20/03/2023

ADAM LOGAN

horizontal line

# Placeholder image

**Appointment Hack The Box Report**

Table of Contents

[1.1 Executive Summary 2](#_heading=h.3dy6vkm)

[1.2 Testing Performed 2](#_heading=h.1t3h5sf)

[1.3 Vulnerabilities Detected 3](#_heading=h.4d34og8)

[1.4 Mitigation Techniques 3](#_heading=h.2s8eyo1)

[1.5 Actions Taken 4](#_heading=h.17dp8vu)

[1.6 Preventative Measures 4](#_heading=h.3rdcrjn)

[1.7 Conclusion 4](#_heading=h.26in1rg)

[1.8 Appendices 5](#_heading=h.2jxsxqh)

[1.8.1 Penetration Testing 5](#_heading=h.z337ya)

[1.8.1.1 Planning 5](#_heading=h.jxb97srv7f8j)

[1.8.1.2 Discovery 6](#_heading=h.3j2qqm3)

[1.8.1.2.1 nmap 6](#_heading=h.1y810tw)

[1.8.1.2.2 gobuster 7](#_heading=h.2xcytpi)

[1.8.1.2.3 Wappalyzer 9](#_heading=h.4i7ojhp)

[1.8.1.3 Attack 10](#_heading=h.1ci93xb)

[1.8.1.3.1 Default Credentials 10](#_heading=h.3whwml4)

[1.8.1.3.2 Checking error messages 11](#_heading=h.2bn6wsx)

[1.8.1.3.3 Using Sleep 11](#_heading=h.qsh70q)

[1.8.1.3.4 Gaining Access 13](#_heading=h.3as4poj)

[1.8.1.3.5 Gaining Administrator access 14](#_heading=h.1pxezwc)

[1.8.2 Code Rewrite 16](#_heading=h.ilnn9f6oczu8)

[1.8.2.1 Input Sanitisation 16](#_heading=h.hyx0my4gpumh)

[1.8.2.2 Parameterised Input 17](#_heading=h.72brqwmgxhjy)

[1.8.2.3 Stored Procedures 18](#_heading=h.m2gdb1u9gbto)

[1.8.2.4 Combined Techniques 18](#_heading=h.3fw9etbluhri)

[1.8.3 Glossary 19](#_heading=h.3om8yym9wbzk)

[1.9 Bibliography 20](#_heading=h.49x2ik5)

Appointment

## 1.1 Executive Summary

This paper has shown the application is vulnerable to a blind SQL injection (SQLi) [[4](#bookmark=id.8a7pjg6gkrqu)], by using the [NIST](#bookmark=id.xxvjtzkd9arn) penetration testing methodology [[9](#bookmark=id.gotje4thskrv)]. This paper demonstrates a malicious actor can gain administrator access through this vulnerability, which is caused by low coding standards alongside misconfiguration.

The impact on security of this is enormous, as a malicious actor will have unfettered access to the application. The integrity, one of the core principles of the CIA (Confidentiality, Integrity, and Availability) triad, has been violated by this vulnerability.

Therefore, the application is not safe to ship, although the cost to resolve this threat is low, as shown in the code rewrite, [[1.8.2](#_heading=h.ilnn9f6oczu8)] that has occurred. This cannot be said for the cost of prevention, as this paper recommends, a change in both the way projects are managed and how the application is developed. The recommendations are to integrate DevSecOps into projects and to carry out test-driven development (TDD) [[12](#bookmark=id.aalidjvfyyx7)].

## 1.2 Testing Performed

The [NIST](#bookmark=id.xxvjtzkd9arn) penetration test methodology was used to locate vulnerabilities, [[9](#bookmark=id.gotje4thskrv)] within the application. This consists of four stages: planning [[1.8.1.1](#_heading=h.jxb97srv7f8j)], discovery [[1.8.1.2](#_heading=h.3j2qqm3)], attack [[1.8.1.3](#_heading=h.1ci93xb)] and reporting. Within this section only, the discovery and attack stages will be detailed.

Within the discovery stage, it was revealed, using nmap, that an Apache HTTP server was running on the IP address [[1.8.1.2.1](#_heading=h.1y810tw)]. gobuster [[1.8.1.2.2](#_heading=h.2xcytpi)] was then used to examine the file structure of the application, where nothing of note was found. The IP is visited within a web browser where wappalyzer is used to further identify the technologies used [[1.8.1.2.3](#_heading=h.4i7ojhp)].

