UKRAINIAN CATHOLIC UNIVERSITY

BACHELOR THESIS

Authorization server with OAuth support and recommended usage patterns

Author:
Bohdan KOVALCHUK

Supervisor:
Yurii YUNIKOV

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science

in the

Department of Computer Sciences
Faculty of Applied Sciences

Lviv 2020

Declaration of Authorship

I, Bohdan KOVALCHUK, declare that this thesis titled, "Authorization server with OAuth support and recommended usage patterns" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

UKRAINIAN CATHOLIC UNIVERSITY

Faculty of Applied Sciences

Bachelor of Science

Authorization server with OAuth support and recommended usage patterns

by Bohdan KOVALCHUK

Abstract

This thesis covers the history of digital authentication and authorization and considers the main changes in different versions of the OAuth protocol. It also describes the implementation details of custom OAuth authorization servers and clients that demonstrate usage examples of different authorization grant types.

Code can be found here: Github repository

Acknowledgements

I am grateful to my family, who supported me during my student life. I would like to thank Yurii Yunikov, who suggested the theme of this thesis and helped me understand the difficult moments of the authorization. I want to thank Yaroslav Kovalchuk, who helped me organize the work process and Andrii Snitsaruk for the text editing initiative. Special thanks to my girl Kate whose care helped me entirely focus on writing this paper.

Contents

List of Figures

List of Abbreviations

API	**A**pplication **P**rogramming **I**nterface
URL	**U**niform **R**esource **L**ocator
HTTP	**H**yper**t**ext **T**ransfer **P**rotocol
TLS	**T**ransport **L**ayer **S**ecurity
IETF	**I**nternet **E**ngineering **T**ask **F**orce
JWT	**J**SON **W**eb **T**oken
PKCE	**P**roof **K**ey for **C**ode **E**xchange
SPA	**S**ingle **P**age **A**pplication
CSRF	**C**ross-**S**ite **R**equest **F**orgery
JWK	**J**SON **W**eb **K**ey

Dedicated to people who make the digital world more secure

Chapter 1

Introduction

1.1 Context

We live in a world where the amount of our digital data is increasing every day.

The importance of data security increases, as well. Stealing or damaging the data can lead to harmful consequences such as financial or moral damage or even a threat of life.

It is especially important in the world that becomes more global and connected, and hacking techniques become more cunning.

Developers face the problem of securely connecting different components while leaving interaction with the end-user as simple as possible.

The problem can be effectively solved using authentication and authorization. In this paper, these two terms will be referred to as *"auth"*.

1.2 Problem

At least a basic understanding of auth standards and protocols is required to use them in computer software. Implementing auth can be challenging, especially if there are exotic constraints or extraordinary conditions. The complexity of proper auth implementation leads to a higher probability of getting a solution with vulnerabilities.

Having more learning resources with clear and complete examples increases the popularity of auth. It makes the entry threshold lower, which should lead to the spread of modern auth and better protection of individuals data.

1.3 Goals

- Analyze the history of auth and select recommendations for use
- Create auth server on Node.js to use as a core for demo cases
- Create demo cases for different OAuth grant types setups

Chapter 2

Background information

2.1 The history of auth

The first modern usage of authentication was in 1961 at the Massachusetts Institute of Technology, for use with the Compatible Time-Sharing System (CTSS). This system made it possible to share the resources of one computer among many users. Each user had its own set of files with a password on as a lock; in other words, it was the first known system to implement password login. (*The World's First Computer Password? It Was Useless Too*)

It has to be said that this approach was hacked in 1962 after one year of existence. It was possible because all passwords were stored purely in a file that could be printed without special permission. (Walden and Vleck, 2001) This showcase demonstrates that auth and hacking go side by side from their beginning.

The next step was in the 1970s when Bell Labs researcher Robert Morris found a way to store passwords securely using a cryptographic idea of a hash function. (Morris and Laboratories, 1979) Each user password was used as an input for a particular hash function. After encryption, the output was saved to the password file. Then on every user login, the prompted password was encrypted and compared to the stored one. If both outputs were the same, then the login attempt was accepted.

In the 1970s, asymmetric cryptography and public/private keys were discovered during government research from the Government Communications Headquarters (GCHQ), but research was classified until the 90s. Fortunately, there were other public researchers, and in 1977 Ron Rivest, Adi Shamir, and Leonard Adleman published their RSA asymmetric key algorithm. (*RSA (cryptosystem)*) The idea behind such algorithms is a pair of public and private keys. The public key might be shared with anyone, while the private key is protected and used only by the owner. Such algorithms give two powerful abilities at once:

- *Public key encryption* when anyone who has a public key encrypts data and sends a result of encryption to the owner. The only owner of the private key, might decrypt the message and read data.
- *Digital signature* when a person shares the public key with others to give them the ability to verify the identity of this person; the private key is used by the person to sign data; then signed data might be verified by a person's public key to prove person's ownership of signed data.

In the 80s, companies came to the conception of persistent password security problems because if attackers guess or stole a password, they could make a replay attack and compromise the user. It will be even worse if other services use a stolen password. That is why a one-time password (OTP) technologies were actively developed at that time. (*One-time password)*) OTP allows using a password that is valid

only for a single session and usually for a particular period. Early OTP often requires a user to have a specific device that generates a password based on different techniques as time, previous password, etc. But later new standards were created to pass OTP through other channels that also influences more recent auth protocols (*Digital authentication: The past, present and uncertain future of the keys to online identity*)

In the 90s, public-key infrastructure (PKI) was created to solve the problem introduced in the 70s with the start of using public/private keys: how to prove that public key retrieved from the owner of a private key definitely belongs to the owner and is not replaced by an attacker. PKI is a set of technologies for digital certificates management, where the digital certificates are used to prove public key ownership. PKI played an important role in the evolution of auth protocols because it allows using TLS, which reduces complexity related to secure communication establishment.

In the next years, many good ideas were invented to make auth more secure. Still, it is essential to say that all that time, the primary purpose behind auth was the same user should give something, usually credentials (password), from some particular system to get access to resources provided by that system. The difference was only in details of how this exchange was implemented.

At the beginning of the second millennium, the world was becoming more global. The Internet spread together with new social networks, which were gaining popularity. More often, there was a need for integration of one service with another. And one of the most popular approaches of that time was asking user credentials from one of the services by the second service to use it together with API requests to the first service later. This approach is also known as the *"password anti-pattern"*, and it has many disadvantages:

1. Compromising of one service will lead to a compromise of the user's account across all systems (Richer and Sanso, 2017a). If an attacker breaks one service and gets user credentials used in many services, there is no way to protect other services and their users from unauthorized access.

2. Service is impersonating user, and API service has no possibility of making a difference between direct user and other services' calls (Richer and Sanso, 2017a)

3. A user cannot revoke access for a particular service except change password that was previously sent to the compromised service.

4. A user cannot restrict access to some part of API service, that is why another service which got user credentials has unlimited access to any part of API service.

To avoid this many companies used their authentication methods to work around this problem (among them are Flickr Auth, Google AuthSub, Yahoo! BBAuth)(*OAuth 2.0 Simplified: Background*)

In 2005 Brad Fitzpatrick developed OpenID open standard and decentralized authentication protocol to give one unified approach of authentication. (Fitzpatrick, 2005)

It allows a user to be authenticated to different sites using third-party service. The only thing required from the user for this is to provide an OpenID identifier URL registered in any OpenID identity provider (even custom provider) and authenticate on the identity provider's login page.

