CSE 5382 Secure Programming - Input Validation

Navneeth Krishna
1002050459

1 Introduction

The idea behind this project is to implement input validation, harden the existing application in terms of API and application security and make it more robust against most common attacks especially that arise from user's input. Input validation holds a central role in application security as it gives the user a direct access into the application's functionalities and allows user to feed the application with anything. As the primary assumption that need to be made in terms of application security, the user inputs must not be trusted. A user may have malicious intent, maybe the user might want to steal information or even disrupt the application availability. Keeping all these things in mind, this project serves not only as a learning curve, but also as a proof of concept of the effectiveness of utilizing security measures in protecting the application.

2 Technical Requirements

1. This project has been implemented with the help of **Python** programming language and will use Python and it's libraries as mentioned in the *requirements.txt* file. You can download Python from here.
2. **Postman** application has been used to test the application endpoints and "postman collections" have been used to write the unit tests in this application. Postman can be downloaded from here.
3. **Docker** has been used to containerize the application for easy deployment and testing. You can get Docker from here.
4. **SQLite3** has been used in conjunction with Python for database to store the data.

3 Security Features

The application has been modified from the way it was supplied with the project. It has been hardened by implementing input validation, secure databases, logging and appropriate error messages. The following sections will give a detailed overview on what changes are implemented.

3.1 Input Validation

The application has implemented a robust input validation that handles what inputs are valid and what are not. This is done with the help of robust regexes. Regexes allow for pattern matching. Before the input is fed into the database, they will have to go through a set of regexes for both the name and phone number fields. The following regex has been used to validate name:

```
"^[A-Za-z]+['\'-]?[A-Za-z]+[.]?( [A-Za-z]+(['\'-]?[A-Za-z]+[.]?){0,2}){0,2}$",
"^[A-Za-z]+['\'-]?[A-Za-z]+[.]?,( [A-Za-z]+(['\'-]?[A-Za-z]+[.]?){0,2}){0,2}$",
"^[A-Za-z]+['\'-]?[A-Za-z]+[.]?( [A-Za-z]+[.]?){0,2}$",
"^[A-Za-z]+['\'-]?[A-Za-z]+[.]?,( [A-Za-z]+[.]?){1,2}$"
```

This will make sure that there will be no conditions where a dangling single quote, multiple spaces, SQL injection attacks are not fed as inputs to the application. The following regex has been used to validate the phone number:

```
"^[0-9]{5}( [0-9]{5})?$",
"^[0-9]{5}\\.[0-9]{5}$",
"(^\\+[1-9]{1,2}[ ]?\\(|^[1][ ]?\\(|^[0][1][1][ ][1]?[ ]?\\(?|^\\(?)"
"([1-9][0-9]{1,2})?\\)?[- ]?([0-9]{3})[-| ]([0-9]{4})$",
"^(\\+[0-9]{2} )?([0-9]{2} ){3}[0-9]{2}$",
"^1?(\\([0-9]{3}\\)|[ ]?[0-9]{3} |\\.?[0-9]{3}\\.|-?[0-9]{3}-)[0-9]{3}[ .-]?[0-9]{4}$"
```

This regex has been developed keeping in mind that the application also gets international phone numbers. The regex will filter out any out of bound inputs. This has been implemented as a part of input validation application hardening.

3.2 Database with SQLite3

There are two database tables that are used to harden the application. One database table stores the data pertaining to user inputs and the other one stores the users who are authorized to access this application. So, every request to the application has to have the username and password in order for that request to be successful. The schema of PHONEBOOK table is as below:

```
CREATE TABLE IF NOT EXISTS PHONEBOOK (
    Id INTEGER PRIMARY KEY AUTOINCREMENT,
    Name TEXT NOT NULL,
    PhoneNumber TEXT UNIQUE NOT NULL);
```

Id is auto generated and incremented for each user input, Name is a text and PhoneNumber is unique and not null. This database is queried every time an API call is made. This will be discussed in the later section.

3.3 Logging Messages

Logging error messages are a key to understanding what happened with the application. In order to achieve this, a brisk logging mechanism has been implemented in this application.

```
logging.basicConfig(filename="phonebook.log", format='%(asctime)s %(message)s',
filemode='a', level=logging.INFO)
logger = logging.getLogger()
```

The logger logs error messages into *phonebook.log* file.

4 Application Working

A bulletin of how the application works is below:

- The application, built on Flask, adeptly manages requests at a predefined IP and port, providing a robust foundation for handling diverse HTTP methods.
- Flask application showcases versatility by seamlessly accommodating GET, POST, and PUT requests, ensuring flexibility in interaction with the API.
- Distinguishing API endpoints—'list,' 'add,' 'deleteByName,' and 'deleteByNumber'—efficiently trigger corresponding functions within the Flask application, tailoring responses to specific user actions.
- In the initial stages, the request data undergoes validation in the login() function, designed for validating user credentials.
- The absence of a username or password triggers a 400 status code, accompanied by a response message signaling, "Username/Password missing!"
- Within the login() function, the validation process involves extracting and assessing the username and password against predefined values.
- In case of a matching username, the function further checks the password, returning True upon success; otherwise, a "Invalid Credentials" message is issued with a status_code of 400.
- Once user credentials are successfully validated, the relevant values are extracted from the request.
- Absence of these values prompts the issuance of a 400 status code, accompanied by a message indicating the missing information.
- The extracted values like Name and PhoneNumber undergo validation using established regular expressions, paving the way for the execution of appropriate functions.

