OAuth2 and OpenID Connect: protocols and acceptance criteria

Lars Wirzenius

2022-05-14 07:23

Contents

Chapter 1

Overview

This a description of two authentication and authorization protocols, and a sketch of acceptance criteria for an implementation of them.

This is very much work in progress.

1.1 Concepts

Some basic concepts in this document:

- **identity** – data about who you are to tell you apart from everyone else
- **authentication** – proving your identity
- **authorization** – giving you permission to do something

FIXME: These could do with citations.

1.2 The protocols: OAuth and OpenID Connect

The OAuth[1] 2.0 protocol is for authorization, not authentication, and assumes an already existing way to authenticate users. It's mainly for giving a service or application permission to do something on your behalf.

The OpenID Connect[2] 1.0 (OIDC) protocol is for authenticating yourself to one service or application by using a third party service. This allows one authentication service (or identity provider) be used for any number of other services or applications. Further, since the identity provider can keep a login session open independently of the other services and applications, this provides a single sign-on experience.

[1] `https://tools.ietf.org/html/rfc6749`
[2] `https://openid.net/specs/openid-connect-core-1_0.html`

We discuss here only these specific versions of these protocols, and even then only subsets chosen based mainly from the point of security.

1.3 Entities involved in the protocols

The protocols involves the following entities:

- the **end user**, who is trying to do something; also known as the resource owner
- the **web browser**, used by the user; might be a mobile or command line application instead of a browser as such; also know as the user agent
- the **application**, which the user uses to do things, and as part of that access resources; also know as the requesting party and the facade
- the **resource provider**, where the resources are, and which allows access to them via a web API
- the **identity provider** (IDP), which authenticates the user

The protocol specifications use different terminology, to be more generic. The above have been chosen to make this document easier to understand.

Chapter 2

The OAuth 2.0 protocol

The OAuth2 protocol is a way for an end user to allow one service controlled access to their data on another service. It does not authenticate the end user in any way.

As an example, imagine if Alice uses an email service, but would also like to make use of another service that typesets and prints emails into really nice, impressive, beautiful, heirloom quality, leather bound hardcover books. The book service needs to be able to read Alice's emails, but not delete them or send email as Alice.

In a kinder world than ours Alice could just give their email password to the book service, but in our world, this needs to be done in a more complicated way. Otherwise someone at the book service would abuse access to Alice's email account by deleting all the email, or worse.

The gist of OAuth2 is that Alice can tell their email service that the book service is allowed to read all her correspondence, but do nothing else.

In a simplistic way, Alice logs into the email service, and asks for an *access token*, then passes that onto the book service. The book service logs into the email service, and gives the access token to gain access to Alice's emails. The email service knows that the access token only allows read-only access: no deletion or sending.

However, such a system would be cumbersome to use. Alice would have to manually navigate to the email service's access token generation page, copy the token, and have a way to communicate the token to the book service. This is too much manual work, with too many steps, and too much can go wrong.

Instead, Alice logs into the book service, and tells them which email service to get emails from. The book service redirects Alice's to the email service in a way that tells the email service that a) an access is required b) by the book service

c) for read-only access d) to Alice's emails. The mail service checks that Alice is already logged in, or else asks Alice to log in, and that Alice is OK with giving the book service the access requested. If all that goes well, the email service generates the token, and redirects Alice's web browser back to the book service in such a way that the token is carried along. The book service can now access the email service API with the access token, and get what they need to print the books.

RFC 6749[1] describes the OAuth2 protocol in detail. This chapter condenses that into a shorter, and opinionated description.

2.1 OAuth2 protocol variants (grant types)

OAuth2 allows for several different kinds of use cases, by providing different ways to get an "authorization grant". An authorization grant is a thing that provides access to a protected resource (see section 1.3 in RFC 6749[2].

