FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Faculdade de Engenharia da Universidade do Porto

SSIN: Segurança em Sistemas Informáticos

Practical Assignment: OAuth 2.0 Protocol Implementation

May 25, 2020

Autores	*Identificação*	*Email*
Luís Miguel Sousa	up201605770	lm.sousa@fe.up.pt
Tiago Ribeiro	up201605619	up201605619@fe.up.pt
Nuno Oliveira	up201506487	up201506487@fe.up.pt

A System Architecture and Implementation

This OAuth 2.0 implementation was made using the Django framework, being splitted into three separate components: Authentication server, Client Application and Resource Server. All of these run simultaneously on different local ports.

The client application is responsible to initialize all the process, interacting with the user, requesting and receiving authorizations in order to obtain the needed protected resources. The authentication server is responsible to store all user credentials, generate and check all the authorization codes and access tokens generated, in order to ensure that all the operations are performed safely and privately. The Resource server keeps all the protected resources safe, using the authentication server to validate any incoming request.

All the actions performed can be monitored using the logging screen.

B Implemented features

B.1 Authorization Code Grant

On this first phase of the protocol, the client application requests the permission to request an access token from the authorization server, which would grant access permissions to the protected resources.

This permission comes in the form of an authorization code, which grants temporary permission for that particular app to request a token to access the protected resources of that particular user, bearing in mind the conceded permissions.

The first client screen allows him to choose exactly what permissions does he need (this will be the permissions requested to the authorization server).

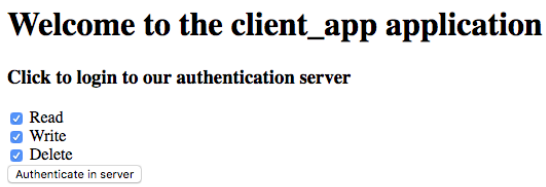

Figure 1: Client Application initial screen

He then sends the request for an authorization code to the authorization server. This request contains the client application information, the URL to where the authentication server should redirect the browser afterwards and a state. This state variable is used to prevent the threat of CSRF attacks by uniquely identifying each request.

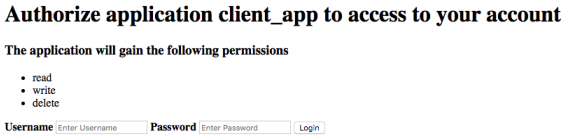

Figure 2: Authorization server confirmation screen

A page from the server it's displayed, confirming all the permissions that are about to be granted for the client application he's using. This allows to prevent the threat of the user unintentionally granting too much

access scope. If everything is alright, he can introduce his credentials to log in the server and if the login is successful, the browser is redirected back to the client application with the authorization code. Otherwise, this redirection is made with an error.

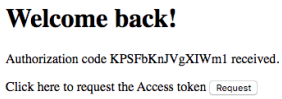

Figure 3: Client Application authorization code success screen

Figure 4: Client Application authorization code error screen

The authorization code is stored in the authorization server with the associated information and an expiration date, meaning that it must be used before this. The application can now exchange this code for an access token that can be used to access protected resources.

B.2 Issuing an Access Token

The tokens used were generated using Simple JWT for the Django framework [JWT].

After successfully receiving an authentication code, the client may now request an access token to the authentication server. In order to do this, it must send a POST request to the /token endpoint of the authentication server with the following parameters:

- grant_type - ”authentication_code” in our case
- code - the authentication code received before
- client_id - the id the client used to register in the authentication server
- client_secret - the secret of the client, used to register in the authentication server
- redirect_uri - the URI to be redirected to after validating the request

The authentication server will the handle the request in the following order:

1. Confirm the grant type and if it is a valid one
2. Authenticate the client with its credentials (client_id & client_secret)
3. Validate the authentication code and fetch its scope
4. Generate a new token pair (access & refresh) with claims for the user_id and scope

If any of the validation steps fail, it will redirect to the redirect_uri with an error message.

If everything succeeds, the authentication redirects to the redirect_uri with access and refresh tokens in its body.

From this point onward, the client may now request access to the protected resource or refresh the current access token using the /token/refresh endpoint of the authentication server.

