Authentication and Authorization Mechanism Implementation as a Variation of OAuth Protocol

Secure Software Engineering

André Costa Lima, up202008169@up.pt
Eduardo Luís Tronjo Ramos, up201906732@up.pt
Fábio Araújo de Sá, up202007658@up.pt
Inês Sá Pereira Estêvão Gaspar, up202007210@up.pt

December 2024

Contents

1 Introduction

The aim of this project was to design and implement a security mechanism, based on the OAuth protocol, which would make it possible to deal with the concepts of authentication and authorization of users of a web application when accessing resources stored on another server.

Figure 1 shows a diagram representing the organization of the system, as well as the actors and operations implemented in the proof of concept.

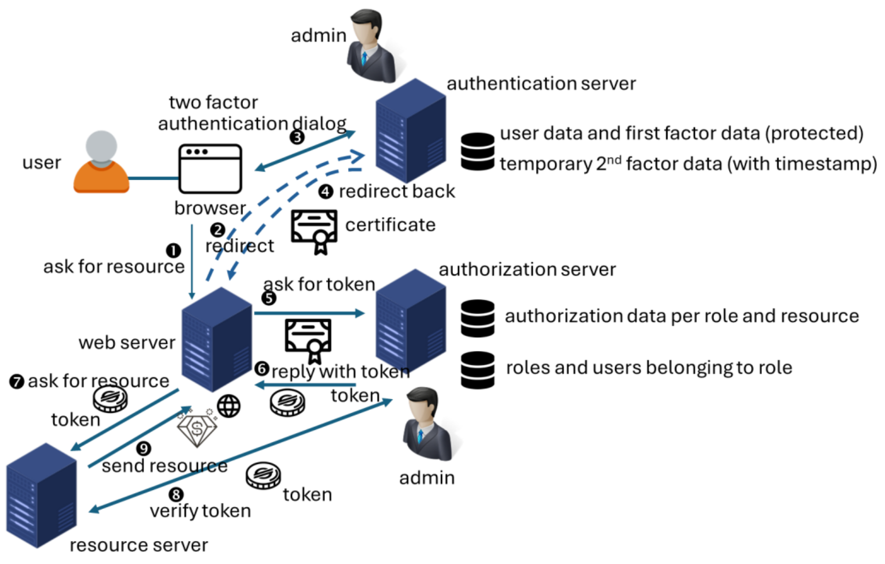


Figure 1: Diagram representing the proof of concept developed

2 System Requirements and Key Workflows

Several actors and components had to work together to implement authentication and authorization functionality based on the OAuth protocol, and certain functional and security requirements were met.

2.1 Actors and Components

The system was based on four essential parts that cooperated to guarantee secure authorization and authentication procedures. As the user's gateway, the client communicated with the web server to make access requests. Identity verification, role administration, and safe data storage were handled by the authentication, authorization, and resource servers, which ensured that every request was verified and appropriately managed.

- **Web Application (Client)**: This web application acts as the intermediary between the user and resource server and is also able to redirect the user to the Authentication server for login purposes. It will also communicate with the authorization server to obtain resource access tokens.
- **Authentication Server (AS)**: This server handles authentication with two-factor authentication by emailing the user a nonce. The server also comes with an SMTP service in order for the server to send and user to receive these emails. The server, through its certificate authority (CA), also creates short-lived certificates for the user.
- **Authorization Server (AuthZ Server)**: This server keeps a database with the roles of the users and resources that are accessible to each role. When the web server wants to exchange a certificate issued by the authentication server for a token, this server first ensures that the certificate is valid and that it is signed by the expected CA. Then, a token with the role of the user for which the certificate was issued is created and provided to the web server. When the resource server wants to authorize an operation, this server validates the token received and checks if the role in the token has access to the requested resource.
- **Resource Server**: This server stores the resources and verifies the authenticity of the tokens received. Finally, it delivers the resource to an authorized user, but only after authorizing the operation with the authorization server.

2.2 Functional Requirements

To guarantee appropriate authorization and authentication procedures, the system must meet a number of functional requirements in the following workflows:

- **Authentication Workflow**:
  - The user requests access to a resource via web browser.
  - The web server redirects the user to the Authentication Server.
  - The Authentication Server performs two-factor authentication:
    1. **First factor**: Username and password.
    2. **Second factor**: OTP via email.
  - On successful authentication, the Authentication Server:
    1. Issues a **short-lived certificate** containing the user's identification, a timestamp, and validity.
    2. **Signs the certificate** using its private key.

