TP

September 24, 2024

1 General instructions

This TP consists in 8 × slots of 1h30. By two-person teams, you will have to follow and answer the instructions that will be given to you latter. The notation consists in:

- a small PDF report answering the questions that require so.
- the project source code.
- a 10min discussion with the teacher on October 10th (last TP slot).

2 Goal

The DGSE want to track cybersecurity incidents and which of their agent is affected to the investigation.
Your goal is to help them by creating an **REST API** (Application Programming Interface) to handle their data and access them. A REST API is a way to communicate with a server and access resources located in the server. For this REST API, all queries made to communicate to the server and for the responses will be in **HTTP**.
You will implement a **Flask** application to render the server, and **sqlite3** to handle the database (a skeleton of such an application will be provided to you).

3 Introduction

The DGSE has specific requirements for its data. The agents are authenticated using a username and a password. When an attack is discovered, an agent is tasked with collecting information about it, such as the date of the attack, its title, description, type of attack (espionage, data destruction, etc.) and the main sources of information. Next, the agent must gather information on the victims of the attack and the industry sectors that have been targeted. Once this data has been collected, the agent can begin the real investigation by looking for the group of attackers responsible for the attack. He wants to

know which country they come from and whether or not they are sponsored by their government. Sometimes those responsible for the attack cannot be found or clearly identified, so the agent can mention the main suspect of his investigation without confirming it. Finally, the agent should add a possible response from the victim to the attack in the form of a type and source of information.

4 Database design

4.1 Question 1

From the model description above,

1. propose an ER diagram;
2. translate the ER diagram into a collection of relational tables (i.e., the physical model). For each table, specify the primary and foreign keys. The proposed model should be **3NF**.

Ask your supervisor to validate the schema.

5 Data processing

5.1 Question 2

Given the data provided for the database in the CSV file by the supervisors, split the column "Response" into "type of response" and "source of response".

Implement the function *transform_data()* located in **./db/__init__.py** to generate a new CSV file with the new columns.

5.2 Question 3

In the CSV file, there are indications that some of the group of attackers has not been identified with certainty (see column *Affiliations*). Identify them and add a column to determine whether the group responsible for the incident has been confirmed or not.

Update the function *transform_data()* located in **./db/__init__.py** to generate a new CSV file with the new columns.

6 Database initialisation

6.1 Question 4

Complete the function *load_config()* in **utils.py** to load the configuration of the database from the configuration file located in **./config/config/**.

6.2 Question 5

Complete the function called *create_database()* in **./db/__init__.py** to create a database based on the model proposed before.

About **column types**:

- Ids must have the type INTEGER
- Password must have the type BINARY(256)
- Any boolean must have the type INTEGER with either a 0 or a 1

Then run the file **test_create_db.py** and verify the file **./data/incidents.db** is created.

6.3 Question 6 - Populate the database

Complete the Python function called *populate_database()* in **./db/__init__.py** taking in entry the name of the CSV file you created (after the modifications performed the previous question) to populate the database with the data.

6.4 Question 7

Update *init_database()* in **./db/__init__.py** to populate the database after creating it.

7 First communications with the database

7.1 Question 8

Read the code and the comments of function *insert_agent()* in **./db/agents.py** to learn how to run a parameterized query on the database and how to handle the sqlite3.IntegrityError.

7.2 Question 9

The file **test_agents.py** is a file to test the functions implemented in **./db/agents.py**. Run it and check whether Hubert has been added to the database by decommenting *print_agents()* to list all agents in the database.

- How do you explain that you do not see Hubert in the list?
- By looking at code that we used to create the tables, how would you modify the code in **./db/agents.py** in order to fix the problem?
- Rerun the code after the modification and verify that Hubert is now listed by *print_agents()* .

7.3 Question 10

Rerun the code. Which error do you get now? And why do you get it?

7.4 Question 11

Complete the function *update_password()* to update the password of an agent.

7.5 Question 12

What happens when you try to update the password of an agent that is not in the database?

To help you understand, you can use the functions in **test_agents.py**: *test_update_password_existing_agent()* and *test_update_password_non_existing_agent()*

8 Handle incidents in the database

Now that you can create new agents, update their password and retrieve all information about them from the database, you can find in the **./db** folder to communicate with the database.