- The utilization of parameterized queries is a notable feature, facilitating secure and efficient table operations. Database connections are seamlessly managed through the *db_config.ini* file.
- Strong constraints govern operations, such as the impossibility of a delete action if data doesn't exist (status_code = 404) and the prevention of insertions if the phone number is already registered (status_code = 400).
- A log in the *phonebook.log* file captures the operations, providing a comprehensive record for debugging and analysis.
- The code stands out for its integration of user authentication, SQLite database management, parameterized queries, and Docker implementation—culminating in a comprehensive and robust system.
- User credentials are not required for the 'list' operation, ensuring accessibility and simplicity in retrieving all entries.

5 How to run?

The following steps will go in detail on how to run this application.

5.1 Setup

The application uses Docker to setup the application. Follow the following steps to setup the application. Please note that I am assuming that you have already installed Docker on your device and you are able to use Docker CLI.

- Open your terminal and navigate to the project directory where the Dockerfile is present.
- run

  ```
  $docker build -t secure_programming .
  ```

  Please note the period `.` after `secure_programming`. Once the build is complete, run this:

  ```
  $docker run -it -p 5002:5002 secure_programming
  ```

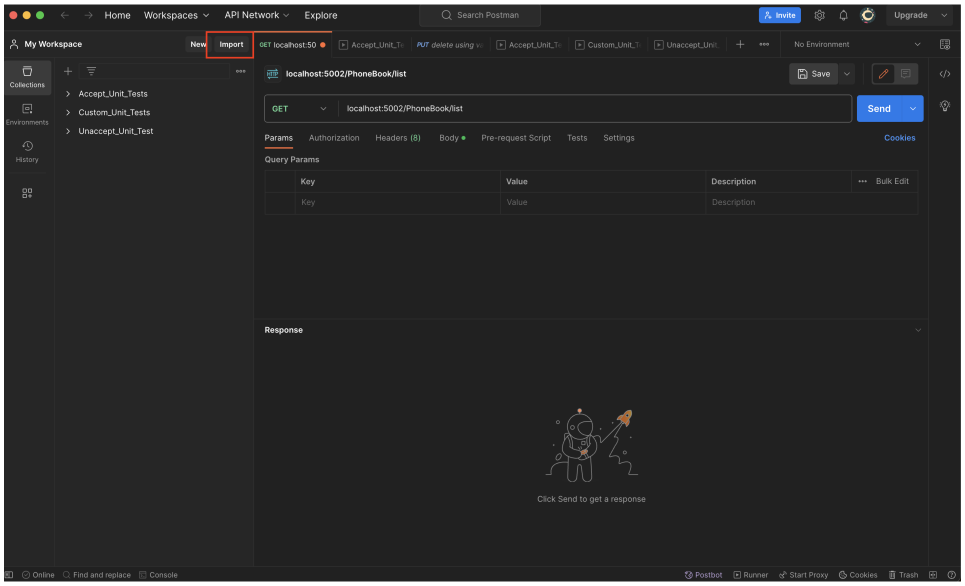
  This should get the application running on the terminal and the application must be active on the endpoint `http://localhost:5002/PhoneBook/`

5.2 Running Automated Tests

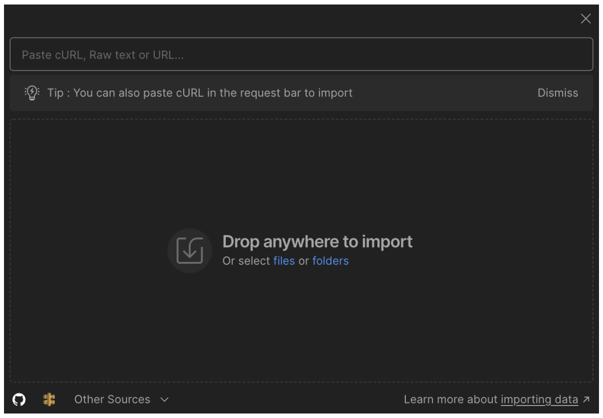
I have created three sets of automated testing suits that test the application in different ways. These test cases have been supplied with the application. It is recommended that you first run the automated tests before checking the functionalities in the application. This is so simply because if you manually enter some data that happen to be similar to the ones present in the testing suit, some of the test cases may fail because of redundant data. To avoid this, first run the automated tests and then manually test the application.

- Open Postman on your device. Make sure that you are logged into the Postman application as one of the following steps require you to be logged into it. If you login, it gives you much more functionalities like loading test suits etc.