Within the attack stage, default login details [[1.8.1.3.1](#_heading=h.3whwml4)] were tested, which all failed. After this, ‘test statements’ [[1.8.1.3.2](#_heading=h.2bn6wsx)][[1.8.1.3.3](#_heading=h.qsh70q)] were used to check for a [SQLi](#bookmark=id.u0no42japqsh), in which a blind [SQLi](#bookmark=id.u0no42japqsh) was found. Then a simple [SQLi](#bookmark=id.u0no42japqsh) was performed, which succeeded in logging into the application [[1.8.1.3.4](#_heading=h.3as4poj)]. Another [SQLi](#bookmark=id.u0no42japqsh) was performed to test if administrator access could be gained, which was successful [[1.8.1.3.5](#_heading=h.1pxezwc)]

The results of the testing can be viewed in section [1.8.1](#_heading=h.z337ya).

## 1.3 Vulnerabilities Detected

The main vulnerability within this application is lack of input sanitisation, which leads to [SQLi](#bookmark=id.u0no42japqsh). The risk of this cannot be understated as a malicious actor can bypass the password mechanism and alter the database, which breaches the ‘integrity’ pillar of the [CIA](#bookmark=id.pxnls1t89to) triad. The severity of an [SQLi](#bookmark=id.u0no42japqsh) can be seen through the attack on Heartland Payment Systems [[4](#bookmark=id.8a7pjg6gkrqu)].

Malicious actors do not, necessarily, have to gain access to an administrator account to breach the integrity of the system as a ‘DROP table’ command can be injected. However, the malicious actor would need to have some knowledge of the database to do this.

Although a MariaDB database is used currently, if the application migrates to Microsoft SQL, OS commands can be executed through SQL statements [[3](#bookmark=id.b2a55qt7jmdw)][[13](#bookmark=id.e2u6r45q6tw8)], if the functionality to do so is enabled. As demonstrated here and in section [1.4](#_heading=h.2s8eyo1), configuration is a low-cost solution to many vulnerabilities.

As injection is a common exploit, being third in [OWASP](#bookmark=id.4cxu71xu224q) 2021 top 10 list [[6](#bookmark=id.3g572trsbbup)], there are many readily available exploits [[11](#bookmark=id.eh9iqql1q3sb)] which further highlights the risk.

## 1.4 Mitigation Techniques

A low-cost technique is to change the default configuration [[1](#bookmark=id.s8xo6vef94zw)]. By having a more inconspicuous administrator username, malicious actors are less likely to ‘guess’ the correct username. A brute force attack will negate this benefit, although protections can be put in place for this, such as timeouts.

In addition, the principle of least privilege should be applied, as if a malicious actor gains access to an account the damage they can do is limited [[1](#bookmark=id.s8xo6vef94zw)] [[7](#bookmark=id.en1mi0fkutyw)], unless privilege escalation occurs.

Another recommendation is to use Object-Relational Mapping (ORM), which protects against [SQLi](#bookmark=id.u0no42japqsh) as developers don’t write SQL statements directly [[4](#bookmark=id.8a7pjg6gkrqu)]. Although it is possible to override [ORM](#bookmark=id.3usr57frun3m) with custom SQL statements which negates this benefit.

A potentially high-cost technique is to use a Web Application Firewall (WAF) to filter through requests and reject potential [SQLis](#bookmark=id.u0no42japqsh) [[8](#bookmark=id.5lzuxl4bhakg)]. This would prevent the [SQLi](#bookmark=id.u0no42japqsh) used in [1.2](#_heading=h.1t3h5sf) but does not guarantee complete protection [[10](#bookmark=id.sp27icdhpfdt)]. There are both paid and open source [WAFs](#bookmark=id.vp4vc1q2ja0c) available and therefore the cost of this protection varies.

By implementing Multi-Factor Authentication (MFA) into the login system, a malicious actor will need to bypass more than just the password. This high-cost technique has its disadvantages as it is still possible to bypass the other authentication methods, such as with the SIM swap fraud [[2](#bookmark=id.cqrqqt5dykad)].