8.1 Question 13

Implement:

1. in **./db/attackers.py**:
   - *insert_attacker()*, adds an attacker in the database.
   - *update_attacker_sponsor()*, updates the attacker sponsor in the database.
   - *get_attackers()*, returns all attackers from the database.
2. in **./db/targets.py**:
   - *insert_target()*, adds a target in the database.
   - *get_targets()*, returns all targets from the database.
3. in **./db/responses.py**:
   - *insert_response()*, adds a response in the database.
   - *get_response()*, returns the response in an incident from the database.
4. in **./db/sources.py**:
   - *insert_source()*, adds a source in the database.
   - *get_sources()*, returns all sources from the database.
5. in **./db/incidents.py**:

- *insert_incident()*, adds a incident in the database.
- *get_incident()*, returns an incidents from the database.
- *update_incident_attacker()*, updates the attacker of an incident in the database.
- *update_incident_response()*, updates the response of an incident in the database.
- *add_incident_target()*, adds a new target to an incident in the database.
- *remove_incident_target()*, removes a target from an incident in the database.
- *add_incident_source()*, adds a new source to an incident in the database.
- *remove_incident_source()*, removes a source from an incident in the database.

9 Create the first API routes

In this section, you have to create the first routes of the API.

9.1 Question 14

Take a look at the file **app.py** and read the code and comments. To start it, run:

```
$ python app.py
```

What does the application do when you run it?

9.2 Question 15

So far, the application only render routes to handle information about the agents. The routes are partially defined in **./routes/agents.py** as well as the code executed when you query one of them.

- What is the path of the route to get all agents?
- What is the type of HTTP request you can make to access it?

9.3 Question 16

Create a script in Python to make the following query to the database using the library **request**:

```
GET localhost:5000/agents/
```

In the request, "localhost" is the address and 5000 is the port number

9.4 Question 17

Complete the functions in **./routes/agents.py**:

- *get_agent()* to retrieve an agent based on its username as well as the list of assigned incidents.
- *patch_password* to update the password of an agent.
- *add_agent()* to add a new agent to the database.

10 Help to debug

10.1 Question 18

In the file **./routes/data.py**, you can find the prototypes of functions to help you debug the database that will be useful in the following questions. Implement them based on the specifications given in the function comments.

11 Manage the incidents

11.1 Question 19

Implement all functions to handle an incident in **./routes/incidents.py**

12 Securing the API

With the previous question, you have now an API to handle incidents and agents. However, it lacks of security. First, the **password** are stored in plain text, which is definitely not a good practice. Moreover, the routes are not secured as **no authentication** is required to make calls to the API you created. Finally, all **communications are in clear** because we use the HTTP protocol which is not secure. In a real life API, all communications between the client and the API server must be encrypted using the HTTPS protocol. In this section, we will focus on the problem of password stored in plain text and the authentication of the user making the queries.

12.1 Question 20

A good practice when creating a database is to store the password hashes rather than the password itself. To do this, Python provide the *bcrypt* library you can install in your virtual environment using:

```
$ python -m install bcrypt
```

Now, using the documentation of bcrypt, complete the functions in **utils.py**:

- *hash_password()* to generate the hash of a password

- *check_password()* to check whether a password provided as plain text is conform to a hash value
- *check_agent()* to check whether the agent username supplied with a plain text password corresponds to an agent in the database and whether the password is correct.

12.2 Question 21

Update function insert_agent() in **./db/agents.py** to store the hash of the password in the database instead of the plain text value.

12.3 Question 22

The second security issue you need to tackle is the authentication. A good practice to handle this issue is to use tokens. The idea is to make an initial request with your password and username to recover a token. Then, add the token to the headers of your HTTP request. For example, to query the agent list using a token, the query is:

```
GET localhost:5000/agents/ HTTP/1.1
Authorization: Bearer <token>
```

First, add the PyJWT library to the virtual environment:

```
$ python -m install PyJWT
```

Then, implement in **utils.py** the functions *generate_token()* and *check_token()*.

12.4 Question 23

To enable a user to get a token, implement the route "/login" in **./routes/auth.py**.

12.5 Question 24

Look at the code in **./routes/auth.py** for the function *token_required()*. The function creates a decorator that can be added to routes to ensure a token is provided and verified before rendering the data.
Add this decorator to the routes you think requires protection and justify it.