacks:

1. 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.

2. Intentionally send a valid authorization request with client_id and redirect_uri controlled by the attacker. After the user authenticates, the AS prompts the user to provide consent to the request. If the user notices an issue with the request and declines the request, the AS still redirects the user agent to the phishing site. In this case, the user agent will be redirected to the phishing site regardless of the action taken by the user.

3. Intentionally send a valid silent authentication request (prompt=none) with client_id and redirect_uri controlled by the attacker. In this case, the AS will automatically redirect the user agent to the phishing site.

The AS MUST take precautions to prevent these threats. The AS MUST always authenticate the user first and, with the exception of the silent authentication use case, prompt the user for credentials when needed, before redirecting the user. Based on its risk assessment, the AS needs to decide whether it can trust the redirect URI or not. It could take into account URI analytics done internally or through some external service to evaluate the credibility and trustworthiness content behind the URI, and the source of the redirect URI and other client data.

The AS 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.

7.13. Authorization Server Mix-Up Mitigation

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.

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, bind this information to the user agent, and check that the authorization response was received from the correct issuer. Clients MUST ensure that the subsequent access token request, if applicable, is sent to the same issuer. 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 [OpenID.Discovery] 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 "their" AS URL, but declare a token endpoint under their own control.

See Section 4.4 of [I-D.ietf-oauth-security-topics] for a detailed description of several types of mix-up attacks.

7.13.1. 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 an optional response parameter iss (see Section 4.1.2).

* 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 [RFC9207].

While this defense may require using an additional parameter to transport the issuer information, it is a robust and relatively simple defense against mix-up.

7.13.2. 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.

8. Native Applications

Native applications are clients installed and executed on the device used by the resource owner (i.e., desktop application, native mobile application). Native applications require special consideration related to security, platform capabilities, and overall end-user experience.

The authorization endpoint requires interaction between the client and the resource owner's user agent. The best current practice is to perform the OAuth authorization request in an external user agent (typically the browser) rather than an embedded user agent (such as one implemented with web-views).

The native application can capture the response from the authorization server using a redirect URI with a scheme registered with the operating system to invoke the client as the handler, manual copy-and-paste of the credentials, running a local web server, installing a user agent extension, or by providing a redirect URI identifying a server-hosted resource under the client's control, which in turn makes the response available to the native application.

Previously, it was common for native apps to use embedded user agents (commonly implemented with web-views) for OAuth authorization requests. That approach has many drawbacks, including the host app being able to copy user credentials and cookies as well as the user needing to authenticate from scratch in each app. See Section 8.5.1 for a deeper analysis of the drawbacks of using embedded user agents for OAuth.

Native app authorization requests that use the system browser are more secure and can take advantage of the user's authentication state on the device. Being able to use the existing authentication session in the browser enables single sign-on, as users don't need to authenticate to the authorization server each time they use a new app (unless required by the authorization server policy).

Supporting authorization flows between a native app and the browser is possible without changing the OAuth protocol itself, as the OAuth authorization request and response are already defined in terms of URIs. This encompasses URIs that can be used for inter-app communication. Some OAuth server implementations that assume all clients are confidential web clients will need to add an understanding of public native app clients and the types of redirect URIs they use to support this best practice.

8.1. Registration of Native App Clients

Except when using a mechanism like Dynamic Client Registration [RFC7591] to provision per-instance secrets, native apps are classified as public clients, as defined in Section 2.1; they 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.

8.1.1. Client Authentication of Native Apps

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, it is NOT RECOMMENDED for authorization servers to require client authentication of public native apps clients 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 native app clients MUST treat the client as a public client (as defined in Section 2.1), and not accept the secret as proof of the client’s identity. Without additional measures, such clients are subject to client impersonation (see Section 7.3.1).

8.2. Using Inter-App URI Communication for OAuth in Native Apps

Just as URIs are used for OAuth on the web to initiate the authorization request and return the authorization response to the requesting website, URIs can be used by native apps to initiate the authorization request in the device’s browser and return the response to the requesting native app.

By adopting the same methods used on the web for OAuth, benefits seen in the web context like the usability of a single sign-on session and the security of a separate authentication context are likewise gained in the native app context. Reusing the same approach also reduces the implementation complexity and increases interoperability by relying on standards-based web flows that are not specific to a particular platform.

Native apps MUST use an external user agent to perform OAuth authorization requests. This is achieved by opening the authorization request in the browser (detailed in Section 8.3) and using a redirect URI that will return the authorization response back to the native app (defined in Section 8.4).