There are four authorization grant types:

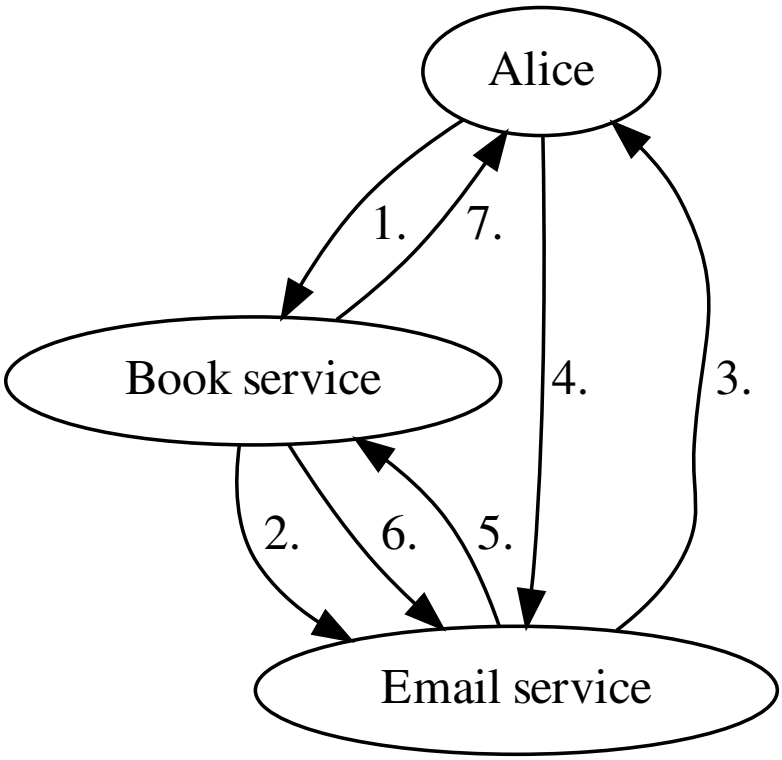
- authorization code
  - this is a slightly simplified description: the spec seems to allow less secure variants, but we'll get to that later
  - application redirects end user's web browser to the identity provider, which returns an authorization code to the application
  - application gives authorization code to resource provider, which gives back an access token
  - browser never gets end the token
  - the authorization code is random, temporary, short-lived, can only be used once, and can be tied to a specific resource provider
  - this allows the application to be authenticated as well as the end user
  - this is the most secure grant type and should be used if possible
- implicit
  - application is typically a JavaScript application running in the browser
  - access token is given directly to the browser
  - this is vulnerable to browser insecurities, which are legion
  - application can't be authenticated in any useful sense
  - don't use this, due to low security
- resource owner password credentials
  - client has the end user's username and password
  - never use this, as it leaks credentials and is inherently insecure
- client credentials
  - client has its own credentials
  - can be suitable when the client itself is the resource owner (such as for server-to-server communication) or the end user authorization has

[1] `#rfc6749`
[2] `#rfc6749`

been arranged beforehand using some other way

In this document we only discuss the authorization code grant, as it's the only one of these we can recommend from a security point of view.

2.2 Overview of a basic OAuth2 transaction



The above diagram represents the steps of Alice getting her book of emails, at a very high level. There are many steps.

1. Alice asks the book service for a book of her emails.
2. Book service asks email service for an access token.
3. Email service asks Alice if book service may access her emails.
4. Alice tells email service "sure".
5. Email service gives book service an access token.
6. Book service uses access token to download all of Alice's emails.
7. Book service sends book to Alice.

The diagram below represents the same transaction, but in a different way. (One of the diagrams will be clearer to the reader, but it depends on the reader which.)

ERROR: Failed to invoke java

2.3 Token types

OAuth has two types of tokens: access and refresh. Access tokens are used by the application to access resource providers. Refresh tokens are used by the application to request a new access token from the identity provider.

OAuth does not specify the format of the tokens, and leaves it to the identity provider. The resource provider needs a way to verify that a token is valid. The validation mechanism is also not specified by OAuth. However, the [JSON Web Token][] (JWT) specification is commonly used. This will be covered in detail in the OIDC part of this document.

2.3.1 Access tokens

An access token is sufficient to access data on a resource provider, on its own. A valid access token itself represents all the authorization to access a protected resource via a resource provider. Every access token granted on behalf of a resource owner does not necessarily grant full and complete access to every resource owned by end user, but may be limited in some way: a particular token may give access only to a particular resource, or only certain kinds of access (read vs write vs delete). Access tokens may also have a limited life time, after which the resource provider refuses to accept them.

Access tokens are "bearer tokens", which means that any person or software with a copy of the token can use it, within the constraints encoded in the token itself. As such, access tokens are to be considered sensitive data, just like all other credentials.

