Web Authorization Protocol
Internet-Draft
Intended status: Standards Track
Expires: 1 September 2026

A. Parecki
Okta
G. Fletcher
Practical Identity LLC
P. Kasselman
Defakto Security

28 February 2026

OAuth 2.0 for First-Party Applications
draft-ietf-oauth-first-party-apps-03

Abstract

This document defines the Authorization Challenge Endpoint, which supports clients that want to control the process of obtaining authorization from the user using a native experience.

In many cases, this can provide an entirely browserless OAuth 2.0 experience suited for native applications, only delegating to the browser in unexpected, high risk, or error conditions.

About This Document

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

The latest revision of this draft can be found at https://drafts.oauth.net/oauth-first-party-apps/draft-ietf-oauth-first-party-apps.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-ietf-oauth-first-party-apps/.

Discussion of this document takes place on the Web Authorization Protocol Working Group mailing list (mailto:oauth@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/oauth/. Subscribe at https://www.ietf.org/mailman/listinfo/oauth/.

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

Status of This Memo

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

This Internet-Draft will expire on 1 September 2026.

Copyright Notice

Table of Contents

1. Introduction 4
  1.1. Usage and Applicability 4
  1.2. Limitations of this specification 5
  1.3. User Experience Considerations 5
2. Conventions and Definitions 6
  2.1. Terminology 6
3. Protocol Overview 6
  3.1. Initial Authorization Request 6
  3.2. Refresh Token Request 8
  3.3. Resource Request 8
4. Protocol Endpoints 8
  4.1. Authorization Challenge Endpoint 8
  4.2. Token endpoint 9
5. Authorization Initiation 10
  5.1. Authorization Challenge Request 10
  5.2. Authorization Challenge Response 11
    5.2.1. Authorization Code Response 11
    5.2.2. Error Response 12
  5.3. Intermediate Requests 14
    5.3.1. Auth Session 15

6. Token Request 15
6.1. Token Endpoint Successful Response 16
6.2. Token Endpoint Error Response 16
7. Resource Server Error Response 17
8. Authorization Server Metadata 18
9. Security Considerations 18
9.1. First-Party Applications 18
9.2. Phishing 18
9.3. Credential Stuffing Attacks 19
9.4. Client Authentication 19
9.5. Sender-Constrained Tokens 20
9.5.1. DPoP: Demonstrating Proof-of-Possession 20
9.5.2. Other Proof of Possession Mechanisms 21
9.6. Auth Session 21
9.6.1. Auth Session DPoP Binding 21
9.6.2. Auth Session Lifetime 22
9.7. Multiple Applications 22
9.7.1. User Experience Risk 22
9.7.2. Technical Risk 22
9.7.3. Mitigation 22
9.8. Single Page Applications 23
10. IANA Considerations 23
10.1. OAuth Parameters Registration 23
10.2. OAuth Server Metadata Registration 23
11. References 24
11.1. Normative References 24
11.2. Informative References 26
Appendix A. Example User Experiences 26
A.1. Passkey 27
A.2. Redirect to Authorization Server 27
A.3. Passwordless One-Time Password (OTP) 28
A.4. E-Mail Confirmation Code 28
A.5. Mobile Confirmation Code 29
A.6. Re-authenticating to an app a week later using OTP . . . 30
A.7. Step-up Authentication using Confirmation SMS 31
A.8. Registration 32
Appendix B. Example Implementations 34
B.1. Authorization Challenge Request Parameters 34
B.2. Authorization Challenge Response Parameters 35
B.3. Example Sequence - Initial Authorization 35
B.4. Example Sequence - Refresh Token Request Triggering Additional Authorization 36
Appendix C. Design Goals 38
Appendix D. Document History 39
Acknowledgments 40
Authors' Addresses 40

1. Introduction

This document, OAuth for First-Party Apps (FiPA), extends the OAuth 2.0 Authorization Framework [RFC6749] with a new endpoint to support applications that want to control the process of obtaining authorization from the user using a native experience, with browser redirection as described in OAuth 2.0 for Native Apps [RFC8252] used only as a fallback when needed.

The client collects any initial information from the user and POSTs that information as well as information about the client's request to the Authorization Challenge Endpoint, and receives either an authorization code (as defined in Section 1.3.1 of [RFC6749] or an error code in response. The error code may indicate that the client can continue to prompt the user for more information, or can indicate that the client needs to launch a browser to have the user complete the flow in a browser.

The Authorization Challenge Endpoint is used to initiate the OAuth flow in place of redirecting or launching a browser to the authorization endpoint.

While a fully-delegated approach using the redirect-based Authorization Code grant is generally preferred, this draft provides a mechanism for the client to directly interact with the user. This requires a high degree of trust between the authorization server and the client, as there typically is for first-party applications. It should be considered only when a redirect-based approach introduces usability issues, for example, when switching context between a native application and the browser, disrupting the user journey and preventing task completion.

This draft also extends the token response (typically for use in response to a refresh token request) and resource server response to allow the authorization server or resource server to indicate that the client should re-request authorization from the user. This can include requesting step-up authentication by including parameters defined in [RFC9470] as well.

1.1. Usage and Applicability

This specification is designed for the security model of first-party applications. First-party applications are applications that are controlled by the same entity as the authorization server and the user understands them both as the same entity. This specification is designed to be used by first-party native applications, which includes both mobile and desktop applications.

Profiles of this specification that extend the usage to non-first-party use cases MUST describe how their application of this specification avoids the risks associated with third-party apps directly interacting with the user. For example, an extension of this specification that enables federation between native apps never actually asks any third-party app to collect credentials from the user, so avoids these risks.

Using this specification in scenarios other than those described may lead to unintended security and privacy problems for users and service providers.

If a service provides multiple apps, and expects users to use multiple apps on the same device, there may be better ways of sharing a user's login between the apps other than each app implementing this specification or using an SDK that implements this specification. For example, [OpenID.Native-SSO] provides a mechanism for one app to obtain new tokens by exchanging tokens from another app, without any user interaction. See Section 9.7 for more details.

Please review the entirety of Section 9, and when more than one first-party application is supported, Section 9.7.

1.2. Limitations of this specification

This draft defines the overall framework for delivering a native OAuth user authentication experience. The precise client宴都erver interactions used to authenticate the user (e.g., prompts, challenges, and step sequencing) are intentionally left to individual deployments and are out of scope for this specification.

This specification is intended to be profiled to standardize specific interaction patterns enabling a complete interoperable solution.

1.3. User Experience Considerations

It is important to consider the user experience implications of different authentication challenges as well as the device with which the user is attempting to authorize.

For example, requesting a user to enter a password on a limited-input device (e.g. TV) creates a lot of user friction while also exposing the user's password to anyone else in the room. On the other hand, using a challenge method that involves, for example, a fingerprint reader on the TV remote allowing for a FIDO2 passkey authentication would be a good experience.