8.3. Initiating the Authorization Request from a Native App

Native apps needing user authorization create an authorization request URI with the authorization code grant type per Section 4.1 using a redirect URI capable of being received by the native app.

The function of the redirect URI for a native app authorization request is similar to that of a web-based authorization request. Rather than returning the authorization response to the OAuth client’s server, the redirect URI used by a native app returns the response to the app. Several options for a redirect URI that will return the authorization response to the native app in different platforms are documented in Section 8.4. Any redirect URI that allows the app to receive the URI and inspect its parameters is viable.

After constructing the authorization request URI, the app uses platform-specific APIs to open the URI in an external user agent. Typically, the external user agent used is the default browser, that is, the application configured for handling http and https scheme URIs on the system; however, different browser selection criteria and other categories of external user agents MAY be used.

This best practice focuses on the browser as the RECOMMENDED external user agent for native apps. An external user agent designed specifically for user authorization and capable of processing authorization requests and responses like a browser MAY also be used. Other external user agents, such as a native app provided by the authorization server may meet the criteria set out in this best practice, including using the same redirect URI properties, but their use is out of scope for this specification.

Some platforms support a browser feature known as "in-app browser tabs", where an app can present a tab of the browser within the app context without switching apps, but still retain key benefits of the browser such as a shared authentication state and security context. On platforms where they are supported, it is RECOMMENDED, for usability reasons, that apps use in-app browser tabs for the authorization request.

8.4. Receiving the Authorization Response in a Native App

There are several redirect URI options available to native apps for receiving the authorization response from the browser, the availability and user experience of which varies by platform.

8.4.1. Claimed "https" Scheme URI Redirection

Some operating systems allow apps to claim https URIs (see Section 4.2.2 of [RFC9110]) in the domains they control. When the browser encounters a claimed URI, instead of the page being loaded in the browser, the native app is launched with the URI supplied as a launch parameter.

Such URIs can be used as redirect URIs by native apps. They are indistinguishable to the authorization server from a regular web-based client redirect URI. An example is:

https://app.example.com/oauth2redirect/example-provider

As the redirect URI alone is not enough to distinguish public native app clients from confidential web clients, it is REQUIRED in Section 8.1 that the client type be recorded during client registration to enable the server to determine the client type and act accordingly.

App-claimed https scheme redirect URIs have some advantages compared to other native app redirect options in that the identity of the destination app is guaranteed to the authorization server by the operating system. For this reason, native apps SHOULD use them over the other options where possible.

8.4.2. Loopback Interface Redirection

Native apps that are able to open a port on the loopback network interface without needing special permissions (typically, those on desktop operating systems) can use the loopback interface to receive the OAuth redirect.

Loopback redirect URIs use the http scheme and are constructed with the loopback IP literal and whatever port the client is listening on.

That is, http://127.0.0.1:{port}/{path} for IPv4, and http://[::1]:{port}/{path} for IPv6. An example redirect using the IPv4 loopback interface with a randomly assigned port:

http://127.0.0.1:51004/oauth2redirect/example-provider

An example redirect using the IPv6 loopback interface with a randomly assigned port:

http://[::1]:61023/oauth2redirect/example-provider

While redirect URIs using the name localhost (i.e., http://localhost:{port}/{path}) function similarly to loopback IP redirects, the use of localhost is NOT RECOMMENDED. Specifying a redirect URI with the loopback IP literal rather than localhost avoids inadvertently listening on network interfaces other than the loopback interface. It is also less susceptible to client-side firewalls and misconfigured host name resolution on the user's device.

The authorization server MUST allow any port to be specified at the time of the request for loopback IP redirect URIs, to accommodate clients that obtain an available ephemeral port from the operating system at the time of the request.

Clients SHOULD NOT assume that the device supports a particular version of the Internet Protocol. It is RECOMMENDED that clients attempt to bind to the loopback interface using both IPv4 and IPv6 and use whichever is available.

8.4.3. Private-Use URI Scheme Redirection

Many mobile and desktop computing platforms support inter-app communication via URIs by allowing apps to register private-use URI schemes (sometimes colloquially referred to as "custom URL schemes") like com.example.app. When the browser or another app attempts to load a URI with a private-use URI scheme, the app that registered it is launched to handle the request.

Many environments that support private-use URI schemes do not provide a mechanism to claim a scheme and prevent other parties from using another application's scheme. As such, clients using private-use URI schemes are vulnerable to potential attacks on their redirect URIs, so this option should only be used if the previously mentioned more secure options are not available.

To perform an authorization request with a private-use URI scheme redirect, the native app launches the browser with a standard authorization request, but one where the redirect URI utilizes a private-use URI scheme it registered with the operating system.