The recommendation of this paper is to incorporate all the above security techniques to achieve defence in depth [[4](#bookmark=id.8a7pjg6gkrqu)], this will reduce the threat as multiple security protocols will need to fail for an attack to succeed.

## 1.5 Actions Taken

A code rewrite [[1.8.2](#_heading=h.ilnn9f6oczu8)] was undertaken, which included adding input sanitisation, parameterised statements [[4](#bookmark=id.8a7pjg6gkrqu)]|[[15](#bookmark=id.wzdv0p6gu7fg)] and stored procedures. Input sanitisation was achieved by creating a blacklist of common characters used in [SQLis](#bookmark=id.u0no42japqsh) and by using character escaping functions [[14](#bookmark=id.4vgg1ncrdg19)]. A reduction in threat is achieved as the user input is not only limited in the control characters that are valid, but also the user input is treated as input by the database driver [[4](#bookmark=id.8a7pjg6gkrqu)].

These techniques should be used in conjunction [[1.8.2.4](#_heading=h.3fw9etbluhri)] according to the defence in depth [[4](#bookmark=id.8a7pjg6gkrqu)] strategy.

## 1.6 Preventative Measures

A prevention technique is to integrate security into the software development lifecycle (SDLC) by adopting the DevSecOps model [[12](#bookmark=id.aalidjvfyyx7)][[5](#bookmark=id.m4yj0ebbnvpq)]. This keeps security in the forefront of project managers minds, which reduces the likelihood of future vulnerabilities.

A [TDD](#bookmark=id.h7teqf23eqcs) model, used in conjunction with DevSecOps, will also reduce the threat level [[12](#bookmark=id.aalidjvfyyx7)], as code will be developed to pass the tests and therefore would fail when an [SQLi](#bookmark=id.u0no42japqsh) test is presented. The vulnerability would then be closed immediately, rather than being found later in the [SDLC](#bookmark=id.8x6nhrc05smo). A disadvantage with this approach is that development time is increased, resulting in this being a higher cost technique.

## 1.7 Conclusion

As detailed within this report: input sanitisation, parameterised statements and changes to the configuration file should all occur to achieve defence in depth [[4](#bookmark=id.8a7pjg6gkrqu)].

A weakness in the report is the inability to view the exact privileges that belong to the ‘admin’ account, and therefore the damage, if a malicious user gained access to this account, could not be accurately assessed. Another limitation of the report is that the recommendations made do not make the application immune to [SQLi](#bookmark=id.u0no42japqsh) but only more resistant.

To assist prevention of vulnerabilities it is recommended that the operational model should transition towards a DevSecOps approach along with the development model moving towards [TDD](#bookmark=id.h7teqf23eqcs).

## 1.8 Appendices

### 1.8.1 Penetration Testing

#### 1.8.1.1 Planning

The expectation of this testing is that a [SQLi](#bookmark=id.u0no42japqsh) vulnerability will be located, within the login page of the application. This will be the main focus of the testing, but general reconnaissance will be done on the whole application.

An attempt to discover the technologies used within the application will be made. Following this the file structure will be explored to check for hidden directories.

Depending on the information provided in the previous steps, the default configuration of the technologies will be researched for any potential vulnerabilities. As a [SQLi](#bookmark=id.u0no42japqsh) vulnerability is expected error messages will be examined to extract any relevant information. Failing this a ‘sleep’ command will be used to check for a successful injection before an attempt to login is made.

Then an actual attempt will be made to login, first with a non-administrator account. The final step is to gain administrator access, by attempting to log into an administrator account.