From the technical point of view (see figure 2.1), a consumer (a web service that wants proof that the user owns the identifier URL) requests a user profile data via

URL provided by the user. After that, the consumer validates the parameters in the response. Validation is handled either with the help of the shared secret (retrieved during initial association using Diffie-Hellman algorithms) or makes additional check_authentication requests to validate the signature provided together with profile information. (*OpenID Authentication 1.1*)

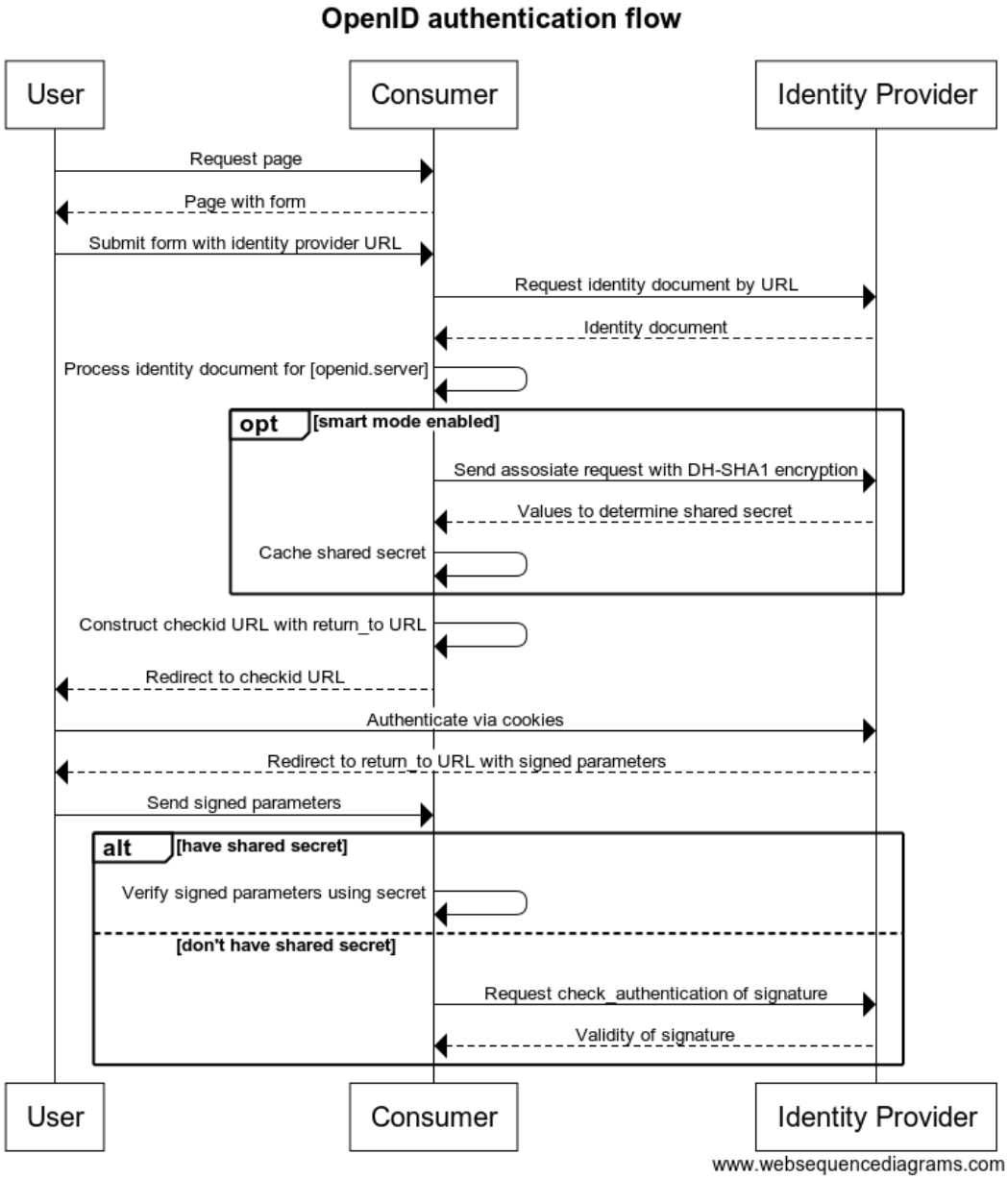

FIGURE 2.1: OpenID authentication flow

In 2006 several companies, including Twitter and Ma.Gnolia, decided to give users the ability to connect companies' applications together. Despite the fact of using OpenID in their systems, they cannot use it for the new tasks, because OpenID had nothing about delegating access (no password-related data was present in response from identity provider).

Developers created a new open standard access delegating protocol that was named OAuth 1.0 to solve this problem.

2.2 OAuth 1.0

The OAuth 1.0 protocol was developed with a focus on traditional web applications; that is why a web browser and a callback URL is required for authorization. A huge part of the protocol specification is about digital signatures that are generated for each request. The client might verify identity and protect data from unauthorized modifications by creating a signature for request data using previously mentioned hashing functions or asymmetric algorithms. Signatures may be one of the most complicated parts of the specification, and also the most important for HTTP without TLS communication.

In comparison with OpenID that is decentralized and allows a user to be authenticated via OpenID identity provider without any prerequisites on the side of OpenID client, OAuth client firstly should be registered at the preselected OAuth identity provider for this.

In other words, if OpenID allows the end-user to choose any identity provider to authenticate, OAuth requires the developer to choose identity providers and register a client during app creation.

The first version of the protocol is based on the following components:

- **Protected resource** - An access-restricted resource that can be obtained from the server using an OAuth-authenticated request.
- **Resource owner** - An entity capable of accessing and controlling protected resources by using credentials to authenticate with the server.
- **Server** - An HTTP server capable of accepting OAuth-authenticated requests.
- **Client** - An HTTP client capable of making OAuth-authenticated requests.
- **Credentials** - a pair of a unique identifier and a matching shared secret. OAuth defines three classes of credentials: client, temporary, and token, used to identify and authenticate the client making the request, the authorization request, and the access grant, respectively.
- **Token** - A unique identifier issued by the server and used by the client to associate authenticated requests with the resource owner whose authorization is requested or has been obtained by the client.

The Authorization flow (see figure 2.2) consists of three consecutive requests: Temporary Credentials Acquisition, Resource Owner Authorization, and Token Exchange.

Temporary Credentials Acquisition is used to bind the client with a new attempt to get a server's access token credentials. Client exchanges *consumer_key* and *callback* URL for a temporary *token*. As a result of this request, the server saved callback for future usage, and the client got a token that might be considered as an identifier of attempt.

Resource Owner Authorization is used to bind the resource owner with the attempt of the client to get the server's access token credentials. The client initiates the request by redirecting the resource owner to the server. Resource owner authorized in the server, usually by providing email and password, and allowed access for the client by clicking the button on the trust screen (consense screen). After granting access, the server redirects the page to the callback URL saved in the previous request. As a result, the server changed the status of the attempt to be approved by the resource owner, and the client got a verifier code.

Token Exchange is used by the client to exchange verifier code obtained in the previous request for an access token. This is the last step in the authorization flow.

After that, to retrieve a protected resource client requests it via a particular API endpoint and provides the access token, received during authorization flow.