When choosing a URI scheme to associate with the app, apps MUST use a URI scheme based on a domain name under their control, expressed in reverse order, as recommended by Section 3.8 of [RFC7595] for private-use URI schemes.

For example, an app that controls the domain name app.example.com can use com.example.app as their scheme. Some authorization servers assign client identifiers based on domain names, for example, client1234.usercontent.example.net, which can also be used as the domain name for the scheme when reversed in the same manner. A scheme such as myapp, however, would not meet this requirement, as it is not based on a domain name.

When there are multiple apps by the same publisher, care must be taken so that each scheme is unique within that group. On platforms that use app identifiers based on reverse-order domain names, those identifiers can be reused as the private-use URI scheme for the OAuth redirect to help avoid this problem.

Following the requirements of Section 3.2 of [RFC3986], as there is no naming authority for private-use URI scheme redirects, only a single slash (/) appears after the scheme component. A complete example of a redirect URI utilizing a private-use URI scheme is:

com.example.app:/oauth2redirect/example-provider

When the authorization server completes the request, it redirects to the client’s redirect URI as it would normally. As the redirect URI uses a private-use URI scheme, it results in the operating system launching the native app, passing in the URI as a launch parameter. Then, the native app uses normal processing for the authorization response.

8.5. Security Considerations in Native Apps

8.5.1. Embedded User Agents in Native Apps

Embedded user agents are a technically possible method for authorizing native apps. These embedded user agents are unsafe for use by third parties to the authorization server by definition, as the app that hosts the embedded user agent can access the user’s full authentication credentials, not just the OAuth authorization grant that was intended for the app.

In typical web-view-based implementations of embedded user agents, the host application can record every keystroke entered in the login form to capture usernames and passwords, automatically submit forms to bypass user consent, and copy session cookies and use them to perform authenticated actions as the user.

Even when used by trusted apps belonging to the same party as the authorization server, embedded user agents violate the principle of least privilege by having access to more powerful credentials than they need, potentially increasing the attack surface.

Encouraging users to enter credentials in an embedded user agent without the usual address bar and visible certificate validation features that browsers have makes it impossible for the user to know if they are signing in to the legitimate site; even when they are, it trains them that it’s OK to enter credentials without validating the site first.

Aside from the security concerns, embedded user agents do not share the authentication state with other apps or the browser, requiring the user to log in for every authorization request, which is often considered an inferior user experience.

8.5.2. Fake External User-Agents in Native Apps

The native app that is initiating the authorization request has a large degree of control over the user interface and can potentially present a fake external user agent, that is, an embedded user agent made to appear as an external user agent.

When all good actors are using external user agents, the advantage is that it is possible for security experts to detect bad actors, as anyone faking an external user agent is provably bad. On the other hand, if good and bad actors alike are using embedded user agents, bad actors don’t need to fake anything, making them harder to detect. Once a malicious app is detected, it may be possible to use this knowledge to blacklist the app’s signature in malware scanning software, take removal action (in the case of apps distributed by app stores) and other steps to reduce the impact and spread of the malicious app.

Authorization servers can also directly protect against fake external user agents by requiring an authentication factor only available to true external user agents.

Users who are particularly concerned about their security when using in-app browser tabs may also take the additional step of opening the request in the full browser from the in-app browser tab and complete the authorization there, as most implementations of the in-app browser tab pattern offer such functionality.

8.5.3. Malicious External User-Agents in Native Apps

If a malicious app is able to configure itself as the default handler for https scheme URIs in the operating system, it will be able to intercept authorization requests that use the default browser and abuse this position of trust for malicious ends such as phishing the user.

This attack is not confined to OAuth; a malicious app configured in this way would present a general and ongoing risk to the user beyond OAuth usage by native apps. Many operating systems mitigate this issue by requiring an explicit user action to change the default handler for http and https scheme URIs.

8.5.4. Loopback Redirect Considerations in Native Apps

Loopback interface redirect URIs MAY use the http scheme (i.e., without TLS). This is acceptable for loopback interface redirect URIs as the HTTP request never leaves the device.

Clients should open the network port only when starting the authorization request and close it once the response is returned.

Clients should listen on the loopback network interface only, in order to avoid interference by other network actors.

Clients should use loopback IP literals rather than the string localhost as described in Section 8.4.2.

9. Browser-Based Apps

Browser-based apps are are clients that run in a web browser, typically written in JavaScript, also known as "single-page apps". These types of apps have particular security considerations similar to native apps.

TODO: Bring in the normative text of the browser-based apps BCP when it is finalized.