2.3.2 Refresh tokens

A refresh token is used by the application to get a new access token, without requiring the end user to grant one. When the end user grants a token it is an interactive process, and may involve re-authenticating them. This means it can be quite tedious and irritating, if it happens often.

An application may need a new access token for various reasons. For example:

- If access tokens expire, the application will use its refresh token to get a new access token.
- An access token may be revoked, but a refresh token may allow the application to get a new one. The old access token revocation may be due to the resource provider noticing suspicious activity. For example, if the access token might be tied to a specific IP address, but when a mobile application moves to a new network, its address changes, invalidating the old access token.

  Revoking access tokens can be tricky, as the revocation needs to be communicated to all the resource providers. For this reason, access tokens

often have a short life time. Refresh tokens are easier to revoke, as only the identity provider needs to know about the revocation. Thus a combination of short-lived access tokens and longer-lived refresh tokens allows fairly rapid, but not instant, revocations in a simple manner, without making the end user re-authenticate themselves many times an hour.

- The application may need to give an access token to another application, but give them less access. In our book printing example, the usual email application Alice uses to process all her email might have a button to order a book, and the application could automatically get a read-only access token to give to the book printing service. The application could use a refresh token to get the extra read-only access token without Alice having to interact again with the identity provider directly.

When the application uses a refresh token, the identity provider validates it and, if everything is OK, responds with a new access token, and possibly a new refresh token.

The sequence diagram shows the protocol flow for when Alice's web mail application orders a book for her.

ERROR: Failed to invoke java

2.4 Finding protocol endpoints

The OAuth2 protocol requires the client to make certain requests to the authorization server to get access and refresh tokens. We can assume that the client knows the address of the authorization server, but it still needs to find the actual complete URLs to the endpoints.

In the original OAuth 2.0 specification, this was left unspecified. Each client needed to have inherent knowledge of the endpoints for each authorization server. RFC 8414[3] adds a way for the client to find the endpoints automatically, once it knows the authorization server location.

The process is complex enough that we won't go into all the details here. In the simple, straightforward case given an authorization server at `https://server.example.com`, the client retrieves the JSON document at the following URL:

```
https://server.example.com/.well-known/oauth-authorization-server
```

The JSON document might look like the following:

```
{
  "issuer": "https://server.example.com",
  "authorization_endpoint": "https://server.example.com/authorize",
  "token_endpoint": "https://server.example.com/token",
```

[3] `#rfc8414`

```
  "token_endpoint_auth_methods_supported": [
    "client_secret_basic",
    "private_key_jwt"
  ],
  "token_endpoint_auth_signing_alg_values_supported": [
    "RS256",
    "ES256"
  ],
  "userinfo_endpoint": "https://server.example.com/userinfo",
  "jwks_uri": "https://server.example.com/jwks.json",
  "registration_endpoint": "https://server.example.com/register",
  "scopes_supported": [
    "openid",
    "profile",
    "email",
    "address",
    "phone",
    "offline_access"
  ],
  "response_types_supported": [
    "code",
    "code token"
  ],
  "service_documentation": "http://server.example.com/service_documentation.html",
  "ui_locales_supported": [
    "en-US",
    "en-GB",
    "en-CA",
    "fr-FR",
    "fr-CA"
  ]
}
```

Some of the complexity we elide here involves compatibility with the OpenID Connect protocol, and extending the OIDC approach. There is also provisions for dealing with multiple authorization servers on the same URL, or server that aren't rooted at the base of the domain. There is also a possibility of using digitally signed metadata, as a signed JSON Web Token. For all of this, detailed reading of the RFC specification is needed to get all correct. Here we aim at giving an overview.

2.5 HTTP transactions

This chapter shows examples of the actual HTTP transactions to implement the OAuth protocol. The examples use HTTP/1.1, for simplicity, but translating

them to newer versions of HTTP should be straightforward.

The examples assume that the server is `auth.example.com` and that the token endpoint is `/token` on that server.

2.5.1 Client credentials grant

The client credentials grant is quite simple: the client sends a request, the server checks that it's a valid request, and responds either with an error or an access token.

The client makes a POST request to the *token* endpoint, with an HTML form-encoded body that specifies that it wants to use the client credentials grant. Specifically, it sets `grant_type` to `client-credentials`. No other form fields are needed.

The client provides its credentials as using HTTP Basic Auth (see RFC 7616[4]), in the `Authorization` header.