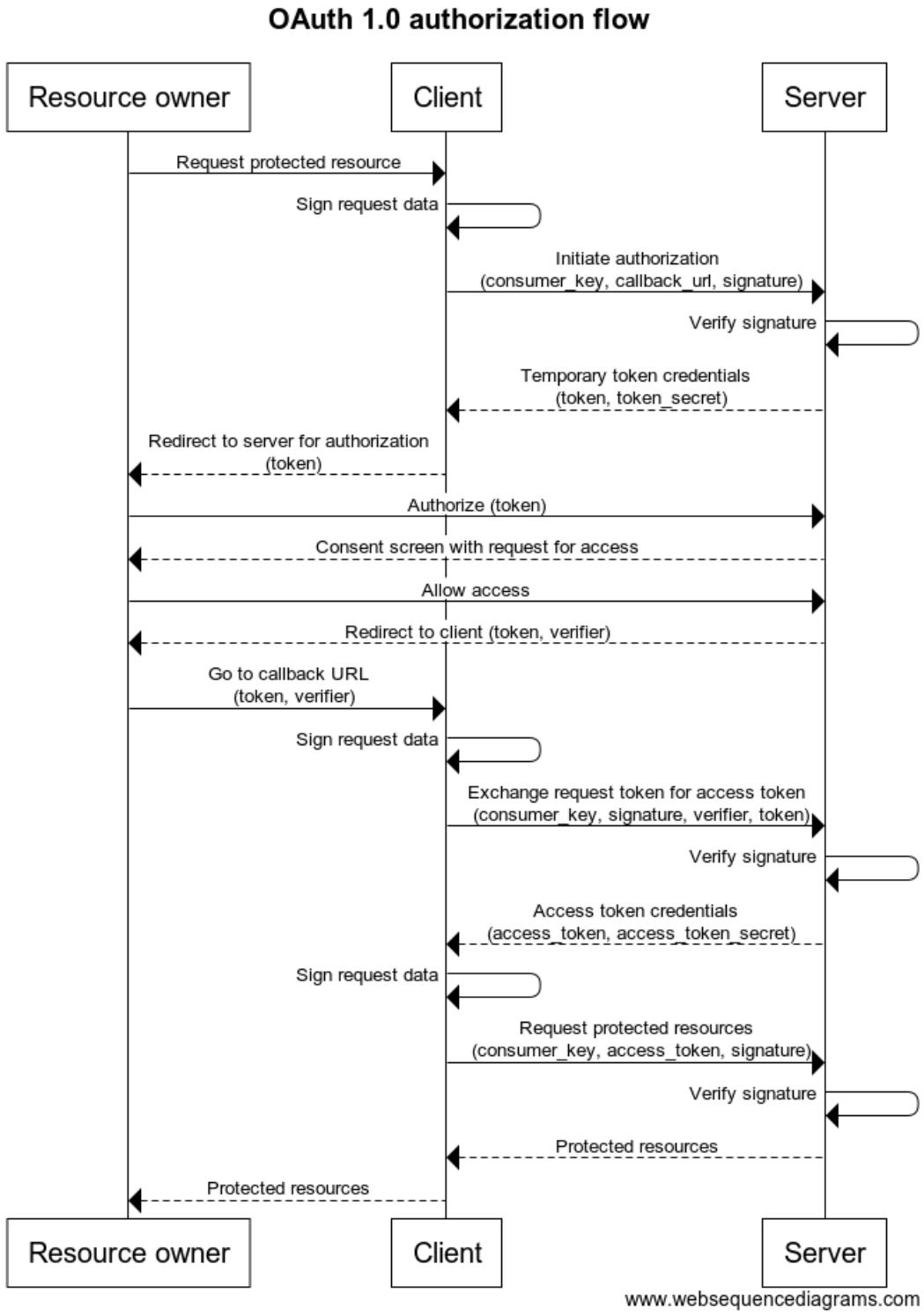

FIGURE 2.2: OAuth 1.0 authorization flow

An important aspect of authorization flow is the usage of additional parameters: *nonce*, *timestamp*, and *signature*.

The *nonce* parameter is used to prevent replay attacks: the server saves random value generated by the client; this value must be used in requests only once per session. Requests with a *nonce* value that was already used should be considered as a possible replay attack and must be rejected.

The server uses a *timestamp* parameter to avoid saving an infinite number of *nonce* values. After some time, specified by the server, requests will expire, and the server will reject them. That is why nonces related to timestamps also will expire and might be removed from the storage.

The signature parameter is calculated by using either HMAC-SHA1 or RSA-SHA1 hashing algorithms on HTTP request elements. These algorithms also use shared-secret obtained during the client registration or together with the token received during authorization flow. The protocol also allows not to use a signature for requests sent with HTTPS. For that purpose, the client should set a value of *signature_method* parameter to PLAINTEXT.

There is only one authorization flow in OAuth 1.0, however for clients without the ability to handle callback URL, there is its configuration named Out-of-Band flow. To use such a configuration client should send the *callback* parameter with value oob during the initial request. The server that got this value should show the verifier code on the screen after the resource owner authorization. The code should be manually provided to the client to finish the authorization flow. This configuration was the first attempt to solve non-standard device authorization problems and, in the future, was reinvented with Device Code Grant of OAuth 2.0. New flow reduced the number of required steps from the user by replacing manual code providing into client code pulling.

OAuth 1.0 has one huge advantage - it enables usage of protocol for non-HTTPS clients and is still used nowadays. (*Why OAuth 1.0a?*)

OAuth 1.0 was replaced by a new OAuth 2.0 because it has many disadvantages:

- it is monolithic and could not cover many specific cases (Richer and Sanso, 2017b)
- has no support for SPA
- has nothing about managing scopes of granted access.

The current official status of OAuth 1.0 is obsolete.

2.3 OAuth 2.0

The second version of the protocol is not backward compatible with the first version. Complicated signatures system was removed in favor of using TLS, the new flows were added, and tokens became more short-lived. In general, the protocol became more modular, which allowed using it the same way as OAuth 1.0 was in practice, but without twisting core aspects of the protocol. (Richer and Sanso, 2017b)

It has the same base set of components as in the first version: Server, Resource owner, Authorization server, Client. However, some of them were re-named, and their explanation was changed:

- **Resource server** - The server hosting the protected resources, capable of accepting and responding to protected resource requests using access tokens.
- **Resource owner** - An entity capable of granting access to a protected resource. When a resource owner is a person, it is referred to as an end-user.
- **Authorization server** - The server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization.
- **Client** - An application making protected resource requests on behalf of the resource owner and with its authorization. The term "client" does not imply any particular implementation characteristics (e.g., whether the application executes on a server, a desktop, or other devices).

  There are two types of clients based on the ability to provide confidentiality of credentials: a *public* that is executed on devices owned by the resource owner (SPA, native app), and *confidential* (executed on a secure server).
- **Tokens** - Credentials obtained during authorization flow that used to access a protected resource (access token) or to get a new access token (refresh token). Usually presented in the form of bearer JWT token or MAC token
- **Scopes** - Strings that are defined by the authorization server. The client uses it to request the scope of granted access. Authorization server uses it to notify a user about the scope of access token issued. The resource server uses it to validate permissions of requests.
- **Flows** - sequences of steps to obtain an access token from a resource server that relies on using authorization grant therefore often called authorization grant flows (or grant types).

The Authorization flow consists of 2 steps: Obtaining Authorization Grant and Obtaining Authorization Token. In comparison to the first version of the protocol, the number of steps was reduced: Temporary Credentials Acquisition and Resource Owner Authorization steps from the OAuth 1.0 were united into one Obtaining Authorization Grant step from the OAuth 2.0.

Obtaining Authorization Grant is used to get a resource owner grant in the form of credentials that might be exchanged further on access token. However, this step might not involve the resource owner directly. It even might be omitted in case if the client already has predefined grant credentials (resource owner login credentials or own grant credentials).

Obtaining Authorization Token is used to get access token from the authorization server for further use with API requests to the resource owner. This step requires the client to choose one of the available grant types and provide appropriate authorization grant credentials.

The OAuth 2.0 standard was developed in an extendable way, so it makes it possible to use different grant types; some of them might be mentioned in the specification, others might be non-standard. To use a non-standard grant type client should set absolute URI value to *grant_type* parameter that was defined by an authorization server, for example, *urn:ietf:params:oauth:grant-type:device_code* is used for Device Code Flow.

The following standard grant flows were defined in the specification of OAuth 2.0 (*The OAuth 2.0 Authorization Framework*):

Authorization Code Grant Flow(see figure 2.3) is suitable for OAuth clients that can keep the tokens confidential. These are the clients that are generally deployed in a secure server.

It provides a few significant security benefits such as the ability to authenticate the client and transmission of the access token directly to the client without passing it through the resource owner's user-agent and potentially exposing it to others (including the resource owner).(*Authorization Code Grant*)

Instead of requesting authorization directly from the resource owner, the client directs the resource owner to an authorization server. The resource owner is then redirected back to the client with the authorization code, which the client will capture and exchange for an access token in the background. Since this is a redirection-based flow, the client must be able to interact with the resource owner's user-agent (typically a Web browser) and receive incoming requests (via redirection) from the authorization server (can act as an HTTP server)

The authorization code flow is as follows:

1. The client directs the resource owner's user-agent to the authorization endpoint and authorizes the client to access data on their behalf. The client includes its client identifier, requested scope, local state, and a redirection URI to which the authorization server will send the user-agent back once access is granted (or denied).

2. After the resource owner approves access and provides approval for the requested scopes, the user-agent redirects back to the client using the redirection URI provided earlier (in the request or during client registration). The redirection URI includes an authorization code and any local state provided by the client in the request. The client compares the value of the state to ensure that it's the same value that was sent before to avoid any cross-site request forgery attack.

3. After obtaining the authorization code, the client passes back the authorization code to obtain an access token response. For the Access Token Request following parameters are required: grant_type=authorization_code, code itself, redirect_uri and client_id.

4. After validating the authorization code and ensuring that the redirect_uri parameter is present, the authorization server passes back a token response to the client. When making the request, the client authenticates with the authorization server. The client includes the redirection URI used to obtain the authorization code for verification.