- Click on the import button on the left hand side as shown in the image:

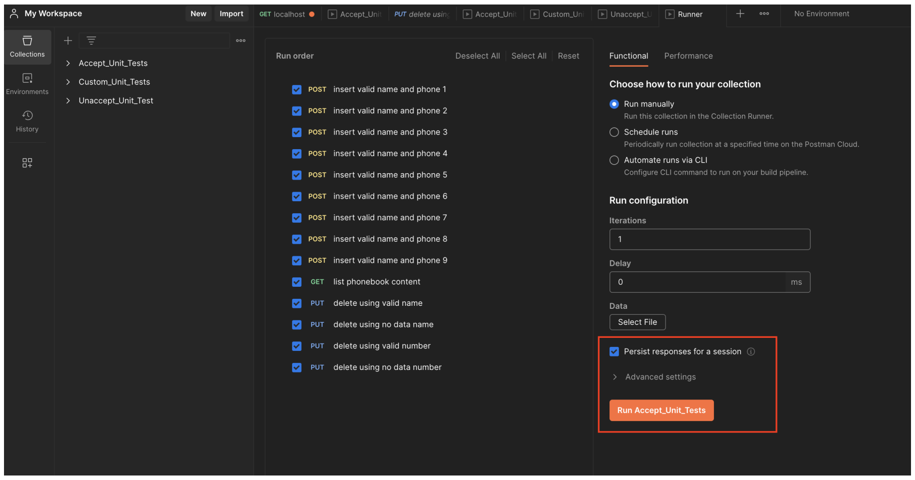


- In the window shown below, drag and drop the three testing suits provided in `./test_cases/` folder and then click on Import button.


Paste cURL, Raw text or URL...
Tip : You can also paste cURL in the request bar to import
Dismiss
Drop anywhere to import
Or select files or folders
Other Sources
Learn more about importing data

This should import all the automated test cases into the Postman.

- Now the three testing suits must be visible on the left side panel. For every test suit, when you hover over it, three dots must be visible. Click on the three dots and click 'Run collection' and the following windown must open.



POST insert valid name and phone 1
POST insert valid name and phone 2
POST insert valid name and phone 3
POST insert valid name and phone 4
POST insert valid name and phone 5
POST insert valid name and phone 6
POST insert valid name and phone 7
POST insert valid name and phone 8
POST insert valid name and phone 9
GET list phonebook content
PUT delete using valid name
PUT delete using no data name
PUT delete using valid number
PUT delete using no data number

Choose how to run your collection

[✓ Run manually] — Run this collection in the Collection Runner.
Schedule runs — Periodically run collection at a specified time on the Postman Cloud.
Automate runs via CLI — Configure CLI command to run on your build pipeline.

Run configuration

Iterations: 1
Delay: 0 ms
Data

✓ Persist responses for a session

Make sure that you check the 'persist responses for a session' and the click on 'Run' orange button. This should execute all the test cases in the suit and give you appropriate result. If you want, you can click on the individual test case and see what was sent and what was the response from the application. You can also re-send the request to see how the application behaves. It is worthy to note that the test cases that have passed may fail if you run them again. This is because of the duplicate entries that the testing suit has entered into the database.

5.3 Request Sample

A sample of the request is shown below. This is the body of the request that is used by the API. For simplicity, there is currently only one user with the given username and password.

```
{
"username": "testuser",
"password": "123@test",
"name" : "Schneier, Bruce",
"phonenumber" : "+1(703)998-4332"
}
```

You can use this payload on the following URLs and their corresponding methods. list doesn't use any payload. Please use Postman to test the application as you need.

```
/PhoneBook/list - PUT : Lists all data present in the database
/PhoneBook/add - POST : Adds a user to database
/PhoneBook/deleteByName - PUT : Deletes record by the name
/PhoneBook/deleteByNumber - PUT : Deletes record by phonenumber
```

6 Assumptions, Pros & Cons

6.1 Assumptions

- This application is not designed to be GUI based, it is an API security focused application and hence, is expected to be tested in that manner.
- The scope of input validation is limited to what is outlined in the project description as laid out by the instructor. It is not 'state-of-art' application and has to be assumed to have some vulnerabilities.
- Currently only one user has been configured and is assumed to have full operational rights on the application. No functionality to add or remove users has been implemented.
- There is no limit on number of requests and hence, no measures against DoS attacks have been configured.
- HTTP error codes have been implemented to the best of knowledge and is not expected to handle every scenario.
- I have taken help from the resources available on the internet to achieve this project.

6.2 Pros & Cons

Pros:

- The user authorization makes sure that the request is sent by an authorized user before accepting the data that it sends. This gives and extra layer of security.
- Parameterized queries are used in the application so that the data and code are separated after the data goes through a set of strong regular expressions for input validation.
- All actions are logged into the logging file so that there is a trace of what happened in the case of an attack.

Cons:

- The scope of input validation is limited. Even the regexes used are not universally protective.
- There is no protection from DoS attacks and this can strain the server resources in case of an attack.
- Adding more valid usernames and passwords is missing.