- **Authorization Workflow**:
  - The web server sends the certificate to the Authorization Server to request a token.
  - The Authorization Server:
    1. Issues a **token containing the user's role(s)** and any required metadata.
    2. Ensures the token is **encrypted** and **integrity-protected** to prevent tampering or inspection by unauthorized parties.
  - The token is used by the web server to request resources from the Resource Server.
  - The Resource Server:
    1. **Validates the token**.
    2. Checks with the Authorization Server if the operation that is being performed is allowed, or not, for the token that was provided.
  - If authorized, the Resource Server grants access to the resource.

2.3 Administration Operations

The system required administrative operations to manage users and permissions.

- Authentication Server:
  - Register users with:
    1. Username/password.
    2. Side-channel (email) verification at registration (nonce-based).
- Authorization Server:
  - Manage roles.
  - Assign roles to users.
  - Define permissions for each role, specifying allowed resources and operations.

2.4 Security Requirements

Strict security measures were necessary for the system to operate safely and dependably. These precautions were centered on data encryption, secure transmission, token protection, and certificate processing. Every criterion was created to protect data integrity and prevent unwanted access.

- **Certificate Handling**:
  1. The Authentication Server acts as a Certificate Authority (CA) and signs short-lived certificates.

2. Public keys of all servers (Authentication, Authorization, Resource, Web) must be verified and trusted.
3. Every server request includes:
   - ID and address of the client-server.
   - Signature of the request using the client server's private key.

- **Token Security**:
  1. Tokens must have:
     - **Confidentiality**: Ensure only authorized parties can read the token.
     - **Integrity**: Prevent token tampering.
  2. Tokens should be cryptographically signed by the Authorization Server.
- **Communication Security**:
  1. Use TLS for all communications to ensure confidentiality and integrity.
  2. Servers must authenticate each other using certificates issued by a common CA.
- **Data Protection**:
  1. Protect user credentials using industry standards.
  2. Encrypt sensitive data in transit and at rest.

With those requirements in mind, we started to develop our architecture and design our solution, as we will describe in the sections below.

3 Architecture

Regarding the architecture of the system, our implementation is divided into four separate servers, each implementing a key responsibility in the protocol:

- Authentication server, responsible for authenticating and providing an identity proof of the user.
- Authorization server, responsible for authorizing user operations.
- Resource server, responsible for storing and providing resources to users, while delegating authorization to the authorization server.
- Web server, responsible for providing an interface for the user and delegating responsibilities to the other servers.

Since these servers are meant to be deployed separately, we used Docker containers to simulate their isolation in terms of file systems and networks.

3.1 Communications protection

In a distributed setting, it is important to ensure integrity, confidentiality and authentication in the communications between servers. As such, for inter-server communication, TLS with client-side and server-side authentication was used. However, to do this, we first needed to have certificates with which we could authenticate the servers.

A self-signed "Communications Certificate Authority" was created and we used it to sign four different certificates, one for each server, for the following DNS names: *authentication-server.local*, *authorization-server.local*, *resource-server.local*, and *web-server.local*. Then, the HTTPS servers on each server were configured to use these certificates and the corresponding private key. This provides server-side authentication for all communications.

Regarding client-side authentication, a "secure_requests" library was developed. For a given request, the receiving server extracts the Common Name of the certificate of the sending server and checks if it is in the allowed list of common names for that endpoint. If it isn't, the request is aborted.

3.2 Proofs of Identity

To implement the proofs of identity using certificates, the Authentication server needs a Certificate Authority to issue the certificates - the "Authentication Server Identity Certificate Authority". This certificate for this Certificate Authority is preloaded into the authentication server, along with its private key. On the authorization server side, only the certificate for it is preloaded, as this is all that is needed to verify the certificates for the proofs of identity. Sharing the private key would be a security risk.

4 Design Decisions and Implementation

In this section, we will describe the design and the implementation strategies to ensure that the requested security requirements were met, explaining how we ensured compliance with the principles of software security, CIA (Confidentiality, Integrity, and Availability).