The Authorization Server SHOULD consider the user’s device when presenting authentication challenges and developers SHOULD consider whether the device implementing this specification can provide a good experience for the user. If the combination of user device and authentication challenge methods creates a lot of friction or security risk, consider using a specification like OAuth 2.0 Device Authorization Grant [RFC8628]. If selecting OAuth 2.0 Device Authorization Grant [RFC8628] which uses a cross-device authorization mechanism, please incorporate the security best practices identified in Cross-Device Flows: Security Best Current Practice [I-D.ietf-oauth-cross-device-security].

This specification also allows for the Authorization Server (AS) to direct the user to a web browser based authorization experience if the AS is not able to authorize the user via the requesting client app. This "redirect-to-web" experience is necessary to allow the AS to manage the security and privacy risks associated with any specific authorization requested by the user’s client.

2. 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.

2.1. Terminology

This specification uses the terms "Access Token", "Authorization Code", "Authorization Endpoint", "Authorization Server" (AS), "Client", "Client Authentication", "Client Identifier", "Client Secret", "Grant Type", "Protected Resource", "Redirection URI", "Refresh Token", "Resource Owner", "Resource Server" (RS) and "Token Endpoint" defined by [RFC6749].

TODO: Replace RFC6749 references with OAuth 2.1

3. Protocol Overview

There are three primary ways this specification extends various parts of an OAuth system.

3.1. Initial Authorization Request

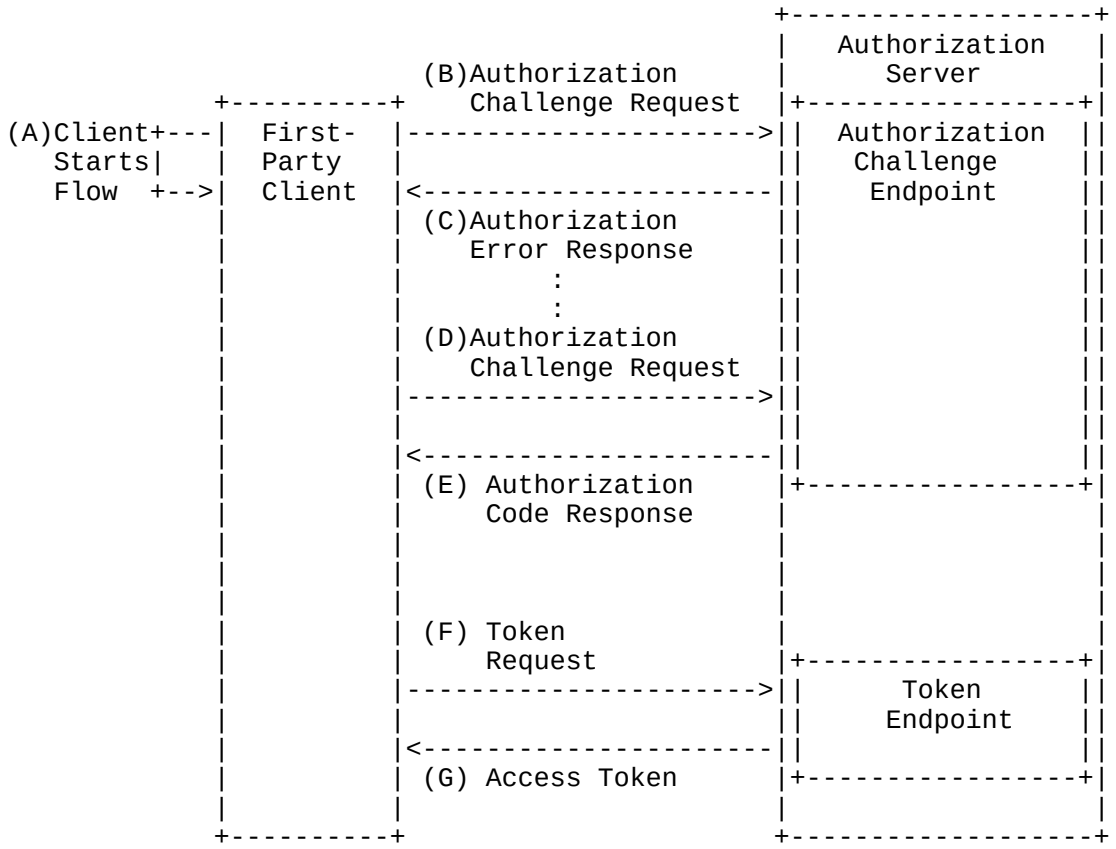

Figure: First-Party Client Authorization Code Request

* (A) The first-party client starts the flow, by presenting the user with a "sign in" button, or collecting information from the user, such as their email address or username (see Section 5.

* (B) The client initiates the authorization request by making a POST request to the Authorization Challenge Endpoint (see Section 5.1, optionally with information collected from the user (e.g. email or username)

* (C) The authorization server determines whether the information provided to the Authorization Challenge Endpoint is sufficient to grant authorization, and either responds with an authorization code or responds with an error (see Section 5.2). In this example, it determines that additional information is needed and responds with an error. The error may contain additional information to guide the Client on what information to collect

next. This pattern of collecting information, submitting it to the Authorization Challenge Endpoint and then receiving an error or authorization code may repeat several times.

* (D) The client gathers additional information (e.g. signed passkey challenge, or one-time code from email) and makes a POST request to the Authorization Challenge Endpoint.

* (E) The Authorization Challenge Endpoint returns an authorization code.

* (F) The client sends the authorization code received in step (E) to obtain a token from the Token Endpoint (see Section 6).

* (G) The Authorization Server returns an Access Token from the Token Endpoint.

3.2. Refresh Token Request

When the client uses a refresh token to obtain a new access token, the authorization server MAY respond with an error to indicate that re-authentication of the user is required.

3.3. Resource Request

When making a resource request to a resource server, the resource server MAY respond with an error according to OAuth 2.0 Step-Up Authentication Challenge Protocol [RFC9470], indicating that re-authentication of the user is required.

The use of [RFC9470] in this specification is for interoperability with its defined error signaling and does not propose changes to [RFC9470] itself.

4. Protocol Endpoints

4.1. Authorization Challenge Endpoint

The authorization challenge endpoint is a new endpoint defined by this specification which the first-party application uses to obtain an authorization code.

The authorization challenge endpoint is an HTTP API at the authorization server that accepts HTTP POST requests with parameters in the HTTP request message body using the application/x-www-form-urlencoded format. This format has a character encoding of UTF-8, as described in Appendix B of [RFC6749]. The authorization challenge endpoint URL MUST use the "https" scheme.

If the authorization server requires client authentication for this client on the Token Endpoint, then the authorization server MUST also require client authentication for this client on the Authorization Challenge Endpoint. See Section 9.4 for more details.

Authorization servers supporting this specification SHOULD include the URL of their authorization challenge endpoint in their authorization server metadata document [RFC8414] using the authorization_challenge_endpoint parameter as defined in Section 8.