10. Differences from OAuth 2.0

This draft consolidates the functionality in OAuth 2.0 [RFC6749], OAuth 2.0 for Native Apps ([RFC8252]), Proof Key for Code Exchange ([RFC7636]), OAuth 2.0 for Browser-Based Apps ([I-D.ietf-oauth-browser-based-apps]), OAuth Security Best Current Practice ([I-D.ietf-oauth-security-topics]), and Bearer Token Usage ([RFC6750]).

Where a later draft updates or obsoletes functionality found in the original [RFC6749], that functionality in this draft is updated with the normative changes described in a later draft, or removed entirely.

A non-normative list of changes from OAuth 2.0 is listed below:

* The authorization code grant is extended with the functionality from PKCE ([RFC7636]) such that the default method of using the authorization code grant according to this specification requires the addition of the PKCE parameters

* Redirect URIs must be compared using exact string matching as per Section 4.1.3 of [I-D.ietf-oauth-security-topics]

* The Implicit grant (response_type=token) is omitted from this specification as per Section 2.1.2 of [I-D.ietf-oauth-security-topics]

* The Resource Owner Password Credentials grant is omitted from this specification as per Section 2.4 of [I-D.ietf-oauth-security-topics]

* Bearer token usage omits the use of bearer tokens in the query string of URIs as per Section 4.3.2 of [I-D.ietf-oauth-security-topics]

* Refresh tokens for public clients must either be sender-constrained or one-time use as per Section 4.13.2 of [I-D.ietf-oauth-security-topics]

* The token endpoint request containing an authorization code no longer contains the redirect_uri parameter

10.1. Removal of the OAuth 2.0 Implicit grant

The OAuth 2.0 Implicit grant is omitted from OAuth 2.1 as it was deprecated in [I-D.ietf-oauth-security-topics].

The intent of removing the Implicit grant is to no longer issue access tokens in the authorization response, as such tokens are vulnerable to leakage and injection, and are unable to be sender-constrained to a client. This behavior was indicated by clients using the response_type=token parameter. This value for the response_type parameter is no longer defined in OAuth 2.1.

Removal of response_type=token does not have an effect on other extension response types returning other artifacts from the authorization endpoint, for example, response_type=id_token defined by [OpenID].

10.2. Redirect URI Parameter in Token Request

In OAuth 2.0, the request to the token endpoint in the authorization code flow (section 4.1.3 of [RFC6749]) contains an optional redirect_uri parameter. The parameter was intended to prevent an authorization code injection attack, and was required if the redirect_uri parameter was sent in the original authorization request. The authorization request only required the redirect_uri parameter if multiple redirect URIs were registered to the specific client. However, in practice, many authorization server implementations required the redirect_uri parameter in the authorization request even if only one was registered, leading the redirect_uri parameter to be required at the token endpoint as well.

In OAuth 2.1, authorization code injection is prevented by the code_challenge and code_verifier parameters, making the inclusion of the redirect_uri parameter serve no purpose in the token request. As such, it has been removed.

For backwards compatibility of an authorization server wishing to support both OAuth 2.0 and OAuth 2.1 clients, the authorization server MUST allow clients to send the redirect_uri parameter in the token request (Section 4.1.3), and MUST enforce the parameter as described in [RFC6749]. The authorization server can use the client_id in the request to determine whether to enforce this behavior for the specific client that it knows will be using the older OAuth 2.0 behavior.

A client following only the OAuth 2.1 recommendations will not send the redirect_uri in the token request, and therefore will not be compatible with an authorization server that expects the parameter in the token request.

11. IANA Considerations

This document does not require any IANA actions.

All referenced registries are defined by [RFC6749] and related documents that this work is based upon. No changes to those registries are required by this specification.

12. References

12.1. Normative References

[BCP195] Saint-Andre, P., "Recommendations for Secure Use of Transport Layer Security (TLS)", 2015.

[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-24, 23 October 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-oauth-security-topics-24>.

[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>.

[RFC2617] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication", RFC 2617, DOI 10.17487/RFC2617, June 1999, <https://www.rfc-editor.org/info/rfc2617>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <https://www.rfc-editor.org/info/rfc3629>.

[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>.

[RFC4949] Shirey, R., "Internet Security Glossary, Version 2", FYI 36, RFC 4949, DOI 10.17487/RFC4949, August 2007, <https://www.rfc-editor.org/info/rfc4949>.

[RFC5234] 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>.

[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>.

[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>.

[RFC7235] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Authentication", RFC 7235, DOI 10.17487/RFC7235, June 2014, <https://www.rfc-editor.org/info/rfc7235>.