5. After the access token and optionally, a refresh token is granted, the client accesses their data.

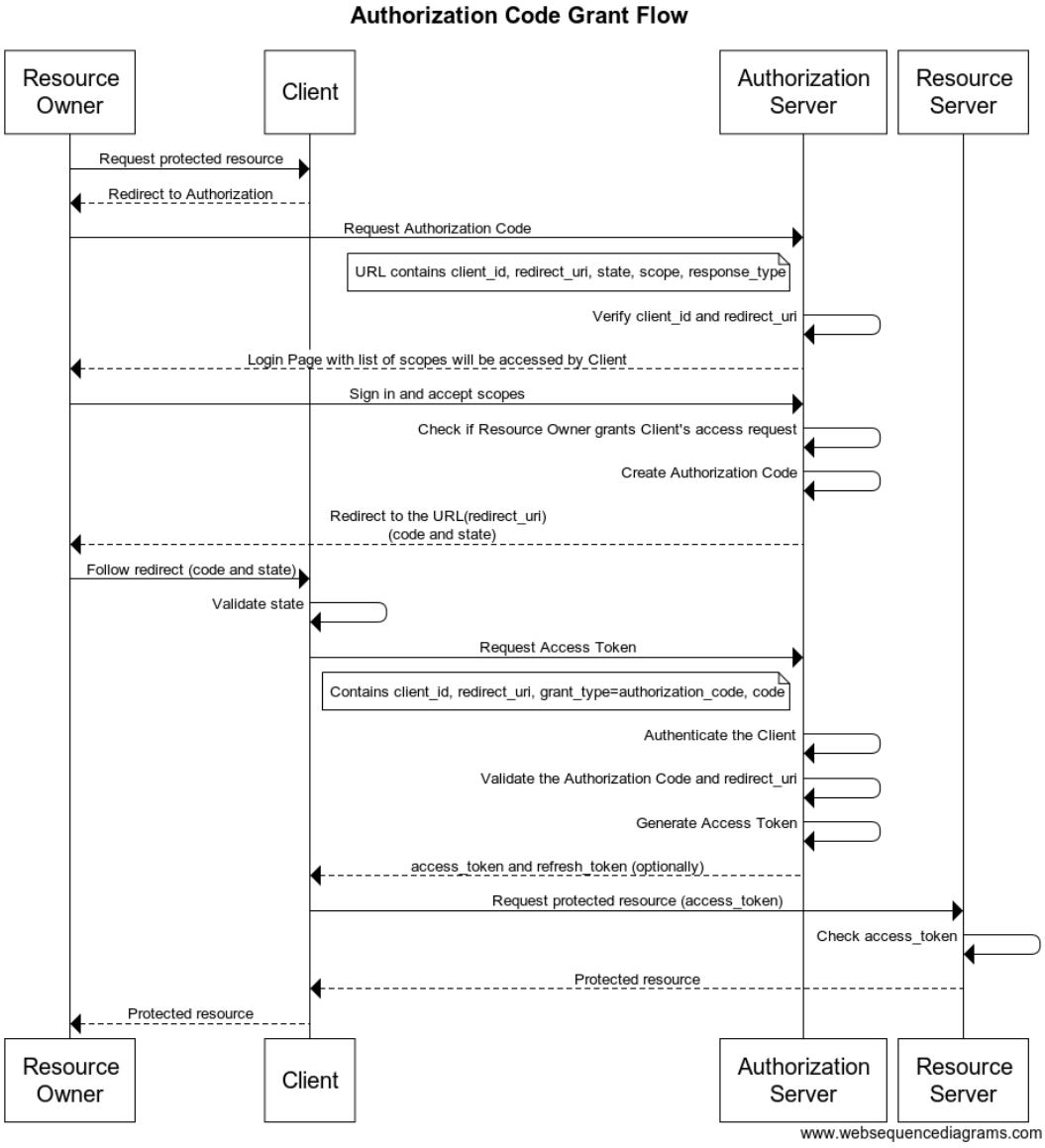

FIGURE 2.3: Authorization Code Grant Flow

Implicit Grant Flow (see figure 2.4) authentication flow is used by client applications (consumers) residing in the user's device. This could be implemented in a browser using a scripting language such as JavaScript, or from a mobile device or a desktop application. These types of applications cannot keep the client secret confidential (application password or private key). Anyone can easily access as these are client-side web apps, for example, like single page web sites.

The primary benefit of implicit grant flow is that it allows the app to get tokens without performing a backend server credential exchange. (*OAuth 2.0 implicit grant flow - Microsoft*)

The implicit grant type is similar to the authorization code grant type, as it will be redirected to an authorization server. However, unlike the authorization code grant type, it will be redirected along with an access token instead of an authorization

code. The implicit grant type does not authenticate the client and instead relies on the presence of the resource owner and the registration of the redirection URI.

To obtain access token client should send a request to the access token endpoint of with following required parameters: *response_type*, which must be "token" and *client_id*, which authorization server is used to verify the client. Also, it is recommended to use a *state* parameter to avoid cross-site request forgery (CSRF) attack. Also, optional parameters are *redirect_uri* - the URL where the server will redirect the user after the authorization process completes and *scope* parameters.

Once all checks and validation are done, the authorization server redirects the user's browser to the URL specified in *redirect_uri*, and this response contains the access token with the following information if required: *expires_in* (recommended), *scope* and *state*.

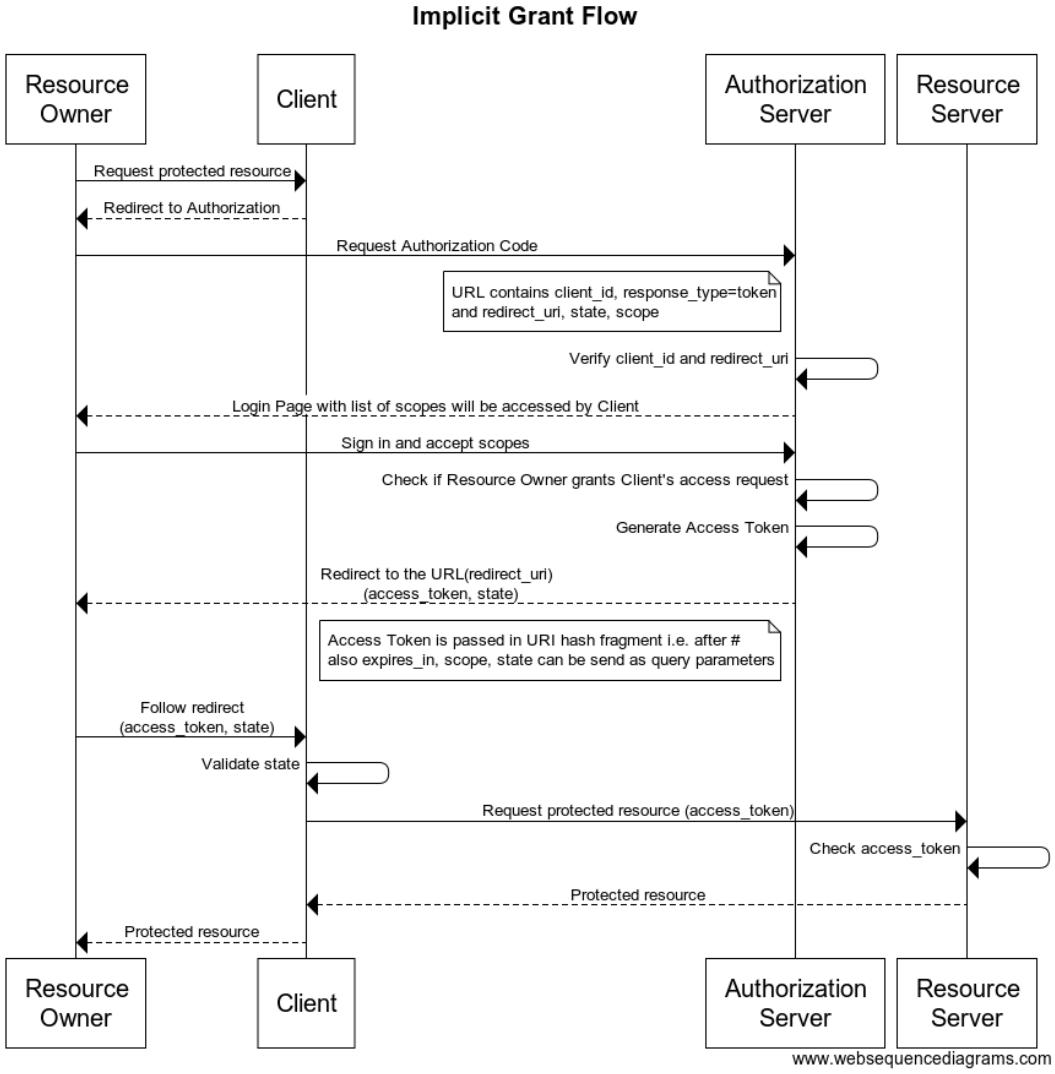

FIGURE 2.4: Implicit Grant Flow

In the implicit grant type flow, the access token is directly returned to the client as a fragment part of the redirect URI. It is assumed that the token is not sent to the redirect URI target, as HTTP user agents do not send the fragment part of URIs to HTTP servers. Thus, an attacker cannot eavesdrop the access token on this communication path, and the token cannot leak through HTTP referer headers.