The authorization challenge endpoint MUST accept the authorization request parameters as defined in [RFC6749] for the authorization endpoint as well as any authorization endpoint extensions supported by the authorization server. Examples of such extensions include Proof Key for Code Exchange (PKCE) [RFC7636], Resource Indicators [RFC8707], and OpenID Connect [OpenID]. Note that some extension parameters have meaning in a web context but don't have meaning in a native mechanism (e.g. response_mode=query). It is out of scope as to what the authorization server does in the case that an extension defines a parameter that has no meaning in this use case.

The client initiates the authorization flow with or without information collected from the user (e.g. a signed passkey challenge or MFA code).

The authorization challenge endpoint response is either an authorization code or an error code, and may also contain an auth_session which the client uses on subsequent requests.

Further communication between the client and authorization server MAY happen at the Authorization Challenge Endpoint or any other proprietary endpoints at the authorization server.

4.2. Token endpoint

The token endpoint is used by the client to obtain an access token by presenting its authorization grant or refresh token, as described in Section 3.2 of OAuth 2.0 [RFC6749].

This specification extends the token endpoint response to allow the authorization server to indicate that further authentication of the user is required.

5. Authorization Initiation

A client may wish to initiate an authorization flow by first prompting the user for their user identifier or other account information. The authorization challenge endpoint is a new endpoint to collect this login hint and direct the client with the next steps, whether that is to do an MFA flow, or perform an OAuth redirect-based flow. If the authorization server directs the client to complete the flow using a redirect-based authorization request in a browser, the client and authorization server SHOULD follow applicable best current practices for native apps (e.g., [RFC8252] and its successors) for redirect URI selection and external user-agent usage.

In order to preserve the security of this specification, the Authorization Server MUST verify the "first-partyness" of the client before continuing with the authentication flow. Please see Section 9.1 for additional considerations.

5.1. Authorization Challenge Request

The client makes a request to the authorization challenge endpoint by adding the following parameters, as well as parameters from any extensions, using the application/x-www-form-urlencoded format with a character encoding of UTF-8 in the HTTP request body:

"client_id": REQUIRED, unless the client is authenticating to the authorization server in a manner that unambiguously identifies the client, or the request includes an auth_session value associated with an existing session from which the authorization server can determine the client identity. The client MAY include the client_id even when one of these conditions applies. If it does, the authorization server MUST verify that the client_id identifies the same client as otherwise determined for the request, and MUST reject the request if it does not.

"scope": OPTIONAL. The OAuth scope defined in [RFC6749].

"auth_session": OPTIONAL. If the client has previously obtained an auth session, described in Section 5.3.1.

"code_challenge": OPTIONAL. The code challenge as defined by [RFC7636]. See Section 5.2.2.1.1 for details.

"code_challenge_method": OPTIONAL. The code challenge method as defined by [RFC7636]. See Section 5.2.2.1.1 for details.

"response_type": REQUIRED. Per section 3.1.1 of [RFC6749]

response_type is required, and for this specification MUST contain the value of code.

Specific implementations as well as extensions to this specification MAY define additional parameters to be used at this endpoint.

For example, the client makes the following request to initiate a flow given the user's phone number, line breaks shown for illustration purposes only:

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

login_hint=%2B1-310-123-4567&scope=profile
&client_id=bb16c14c73415
```

5.2. Authorization Challenge Response

The authorization server determines whether the information provided up to this point is sufficient to issue an authorization code, and if so responds with an authorization code. If the information is not sufficient for issuing an authorization code, then the authorization server MUST respond with an error response.

5.2.1. Authorization Code Response

The authorization server issues an authorization code 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:

"authorization_code": REQUIRED. The authorization code issued by the authorization server.

For example,

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

{
  "authorization_code": "uY29tL2F1dGhlbnRpY"
}
```

5.2.2. Error Response

If the request contains invalid parameters or incorrect data, or if the authorization server wishes to interact with the user directly, the authorization server responds with an HTTP 400 (Bad Request) status code (unless specified otherwise below) and includes the following parameters with the response.

Response parameters error, error_description, and error_uri are defined and used according to [RFC6749]. request_uri and expires_in are defined and used according to [RFC9126]. This specification defines the auth_session response parameter.

"error": REQUIRED. A single ASCII [USASCII] error code as described in in section Section 5.2.2.1

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

The authorization server MAY extend these error codes with custom messages based on the requirements of the authorization server.

"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.

"auth_session": OPTIONAL. The auth session allows the authorization server to associate subsequent requests by this client with an ongoing authorization request sequence. The client MUST include the auth_session in follow-up requests to the authorization challenge endpoint if it receives one along with the error response.

"request_uri": OPTIONAL. A request URI as described by [RFC9126] Section 2.2.

"expires_in": OPTIONAL. The lifetime of the request_uri in seconds, as described by [RFC9126] Section 2.2.

This specification requires the authorization server to define new error codes that relate to the actions the client must take in order to properly authenticate the user. These new error codes are specific to the authorization server's implementation of this specification and are intentionally left out of scope.

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.

The authorization server MAY define additional parameters in the response depending on the implementation. The authorization server MAY also define more specific content types for the error responses as long as the response is JSON and conforms to application/<AS-defined>+json.

5.2.2.1. Error Codes

This specification supports the use of error codes defined by [RFC6749] and other error codes defined by OAuth extensions supported by the Authorization Server.

This specification defines the following error codes.

"invalid_session": : The provided auth_session is invalid, expired, revoked, or is otherwise invalid.

"insufficient_authorization": : The presented authorization is insufficient, and the authorization server is requesting the client to take additional steps to complete the authorization.

"redirect_to_web": : The request is not able to be fulfilled with any further direct interaction with the user. Instead, the client should initiate a new authorization code flow so that the user interacts with the authorization server in a web browser. See Section 5.2.2.1.1 for details.

5.2.2.1.1. Redirect to Web Error Response

The authorization server may choose to interact directly with the user based on a risk assesment, the introduction of a new authentication method not supported in the application, or to handle an exception flow like account recovery. To indicate this error to the client, the authorization server returns an error response as defined above with the redirect_to_web error code.

The response MAY include a request_uri, in which case the client is expected to use it to initiate an authorization request as described in Section 4 of [RFC9126].

If no request_uri is returned, the client is expected to initiate a new OAuth Authorization Code flow with PKCE according to [RFC6749] and [RFC7636].

If the client expects the frequency of this error response to be high, the client SHOULD include a PKCE [RFC7636] code_challenge in the initial authorization challenge request.

If the client does not include a PKCE code_challenge in the initial authorization challenge request, the authorization server MUST NOT return a request_uri in the redirect_to_web error response, as that would effectively be the same as a PAR request without PKCE.

5.3. Intermediate Requests

If the authorization server returns an insufficient_authorization error as described above, this is an indication that there is further information the client should request from the user, and continue to make requests to the authorization server until the authorization request is fulfilled and an authorization code returned.

These intermediate requests are out of scope of this specification, and are expected to be defined by the authorization server. The format of these requests is not required to conform to the format of the initial authorization challenge requests (e.g. the request format may be application/json rather than application/x-www-form-urlencoded).