4.1 Security Mechanisms

In order to ensure the most general security mechanisms at the server and communications level, the following solution was found:

- **Server reliability and authentication:** all servers have a certificate signed by a CA in order to authenticate them and prove that they are valid and trustworthy, as well as being used to store each server's public and private keys;

- **Confidentiality and Integrity in communications:** all communications ensure these properties since all messages exchanged in the system are protected using TLS, which provides confidentiality and integrity. On top of that, all messages are signed by the sender with its private key, and the signature is verified with its public key on the receiver. In addition to that, every message also contains the sender's ID and address. With client-side authentication in TLS, this guarantees that the message came from the sender in question and that it cannot have been altered (integrity), since no one else has access to its private key.

With these approaches, we can guarantee data protection, confidentiality, and integrity when transmitting information between servers.

4.2 Certificate Management

A custom Python class, **CertManager**, was implemented to handle certificate management. Leveraging the cryptography[1] Python library, this class provides functionalities such as loading certificates and keys from files or raw data, creating certificate authorities (CAs) for issuing new certificates, generating certificates using existing CAs, and verifying certificates by validating their signatures and issuers. The 'cryptography' library was selected due to its comprehensive feature set and ease of use. Furthermore, as a backend, it uses OpenSSL, which is a widely used library for performing cryptographic operations securely.

Certificates and keys are stored in the PEM format, and robust security measures are in place to safeguard private keys. While the **Certificate** class encapsulates a certificate, offering verification and conversion methods, the **Key** class represents a private key, providing signing capabilities. The **CA** class represents a certificate authority, responsible for certificate issuance and is used across servers.

4.3 Authentication

As described in the section above, the Authentication Server is tasked with dealing with the user's login and registration. It must confirm the identity of a user by both its username and password and by a two-factor by way of a nonce sent by mail to the user. A user can be redirected back to the web server and start the authorization process only when confirmed. The server also has to allow an admin to add accounts and remove them as they see fit. In the subsection below we will go over our approach to this server.

4.3.1 Approach

The server runs inside a docker container and was written with the Python library Flask[2]. This library provides us with a web framework that allows us to access the RESTful API and the flexibility with other Python libraries. One of the first practical ones was the SQLite3[3] library, which enabled us to

[1]Available online in: `https://pypi.org/project/cryptography/`
[2]Available on `https://flask.palletsprojects.com/en/stable/`
[3]Available online in: `https://www.sqlite.org/index.html`

configure the tables storing the users's usernames, passwords, and emails. We also store a table to store/archive the generated nonces as part of the two-factor authentication.

If the user tries to log in through the web server, instead he is redirected to the authentication server. There the credentials are validated and, if they are correct, a nonce is generated using the system's random number generator and sent by email to the user. The mail we did was to establish an SMTP service on the server side and utilize the mail.mime.text[4] library. The user can then access their email by accessing the server IP in the web server's SMTP port. The user is now redirected back to the web server with the temporary certificate, generated by the CA, stored in the query parameters if the correct nonce is provided. However, if one of the steps above fails the user is bounced back to the beginning but no certificate is produced and the rest of the system is inaccessible.

For the sake of guarantees that security principles are protected, any sensible stored value, for example, a password, is hashed/salted). Certificate delivery is secured also by the use of encryption by the public keys of each server to preserve the integrity and confidentiality of the information that is being exchanged.

4.3.2 Results and Examples

In the actual project, a registration and login endpoint was implemented in the authentication server. The login page appears in this way:

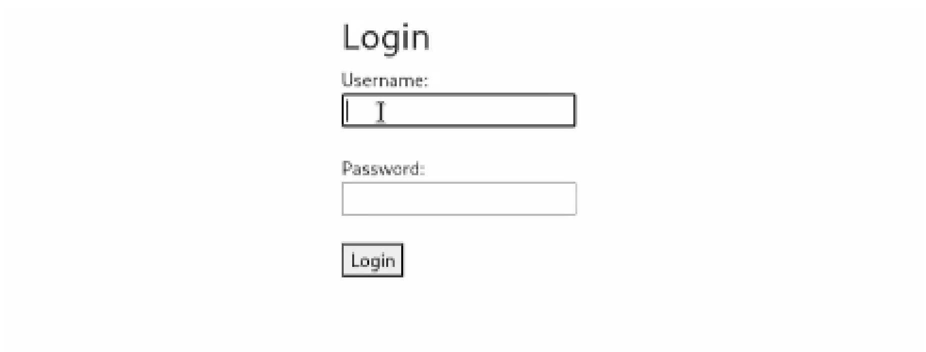
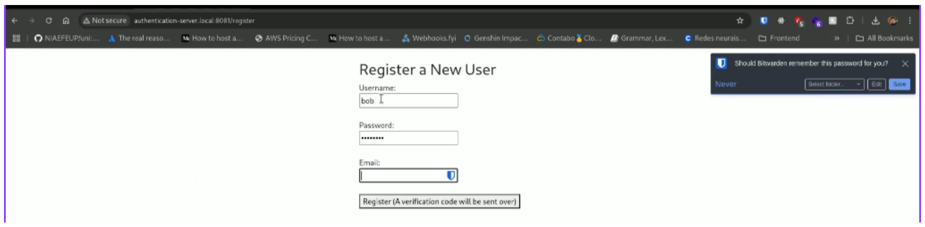


Figure 2: User Login Page

The registration page is shown like this:

[4]Available online in: `https://docs.python.org/3/library/email.html`


authentication-server.local:8081/register

Register a New User

Username: bob

Password: ••••••••

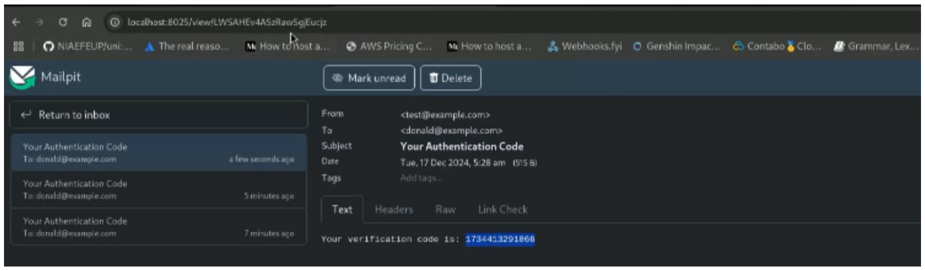
Email:

Register (A verification code will be sent over)

Should Bitwarden remember this password for you?

Figure 3: User Registration Page

Both the registration and login pages produce a nonce that is then used to verify the purported identity. This nonce, generated using the `nonce` Python library. This is a custom library written by us, that takes the current time and uses the system's entropy to ensure that a truly random number is employed in the generation of a unique nonce. After its creation, it is then sent over by email using the server's built-in SMTP. The email looks like this:


localhost:8025/view/LWSAHEv4ASzRawSgjEucjz

Mailpit

Your Authentication Code
To: donald@example.com — a few seconds ago

Your Authentication Code
To: donald@example.com — 5 minutes ago

Your Authentication Code
To: donald@example.com — 7 minutes ago

From	<test@example.com>
To	<donald@example.com>
Subject	Your Authentication Code
Date	Tue, 17 Dec 2024, 5:28 am (515 B)
Tags	Add tags...

```
Your verification code is: 1734413291866
```

Figure 4: Example of email sent

And the nonce verification page looks as such:

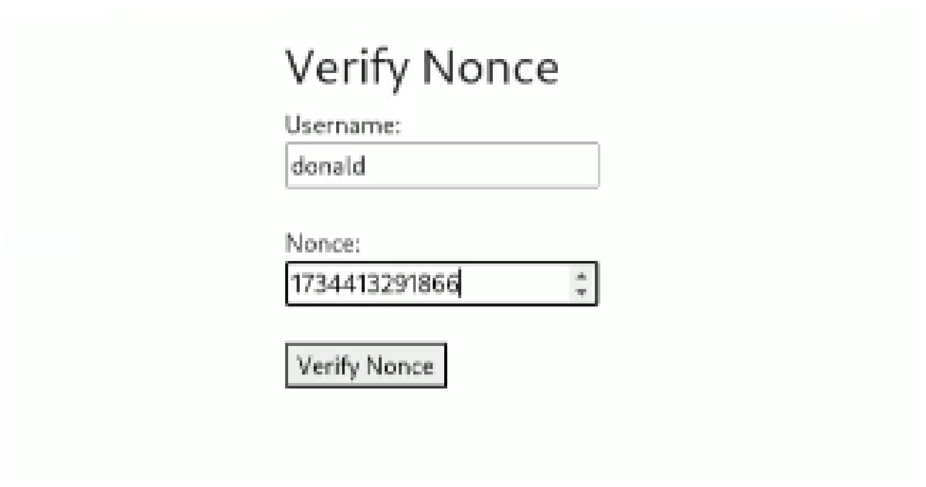


Figure 5: Nonce Verification Page

The nonce takes into account the time of the request for its creation, which can lead to some security concerns, as we will discuss further. After the nonce is sent and the user receives their email, they are redirected to a verification page where the nonce will be requested. After a successful verification, a user will either be logged in or registered, depending on the requested operation.

The admin can then change the user's permissions using the admin panel contained in the server. In the current implementation, this panel is unprotected for ease of use and testing in a development environment. In a real-world application, we would restrict its access using an `.htaccess` file in the Apache servers hosting the project. This file would allow us to restrict the endpoints accessed by users, only permitting certain IPs, such as the one used by the admin.

After the user is logged in, a certificate is issued for the user using the server's Certificate Authority (CA). The CA's private key is stored within the server's code. The certificate is then sent in the headers while the user is redirected back to the web server.

Any user who incorrectly inputs the nonce or any other authentication information, such as the password or email, will be prohibited from logging in and accessing the protected resources.

4.3.3 Security Concerns

Admin Panel Accessibility

The admin panel is currently public and accessible to all. This can be addressed by the method explained above, using `.htaccess` to restrict access.

Nonce Brute Force Attack

If an attacker knows someone is making a login attempt and knows the user's email, they could attempt to brute-force their way through the verification page by trying all possible nonce values before the expiration time of 5 minutes. However, this attack is challenging due to several factors:

- The large number of possible values for the nonce.
- The high entropy of the nonce ensures its high degree of randomness, making it extremely difficult to predict or anticipate.
- The short time frame for the nonce to expire.
- The legitimate user will also be trying to log in, which will consume the nonce, making further attempts by the attacker pointless.

While this attack is technically possible, it is very difficult to accomplish as the attacker would need to know the victim's email, and the exact time of the login attempt, and have the luck to find the correct value within the short expiration window.

Storage of Sensitive Information

All the passwords of the users are stored in hashed form in the authentication server's database. The hashes also use salt, making them much harder to brute force if any data leak were to happen. We use the bcrypt[5] library to not only generate the salt, but also, hash the passwords, ensuring that they are properly hashed. However, they are still vulnerable to rainbow tables. Although a much more cumbersome method of brute-forcing, it remains a very real possibility.

To mitigate this attack, pepper was going to be used, but due to time constraints, we were forced to abandon this issue as more important features required attention. With that said, we still believe that the salted hashed passwords offer strong protection that meets the needs of this project.

For the reasons stated above, we can confidently state that our authentication approach offers the necessary security requirements for the project at hand.

4.4 Authorization

The Authorization Server, as mentioned in previous sections, has the main function of verifying user permissions to access resources. In addition to this function, it is also responsible for validating certificates and sending authorization tokens. This server must allow the administrator to manage the system's resources and users.

[5]Available online in: `https://pypi.org/project/bcrypt/`

4.4.1 Approach

This server runs in an independent Docker container and is written in Flask. In terms of implementing Role-Based Access Control, it was done as follows: a user has a role, and a role has operations. In this case, operations are just the resources that can be accessed. Here's an example of this structure:

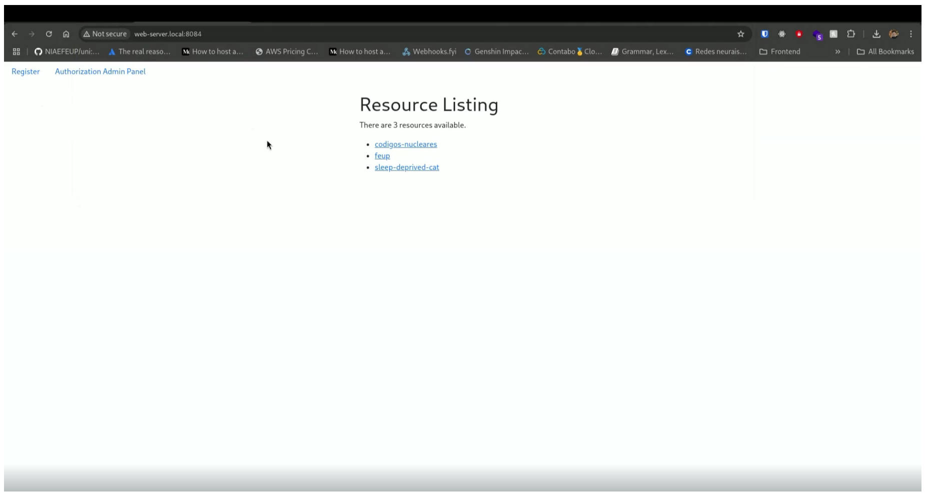
$$UserA- > Role1- > Operation1 : feup.png$$

$$UserB- > Role2- > Operation2 : feup.png, codigos\_nucleares.text$$

In terms of functionality, the authorization server receives a message from the web server with the certificate of the user who has authenticated to the system and validates the user's certificate (from the authentication server) by verifying the signature, thus confirming that the authentication came from a trusted server. Once the verification is complete, the authorization server sends a token containing the user's role encrypted symmetrically with their key to the web server. From there, the web server forwards the encrypted token to the resource_server, which in turn forwards the encrypted token to the authorization_server along with the resource the user wants to access. Thus, the authorization_server receives the encrypted token and the resource. From there, it decrypts the token and gains access to the user's role, which it uses to check whether they have permission to access the resource sent. If the user has the necessary permissions, they are redirected to the file download route; if they don't, they are redirected to a page informing them that they don't have permission to carry out the desired operation.

4.4.2 Results and Examples

On to the implementation and access pages for the resources. Access begins when the user goes to the home page and clicks on the resource they want to download (Figure 6). From there, the user is redirected to the login page which proceeds with the rest of the authentication functionality explained in Section 4.3.2.



Register Authorization Admin Panel

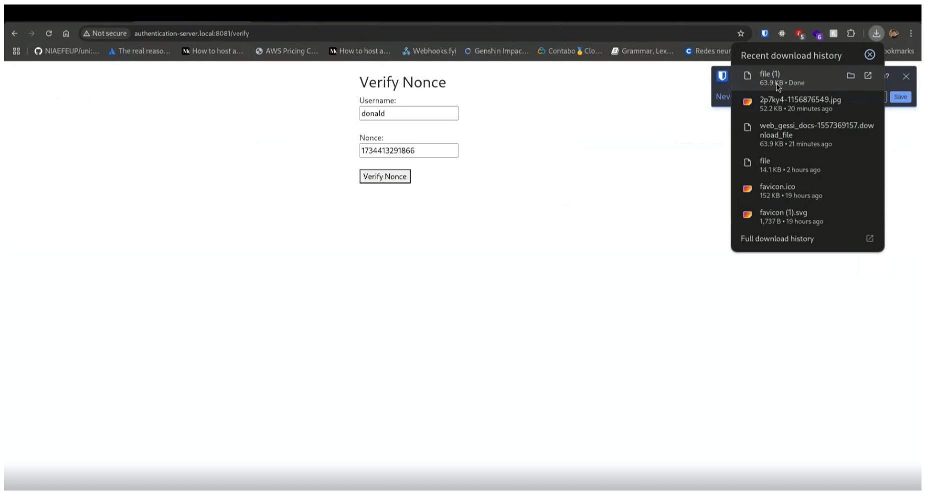
Resource Listing

There are 3 resources available.

- codigos-nucleares
- feup
- sleep-deprived-cat

Figure 6: Main Page with a list of resources

After logging in, the user is redirected to the main page and if the user has access to the resource they clicked on, the document is downloaded (Figure 7 and 8); if they don't, they are redirected to a page informing them that they don't have permissions (Figure 9).