However, the access token can be leaked in other ways (*The OAuth 2.0 Authorization Framework*):

- parsing the returned URI if the communication is not secured
- from the browser's history
- a malicious client could attempt to obtain a token by fraud.
- replacing or modifying the actual implementation of the client (script)
- CSRF Attack against redirect-uri

To minimize these risks the authorization server should authenticate the client, if possible; require public clients and confidential clients to pre-register their redirect URIs and validate against the registered redirect URI in the authorization request; use short expiry time for tokens; make responses non-cacheable; use state parameters in authorization requests.

Also, the implicit grant flow does not issue refresh tokens, mostly for security reasons. A refresh token isn't as narrowly scoped as access tokens, granting far more power hence inflicting far more damage in case if it is leaked out. In the implicit flow, tokens are delivered in the URL. Therefore the risk of interception is higher than in the authorization code grant. (*OAuth 2.0 implicit grant flow - Microsoft*)

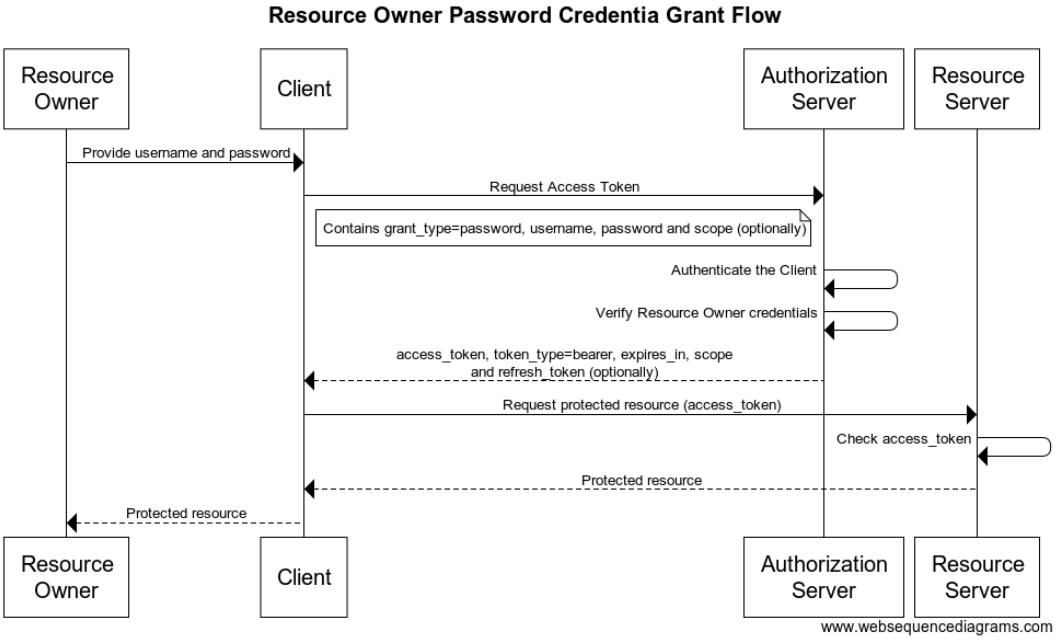

FIGURE 2.5: Resource Owner Password Credentials Grant Flow

Resource Owner Password Credentials Grant Flow (see figure 2.5) is suitable in cases where the resource owner has a trust relationship with the client (e.g., a service's mobile client, the device operating system or a highly privileged application) and in situations where the client can obtain the resource owner credentials.

It is also used to migrate existing clients using direct authentication schemes such as HTTP Basic or Digest authentication to OAuth by converting the stored credentials to an access token.

This grant type must not be used when the service is going to be used by a third-party app. There should be no mistrust between the client app, the authorization server, and the resource owner.

Instead of redirecting the user to the authorization server, the client itself will ask the user for the resource owner's username and password. (*Resource Owner Password Credentials Grant*) The client will then send these credentials to the authorization server along with the client's credentials. To request, it should add *grant_type*=password and resource owner credentials. Once the authorization server authenticates the client and validates the resource owner password credentials, it sends back an access token. Optionally the resource owner password credentials grant might return a refresh token.

Since this access token request utilizes the resource owner's password, the authorization server MUST protect the endpoint against brute force attacks (e.g., using rate-limitation or generating alerts).

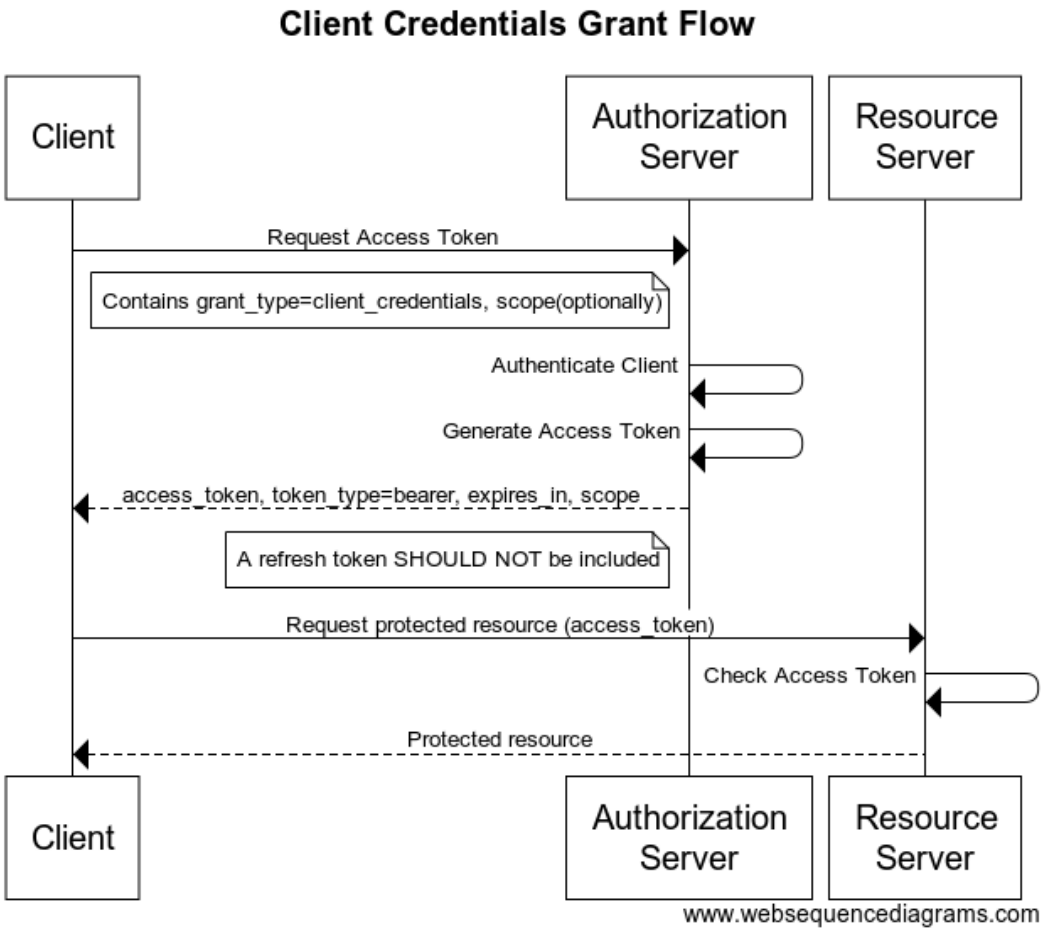

FIGURE 2.6: Client Credentials Grant Flow

Client Credentials Grant Flow (see figure 2.6), sometimes called two-legged OAuth, is suitable for machine-to-machine authentication or for a client making requests to an API that does not require the user's permission. (*Client Credentials Grant*)

This type of grant is commonly used for server-to-server interactions that must run in the background, without immediate interaction with a user. These types of

applications are often referred to as daemons or service accounts. There is no end-user entity participating in the grant type. This grant should be allowed for use only by trusted clients. (*OAuth 2.0 client credentials flow - Microsoft*)

The client can request an access token using only its client credentials (or other supported means of authentication), instead of impersonating a user, when the client is requesting access to the protected resources under its control.