These intermediate requests MAY also be sent to proprietary endpoints at the authorization server rather than the Authorization Challenge Endpoint.

5.3.1. Auth Session

The auth_session is a value that the authorization server issues in order to be able to associate subsequent requests from the same client. It is intended to be analagous to how a browser cookie associates multiple requests by the same browser to the authorization server.

The auth_session value is completely opaque to the client, and as such the authorization server MUST adequately protect the value from inspection by the client.

If the client has an auth_session, the client MUST include it in future requests to the authorization challenge endpoint. The client MUST store the auth_session beyond the issuance of the authorization code to be able to use it in future requests.

Every response defined by this specification may include a new auth_session value. Clients MUST NOT assume that auth_session values are static, and MUST be prepared to update the stored auth_session value if one is received in a response.

To mitigate the risk of session hijacking, the auth_session SHOULD be bound to the device, and the authorization server SHOULD reject an auth_session if it is presented from a different device than the one it was bound to. One method of binding the auth_session to the device is described in Section 9.6.1.

The AS MUST ensure that the auth_session value is unique to the session and protected from accidental collisions. For example, if the AS is using a random string for the auth_session value, the value SHOULD have a minimum of 256 bits of entropy.

See Section 9.6 for additional security considerations.

6. Token Request

The client makes a request to the token endpoint using the authorization code it obtained from the authorization challenge endpoint.

This specification does not define any additional parameters beyond the token request parameters defined in Section 4.1.3 of [RFC6749]. However, notably, the redirect_uri parameter will not be included in this request, because no redirect_uri parameter was included in the authorization request.

6.1. Token Endpoint Successful Response

This specification extends the OAuth 2.0 [RFC6749] token response defined in Section 5.1 with the additional parameter auth_session, defined in Section 5.3.1.

An example successful token response is below:

```
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",
  "auth_session": "uY29tL2F1dGhlbnRpY"
}
```

The response MAY include an auth_session parameter which the client is expected to include on any subsequent requests to the authorization challenge endpoint, as described in Section 5.3.1. The auth_session parameter MAY also be included even if the authorization code was obtained through a traditional OAuth authorization code flow rather than the flow defined by this specification.

The auth_session mechanism described in Section 5.3.1 is an optional feature the authorization server can leverage in order to enable flows such as step-up authentication [RFC9470], so that the authorization server can restore the context of a previous session and prompt only for the needed step-up factors. See Appendix A.7 for an example application.

6.2. Token Endpoint Error Response

Upon any request to the token endpoint, including a request with a valid refresh token, the authorization server can respond with an authorization challenge instead of a successful access token response.

An authorization challenge error response is a particular type of error response as defined in Section 5.2 of OAuth 2.0 [RFC6749] where the error code is set to the following value:

"error": "insufficient_authorization": The presented authorization is insufficient, and the authorization server is requesting the client take additional steps to complete the authorization.

The response MAY also contain an auth_session parameter which the client is expected to include on a subsequent request to the authorization challenge endpoint.

"auth_session": OPTIONAL. The optional auth session value allows the authorization server to associate subsequent requests by this client with an ongoing authorization request sequence. The client MUST include the auth_session in follow-up requests to the challenge endpoint if it receives one along with the error response.

Additionally, the response MAY contain custom values that describe instructions for how the client should proceed to interact with the user.

For example:

```
HTTP/1.1 403 Forbidden
Content-Type: application/json
Cache-Control: no-store

{
  "error": "insufficient_authorization",
  "auth_session": "uY29tL2F1dGhlbnRpY",
  "otp_required": true
}
```

7. Resource Server Error Response

Step-Up Authentication [RFC9470] defines error code values that a resource server can use to tell the client to start a new authorization request including acr_values and max_age from [OpenID]. This specification reuses the Step-Up Authentication [RFC9470] error response to initiate a first party authorization flow to satisfy the step-up authentication request.

Upon receiving this error response, the client starts a new first-party authorization request at the authorization challenge endpoint, and includes the acr_values, max_age and scope that were returned in the error response.

This specification does not update or alter [RFC9470] resource server error behaviour and does not define any new parameters for the resource server error response beyond those defined in [RFC9470] and [RFC6750]. It only defines first party client behavior for continuing authorization at the authorization challenge endpoint when such an error is received.

8. Authorization Server Metadata

The following authorization server metadata parameters [RFC8414] are introduced to signal the server’s capability and policy with respect to first-party applications.

"authorization_challenge_endpoint": The URL of the authorization challenge endpoint at which a client can initiate an authorization request and eventually obtain an authorization code.

9. Security Considerations

9.1. First-Party Applications

First-party applications are applications that are controlled by the same entity as the authorization server used by the application, and the user understands them both as the same entity.

For first-party applications, it is important that the user recognizes the application and authorization server as belonging to the same brand. For example, a bank publishing their own mobile application.

Because this specification enables a client application to interact directly with the end user, and the application handles sending any information collected from the user to the authorization server, it is expected to be used only for first-party applications when the authorization server also has a high degree of trust of the client.

This specification is not prescriptive on how the Authorization Server establishes its trust in the first-partyness of the application. For mobile platforms, most support some mechanism for application attestation that can be used to identify the entity that created/signed/uploaded the app to the app store. App attestation can be combined with mechanisms such as Attestation-Based Client Authentication [[I-D.ietf-oauth-attestation-based-client-auth]] or Dynamic Client Registration [RFC7591] to enable strong client authentication in addition to client verification (first-partyness). The exact steps required are out of scope for this specification. Note that applications running inside a browser (e.g. Single Page Apps) context it is much more difficult to verify the first-partyness of the client. Please see Section 9.8 for additional details.

9.2. Phishing

There are two ways using this specification increases the risk of phishing.

1. Malicious application: With this specification, the client interacts directly with the end user, collecting information provided by the user and sending it to the authorization server. If an attacker impersonates the client and successfully tricks a user into using it, they may not realize they are giving their credentials to the malicious application.

2. User education: In a traditional OAuth deployment using the redirect-based authorization code flow, the user will only ever enter their credentials at the authorization server, and it is straightforward to explain to avoid entering credentials in other "fake" websites. By introducing a new place the user is expected to enter their credentials using this specification, it is more complicated to teach users how to recognize other fake login prompts that might be attempting to steal their credentials.

Because of these risks, the authorization server MAY decide to require that the user go through a redirect-based flow at any stage of the process based on its own risk assessment.

9.3. Credential Stuffing Attacks

The authorization challenge endpoint is capable of directly receiving user credentials and other authentication material like OTPs. This exposes a new vector to perform credential stuffing or brute force attacks if additional measures are not taken to ensure the authenticity of the application.

An authorization server may already have a combination of built-in or 3rd party security tools in place to monitor and reduce this risk in browser-based authentication flows. Implementors SHOULD consider similar security measures to reduce this risk in the authorization challenge endpoint. Additionally, the attestation APIs SHOULD be used when possible to assert a level of confidence to the authorization server that the request is originating from an application owned by the same party.

Implementors SHOULD rate-limit requests from the same auth_session.