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

grant_type=client_credentials
```

The response uses HTTP status codes to indicate success or failure. A 200 status code means success. A 400 series status code means failure.

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

{
  "access_token":"2YotnFZFEjr1zCsicMWpAA",
  "token_type":"bearer",
  "expires_in":3600,
}
```

In the sample response above, the token is a bearer token, which means the client should use it in requests with the `Authorization` header:

```
Authorization: Bearer 2YotnFZFEjr1zCsicMWpAA
```

In the sample response above, the token is valid for an hour. After that it expires and the resource server should refuse it. The token may become invalid earlier.

[4] `#rfc7616`

Note that for client credentials grants no refresh token is returned, it would be fairly pointless. The client can just get a new access token the same way it got the first one.

2.5.2 Error responses

A token request can fail for a variety of reasons. The 400 status code is used for anything the client did wrong. Other HTTP status codes are used for other errors, as specified by HTTP. For example, if the server has an internal problem, 500 is returned.

For OAuth, a 400 response returns a more detailed indication about what the client did wrong, in the `error` field in the JSON body of the response. RFC 6749[5] lists the error codes[6] and other fields. For example, if the client has the wrong credentials, the response would look something like this:

```
1 HTTP/1.1 400 Bad Request
2 Content-Type: application/json;charset=UTF-8
3 Cache-Control: no-store
4 Pragma: no-cache
5
6 {
7    "error":"invalid_client"
8 }
```

The client can use the detailed error code to inform what it should do about the error. Few, if any, errors warrant the client re-trying the request, especially soon, but it might alert a human, or just log the failure.

[5] `#rfc6749`
[6] `https://tools.ietf.org/html/rfc6749#section-5.2`

Chapter 3

The OIDC 1.0 protocol: authorization code

FIXME: write this

Chapter 4

Acknowledgement

Thank you to Ivan Dolgov and Pyry Heiskanen for reviewing merge requests and general support when writing this document.

Chapter 5

References

5.1 RFC 7519 – JSON Web Token (JWT)

- https://tools.ietf.org/html/rfc7519
- M. Jones, J. Bradley, N. Sakimura
- 2015

Abstract:

JSON Web Token (JWT) is a compact, URL-safe means of representing claims to be transferred between two parties. The claims in a JWT are encoded as a JSON object that is used as the payload of a JSON Web Signature (JWS) structure or as the plaintext of a JSON Web Encryption (JWE) structure, enabling the claims to be digitally signed or integrity protected with a Message Authentication Code (MAC) and/or encrypted.

5.2 RFC 7616 – The 'Basic' HTTP Authentication Scheme

- https://tools.ietf.org/html/rfc7617[1]
- J. Reschke
- 2015

Abstract:

This document defines the "Basic" Hypertext Transfer Protocol (HTTP) authentication scheme, which transmits credentials as user-id/ password pairs, encoded using Base64.

[1] `https://tools.ietf.org/html/rfc7617`

5.3 RFC 6759 – The OAuth 2.0 Authorization Framework

- https://tools.ietf.org/html/rfc6749[2]
- D. Hardt, Ed.
- 2012

This is the core OAuth specification upon which all other ones build. Abstract:

> The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf. This specification replaces and obsoletes the OAuth 1.0 protocol described in RFC 5849.

5.4 RFC 8252 – OAuth 2.0 for Native Apps

- https://tools.ietf.org/html/rfc8252[3]
- W. Denniss, J. Bradley
- 2017

Abstract:

> OAuth 2.0 authorization requests from native apps should only be made through external user-agents, primarily the user's browser. This specification details the security and usability reasons why this is the case and how native apps and authorization servers can implement this best practice.

5.5 RFC 8414 – OAuth 2.0 Authorization Server Metadata

- https://tools.ietf.org/html/rfc8414[4]
- M. Jones, N. Sakimura, J. Bradley
- 2018

Abstract:

> This specification defines a metadata format that an OAuth 2.0 client can use to obtain the information needed to interact with an OAuth 2.0 authorization server, including its endpoint locations and authorization server capabilities.

[2] `https://tools.ietf.org/html/rfc6749`
[3] `https://tools.ietf.org/html/rfc8252`
[4] `https://tools.ietf.org/html/rfc8414`

Chapter 6

Links to OAuth2 or OIDC implementations

- Ory[1]

[1] https://www.ory.sh/