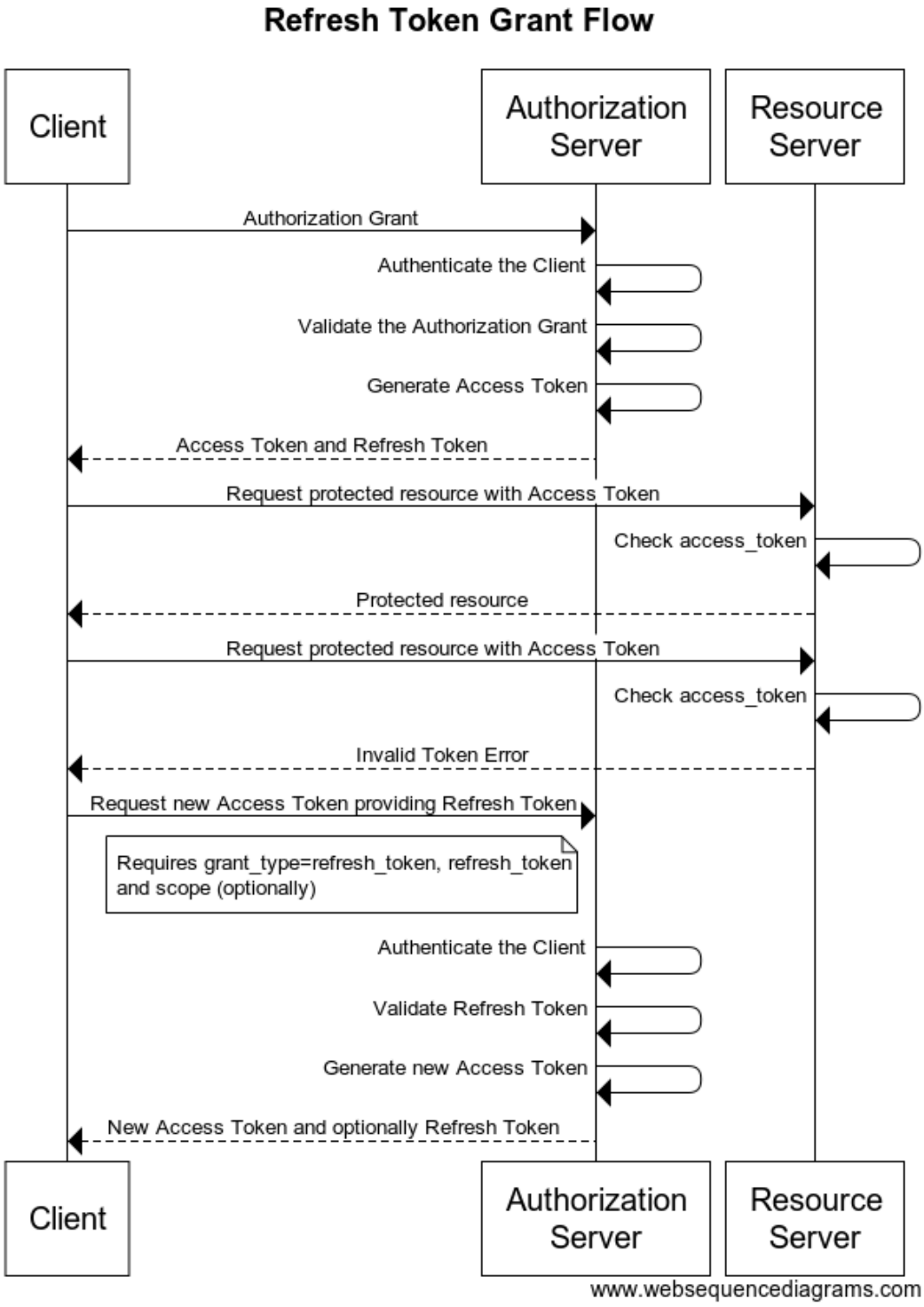

FIGURE 2.7: Refresh Grant Flow

It is similar to the resource owner password credentials grant type except, in this case, only the client's credentials are used to authenticate a request for an access token. Since the client authentication is used as the authorization grant, no additional authorization request is needed. This grant does not support refresh tokens.

Refresh Grant Flow (see figure 2.7) is not a typical grant flow because it requires a refresh token and only uses when the usual access token becomes invalid or expires. The refresh token can also be used to obtain additional access tokens with an identical or narrower scope.

In case the access token expired client can initiate token refresh flow by itself. Otherwise, it makes another protected resource request, and the resource server returns an invalid token error. Then the client invokes the token endpoint with the *grant_type* value *refresh_token* and the refresh token itself.

A refresh token is a string representing the authorization granted to the client by the resource owner. This string usually should not be transparent to the client. It should be sufficiently complicated for other unauthorized parties to be able to generate, modify, or guess it. Unlike access tokens, refresh tokens should never be sent to resource servers.

Because refresh tokens are typically long-living credentials used to request additional access tokens, the refresh token is bound to the client to which it was issued. That's why the authorization server must verify the binding between the refresh token and client identity whenever the client identity can be authenticated.

2.4 OAuth 2.1

OAuth2.1 is an in-progress version of the protocol, the primary purpose of which is to simplify OAuth 2.0 and unite ideas of several relevant papers under the one document. (*The OAuth 2.1 Authorization Framework*)

It deprecates the Implicit Grant Flow and Resource Owner Password Credentials Grant Flow introduced in OAuth 2.0 because of the lack of security. (*OAuth 2.0 Security Best Current Practice*) For Implicit Grant Flow, the reason is returning access tokens in the authorization response that is vulnerable for access token leakage (via open redirect, referer header, or browser history) and access token replay attack. Resource Owner Password Credentials Grant Flow exposes resource owner credentials that increase the attack surface, and also it complicates the support of two-factor authentications.

Instead, it adds two new flows:

Authorization Code Grant Flow with PKCE is based on proposed standard RFC 7636 for public clients.(*Proof Key for Code Exchange by OAuth Public Clients*) The flow is identical to Authorization Code Grant Flow except for additional protection from the authorization code interception attack by using randomly generated codes and their verification on the authorization server.

This extension was introduced because, in the OAuth 2.0 specification, public clients cannot prove that requests that they make belong to them. Authorization servers also have no way to verify that the request was made by a particular client.

Confidential clients have an additional client_secret parameter for that purpose. But the public client should not use it because the attacker might inspect the source code of SPA as well as Native application and find client_secret value even if the code of applications is obfuscated and compiled.

That is why if an attacker intercepts a response from an authorization server that contains authorization code, the attacker might exchange it on access token without any restrictions.

When using PKCE extension client generates *code_verifier* value and retrieves code_challenge derived from*code_verifier*. The idea is the same as using a hash algorithm for password hashing, where *code_verifier* is randomly generated password, and *code_challenge* is hash of the password.

After code generation, the client requests authorization code and sends code_challenge together with request data. Authorization server saves code_challenge and binds it to authorization code. Then the client gets the authorization code and exchanges it on access token by sending authorization code together with *code_verifier*. Authorization server calculates *code_challenge* value from *code_verifier* and compares it to a *code_challenge* saved in storage. If they are identical and authorization code is correct, the authorization server returns an access token to the client.

PKCE protects clients from authorization code interception attacks and is recommended to use even for confidential clients.

Device Authorization Grant Flow is currently the OAuth 2.0 protocol extension, and Aaron Parecki, the contributor of OAuth 2.1, mentioned a possibility to add this flow to the new version of the protocol as top-level grant option.(*It's Time for OAuth 2.1*)

The flow was designed for devices with limited input capabilities or lack of a suitable browser. (*OAuth 2.0 Device Authorization Grant*)

In comparison with other flows where a client obtains a resource owner's grant via redirection (Authorization Code Grant) or implicitly (Implicit Grant, Password Grant), this flow uses HTTP polling. This technique allows the device to repeatedly request the status of user authorization and access token within the interval specified by the resource server. This process is asynchronous and independent of the resource owner authentication process.

In this flow (see figure 2.8), client applications via particular devices (TV, media consoles, printers, etc.) requests *device_code*, *user_code*, and *verification_uri* from the authorization server. Then the client should provide resource owner *verification_uri* and *user_code*, usually by displaying this information on some screen. The resource owner authenticates via *verification_uri* and enters *user_code*.

In parallel to resource owner authorization, the client starts the polling process and continues it until the resource owner will return *access_token* or error. The client should interrupt polling if the *device_code* is expired; otherwise, the resource server will reject the request.