9.4. Client Authentication

Typically, mobile and desktop applications are considered "public clients" in OAuth, since they cannot be shipped with a statically configured set of client credentials [RFC8252]. Because of this, client impersonation should be a concern of anyone deploying this pattern. Without client authentication, a malicious user or attacker can mimick the requests the application makes to the authorization server, pretending to be the legitimate client.

Implementers SHOULD consider additional measures to limit the risk of client impersonation, such as using attestation APIs available from the operating system.

9.5. Sender-Constrained Tokens

Tokens issued in response to an authorization challenge request SHOULD be sender constrained to mitigate the risk of token theft and replay.

Proof-of-Possession techniques constrain tokens by binding them to a cryptographic key. Whenever the token is presented, it MUST be accompanied by a proof that the client presenting the token also controls the cryptographic key bound to the token. If a proof-of-possession sender constrained token is presented without valid proof of possession of the cryptographic key, it MUST be rejected.

9.5.1. DPoP: Demonstrating Proof-of-Possession

DPoP [RFC9449] is an application-level mechanism for sender-constraining OAuth [RFC6749] access and refresh tokens. If DPoP is used to sender constrain tokens, the client SHOULD use DPoP for every token request to the Authorization Server and interaction with the Resource Server.

DPoP includes an optional capability to bind the authorization code to the DPoP key to enable end-to-end binding of the entire authorization flow. Given the back-channel nature of this specification, there are far fewer opportunities for an attacker to access the authorization code and PKCE code verifier compared to the redirect-based Authorization Code Flow. In this specification, the Authorization Code is obtained via a back-channel request. Despite this, omitting Authorization Code binding leaves a gap in the end-to-end protection that DPoP provides, so DPoP Authorization Code binding SHOULD be used.

The mechanism for Authorization Code binding with DPoP is similar as that defined for Pushed Authorization Requests (PARs) in Section 10.1 of [RFC9449]. In order to bind the Authorization Code with DPoP, the client MUST add the DPoP header to the Authorization Challenge Request. The authorization server MUST check the DPoP proof JWT that was included in the DPoP header as defined in Section 4.3 of [RFC9449]. The authorization server MUST ensure that the same key is used in all subsequent Authorization Challenge Requests and in the eventual token request. The authorization server MUST reject subsequent Authorization Challenge Requests, or the eventual token request, unless a DPoP proof for the same key presented in the original Authorization Challenge Request is provided.

The above mechanism simplifies the implementation of the client, as it can attach the DPoP header to all requests to the authorization server regardless of the type of request. This mechanism provides a stronger binding than using the dpop_jkt parameter, as the DPoP header contains a proof of possession of the private key.

9.5.2. Other Proof of Possession Mechanisms

It may be possible to use other proof of possession mechanisms to sender constrain access and refresh tokens. Defining these mechanisms are out of scope for this specification.

9.6. Auth Session

Binding the auth_session to the device requesting authorization is important to prevent session hijacking and replay of the auth_session value. Without the device binding a captured auth_session could be replayed from another device. The following section describes one way to bind the auth_session to the requesting device. Other device binding methods are available and useable to prevent this potential security exposure.

9.6.1. Auth Session DPoP Binding

If the client and authorization server are using DPoP binding of access tokens and/or authorization codes, then the auth_session value SHOULD be protected as well. The authorization server SHOULD associate the auth_session value with the DPoP public key. This removes the need for the authorization server to include additional claims in the DPoP proof, while still benefitting from the assurance that the client presenting the proof has control over the DPoP key. To associate the auth_session value with the DPoP public key, the authorization server:

* MUST check that the same DPoP public key is being used when the client presents the DPoP proof.

* MUST verify the DPoP proof to ensure the client controls the corresponding private key whenever the client includes the auth_session in an Authorization Challenge Request as described in Section 5.1.

DPoP binding of the auth_session value ensures that the context referenced by the auth_session cannot be stolen and reused by another device.

9.6.2. Auth Session Lifetime

This specification makes no requirements or assumptions on the lifetime of the auth_session value. The lifetime and expiration is at the discretion of the authorization server, and the authorization server may choose to invalidate the value for any reason such as scheduled expiration, security events, or revocation events.

Clients MUST NOT make any assumptions or depend on any particular lifetime of the auth_session value.

9.7. Multiple Applications

When multiple first-party applications are supported by the AS, then it is important to consider a number of additional risks. These risks fall into two main categories: Experience Risk and Technical Risk which are described below.

9.7.1. User Experience Risk

Any time a user is asked to provide the authentication credentials in user experiences that differ, it has the effect of increasing the likelihood that the user will fall prey to a phishing attack because they are used to entering credentials in different looking experiences. When multiple first-party applications are supported, the implementation MUST ensure the native experience is identical across all the first-party applications.

Another experience risk is user confusion caused by different looking experiences and behaviors. This can increase the likelihood the user will not complete the authentication experience for the first-party application.

9.7.2. Technical Risk

In addition to the experience risks, multiple implementations in first-party applications increases the risk of an incorrect implementation as well as increasing the attack surface as each implementation may expose its own weaknesses.

9.7.3. Mitigation

To address these risks, when multiple first-party applications must be supported, and other methods such as [OpenID.Native-SSO] are not applicable, it is RECOMMENDED that a client-side SDK be used to ensure the implementation is consistent across the different applications and to ensure the user experience is identical for all first-party apps.

9.8. Single Page Applications

Single Page Applications (SPA) run in a scripting language inside the context of a browser instance. This environment poses several unique challenges compared to native applications, in particular:

* Significant attack vectors due to the possibility of Cross-Site Scripting (XSS) attacks

* Fewer options to securely attest to the first-partyness of a browser based application

See [I-D.ietf-oauth-browser-based-apps] for a detailed discussion of the risks of XSS attacks in browsers.

Additionally, the nature of a Single-Page App means the user is already in a browser context, so the user experience cost of doing a full page redirect or a popup window for the traditional OAuth Authorization Code Flow is much less than the cost of doing so in a native application. The complexity and risk of implementing this specification in a browser likely does not outweigh the user experience benefits that would be gained in that context.

For these reasons, it is NOT RECOMMENDED to use this specification in browser-based applications.

10. IANA Considerations

10.1. OAuth Parameters Registration

IANA has (TBD) registered the following values in the IANA "OAuth Parameters" registry of [IANA.oauth-parameters] established by [RFC6749].

Parameter name: auth_session

Parameter usage location: token response

Change Controller: IETF

Specification Document: Section 5.4 of this specification

10.2. OAuth Server Metadata Registration

IANA has (TBD) registered the following values in the IANA "OAuth Authorization Server Metadata" registry of [IANA.oauth-parameters] established by [RFC8414].

Metadata Name: authorization_challenge_endpoint

Metadata Description: URL of the authorization server's authorization challenge endpoint.

