Part A: Implementing a Secure API

Dilan Coral[1], Juan Diaz [2]Steven Zambrano[3]
School of Mathematical and Computational Science
Universidad Yachay Tech, Urcuquí - Ecuador

December 10, 2023.

Following the guide to implement a secure API architecture and the Flask prototype implemented during the class, do the following:

1 Implement each one of the layers: Rate-Limiting, Authentication, Audit Log, and Access Control. Explore and investigate more advanced techniques or configurations for each layer to set new security levels.

1.1 Rate-Limiting

For the implementation of the Rate Limit, the **flask-limiter** library was used. It is a Flask extension that provides functionality to limit the frequency of requests to a Flask application. It helps prevent abuse and protect against denial of service (DoS) attacks by limiting the number of requests a client can make in a given time period.

1.2 Authentication

For the implementation of Authentication, we used Flask-BasicAuth's **BasicAuth** library provides basic HTTP authentication functionality for Flask applications. HTTP Basic authentication is a simple authentication method in which the client sends its credentials (username and password) in the Authorization header of the HTTP request.

Also for password security, the **werkzeug.security** library was used, which provides the **generate_password_hash** and **check_password_hash** functions, which are useful for working with passwords securely. These functions are designed for the secure storage of passwords in a database, using hash and salt algorithms to improve security.

1.3 Audit Log

For the log audit, the **logging** module was used to record relevant information about the operation of your application. This includes debug messages, information, warnings, errors, etc. The logged information can be useful for diagnosing problems, tracking the execution flow of the application, and monitoring its behavior.

In turn, a table in the database was used to manage these logs according to the date they were issued.

1.4 Access Control

To manage access control, it was necessary to use a database, in which we store the users, the roles and the tasks that these users can perform. This can be seen better below:

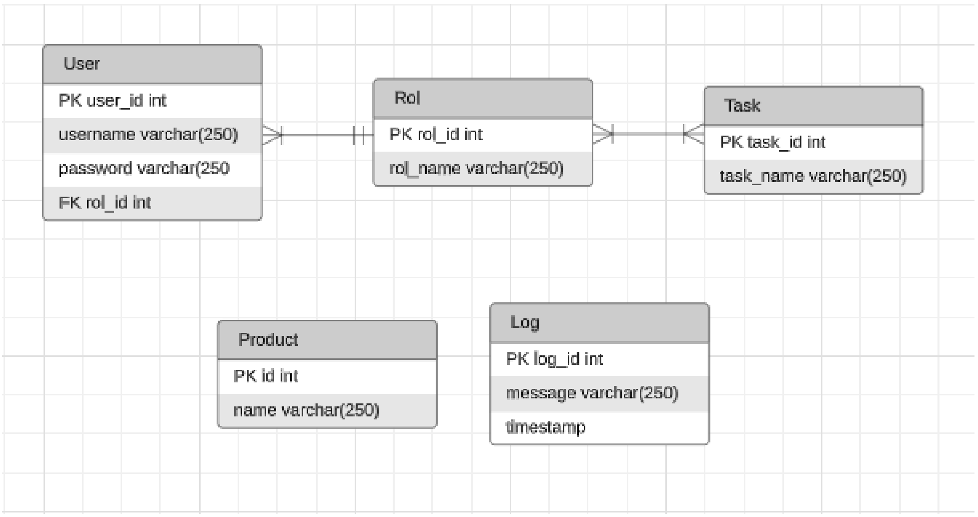

Figura 1: Data base of the api

1.5 Functions of the api

Within the API we find the functions that will allow us to authenticate and authorize. These are the Register and Login functions. In turn, there are functions for viewing, editing, creating and deleting the products present in the 'products' table according to the roles and tasks that are allowed to the user.

2 Check your implemented code using a Static Application Security Testing (SAST) tool. Capture and save the vulnerabilities list and implement the recommendations given by the report. You must show evidence that you follow the guidance.

Using Bandit I found the following vulnerabilities.

```
[main]  INFO    profile include tests: None
[main]  INFO    profile exclude tests: None
[main]  INFO    cli include tests: None
[main]  INFO    cli exclude tests: None
[main]  INFO    running on Python 3.9.18
[node_visitor]  WARNING Unable to find qualified name for module: trial.py
Run started:2023-12-10 00:33:28.214943

Test results:
>> Issue: [B201:flask_debug_true] A Flask app appears to be run with debug=True, which exposes the
    Werkzeug debugger and allows the execution of arbitrary code.
   Severity: High   Confidence: Medium
   CWE: CWE-94 (https://cwe.mitre.org/data/definitions/94.html)
   More Info: https://bandit.readthedocs.io/en/1.7.5/plugins/b201_flask_debug_true.html
   Location: trial.py:278:4
277     if __name__ == '__main__':
278         app.run(debug=True, host='0.0.0.0', port=port_number)
                Low: 0
                Medium: 2
                High: 0
Files skipped (0):
```

To solve this security flaw it is necessary to make a change in the code, more specific in the debug section. Such that:

```
if __name__ == '__main__':
    app.run(host='192.168.1.9', port=port_number)
```

Then we do not find vulnerabilities.

```
    (apidb) PS C:\Users\User\anaconda3\envs\apidb\product-service\src> bandit trial.py
[main]  INFO    profile include tests: None
[main]  INFO    profile exclude tests: None
[main]  INFO    cli include tests: None
[main]  INFO    cli exclude tests: None
[main]  INFO    running on Python 3.9.18
[node_visitor]  WARNING Unable to find qualified name for module: trial.py
```

```
                High: 0
        Total issues (by confidence):
                Undefined: 0
                Low: 0
                Medium: 0
                High: 0
Files skipped (0):
```

3 Configure and run your API under HTTPS protocol.

We set **ssl_context** in the app.run() method to enable SSL/TLS support and provide the SSL context information needed to establish a secure HTTPS connection. More specifically we use **ssl_context='adhoc'**: When you use this value for ssl_context, Flask will dynamically generate a self-signed certificate for your application. This certificate is not suitable for production environments as browsers will not trust it by default, but it is useful for local development and testing.