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

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

[RFC7595] Thaler, D., Ed., Hansen, T., and T. Hardie, "Guidelines and Registration Procedures for URI Schemes", BCP 35, RFC 7595, DOI 10.17487/RFC7595, June 2015, <https://www.rfc-editor.org/info/rfc7595>.

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

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

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

[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>.

[RFC9110] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022, <https://www.rfc-editor.org/info/rfc9110>.

[RFC9111] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Caching", STD 98, RFC 9111, DOI 10.17487/RFC9111, June 2022, <https://www.rfc-editor.org/info/rfc9111>.

[RFC9207] Meyer zu Selhausen, K. and D. Fett, "OAuth 2.0 Authorization Server Issuer Identification", RFC 9207, DOI 10.17487/RFC9207, March 2022, <https://www.rfc-editor.org/info/rfc9207>.

[USASCII] Institute, A. N. S., "Coded Character Set -- 7-bit American Standard Code for Information Interchange, ANSI X3.4", 1986.

[W3C.REC-xml-20081126] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and F. Yergeau, "Extensible Markup Language", November 2008, <https://www.w3.org/TR/REC-xml/REC-xml-20081126.xml>.

[WHATWG.CORS] WHATWG, "Fetch Standard: CORS protocol", June 2023, <https://fetch.spec.whatwg.org/#http-cors-protocol>.

[WHATWG.URL] WHATWG, "URL", May 2022, <https://url.spec.whatwg.org/>.

12.2. Informative References

[CSP-2] "Content Security Policy Level 2", December 2016, <https://www.w3.org/TR/CSP2>.

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

[I-D.ietf-oauth-browser-based-apps]
Parecki, A., Waite, D., and P. De Ryck, "OAuth 2.0 for Browser-Based Apps", Work in Progress, Internet-Draft, draft-ietf-oauth-browser-based-apps-15, 23 October 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-oauth-browser-based-apps-15>.

[NIST800-63]
Burr, W., Dodson, D., Newton, E., Perlner, R., Polk, T., Gupta, S., and E. Nabbus, "NIST Special Publication 800-63-1, INFORMATION SECURITY", December 2011, <http://csrc.nist.gov/publications/>.

[OMAP] Huff, J., Schlacht, D., Nadalin, A., Simmons, J., Rosenberg, P., Madsen, P., Ace, T., Rickelton-Abdi, C., and B. Boyer, "Online Multimedia Authorization Protocol: An Industry Standard for Authorized Access to Internet Multimedia Resources", August 2012, <https://www.svta.org/product/online-multimedia-authorization-protocol/>.

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

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

[OpenID.Messages]
Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., Mortimore, C., and E. Jay, "OpenID Connect Messages 1.0", June 2012, <http://openid.net/specs/openid-connect-messages-1_0.html>.

[owasp_redir]
"OWASP Cheat Sheet Series - Unvalidated Redirects and Forwards", 2020, <https://cheatsheetseries.owasp.org/cheatsheets/Unvalidated_Redirects_and_Forwards_Cheat_Sheet.html>.

[RFC6265] Barth, A., "HTTP State Management Mechanism", RFC 6265, DOI 10.17487/RFC6265, April 2011, <https://www.rfc-editor.org/info/rfc6265>.

[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>.

[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>.

[RFC7592] Richer, J., Ed., Jones, M., Bradley, J., and M. Machulak, "OAuth 2.0 Dynamic Client Registration Management Protocol", RFC 7592, DOI 10.17487/RFC7592, July 2015, <https://www.rfc-editor.org/info/rfc7592>.

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

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

[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>.

[RFC8628] Denniss, W., Bradley, J., Jones, M., and H. Tschofenig, "OAuth 2.0 Device Authorization Grant", RFC 8628, DOI 10.17487/RFC8628, August 2019, <https://www.rfc-editor.org/info/rfc8628>.

[RFC8705] Campbell, B., Bradley, J., Sakimura, N., and T. Lodderstedt, "OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens", RFC 8705, DOI 10.17487/RFC8705, February 2020, <https://www.rfc-editor.org/info/rfc8705>.

[RFC8707] Campbell, B., Bradley, J., and H. Tschofenig, "Resource Indicators for OAuth 2.0", RFC 8707, DOI 10.17487/RFC8707, February 2020, <https://www.rfc-editor.org/info/rfc8707>.

[RFC9068] Bertocci, V., "JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens", RFC 9068, DOI 10.17487/RFC9068, October 2021, <https://www.rfc-editor.org/info/rfc9068>.