Change Controller: IESG

Specification Document: Section 4.1 of [[this specification]]

11. References

11.1. Normative References

[I-D.ietf-oauth-cross-device-security]
Kasselman, P., Fett, D., and F. Skokan, "Cross-Device Flows: Security Best Current Practice", Work in Progress, Internet-Draft, draft-ietf-oauth-cross-device-security-15, 23 January 2026, <https://datatracker.ietf.org/doc/html/draft-ietf-oauth-cross-device-security-15>.

[IANA.JWT] "*** BROKEN REFERENCE ***".

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

[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.Native-SSO]
Fletcher, G., "OpenID Connect Native SSO for Mobile Apps", November 2022, <https://openid.net/specs/openid-connect-native-sso-1_0.html>.

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

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

[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/rfc/rfc7159>.

[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/rfc/rfc7515>.

[RFC7519] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <https://www.rfc-editor.org/rfc/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/rfc/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/rfc/rfc7636>.

[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/rfc/rfc8174>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/rfc/rfc8259>.

[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/rfc/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/rfc/rfc8628>.

[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/rfc/rfc8707>.

[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/rfc/rfc9126>.

[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/rfc/rfc9449>.

[RFC9470] Bertocci, V. and B. Campbell, "OAuth 2.0 Step Up Authentication Challenge Protocol", RFC 9470, DOI 10.17487/RFC9470, September 2023, <https://www.rfc-editor.org/rfc/rfc9470>.

[SHS] Technology, N. I. of S. and., ""Secure Hash Standard (SHS)", FIPS PUB 180-4, DOI 10.6028/NIST.FIPS.180-4", August 2015, <http://dx.doi.org/10.6028/NIST.FIPS.180-4>.

[USASCII] Institute, A. N. S., "Coded Character Set -- 7-bit American Standard Code for Information Interchange, ANSI X3.4", 1986.

11.2. Informative References

[I-D.ietf-oauth-attestation-based-client-auth] Looker, T., Bastian, P., and C. Bormann, "OAuth 2.0 Attestation-Based Client Authentication", Work in Progress, Internet-Draft, draft-ietf-oauth-attestation-based-client-auth-07, 15 September 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-oauth-attestation-based-client-auth-07>.

[I-D.ietf-oauth-browser-based-apps] Parecki, A., De Ryck, P., and D. Waite, "OAuth 2.0 for Browser-Based Applications", Work in Progress, Internet-Draft, draft-ietf-oauth-browser-based-apps-26, 3 December 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-oauth-browser-based-apps-26>.

[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/rfc/rfc6750>.

[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/rfc/rfc8252>.

Appendix A. Example User Experiences

This section provides non-normative examples of how this specification may be used to support specific use cases.

A.1. Passkey

A user may log in with a passkey (without a password).

1. The Client collects the username from the user.
2. The Client sends an Authorization Challenge Request (Section 5.1) to the Authorization Challenge Endpoint (Section 4.1) including the username.
3. The Authorization Server verifies the username and returns a challenge
4. The user is prompted for verification with biometrics or a PIN, enabling the Client to sign the challenge using the passkey.
5. The Client sends the signed challenge, username, and credential ID to the Authorization Challenge Endpoint (Section 4.1).
6. The Authorization Server verifies the signed challenge and returns an Authorization Code.
7. The Client requests an Access Token and Refresh Token by issuing a Token Request (Section 6) to the Token Endpoint.
8. The Authorization Server verifies the Authorization Code and issues the requested tokens.

A.2. Redirect to Authorization Server

A user may be redirected to the Authorization Server to perfrom an account reset.

1. The Client collects username from the user.
2. The Client sends an Authorization Challenge Request (Section 5.1) to the Authorization Challenge Endpoint (Section 4.1) including the username.
3. The Authorization Server verifies the username and determines that the account is locked and returns a Redirect error response.
4. The Client parses the redirect message, opens a browser and redirects the user to the Authorization Server performing an OAuth 2.0 flow with PKCE.
5. The user resets their account by performing a multi-step authentication flow with the Authorization Server.

6. The Authorization Server issues an Authorization Code in a redirect back to the client, which then exchanges it for an access and refresh token.

A.3. Passwordless One-Time Password (OTP)

In a passwordless One-Time Password (OTP) scheme, the user is in possession of a one-time password generator. This generator may be a hardware device, or implemented as an app on a mobile phone. The user provides a user identifier and one-time password, which is verified by the Authorization Server before it issues an Authorization Code, which can be exchanged for an Access and Refresh Token.

1. The Client collects username and OTP from user.

2. The Client sends an Authorization Challenge Request (Section 5.1) to the Authorization Challenge Endpoint (Section 4.1) including the username and OTP.

3. The Authorization Server verifies the username and OTP and returns an Authorization Code.

4. The Client requests an Access Token and Refresh Token by issuing a Token Request (Section 6) to the Token Endpoint.

5. The Authorization Server verifies the Authorization Code and issues the requested tokens.

A.4. E-Mail Confirmation Code

A user may be required to provide an e-mail confirmation code as part of an authentication ceremony to prove they control an e-mail address. The user provides an e-mail address and is then required to enter a verification code sent to the e-mail address. If the correct verification code is returned to the Authorization Server, it issues Access and Refresh Tokens.

1. The Client collects an e-mail address from the user.

2. The Client sends the e-mail address in an Authorization Challenge Request (Section 5.1) to the Authorization Challenge Endpoint (Section 4.1).

3. The Authorization Server sends a verification code to the e-mail address and returns an Error Response (Section 5.2.2) including "error": "insufficient_authorization", "auth_session" and a custom property indicating that an e-mail verification code must be entered.

4. The Client presents a user experience guiding the user to copy the e-mail verification code to the Client. Once the e-mail verification code is entered, the Client sends an Authorization Challenge Request to the Authorization Challenge Endpoint, including the e-mail verification code as well as the auth_session parameter returned in the previous Error Response.

5. The Authorization Server uses the auth_session to maintain the session and verifies the e-mail verification code before issuing an Authorization Code to the Client.

6. The Client sends the Authorization Code in a Token Request (Section 6) to the Token Endpoint.

7. The Authorization Server verifies the Authorization Code and issues the Access Token and Refresh Token.

An alternative version of this verification involves the user clicking a link in an email rather than manually entering a verification code. This is typically done for email verification flows rather than inline in a login flow. The protocol-level details remain the same for the alternative flow despite the different user experience. All steps except step 4 above remain the same, but the client presents an alternative user experience for step 4 described below:

* The Client presents a message to the user instructing them to click the link sent to their email address. The user clicks the link in the email, which contains the verification code in the URL. The URL launches the app providing the verification code to the Client. The Client sends the verification code and auth_session to the Authorization Challenge Endpoint.

A.5. Mobile Confirmation Code

A user may be required to provide a confirmation code as part of an authentication ceremony to prove they control a mobile phone number. The user provides a phone number and is then required to enter a confirmation code sent to the phone. If the correct confirmation code is returned to the Authorization Server, it issues Access and Refresh Tokens.

1. The Client collects a mobile phone number from the user.

2. The Client sends the phone number in an Authorization Challenge Request (Section 5.1) to the Authorization Challenge Endpoint (Section 4.1).