B.3 Refreshing an Access Token

In order to refresh an access token, the client must send a POST request to the authentication server's /token/refresh endpoint with the following parameters:

- redirect_uri - the URI to be redirected to after completing the request
- client_id - the ID the client used to register in the authentication server
- client_secret - the secret the client used to register in the authentication server
- refresh_token - the refresh token associated with the current access token

The authentication server then processes the request in the following order:

1. Authenticate the client using its credentials
2. Validate the refresh token and if it is associated with the client
3. Refresh the old access token by generating a new one, and keeping the old refresh token

If the validation failed, it then redirects to the redirect_uri with the error associated.

If not, then it redirects to the same URI with the new access token attached in a GET parameter 'access'.

B.4 Protected Resource Access

Using the provided JWT access token, the client can make requests to the Resource Server embedding the token in the 'Authorization' header.

The Resource Server supports the following endpoints and methods:

1. /admin - Administrative interface for monitoring
2. /resources

   GET - List all resources. ('read' scope required)

   OPTIONS

   HEAD - ('read' scope required)

   POST - Create new resource. ('edit' scope required)
3. /resources/«id»

   GET - Read all info regarding the resource with the provided id. ('read' scope required)

   OPTIONS

   HEAD - ('read' scope required)

   PUT - Amend resource data. ('write' scope required)

   PATCH - Amend resource data. ('write' scope required)

   DELETE - Deletes resource with provided id. ('delete' scope required)

Each Resource is presented a JSON object consisting of an 'ID', a URL to /resources/«resource_id» and a 'data' string.

The request verification flow used is the following:

1. Verify the presence of an 'Authorization' header.
2. Decode the JWT token.
3. Query the Auth Server for confirmation of validity of the token (/token/verify).

4. Verify token type is 'access'.
5. Verify that the required scopes for the request operation are present.

These endpoint will return the code 200 on success, 403 on Authorization Error and 404 if the requested resource does not exist.

B.5 Logging service

We setup a logging service that consists of an endpoint for each service (/logger) that displays logs of what is happening inside the services.

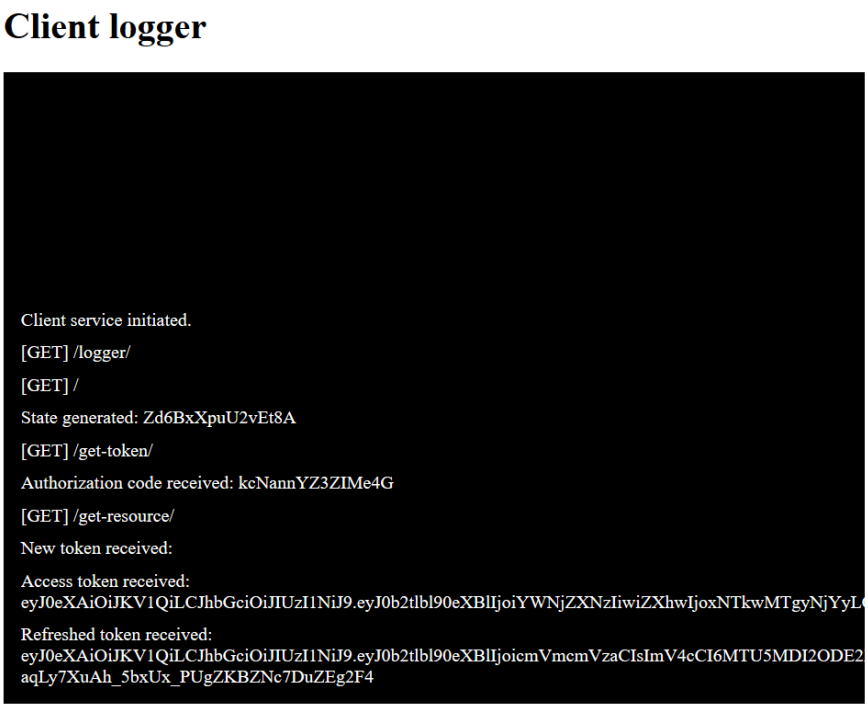

Figure 5: Logger view for the client server

These logs mainly focus on requests to specific endpoints, failed validations, creations of elements such as tokens, users, codes, etc.

Bibliography

[Fra] RFC6749: The OAuth 2.0 Authorization Framework. URL: https://tools.ietf.org/html/rfc6749.

[JWT] Simple JWT. URL: https://django-rest-framework-simplejwt.readthedocs.io/en/latest/index.html.

[MC] RFC6819: OAuth 2.0 Threat Model and Security Considerations. URL: https://tools.ietf.org/html/rfc6819.