[RFC9126] Lodderstedt, T., Campbell, B., Sakimura, N., Tonge, D., and F. Skokan, "OAuth 2.0 Pushed Authorization Requests", RFC 9126, DOI 10.17487/RFC9126, September 2021, <https://www.rfc-editor.org/info/rfc9126>.

[RFC9396] Lodderstedt, T., Richer, J., and B. Campbell, "OAuth 2.0 Rich Authorization Requests", RFC 9396, DOI 10.17487/RFC9396, May 2023, <https://www.rfc-editor.org/info/rfc9396>.

[RFC9449] Fett, D., Campbell, B., Bradley, J., Lodderstedt, T., Jones, M., and D. Waite, "OAuth 2.0 Demonstrating Proof of Possession (DPoP)", RFC 9449, DOI 10.17487/RFC9449, September 2023, <https://www.rfc-editor.org/info/rfc9449>.

[W3C.REC-html401-19991224] Hors, A. L., Ed., Raggett, D., Ed., and I. Jacobs, Ed., "HTML 4.01 Specification", W3C REC REC-html401-19991224, W3C REC-html401-19991224, 24 December 1999, <https://www.w3.org/TR/1999/REC-html401-19991224/>.

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

This section provides Augmented Backus-Naur Form (ABNF) syntax descriptions for the elements defined in this specification using the notation of [RFC5234]. The ABNF below is defined in terms of Unicode code points [W3C.REC-xml-20081126]; these characters are typically encoded in UTF-8. Elements are presented in the order first defined.

Some of the definitions that follow use the "URI-reference" definition from [RFC3986].

Some of the definitions that follow use these common definitions:

```
VSCHAR     = %x20-7E
NQCHAR     = %x21 / %x23-5B / %x5D-7E
NQSCHAR    = %x20-21 / %x23-5B / %x5D-7E
```

A.1. "client_id" Syntax

The client_id element is defined in Section 2.4.1:

```
client-id     = *VSCHAR
```

A.2. "client_secret" Syntax

The client_secret element is defined in Section 2.4.1:

```
client-secret = *VSCHAR
```

A.3. "response_type" Syntax

The response_type element is defined in Section 4.1.1 and Section 6.4:

```
response-type = response-name *( SP response-name )
response-name = 1*response-char
response-char = "_" / DIGIT / ALPHA
```

A.4. "scope" Syntax

The scope element is defined in Section 1.4.1:

```
scope       = scope-token *( SP scope-token )
scope-token = 1*NQCHAR
```

A.5. "state" Syntax

The state element is defined in Section 4.1.1, Section 4.1.2, and Section 4.1.2.1:

```
state      = 1*VSCHAR
```

A.6. "redirect_uri" Syntax

The redirect_uri element is defined in Section 4.1.1, and Section 4.1.3:

```
redirect-uri      = URI-reference
```

A.7. "error" Syntax

The error element is defined in Sections Section 4.1.2.1, Section 3.2.4, 7.2, and 8.5:

```
error             = 1*NQSCHAR
```

A.8. "error_description" Syntax

The error_description element is defined in Sections Section 4.1.2.1, Section 3.2.4, and Section 5.3:

```
error-description = 1*NQSCHAR
```

A.9. "error_uri" Syntax

The error_uri element is defined in Sections Section 4.1.2.1, Section 3.2.4, and 7.2:

```
error-uri         = URI-reference
```

A.10. "grant_type" Syntax

The grant_type element is defined in Section Section 3.2.2:

```
grant-type = grant-name / URI-reference
grant-name = 1*name-char
name-char  = "-" / "." / "_" / DIGIT / ALPHA
```

A.11. "code" Syntax

The code element is defined in Section 4.1.3:

```
code       = 1*VSCHAR
```

A.12. "access_token" Syntax

The access_token element is defined in Section 3.2.3:

```
access-token = 1*VSCHAR
```

A.13. "token_type" Syntax

The token_type element is defined in Section 3.2.3, and Section 6.1:

```
token-type = type-name / URI-reference
type-name  = 1*name-char
name-char  = "-" / "." / "_" / DIGIT / ALPHA
```

A.14. "expires_in" Syntax

The expires_in element is defined in Section 3.2.3:

```
expires-in = 1*DIGIT
```

A.15. "refresh_token" Syntax

The refresh_token element is defined in Section 3.2.3 and Section 4.3:

```
refresh-token = 1*VSCHAR
```

A.16. Endpoint Parameter Syntax

The syntax for new endpoint parameters is defined in Section 6.2:

```
param-name = 1*name-char
name-char  = "-" / "." / "_" / DIGIT / ALPHA
```