3. The Authorization Server sends a confirmation code to the phone number and returns an Error Response (Section 5.2.2) including "error": "insufficient_authorization", "auth_session" and a custom property indicating that a confirmation code must be entered.

4. The Client presents a user experience guiding the user to enter the confirmation code. Once the code is entered, the Client sends an Authorization Challenge Request to the Authorization Challenge Endpoint, including the confirmation code as well as the auth_session parameter returned in the previous Error Response.

5. The Authorization Server uses the auth_session to maintain the session context and verifies the code before issuing an Authorization Code to the Client.

6. The Client sends the Authorization Code in a Token Request (Section 6) to the Token Endpoint.

7. The Authorization Server verifies the Authorization Code and issues the Access Token and Refresh Token.

A.6. Re-authenticating to an app a week later using OTP

A client may be in possession of an Access and Refresh Token as the result of a previous succesful user authentication. The user returns to the app a week later and accesses the app. The Client presents the Access Token, but receives an error indicating the Access Token is no longer valid. The Client presents a Refresh Token to the Authorization Server to obtain a new Access Token. If the Authorization Server requires user interaction for reasons based on its own policies, it rejects the Refresh Token and the Client re-starts the user authentication flow to obtain new Access and Refresh Tokens.

1. The Client has a short-lived access token and long-lived refresh token following a previous completion of an Authorization Grant Flow which included user authentication.

2. A week later, the user launches the app and tries to access a protected resource at the Resource Server.

3. The Resource Server responds with an error code indicating an invalid access token since it has expired.

4. The Client presents the refresh token to the Authorization Server to obtain a new access token (section 6 [RFC6749])

5. The Authorization Server responds with an error code indicating that an OTP from the user is required, as well as an auth_session.

6. The Client prompts the user to enter an OTP.

7. The Client sends the OTP and auth_session in an Authorization Challenge Request (Section 5.1) to the Authorization Challenge Endpoint (Section 4.1).

8. The Authorization Server verifies the auth_session and OTP, and returns an Authorization Code.

9. The Client sends the Authorization Code in a Token Request (Section 6) to the Token Endpoint.

10. The Authorization Server verifies the Authorization Code and issues the requested tokens.

11. The Client presents the new Access Token to the Resource Server in order to access the protected resource.

A.7. Step-up Authentication using Confirmation SMS

A Client previously obtained an Access and Refresh Token after the user authenticated with an OTP. When the user attempts to access a protected resource, the Resource Server determines that it needs an additional level of authentication and triggers a step-up authentication, indicating the desired level of authentication using acr_values and max_age as defined in the Step-up Authentication specification. The Client initiates an authorization request with the Authorization Server indicating the acr_values and max_age parameters. The Authorization Server responds with error messages promptng for additional authentication until the acr_values and max_age values are satisfied before issuing fresh Access and Refresh Tokens.

1. The Client has a short-lived access token and long-lived refresh token following the completion of an Authorization Code Grant Flow which included user authentication.

2. When the Client presents the Access token to the Resource Server, the Resource Server determines that the acr claim in the Access Token is insufficient given the resource the user wants to access and responds with an insufficient_user_authentication error code, along with the desired acr_values and desired max_age.

3. The Client sends an Authorization Challenge Request (Section 5.1) to the Authorization Challenge Endpoint (Section 4.1) including the auth_session, acr_values and max_age parameters.

4. The Authorization Server verifies the auth_session and determines which authentication methods must be satisfied based on the acr_values, and responds with an Error Response (Section 5.2.2) including "error": "insufficient_authorization" and a custom property indicating that an OTP must be entered.

5. The Client prompts the user for an OTP, which the user obtains and enters.

6. The Client sends an Authorization Challenge Request to the Authorization Challenge Endpoint including the auth_session and OTP.

7. The Authorization Server verifies the OTP and returns an Authorization Code.

8. The Client sends the Authorization Code in a Token Request (Section 6) to the Token Endpoint.

9. The Authorization Server verifies the Authorization Code and issues an Access Token with the updated acr value along with the Refresh Token.

10. The Client presents the Access Token to the Resources Server, which verifies that the acr value meets its requirements before granting access to the protected resource.

A.8. Registration

This example describes how to use the mechanisms defined in this draft to create a complete user registration flow starting with an email address. In this example, it is the Authorization Server’s policy to allow these challenges to be sent to email and phone number that were previously unrecognized, and creating the user account on the fly.

1. The Client collects a username from the user.

2. The Client sends an Authorization Challenge Request (Section 5.1) to the Authorization Challenge Endpoint (Section 4.1) including the username.

3. The Authorization Server returns an Error Response (Section 5.2.2) including "error": "insufficient_authorization", "auth_session", and a custom property indicating that an e-mail address must be collected.

4. The Client collects an e-mail address from the user.

5. The Client sends the e-mail address as part of a second Authorization Challenge Request to the Authorization Challenge Endpoint, along with the auth_session parameter.

6. The Authorization Server sends a verification code to the e-mail address and returns an Error Response including "error": "insufficient_authorization", "auth_session" and a custom property indicating that an e-mail verification code must be entered.

7. The Client presents a user experience guiding the user to copy the e-mail verification code to the Client. Once the e-mail verification code is entered, the Client sends an Authorization Challenge Request to the Authorization Challenge Endpoint, including the e-mail verification code as well as the auth_session parameter returned in the previous Error Response.

8. The Authorization Server uses the auth_session to maintain the session context, and verifies the e-mail verification code. It determines that it also needs a phone number for account recovery purposes and returns an Error Response including "error": "insufficient_authorization", "auth_session" and a custom property indicating that a phone number must be collected.

9. The Client collects a mobile phone number from the user.

10. The Client sends the phone number in an Authorization Challenge Request to the Authorization Challenge Endpoint, along with the auth_session.

11. The Authorization Server uses the auth_session parameter to link the previous requests. It sends a confirmation code to the phone number and returns an Error Response including "error": "insufficient_authorization", "auth_session" and a custom property indicating that a SMS confirmation code must be entered.

12. The Client presents a user experience guiding the user to enter the SMS confirmation code. Once the SMS verification code is entered, the Client sends an Authorization Challenge Request to the Authorization Challenge Endpoint, including the confirmation code as well as the auth_session parameter returned in the previous Error Response.

13. The Authorization Server uses the auth_session to maintain the session context, and verifies the SMS verification code before issuing an Authorization Code to the Client.

14. The Client sends the Authorization Code in a Token Request (Section 6) to the Token Endpoint.

15. The Authorization Server verifies the Authorization Code and issues the requested tokens.

Appendix B. Example Implementations

In order to successfully implement this specification, the Authorization Server will need to define its own specific profile for what values clients are expected to send in the Authorization Challenge Request (Section 5.1), as well as AS-defined specific error codes in the Authorization Challenge Response (Section 5.2).

It is expected that service providers will wrap the implementation of this specification in an SDK which will be used by application developers, removing the need for application developers to implement the specification themselves.