Among other changes listed in specification (*OAuth 2.1*):

- Redirect URIs must be compared using exact string matching
- Bearer tokens in query parameters are no longer allowed
- Refresh tokens must either be a sender-constrained or one-time use

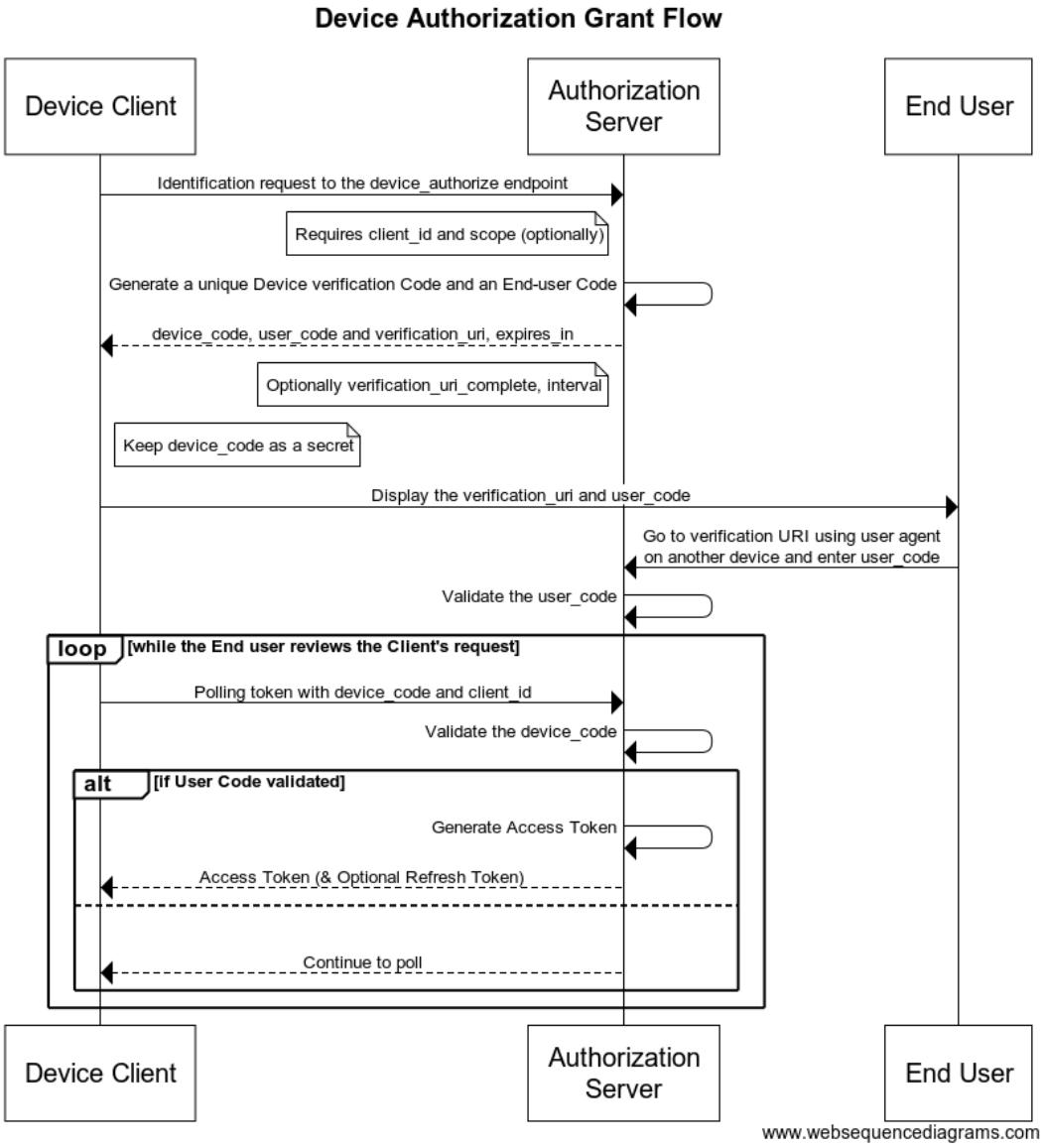

FIGURE 2.8: Device Authorization Grant Flow

Chapter 3

Implementation details

I decided to use Node.js for the purpose of creating my own implementation of the authorization server. Such a decision has several reasons. It allows using JavaScript - the most popular in 2019 language, according to GitHub and one of the top 7 popular languages in May 2020, according to Tiobe Index.

JavaScript is dominant in Web Development, and Node.js is a good choice for creating web servers because of its event-driven design that allows building fast, non-blocking, and scalable applications. Also, I have a background in creating an application using the mentioned languages and technologies. All of this helps to demonstrate examples of using the OAuth protocol, which makes the popularization of auth more effective.

OAuth 2.1 specification was used as a reference for implementation instructions, including information from the reliable plan mentioned by one of the contributors Aaron Parecki. (*It's Time for OAuth 2.1*)

This implementation uses Express.js framework to create a web server because this is the most popular solution in May 2020. (NPM trends express vs others)

3.1 Database schema

Authorization server has the following schema of the database (see figure 3.1):

- **users** - it contains authentication information about a user, a pair of login and password. The password is hashed before saving to the table. To simplify system this implementation does not require email verification from the user.
- **resource_servers** - is used mainly by scopes table to bind scopes to a particular resource server. It contains name column encoded in access token as part of scopes' name and audience list.
- **scopes** - this table represents scopes bind to a particular resource server and is used during resource owner authentication to request an exact grant for permissions. It contains name, descriptions (showed on consent screen), and resource_server_id.
- **clients** - this table contains information about clients. The information is added during client registration by the authorization server admin. There are such columns in the table: name (used on the consent screen and in JWT token), client_id, client_secret, redirect_uri (list of absolute URI), type (used to define whether the client is public or confidential)
- **clients_to_scopes** - this table is used to bind scopes of particular resource server with client

- **codes** - it contains information used during Authorization Code Grant Flow. After the resource owner authentication authorization server generates new authorization code by saving information used in initial request: client_id, redirect_uri, expires_at (expiration date of code, item will be removed after that date), code_challenge (used for PKCE), and code_challenge_method (used for PKCE). client_id and redirect_uri are used for verification during code exchange step.
- **tokens** - this table contains information for refresh tokens. It binds user, client, and code. The parameter code_id allows authorization server revoke tokens if someone tries to exchange authorization code for the second time. Also, it contains expires_at value used to delete refresh token after its expiration.

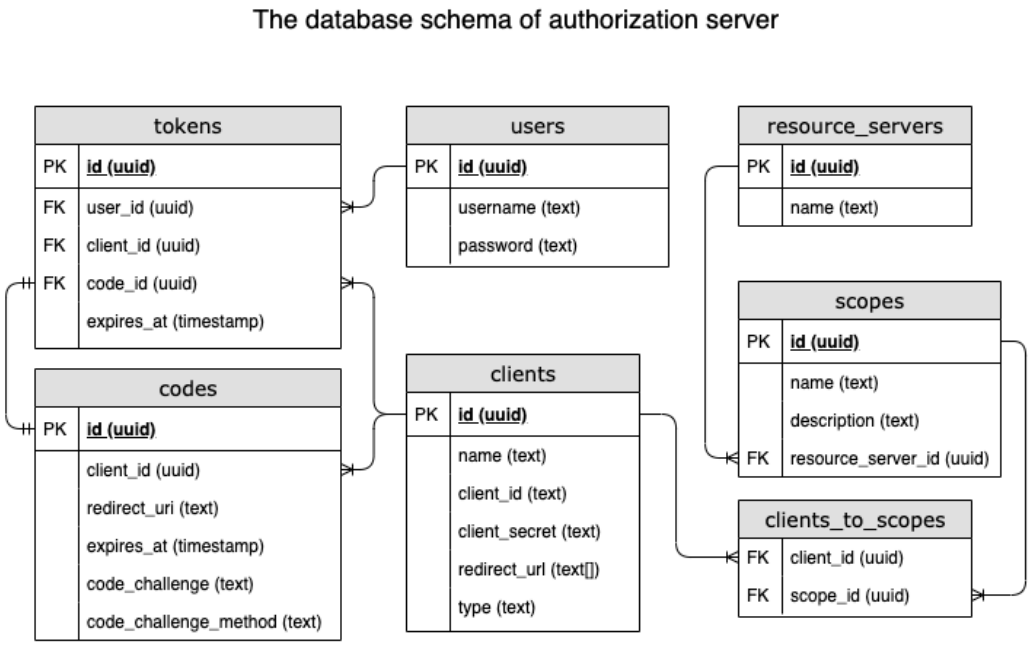

FIGURE 3.1: The database schema of authorization server

3.2 API endpoints

Authorization server provides the following endpoints (see OpenAPI specification):

- **GET /register** - used to get a registration form.