A.17. "code_verifier" Syntax

ABNF for code_verifier is as follows.

```
code-verifier = 43*128unreserved
unreserved = ALPHA / DIGIT / "-" / "." / "_" / "~"
ALPHA = %x41-5A / %x61-7A
DIGIT = %x30-39
```

A.18. "code_challenge" Syntax

ABNF for code_challenge is as follows.

```
code-challenge = 43*128unreserved
unreserved = ALPHA / DIGIT / "-" / "." / "_" / "~"
ALPHA = %x41-5A / %x61-7A
DIGIT = %x30-39
```

Appendix B. Use of application/x-www-form-urlencoded Media Type

At the time of publication of [RFC6749], the application/x-www-form-urlencoded media type was defined in Section 17.13.4 of [W3C.REC-html401-19991224] but not registered in the IANA MIME Media Types registry (http://www.iana.org/assignments/media-types (http://www.iana.org/assignments/media-types)). Furthermore, that definition is incomplete, as it does not consider non-US-ASCII characters.

To address this shortcoming when generating contents using this media type, names and values MUST be encoded using the UTF-8 character encoding scheme [RFC3629] first; the resulting octet sequence then needs to be further encoded using the escaping rules defined in [W3C.REC-html401-19991224].

When parsing data from a content using this media type, the names and values resulting from reversing the name/value encoding consequently need to be treated as octet sequences, to be decoded using the UTF-8 character encoding scheme.

For example, the value consisting of the six Unicode code points (1) U+0020 (SPACE), (2) U+0025 (PERCENT SIGN), (3) U+0026 (AMPERSAND), (4) U+002B (PLUS SIGN), (5) U+00A3 (POUND SIGN), and (6) U+20AC (EURO SIGN) would be encoded into the octet sequence below (using hexadecimal notation):

```
20 25 26 2B C2 A3 E2 82 AC
```

and then represented in the content as:

```
+%25%26%2B%C2%A3%E2%82%AC
```

GitHub discussion: https://github.com/oauth-wg/oauth-v2-1/issues/128 (https://github.com/oauth-wg/oauth-v2-1/issues/128)

Appendix C. Extensions

Below is a list of well-established extensions at the time of publication:

* [RFC9068]: JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens

  - This specification defines a profile for issuing OAuth access tokens in JSON Web Token (JWT) format.

* [RFC8628]: OAuth 2.0 Device Authorization Grant

  - The Device Authorization Grant (formerly known as the Device Flow) is an extension that enables devices with no browser or limited input capability to obtain an access token. This is commonly used by smart TV apps, or devices like hardware video encoders that can stream video to a streaming video service.

* [RFC8414]: Authorization Server Metadata

  - Authorization Server Metadata (also known as OAuth Discovery) defines an endpoint clients can use to look up the information needed to interact with a particular OAuth server, such as the location of the authorization and token endpoints and the supported grant types.

* [RFC8707]: Resource Indicators

  - Provides a way for the client to explicitly signal to the authorization server where it intends to use the access token it is requesting.

* [RFC7591]: Dynamic Client Registration

  - Dynamic Client Registration provides a mechanism for programmatically registering clients with an authorization server.

* [RFC9449]: Demonstrating Proof of Possession (DPoP)

  - DPoP describes a mechanism of binding tokens to the clients they were issued to, and providing proof of that binding in an HTTP header when making requests.

* [RFC8705]: Mutual TLS

  - Mutual TLS describes a mechanism of binding tokens to the clients they were issued to, as well as a client authentication mechanism, via TLS certificate authentication.

* [RFC7662]: Token Introspection

  - The Token Introspection extension defines a mechanism for resource servers to obtain information about access tokens.

* [RFC7009]: Token Revocation

  - The Token Revocation extension defines a mechanism for clients to indicate to the authorization server that an access token is no longer needed.

* [RFC9126]: Pushed Authorization Requests

  - The Pushed Authorization Requests extension describes a technique of initiating an OAuth flow from the back channel, providing better security and more flexibility for building complex authorization requests.

* [RFC9207]: Authorization Server Issuer Identification

  - The iss parameter in the authorization response indicates the identity of the authorization server to prevent mix-up attacks in the client.

* [RFC9396]: Rich Authorization Requests

  - Rich Authorization Requests specifies a new parameter authorization_details that is used to carry fine-grained authorization data in the OAuth authorization request.