#### 1.8.1.2 Discovery

##### 1.8.1.2.1 nmap

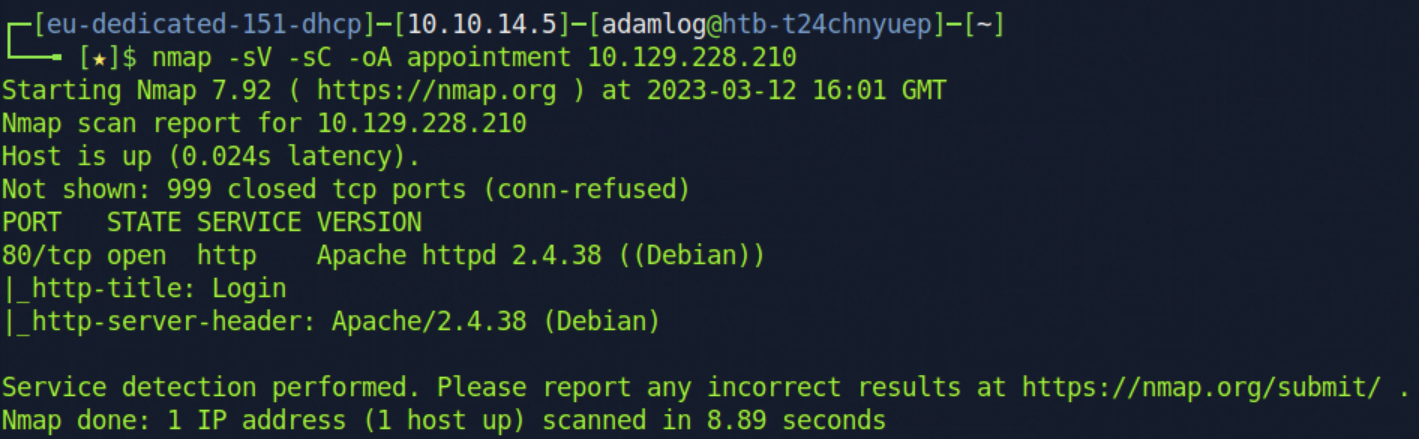


Figure 1.8.1.2.1.1: The results of the nmap scan in the terminal

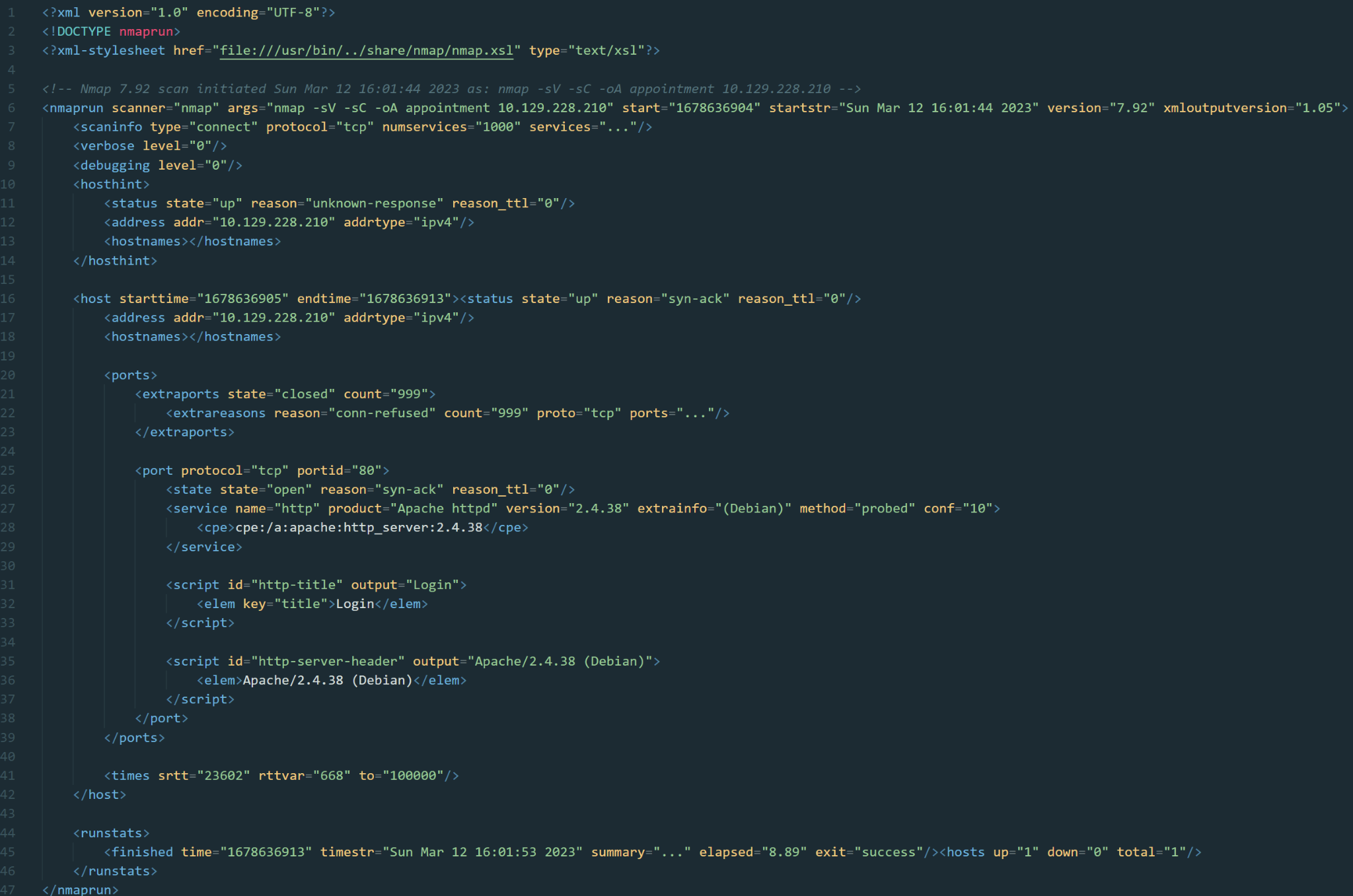


Figure 1.8.1.2.1.2: The results of the nmap scan as an XML file, note that the attributes services, ports and summary have been replaced with ‘...’, for the entire file to fit into the report

##### 1.8.1.2.2 gobuster

===============================================================

Gobuster v3.1.0

by OJ Reeves (@TheColonial) & Christian Mehlmauer (@firefart)

===============================================================

[+] Url: http://10.129.228.210

[+] Method: GET

[+] Threads: 10

[+] Wordlist: directory-list-2.3-medium.txt

[+] Negative Status codes: 404

[+] User Agent: gobuster/3.1.0

[+] Timeout: 10s

===============================================================

2023/03/16 15:35:27 Starting gobuster in directory enumeration mode

===============================================================

/images (Status: 301) [Size: 317] [--> http://10.129.228.210/images/]

/css (Status: 301) [Size: 314] [--> http://10.129.228.210/css/]

/js (Status: 301) [Size: 313] [--> http://10.129.228.210/js/]

/vendor (Status: 301) [Size: 317] [--> http://10.129.228.210/vendor/]

/fonts (Status: 301) [Size: 316] [--> http://10.129.228.210/fonts/]

/server-status (Status: 403) [Size: 279]

===============================================================

2023/03/16 15:40:02 Finished

===============================================================

Figure 1.8.1.2.2.1: The output of gobuster using the wordlist directory-list-2.3-medium.txt

