Workgroup: OAuth
Internet-Draft: draft-jenkins-oauth-public-00
Published: 17 May 2024
Intended Status: Informational
Expires: 18 November 2024
Authors: N.M. Jenkins, Ed.
Fastmail

OAuth Profile for Open Public Clients

Abstract

This document specifies a profile of the OAuth authorization protocol to allow for interoperability between clients and servers using open protocols, such as JMAP, IMAP, SMTP, POP, CalDAV, and CardDAV.

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 18 November 2024.

Table of Contents

1. Introduction

This document pulls together several existing standards and specifies a specific profile using them to allow for interoperable modern authentication for clients of open protocols, such as IMAP, JMAP, SMTP, POP, CalDAV, and CardDAV. For these protocols, there are many servers and many clients with no pre-existing relationship, that need to be able to connect. At the moment, the only interoperable way to do so is with a basic username and password, which have many deficiences from a security standpoint.

1.1. Presumptions

This OAuth flow presumes you have an email address that is used to identify the user, along with:

1. The set of services that may be available for this email address (e.g., JMAP/IMAP/SMTP/POP/CardDAV/CalDAV);
2. The Application Server endpoint to connect to in order to access them; (e.g. a JMAP session endpoint https://api.example.com/jmap/session, or an IMAP endpoint imaps://imap.example.com:993).
3. The authorization server issuer identifier, needed to do OAuth, e.g. https://auth.example.com.

Ideally, the client may use an autodiscovery mechanism to find these given the email address. Such a mechanism is out of scope of this document.

1.2. Scopes

To work interoperably, clients and server must use a standard set of scopes for access. A separate document will bring together an autodiscovery mechanism (to get the details described above), this document (for authorization), and a set of standard scopes.

1.3. 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.

2. The Open Public Client OAuth Profile

2.1. Overview

OAuth 2 can be used in many different ways. This document specifies one particular set of options to ensure interoperability and security. Servers may implement more options, but MUST support the flow as described in this document for interoperability with clients. Similarly, clients may choose to support additional flows but there is no guarantee that this will be interoperable.

The general flow works like this:

1. The OAuth 2.0 Authorization Server Metadata ([RFC8414]) is fetched.
2. The client registers with the authorization server to get a client id using the OAuth 2.0 Dynamic Client Registration Protocol ([RFC7591]).
3. The client authorizes using the Authorization Code Grant flow ([RFC6749], Section 4.1) with PKCE ([RFC7636]), Issuer Identification ([RFC9207]) and Resource Indicators ([RFC8707]).
4. The client gets an access token and refresh token (as per [RFC6749], Section 5).

The access token can now be used as a Bearer token to authenticate requests to the application servers (as per [RFC6750]). When it expires, a new one can be requested using the refresh token (as per [RFC6749], Section 6).

The rest of this document describes each for the above steps in detail.

2.2. Fetching the Authorization Server Data

The authorization server issuer identifier MUST be an origin (i.e. an absolute URL with no path, query parameters or fragment) and MUST use HTTPS. Otherwise, abort the flow. The authorization server metadata may be fetched by a GET request to the path of /.well-known/oauth-authorization-server at this origin.

When fetched, a successful response is indicated by a 200 OK HTTP status code. The reponse MUST have an "application/json" content type. Anything else MUST be treated as an error, and the flow MUST be aborted.

The authorization server metadata is a JSON document with properties as specified in [RFC8414]. It MUST include the following properties:

*issuer

The authorization server's issuer identifier. This MUST be identical to the issuer identifier used to fetch this document (i.e., the origin at which this resource is hosted). For example, if this document was fetched from https://example.com/.well-known/oauth-authorization-server, the issuer MUST be "https://example.com". If not, the flow MUST be aborted.

This MUST be verified again during the authorization flow (see Section XXX), to prevent against mix-up attacks (as defined in [RFC9207]).