Appendix D. Acknowledgements

This specification is the work of the OAuth Working Group, and its starting point was based on the contents of the following specifications: OAuth 2.0 Authorization Framework (RFC 6749), OAuth 2.0 for Native Apps (RFC 8252), OAuth Security Best Current Practice, and OAuth 2.0 for Browser-Based Apps. The editors would like to thank everyone involved in the creation of those specifications upon which this is built.

The editors would also like to thank the following individuals for their ideas, feedback, corrections, and wording that helped shape this version of the specification: Vittorio Bertocci, Michael Jones, Justin Richer, Daniel Fett, Brian Campbell, Joseph Heenan, Roberto Polli, Andrii Deinega, Falko, Michael Peck, Bob Hamburg, Deng Chao, and Karsten Meyer zu Selhausen.

Discussions around this specification have also occurred at the OAuth Security Workshop in 2021 and 2022. The authors thank the organizers of the workshop (Guido Schmitz, Steinar Noem, and Daniel Fett) for hosting an event that’s conducive to collaboration and community input.

Appendix E. Document History

[[To be removed from the final specification]]

-10

* Clarify that the client id is an opaque string

* Extensions may define additional error codes on a resource request

* Improved formatting for error field definitions

* Moved and expanded "scope" definition to introduction section

* Split access token section into structure and request

* Renamed b64token to token68 for consistency with RFC7235

* Restored content from old appendix B about application/x-www-form-urlencoded

* Clarified that clients must not parse access tokens

* Expanded text around when redirect_uri parameter is required in the authorization request

* Changed "permissions" to "privileges" in refresh token section for consistency

* Consolidated authorization code flow security considerations

* Clarified authorization code reuse - an authorization code can only obtain an access token once

-09

* AS MUST NOT support CORS requests at authorization endpoint

* more detail on asymmetric client authentication

* sync CSRF description from security BCP

* update and move sender-constrained access tokens section

* sync client impersonating resource owner with security BCP

* add reference to authorization request from redirect URI registration section

* sync refresh rotation section from security BCP

* sync redirect URI matching text from security BCP

* updated references to RAR (RFC9396)

* clarifications on URIs

* removed redirect_uri from the token request

* expanded security considerations around code_verifier

* revised introduction section

-08

* Updated acknowledgments

* Swap "by a trusted party" with "by an outside party" in client ID definition

* Replaced "verify the identity of the resource owner" with "authenticate"

* Clarified refresh token rotation to match RFC6819

* Added appendix to hold application/x-www-form-urlencoded examples

* Fixed references to entries in appendix

* Incorporated new "Phishing via AS" section from Security BCP

* Rephrase description of the motivation for client authentication

* Moved "scope" parameter in token request into specific grant types to match OAuth 2.0

* Updated Clickjacking and Open Redirection description from the latest version of the Security BCP

* Moved normative requirements out of authorization code security considerations section

* Security considerations clarifications, and removed a duplicate section

-07

* Removed "third party" from abstract

* Added MFA and passwordless as additional motiviations in introduction

* Mention PAR as one way redirect URI registration can happen

* Added a reference to requiring CORS headers on the token endpoint

* Updated reference to OMAP extension

* Fixed numbering in sequence diagram

-06

* Removed "credentialed client" term

* Simplified definition of "confidential" and "public" clients

* Incorporated the iss response parameter referencing RFC9207

* Added section on access token validation by the RS

* Removed requirement for authorization servers to support all 3 redirect methods for native apps

* Fixes for some references

* Updates HTTP references to RFC 9110

* Clarifies "authorization grant" term

* Clarifies client credential grant usage

* Clean up authorization code diagram

* Updated reference for application/x-www-form-urlencoded and removed outdated note about it not being in the IANA registry

-05

* Added a section about the removal of the implicit flow

* Moved many normative requirements from security considerations into the appropriate inline sections

* Reorganized and consolidated TLS language

* Require TLS on redirect URIs except for localhost/custom URL scheme

* Updated refresh token guidance to match security BCP

-04

* Added explicit mention of not sending access tokens in URI query strings

* Clarifications on definition of client types

* Consolidated text around loopback vs localhost

* Editorial clarifications throughout the document

-03

* refactoring to collect all the grant types under the same top-level header in section 4

* Better split normative and security consideration text into the appropriate places, both moving text that was really security considerations out of the main part of the document, as well as pulling normative requirements from the security considerations sections into the appropriate part of the main document

* Incorporated many of the published errata on RFC6749

* Updated references to various RFCs

* Editorial clarifications throughout the document

-02

-01

-00

* initial revision

Authors' Addresses

Dick Hardt
Hell
Email: dick.hardt@gmail.com

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

Torsten Lodderstedt
yes.com
Email: torsten@lodderstedt.net