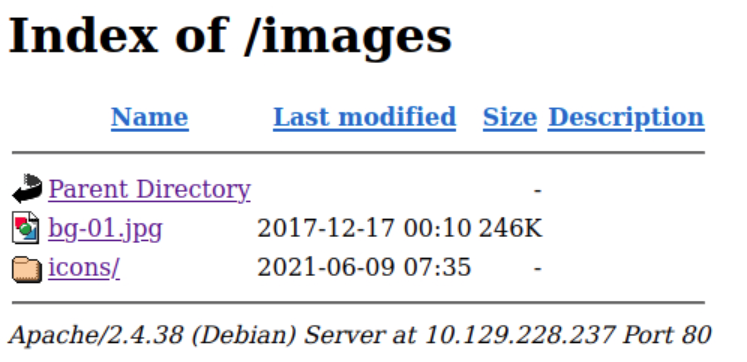


Figure 1.8.1.2.2.2: When /images is visited within the browser

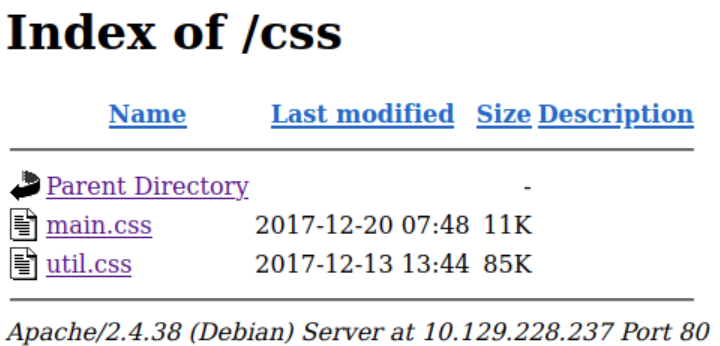


Figure 1.8.1.2.2.3: When /css is visited within the browser



Figure 1.8.1.2.2.4: When /js is visited within the browser



Figure 1.8.1.2.2.5: When /vendor is visited within the browser

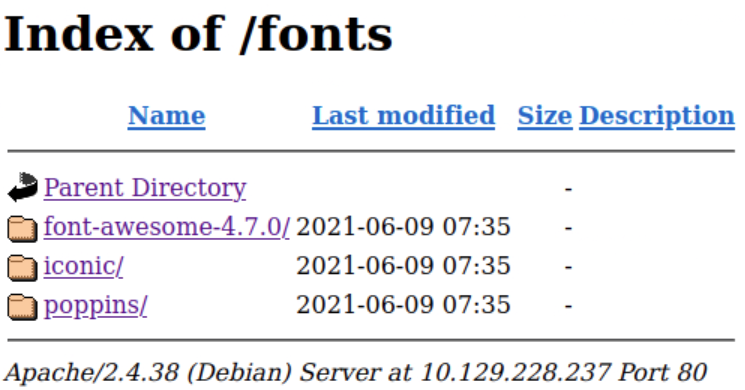


Figure 1.8.1.2.2.6: When /fonts is visited within the browser

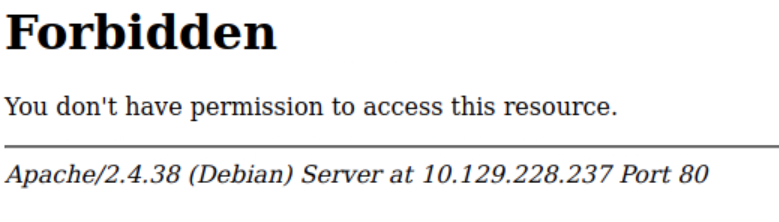


Figure 1.8.1.2.2.7: When /server-status is visited within the browser

##### 1.8.1.2.3 Wappalyzer

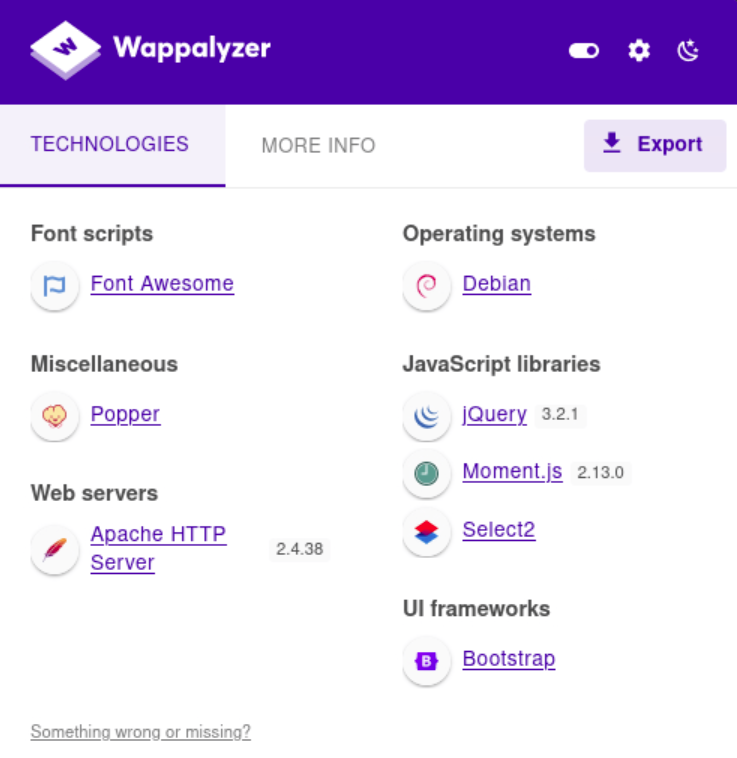


Figure: 1.8.1.2.3.1: A screenshot of wappalyzer used when the IP address is visited on a browser

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **URL** | **Font scripts** | **Miscellaneous** | **Web servers** | **Operating systems** | **JavaScript libraries** | **UI frameworks** |
| http://10.129.228.210 | Font Awesome | Popper | Apache HTTP Server | Debian | jQuery | Bootstrap |
|  |  |  |  |  | Moment.js |  |
|  |  |  |  |  | Select2 |  |