*registration_endpoint

The URL the client will use to register, to get a client id it needs for authentication (see Section XXX).

*authorization_endpoint

The URL the client will use to start the authentication process (see Section XXX) once it has registered.

*token_endpoint

The URL the client will use to get an access token after successful authentication (see Section XXX).

*scopes_supported

An array of supported scopes on the server (see Section XXX).

*response_types_supported

A list of response types supported in the OAuth authorization flow. This is an array of strings that MUST include "code". It may include other types, but they are not relevant for this document.

*grant_types_supported

A list of the OAuth 2.0 grant type values that this authorization server supports. This is an array of strings that MUST include "authorization_code" and "refresh_token". It may include other types, but they are not relevant for this document.

*token_endpoint_auth_methods_supported

This is an array of strings that MUST include "none".

*code_challenge_methods_supported

An array of strings listing Proof Key for Code Exchange (PKCE) [RFC7636] code challenge methods supported by this authorization server. This MUST include "S256".

*authorization_response_iss_parameter_supported

This MUST have the boolean value true.

The document MAY include other properties. It is RECOMMENDED servers support DPoP ([RFC9449]) to allow sender-constrained refresh and access tokens for HTTP-based protocols. Servers that support DPoP MUST include the following property:

*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, as defined in [RFC9449], Section 5.1.

Also of possible interest to client/server implementors following this document:

*revocation_endpoint

The URL the client can use to revoke their tokens, as per [RFC7009].

*revocation_endpoint_auth_methods_supported

If a revocation_endpoint is included, this property MUST be included, and is an array of strings that MUST include "none".

Clients MUST verify the required properties are present and conform to the requirements of this document. If not, the server is not using OAuth in conformance with this document and no compatibility may be presumed. It is RECOMMENDED clients abort the flow in such a case.

2.3. Dynamic Client Registration

To register, the client or developer sends an HTTP POST to the client registration endpoint (as found in the metadata) with a content type of "application/json", and a body consisting of a JSON document with the following properties:

*redirect_uris

An array of URIs the client may use to receive back information at the end of the authorization flow. Each URI MUST satisfy all of these conditions:

-The URI MUST start with one of the following:

ohttp://127.0.0.1/
ohttp://::1/
oA private-use scheme in reverse domain notation, e.g., com.example:/. Such a scheme MUST have at least one dot in it.

-The URI MUST NOT include two consecutive dots (e.g., /../).
-The URI MUST NOT include a fragment part (#).

The URI may include a path and query parameters. Clients MUST generate a unqiue redirect URI for each authorization server they register with to ensure they can protect against mix up attacks (see later).

*token_endpoint_auth_method

This MUST be "none".

*grant_types

This is an array of strings that MUST include "authorization_code" and "refresh_token".

*response_types

This is an array of strings that MUST include "code".

*scope

A string containing a space-separated list of scope values the client may request access for. (Note! This is not a JSON array.)

*client_name

Human-readable string name of the client to be presented to the end-user during authorization.

*client_uri

A URL string of a web page providing information about the client. This MUST use HTTPS.

*logo_uri

A URL for a logo to display for this client. This SHOULD be square, and in a PNG or SVG image format. This MUST use HTTPS.

*tos_uri

A URL that points to a human-readable terms of service document for the client. This MUST use HTTPS.

*policy_uri

A URL that points to a human-readable privacy policy document for the client. This MUST use HTTPS.

*software_id

A unique identifier string (e.g., a Universally Unique Identifier (UUID)) assigned by the client developer or software publisher, used by registration endpoints to identify the client software to be dynamically registered. Unlike "client_id", which is issued by the authorization server and SHOULD vary between instances, the "software_id" SHOULD remain the same for all instances of the client software. The "software_id" SHOULD remain the same across multiple updates or versions of the same piece of software. The value of this field is not intended to be human readable and is usually opaque to the client and authorization server.