Verify Nonce

Username: donald

Nonce: 1734413291866

Verify Nonce

Recent download history

file (1)
63.9 KB • Done

2p7ky4-1156876549.jpg
52.2 KB • 20 minutes ago

web_gessi_docs-1557369157.download_file
63.9 KB • 21 minutes ago

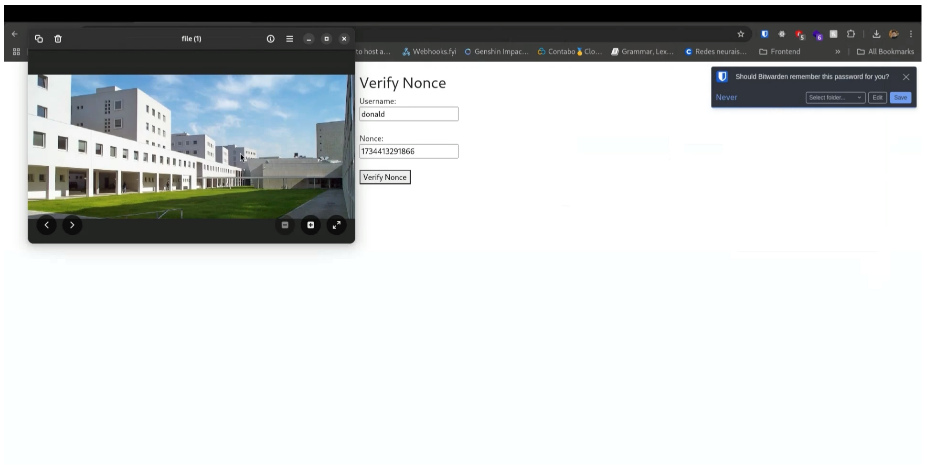
file
14.1 KB • 2 hours ago

favicon.ico
152 KB • 19 hours ago

favicon (1).svg
1,737 B • 19 hours ago

Full download history

Figure 7: Download of the file that the users wanted to access



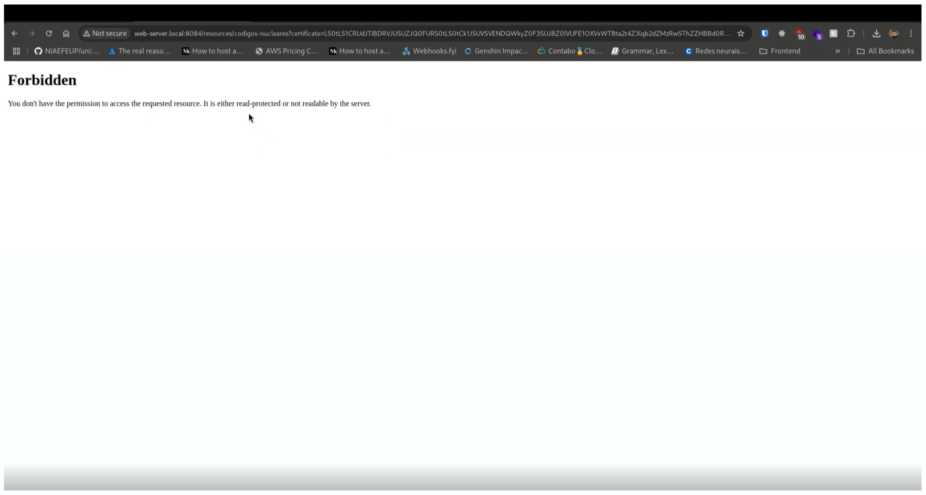
Verify Nonce

Username:
donald

Nonce:
1734413291866

Verify Nonce

Figure 8: Image of the file downloaded



web-server.local:8084/resources/codigos-nucleares?certificate=LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUVSVENDQWkyZ0F3SUJBZ0lVUFE1OXVxWTBta2t4Z3lqb2dZMzRwSThZZHBBd0R...