  Request (in query parameters):

  requires the same set of parameters as passed to POST /authorize

  Response:

  page with registration form

- **POST /register** - used to allow users register in the system without participating of admin

  Request (in query parameters):

  requires the same set of parameters as passed to POST /authorize

Request (in form data):

username - resource owner username, his human-readable identifier in the system

password - resource owner secret, that will be hashed and saved for further identity verification

Response: there are no data except HTTP status code

- **GET /authorize** - used to get the login form.

  Request (in query parameters):

  requires the same set of parameters as passed to POST /authorize

  Response:

  page with login form

- **POST /authorize** - used by the resource owner to authenticate in the system.

  Request (in query parameters):

  response_type - defines what response is expected by the client. The only available value is code.

  client_id - id of a client issued after client registrations.

  redirect_uri - URL-encoded one of absolute uri predefined in the authorization server during client registration. Required for code response_type.

  state - random string generated by client to prevent CSRF attack.

  code_challenge - code created via secret code transformation by the client

  code_challenge_method - a method used to make a transformation of secret code

  Request (in form data):

  username - resource owner username, his human-readable identifier in the system

  password - resource owner secret, that will be hashed and compared with stored one in the system for identity verification

  Response (in query parameters): Authorization server redirects to URL provided in redirect_uri

  code - authorization code generated by the authorization server

  state - string passed by the client in the request

- **POST /token** - used by client to obtain or refresh token pair (access and refresh tokens).

  Request:

  grant_type - the grant type of the authorization flow. Available values are authorization_code, device_code, and client_credentials.

  code - (*if grant_type equals authorization_code*) authorization code

  redirect_uri - (*if grant_type equals authorization_code*) should be the same as in /authorization request

  client_id - identifier of client

client_secret - secret key of client issued during client registration (for confidential clients)

code_verifier - (*if grant_type equals authorization_code*) secret value generated by client before requesting authorization code

Response (in JSON):

access_token - short-lived token used for accessing protected resources

expired_at - the date when access token will be expired

refresh_token - long-lived token used to retrieve new pair of access and refresh tokens

refresh_expires_in - the date when refresh token will be expired

token_type - a type of token, this implementation uses "bearer" type

- **POST /revoke** - used by the client to revoke the refresh token and in such way end resource owner session.

  Request (in JSON):

  token - refresh token to revoke. Session associated with a passed token will be removed

  Response:

  there are no data except HTTP status code

- **GET /.well-known/jwks.json** - used by the resource server to get public keys and verify access token.

  Request:

  does not require any parameters

  Response (in JSON):

  public keys used to verify access token presented in the form of JSON Web Key Set (JWKS))

3.3 Access token

Authorization server uses JWT as a representative of access and refresh tokens.

After success resource owner authorization, the authorization server generates a pair of access and refresh tokens.

The access token is generated by providing payload data and signed by the authorization server private key.

Private and public keys of the authorization server are generated every month. Authorization server generates a new pair of keys, and still stores the previous public key. That is why GET request to /.well-known/jwks.json will return a set of keys that will include actual and previous public keys.

Authorization server provides data object to JWT generation method with the property *resource_access* and its value - list of default scopes available for a particular client and granted by the resource owner.

And also uses the following JWT claims:

- jti (JWT ID) - unique identifier of token (might be omitted for refresh token)
- sub (Subject) - identifier of resource owner in the system

- aud (Audience - list of resource servers which scopes are available for client
- exp (Expiration Time) - time after witch token will be expired
- iss (Issuer) - root URL of the authorization server

Resource server should verify token in several steps:
Verify signature with the help of public key, provided by authorization server via /.well-known/jwks.json endpoint.
Verify claims whether exp is not expired. Verify whether claims contain resource server name in aud claim list because an attacker might use token issued for different client and resource owner.
Verify scopes each action (endpoint) on resource server API should be protected with appropriate scopes.
To simplify implementation and restrict the number of possible bugs the following third-party modules used for different steps of generation:

- generate public/private key with Crypto
- sign access token payload (also used on the client to verify token) with jsonwebtoken
- convert public key to JWK with pem-jwk
- helper to request JWK using jwks-rsa

Chapter 4

Conclusion

I can certainly say that I became familiar with the history of auth, starting from the first modern use of a password. Also, I figured out ideas behind modern auth protocols such as OAuth 2.0.

I created an authorization server on Node.js according to the specification of the OAuth 2.0 protocol.

I described the most known OAuth grant type flows, including the pros and cons of their usage, and provide demo applications that rely on the flows.

However, there are grant types not covered in my work, and existing descriptions might be improved. Still, I believe that I have reached most of my goals and prepared the ground for future contributions in auth popularization.

Bibliography

Authorization Code Grant. URL: https://docs.wso2.com/display/IS530/Authorization+Code+Grant. (accessed: 11.05.2020).

Client Credentials Grant. URL: https://docs.wso2.com/display/IS530/Client+Credentials+Grant. (accessed: 11.05.2020).

Fitzpatrick, Brad (2005). *Distributed Identity: Yadis*. URL: https://web.archive.org/web/20060504054201/http://community.livejournal.com/lj_dev/683939.html. (accessed: 11.05.2020).

Mcmillian, Robert. *The World's First Computer Password? It Was Useless Too*. URL: https://www.wired.com/2012/01/computer-password/. (accessed: 11.05.2020).

Morris, Robert and Ken Thompson Bell Laboratories (1979). *Password Security: A Case History*, pp. 595–596. URL: https://spqr.eecs.umich.edu/courses/cs660sp11/papers/10.1.1.128.1635.pdf.

Nachreiner, Corey. *Digital authentication: The past, present and uncertain future of the keys to online identity*. URL: https://www.geekwire.com/2018/digital-authentication-human-beings-history-trust/. (accessed: 11.05.2020).

OAuth 2.0 client credentials flow - Microsoft. URL: https://docs.microsoft.com/en-us/azure/active-directory/develop/v2-oauth2-client-creds-grant-flow. (accessed: 11.05.2020).

OAuth 2.0 Device Authorization Grant. URL: https://tools.ietf.org/html/rfc8628. (accessed: 11.05.2020).

OAuth 2.0 implicit grant flow - Microsoft. URL: https://docs.microsoft.com/en-us/azure/active-directory/develop/v2-oauth2-implicit-grant-flow. (accessed: 11.05.2020).

OAuth 2.0 Security Best Current Practice. URL: https://tools.ietf.org/html/draft-ietf-oauth-security-topics-15. (accessed: 11.05.2020).

OAuth 2.1. URL: https://oauth.net/2.1/. (accessed: 11.05.2020).

One-time password). URL: https://en.wikipedia.org/wiki/One-time_password. (accessed: 11.05.2020).

OpenID Authentication 1.1. URL: https://openid.net/specs/openid-authentication-1_1.html. (accessed: 11.05.2020).

Parecki, Aaron. *It's Time for OAuth 2.1*. URL: https://aaronparecki.com/2019/12/12/21/its-time-for-oauth-2-dot-1. (accessed: 11.05.2020).

— *OAuth 2.0 Simplified: Background*. URL: https://www.oauth.com/oauth2-servers/background/. (accessed: 11.05.2020).

Proof Key for Code Exchange by OAuth Public Clients. URL: https://tools.ietf.org/html/rfc7636. (accessed: 11.05.2020).

Resource Owner Password Credentials Grant. URL: https://docs.wso2.com/display/IS570/Resource+Owner+Password+Credentials+Grant. (accessed: 11.05.2020).

Richer, Justin and Antonio Sanso (2017a). *OAuth 2 in Action*, pp. 35–36. ISBN: 9781617293276.

— (2017b). *OAuth 2 in Action*, pp. 17–18. ISBN: 9781617293276.

RSA (cryptosystem). URL: https://en.wikipedia.org/wiki/RSA_(cryptosystem). (accessed: 11.05.2020).

The OAuth 2.0 Authorization Framework. URL: https://tools.ietf.org/html/rfc6749. (accessed: 11.05.2020).

The OAuth 2.0 Authorization Framework. URL: https://tools.ietf.org/html/rfc6819#section-4.4.2. (accessed: 11.05.2020).

The OAuth 2.1 Authorization Framework. URL: https://tools.ietf.org/html/draft-parecki-oauth-v2-1-02. (accessed: 11.05.2020).

Walden, David and Tom Van Vleck (2001). *The Compatible Time Sharing System(1961–1973)*, pp. 36–37. URL: https://www.multicians.org/thvv/compatible-time-sharing-system.pdf.

Why OAuth 1.0a? URL: https://oauth1.wp-api.org/docs/introduction/OAuth-1.html. (accessed: 11.05.2020).