*software_version

A version identifier string for the client software identified by "software_id". The value of the "software_version" SHOULD change on any update to the client software identified by the same "software_id". The value of this field is intended to be compared using string equality matching and no other comparison semantics are defined by this specification.

If the server indicated in its metadata that it supports DPoP [RFC9449] and the client is intending to authenticate all requests using DPoP, the client SHOULD also include the following property:

*dpop_bound_access_tokens

true

(If set, the server MUST then require all token requests from this client use DPoP. Note, DPoP is only currently defined for HTTP protocols, so this precludes usage for non-HTTP procols as of time of writing.)

The server will check that all required properties are present and have valid values. Any unknown properties supplied by the client MUST just be ignored. The authorization server MAY replace any of the client's requested metadata values submitted during the registration and substitute them with suitable values.

If there is an exact match for all properties except for software_version, an existing registration may be returned. Otherwise, servers SHOULD create a new registration and client id.

There is no way to verify the authenticity of the information supplied by the client, however the general case of accurate information is still useful to the server, for example to be able to contact client authors to help debug issues if aberrant behaviour is observed. Servers MAY choose to ignore all of the information instead and just return a static client id to all requests.

The redirect URI restrictions MUST be enforced. These ensure the OAuth flow can only be completed by native clients — (U+2014) not web clients. Since a malicious native client could present the user with a custom browser to phish credentials anyway, the lack of verification of client registration details does not provide additional danger beyond existing threats.

If successful, the server responds with an HTTP 201 Created status code and a body of type "application/json", with the content being a JSON object containing all the properties submitted during registration (with their values as set by the server, if overwritten), plus the following property:

*client_id

The OAuth 2.0 client identifier string, used in the authorization flow (see Section XXX).

If the registration fails, the server will respond with an HTTP 400 status code and a JSON body as described in [RFC7591], section 3.2.2.

2.4. Authorization

Clients initiate authorization by opening the authorization_endpoint URL in a web browser, with the following additional query parameters:

*client_id

The client id as returned in the registration.

*redirect_uri

One of the redirect uris registered by the client. This MUST be identical to the registered URI. The one exception to this is if a URI with the prefix "http://127.0.0.1/" or "http://::1/" was registered, in which case the matching URI MUST also include an arbitrary port.

For example, if "http://127.0.0.1/redirect" was registered, then the client could send "http://127.0.0.1:49152/redirect" as the redirect_uri for authorization.

*response_type

This MUST be "code".

*scope

A space delimited set of scopes the client would like access to. This MUST be a subset of the scopes registered for the client.

*code_challenge

A PKCE code challenge as per [RFC7636], using SHA 256. To generate a challenge, first generate a code_verifier: a high-entropy cryptographic random STRING using the unreserved characters [A-Z] / [a-z] / [0-9] / "-" / "." / "_" / "~" from Section 2.3 of [RFC3986], with a minimum length of 43 characters and a maximum length of 128 characters.

The code_challenge is then BASE64URL-ENCODE(SHA256(ASCII(code_verifier)))

*code_challenge_method

This MUST be "S256".

*resource

The URL for the initial resource endpoint (as returned in the autodiscover) you wish to access after successful authorization.

For example, "https://api.example.com/jmap/session". If you wish to use multiple protocols, the client MUST include multiple "resource" query parameters: one for each endpoint. The server MUST verify that all requested endpoints are permitted locations to send access tokens to (i.e., they are the real endpoints resource associated with this authentication server), in order to prevent mix up attacks.

*state

An opaque value used by the client to verify that an authorization response is due to a request that the client initiated. The authorization server will include this value when redirecting the user-agent back to the client. Clients MUST generate a state with a unique, unguessable random string when initiating an authorization request.

*login_hint (optional)

The email address the user originally asked to log in with. The server can prefill this in a login form. Note, the user may choose to log in with a different address.

The authorization server MUST verify all required parameters, and that they conform to the restrictions in this document. Other URL parameters MAY be supplied but will be ignored.