Below is an example profile that allows for a successful implementation that enables the user to log in with a username and OTP. This example is included for illustration purposes only to help AS developers define the profile for their deployment and ecosystem.

B.1. Authorization Challenge Request Parameters

In addition to the request parameters defined in Section 5.1, the authorization server defines the additional parameters below.

"username": REQUIRED for the initial Authorization Challenge

Request.

"otp": The OTP collected from the user. REQUIRED when re-trying an Authorization Challenge Request in response to the otp_required error defined below.

B.2. Authorization Challenge Response Parameters

In addition to the response parameters defined in Section 5.2, the authorization server defines the additional parameter below.

"otp_required": The client should collect an OTP from the user and send the OTP in a second request to the Authorization Challenge Endpoint. The HTTP response code to use with this error value is 401 Unauthorized.

B.3. Example Sequence - Initial Authorization

The client prompts the user to enter their username, and sends the username in an initial Authorization Challenge Request.

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

username=alice
&scope=photos
&client_id=bb16c14c73415
```

The Authorization Server sends an error response indicating that an OTP is required.

```
HTTP/1.1 401 Unauthorized
Content-Type: application/json
Cache-Control: no-store

{
  "error": "insufficient_authorization",
  "auth_session": "ce6772f5e07bc8361572f",
  "otp_required": true
}
```

The client prompts the user for an OTP, and sends a new Authorization Challenge Request.

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

auth_session=ce6772f5e07bc8361572f
&otp=555121
```

The Authorization Server validates the auth_session to find the expected user, then validates the OTP for that user, and responds with an authorization code.

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

{
  "authorization_code": "uY29tL2F1dGhlbnRpY"
}
```

The client sends the authorization code to the token endpoint.

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

grant_type=authorization_code
&client_id=bb16c14c73415
&code=uY29tL2F1dGhlbnRpY
```

The Authorization Server responds with an access token and refresh token.

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

{
  "token_type": "Bearer",
  "expires_in": 3600,
  "access_token": "d41c0692f1187fd9b326c63d",
  "refresh_token": "e090366ac1c448b8aed84cbc07"
}
```

B.4. Example Sequence - Refresh Token Request Triggering Additional Authorization

This example illustrates the use case described in Appendix A.6.

The client sends a refresh token request to obtain a new access token.

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

grant_type=refresh_token
&client_id=bb16c14c73415
&refresh_token=e090366ac1c448b8aed84cbc07
```

The Authorization Server determines that additional authorization is required (for example, step-up or re-verification) before the refresh token can be used, and returns an authorization challenge error response.

```
HTTP/1.1 403 Forbidden
Content-Type: application/json
Cache-Control: no-store

{
  "error": "insufficient_authorization",
  "auth_session": "ce6772f5e07bc8361572f",
  "otp_required": true
}
```

The client prompts the user for an OTP, and sends an Authorization Challenge Request to continue the authorization session identified by auth_session.

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

auth_session=ce6772f5e07bc8361572f
&otp=555121
```

The Authorization Server validates the auth_session to find the expected user, then validates the OTP for that user, and responds with an authorization code.

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

{
  "authorization_code": "uY29tL2F1dGhlbnRpY"
}
```

The client sends the authorization code to the token endpoint.

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

grant_type=authorization_code
&client_id=bb16c14c73415
&code=uY29tL2F1dGhlbnRpY
```

The Authorization Server responds with an access token and new refresh token.

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

{
  "token_type": "Bearer",
  "expires_in": 3600,
  "access_token": "d41c0692f1187fd9b326c63d",
  "refresh_token": "f2b0f7d9a41f4d59a8c1e9b3ab"
}
```

Appendix C. Design Goals

This specification defines a new authorization flow the client can use to obtain an authorization grant. There are two primary reasons for designing the specification this way.

This enables existing OAuth implementations to make fewer modifications to existing code by not needing to extend the token endpoint with new logic. Instead, the new logic can be encapsulated in an entirely new endpoint, the output of which is an authorization code which can be redeemed for an access token at the existing token endpoint.

This also mirrors more closely the existing architecture of the redirect-based authorization code flow. In the authorization code flow, the client first initiates a request by redirecting a browser to the authorization endpoint, at which point the authorization server takes over with its own custom logic to authenticate the user in whatever way appropriate, possibly including interacting with other endpoints for the actual user authentication process. Afterwards, the authorization server redirects the user back to the client application with an authorization code in the query string. This specification mirrors the existing approach by having the client first make a POST request to the Authorization Challenge Endpoint, at which point the authorization server provides its own custom logic to authenticate the user, eventually returning an authorization code.

An alternative design would be to define new custom grant types for the different authentication factors such as WebAuthn, OTP, etc. The drawback to this design is that conceptually, these authentication methods do not map to an OAuth grant. In other words, the OAuth authorization grant captures the user's intent to authorize access to some data, and that authorization is represented by an authorization code, not by different methods of authenticating the user.

Another alternative option would be to have the Authorization Challenge Endpoint return an access token upon successful authentication of the user. This was deliberately not chosen, as this adds a new endpoint that tokens would be returned from. In most deployments, the Token Endpoint is the only endpoint that actually issues tokens, and includes all the implmentation logic around token binding, rate limiting, etc. Instead of defining a new endpoint that issues tokens which would have to have similar logic and protections, instead the new endpoint only issues authorization codes, which can be exchanged for tokens at the existing Token Endpoint just like in the redirect-based Authorization Code flow.

These design decisions should enable authorization server implementations to isolate and encapsulate the changes needed to support this specification.

Appendix D. Document History

-03

* Editorial clarifications and improvements

* Pointed definition of authorization code to section 1.3.1 of RFC 6749

* Updated auth_session binding requirements to SHOULD, and added a reference to DPoP

* Revised introduction and context to clarify this draft is intended for first-party applications but can be extended for third-party in some situations

* Added response_type=code as a required parameter to match RFC 6749

-02

* Updated affiliations and acks

* Editorial clarifications

* Added reference to Attestation-Based Client Authentication

-01

* Corrected "re-authorization of the user" to "re-authentication of the user"

-00

* Adopted into the OAuth WG, no changes from previous individual draft

Acknowledgments

The authors would like to thank the attendees of the OAuth Security Workshop 2023 session in which this was discussed, as well as the following individuals who contributed ideas, feedback, and wording that shaped and formed the final specification:

Alejo Fernandez, Brian Campbell, Dean Saxe, Dick Hardt, Dmitry Telegin, Evert Pot, Janak Amarasena, Jeff Corrigan, John Bradley, Justin Richer, Kristina Yasuda, Martin Besozzi, Matt MacAdam, Mike Jones, Orie Steele, Tim Cappalli, Tobias Looker, Yaron Sheffer, Yaron Zehavi.

Authors' Addresses

Aaron Parecki
Okta
Email: aaron@parecki.com

George Fletcher
Practical Identity LLC
Email: george@practicalidentity.com

Pieter Kasselman
Defakto Security
Email: pieter@defakto.security