Introduction to OAuth 2.0

In this lesson we will learn about the OAuth 2.0 protocol, how it works and why it should become the basis of your microservices security strategy.

Introducing OAuth 2.0

Let's start with the standard definition of OAuth 2.0. OAuth 2.0 is the industry-standard protocol for authorization, allowing a website or application to access resources hosted by other web apps on behalf of a user. In essence, it is a security standard where you give one application permission to access your data in another application.

The framework that describes OAuth 2.0 was released in 2012 and as of today has been adopted by dozens of service providers of all sizes.

Let's see how it works.

The OAuth 2.0 authorization flow

There are multiple authorization flows supported by OAuth 2.0, but here we will focus only in the recommended flow for Single Page Applications (SPAs) and native clients, which are the ones our microservices will integrate with during this course.

Let's first understand which are the participants on an OAuth 2.0 authorization flow:

- **The resource owner** is the person or entity that owns a resource. Like John in our case who owns his inventory bag.

- **The resource server** is the server or service that hosts the protected resources owned by the resource owner. An example here is our Inventory microservice which owns the database that contains the inventory items.

- **The client** is the application that wants access to perform actions on protected resources on behalf of the resource owner. This would be our game client, our web front end and even the Postman app when doing local testing.

- **And, the authorization server** is the server capable of authorizing the client to get access to the protected resources after authenticating the resource owner. This is our Identity microservice.

The flow stars when the resource owner uses the client to initiate an authorization request to perform some action, for instance, to retrieve John'ss inventory bag. The client generates a code verifier, which is a hard to guess value that will be used later in the flow.

The client then generates a code challenge, which is an encoded hash value derived from the code verifier, and sends it as part of the authorization request to the authorization server. The authorization server presents the sign in page where the resource owner can authenticate. After authentication, the authorization server might optionally also present an additional page where the resource owner can provide explicit consent to which actions can be performed on the resources he owns.

After this the authorization server stores the code challenge for future verification and generates an authorization code which it sends back to the client. This authorization code is valid only for a short amount of time (ideally no more than 10 minutes) and is good only for one use.

The client then can use the received authorization code plus the code verifier it had created at the start of the flow to request an authorization token. The authorization server verifies this request by generating the code challenge itself out of the received code verifier and comparing it with the code challenge it had already stored. If this doesn't match, then the request likely comes from a malicious application and an error is returned to the client.

If all looks good, the Authorization server verifies the received authorization code and generates an access token. The client can now use this access token to send a request to the resource server so it can perform the desired action on the protected resources. The resource server validates the received access token and if all looks good it executes the requested action and returns the corresponding response.

This whole flow is known as the Authorization Code Flow with PKCE (pixy), where PKCE means Proof Key for Code Exchange. This flow is ideal for public clients like Single Page Applications (SPAs) and native/mobile applications because:

- There is no need to store a static, hard coded secret in the client, which could easily be stolen
- With no secret, the client can't get an access token right away. It needs to first acquire a valid authorization code only available after the user authenticates and provides consent
- Even if a malicious application steals the authorization code, it would not be able to acquire the code verifier, which is dynamic and is only known to our own app

Let's take a closer look at the elements involved in the initial authorization request sent by the client.

The Authorization Request

This is how an authorization request for the authorization code with PKCE flow looks like. Let's understand what each piece means:

- **https://localhost:5003.** This is the address of the authorization server, which in this example is running in our local box.
- **Authorization endpoint.** This is the endpoint in the authorization server that can be used to request an authorization code. There are many other standard endpoints available in the authorization server as we'll see in a future lesson.
- **response_type=code.** By specifying "code" in the response type we are asking the authorization server to respond with an authorization code, assuming successful authentication by the resource owner.
- **client_id=gameclient.** The client_id is the unique identifier of the client, which was provided to the authorization server when the client was registered.

- **scope=Inventory.** Scope represents the resource or set of resources and the actions on these resources that the client would like access to. In this example the client is asking for access to the Inventory microservice REST API. The scope could be more granular if we wanted to allow only for read access, read write access, or perhaps a more custom scope if needed.
- **redirect_uri**. redirect_uri is the address where the authorization server will redirect the user after successful authentication. Just like the client_id, this URI should have been provided to the authorization server when the client was registered, which prevents phishing attacks.
- **code_challenge**. The code_challenge is the encoded value derived from the hashed code verifier that the authorization server will store for future verification.
- **code_challenge_method**. And finally, code_challenge_method is the method used to hash the code verifier to turn it into the code_challenge.

The Access Token request

Let's now inspect the elements of the access token request, which the client sends to the Authorization server after acquiring the authorization code:

- **https://localhost:5003.** Again, this is the address of the authorization server, our local box in this example.
- **Token endpoint.** This is the endpoint in the authorization server that can be used to request an access token.
- **grant_type=authorization_code.** By specifying "authorization_code" in the grant type we are asking the authorization server to exchange the provided authorization code with an access token that the client can later use to talk to any of our microservices.
- **client_id=gameclient.** As mentioned before, client_id is the unique identifier of the client, which was provided to the authorization server when the client was registered.
- **redirect_uri**. Again, redirect_uri is the address where the authorization server will redirect the user after successful authentication. It must match exactly the value used when requesting the authorization code.
- **code**. Code is the authorization code provided by the authorization server via the authorization request.
- **code_verifier**. And, code_verifier is the original code verifier that the client generated before computing and sending the code challenge as part of the authorization request.

I think it's good to know the elements of both the authorization request and the access token request since they help understand the different pieces involved in the OAuth 2.0 protocol.

In the next lesson we will learn about another important component of the microservices security landscape, called OpenID Connect.