If verified, the authorization server will authenticate the user and ask them if they wish to grant authorization to the client. If successful, the client will receive a response via the redirect_uri, which will include the following query parameters:

*code

The authorization code. This may be exchanged for the refresh token. This code MUST have a limited expiry, which MUST be at least 10 minutes from authorization. It MUST NOT be used again once the client has successfully exchanged it for a refresh token. Doing so may cause the server to detect it as stolen and revoke all associated tokens.

*state

The value of the state parameter that was passed in with the initial request.

*iss

The issuer identifier of the authorization server.

The client MUST verify all of the following:

- The state returned matches exactly the state it sent, to verify that this request was indeed initiated by the client and not an attacker.
- The redirect URI the authorization response came in on was the one they used when generating the request. This prevents mix-up attacks when the client supports more than one authorization server.
- The "iss" returned is identical to the "issuer" property in the metadata object. This is another prevention against mix-up attacks.

If any verification fails, the client MUST abort the flow and not send the authorization code anywhere.

2.5. Obtaining a Refresh Token

Following authorization, the client will obtain initial refresh tokens and access tokens by making a POST request to the token_endpoint URL. The following parameters MUST be present, using the "application/x-www-form-urlencoded" format with a character encoding of UTF-8 in the HTTP request entity-body:

- client_id

  The client id as given to the developer.

- redirect_uri

  The redirect_uri parameter sent with the authorization request from which the code was obtained.

- grant_type

  This MUST be "authorization_code".

- code

  The code returned via the redirect back from authorization.

- code_verifier

  The code_verifier generated for the authorization (the random string generated in the authorization step, as per [RFC7636]).

Other parameters MAY be supplied but will be ignored. If using DPoP, the client must also set a DPoP header in accordance with [RFC9449], Section 5.

The server will verify the parameters and if successful, return a 200 OK response with a content type of application/json. The body will be a JSON object with the following properties:

*access_token

A bearer token used to authenticate API requests. This will be valid for a fixed, limited time.

*token_type

The type of the access token. This will always be "bearer".

*expires_in

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.

*scope

The space delimited set of scopes that this access token may use.

*refresh_token

The refresh token to use next time the client needs to get a new access token.

2.6. Using the access_token

The client is now authenticated. It can connect to the servers given in the discovered auto config with the Bearer scheme [RFC6750]. For HTTP based protocols, this means setting an Authorization header with the value Bearer {access_token}.

If using DPoP, the client must also set a DPoP header in accordance with [RFC9449], Section 7.

2.7. Getting a New Access Token

Client should keep using an access token they have been issued until it expires, which will result in getting a 401 error back.

When the access token expires, the client must get a new one by making another POST request to the authorization server token endpoint. The following parameters MUST be present, using the "application/x-www-form-urlencoded" format with a character encoding of UTF-8 in the HTTP request entity-body:

*client_id

The client id as given to the developer.

*grant_type

This MUST be "refresh_token".

*refresh_token

The refresh token returned last time the client obtained a new access token.

The success and failures responses are identical to those documented in "Obtaining a refresh token" (Section XXX).

A new refresh token will be returned in the response and the client MUST replace their previous refresh token with this. The client MUST NOT try to use an old refresh token again; this SHOULD result in the authorization being revoked as a protection against leaked refresh tokens. If the user has multiple devices, each client MUST obtain separate authorization. You cannot share a refresh token between devices.

3. Security Considerations

Yes, there are security considerations. They will be added in due course.

4. IANA Considerations

This document does not introduce any IANA considerations.

5. Normative References

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

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

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

[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>.

[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>.

[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>.

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

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

[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>.

[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>.

[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>.

Author's Address

Neil Jenkins (editor)
Fastmail
PO Box 234, Collins St West
Melbourne VIC 8007
Australia

Email: neilj@fastmailteam.com
URI: https://www.fastmail.com