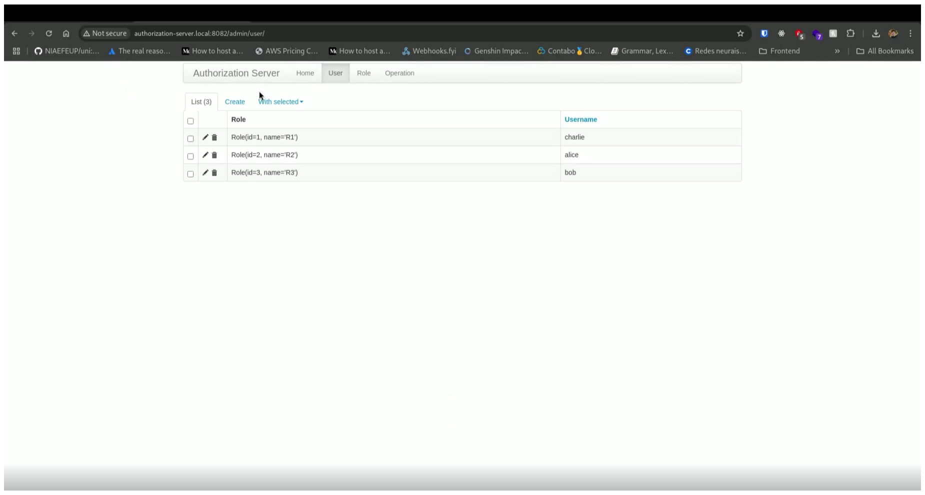
Forbidden

You don't have the permission to access the requested resource. It is either read-protected or not readable by the server.

Figure 9: No permissions page to access the resource

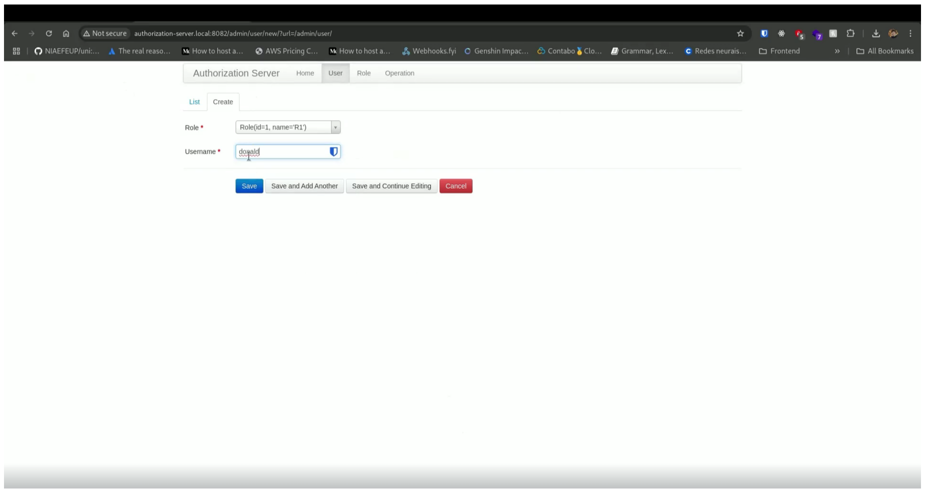
Regarding the admin pages, they were implemented taking into account the admin page that Flask-AdminSQLAlchemy[6] provides. This page allows you to manage users, roles, and operations (resources), such as adding, editing, and removing instances (Figure 10 and 11).

[6]Available online in: `https://flask-admin.readthedocs.io/en/latest/index.html`



Role	Username
Role(id=1, name='R1')	charlie
Role(id=2, name='R2')	alice
Role(id=3, name='R3')	bob

Figure 10: Administration Page that allows the management of users, roles, and operations



Role * Role(id=1, name='R1')

Username * donald

Figure 11: Administration Page that allows the addition of users, roles, and operations

4.4.3 Security Concerns

Admin Panel Accessibility

As stated in the subsection above the admin panel is currently public and accessible by all and thus can be addressed by the method explained above, using `.htaccess` to restrict access.

5 Conclusion

This project successfully delivered an authentication and authorization mechanism based on the OAuth protocol, fulfilling the required functional and security objectives.

The implemented system integrates key components — Authentication Server, Authorization Server, Resource Server, and Web Client — ensuring secure authentication and role-based access control. Features such as two-factor authentication, short-lived certificates, and encrypted communications with nonce were implemented to enhance security and mitigate risks.

The design adheres to essential principles of confidentiality, integrity, and availability, ensuring secure data handling and reliable access management. The project demonstrates a practical and effective approach to implementing a secure and scalable authorization system.