Figure 1.8.1.2.3.2: The table that is produced by wappalyzer when it is exported

#### 1.8.1.3 Attack

##### 1.8.1.3.1 Default Credentials

——————————————————————————————————————————————————————————————————————————————

admin:password

admin:admin

root:password

root:root

administrator:password

administrator:administrator

——————————————————————————————————————————————————————————————————————————————Figure 1.8.1.3.1.1: The common administrator login details that were attempted

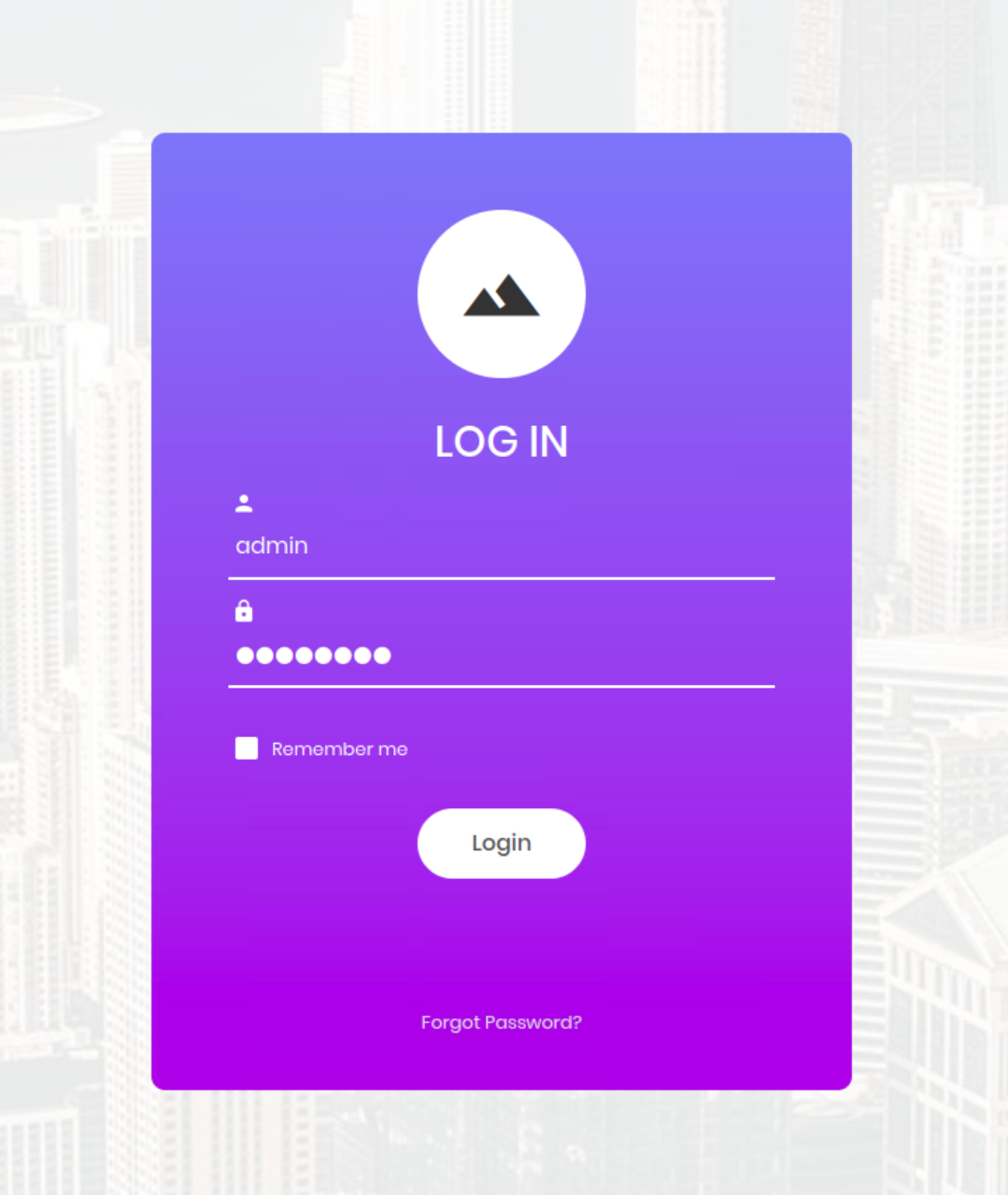


Figure 1.8.1.3.1.2: The attempt to login using admin:password

##### 1.8.1.3.2 Checking error messages

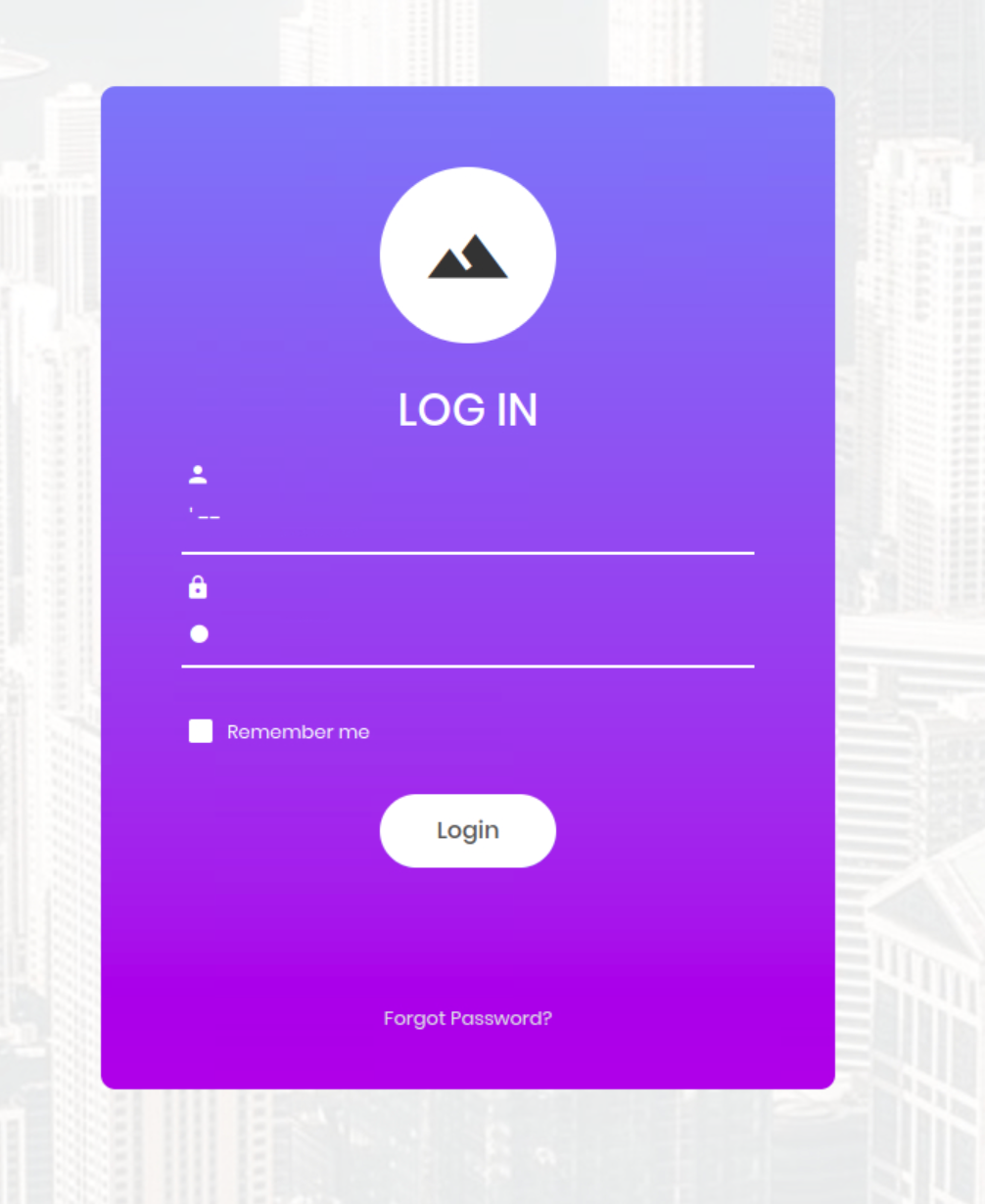


Figure 1.8.1.3.2.1: Attempts to inject a MariaDB comment character

##### 1.8.1.3.3 Using Sleep

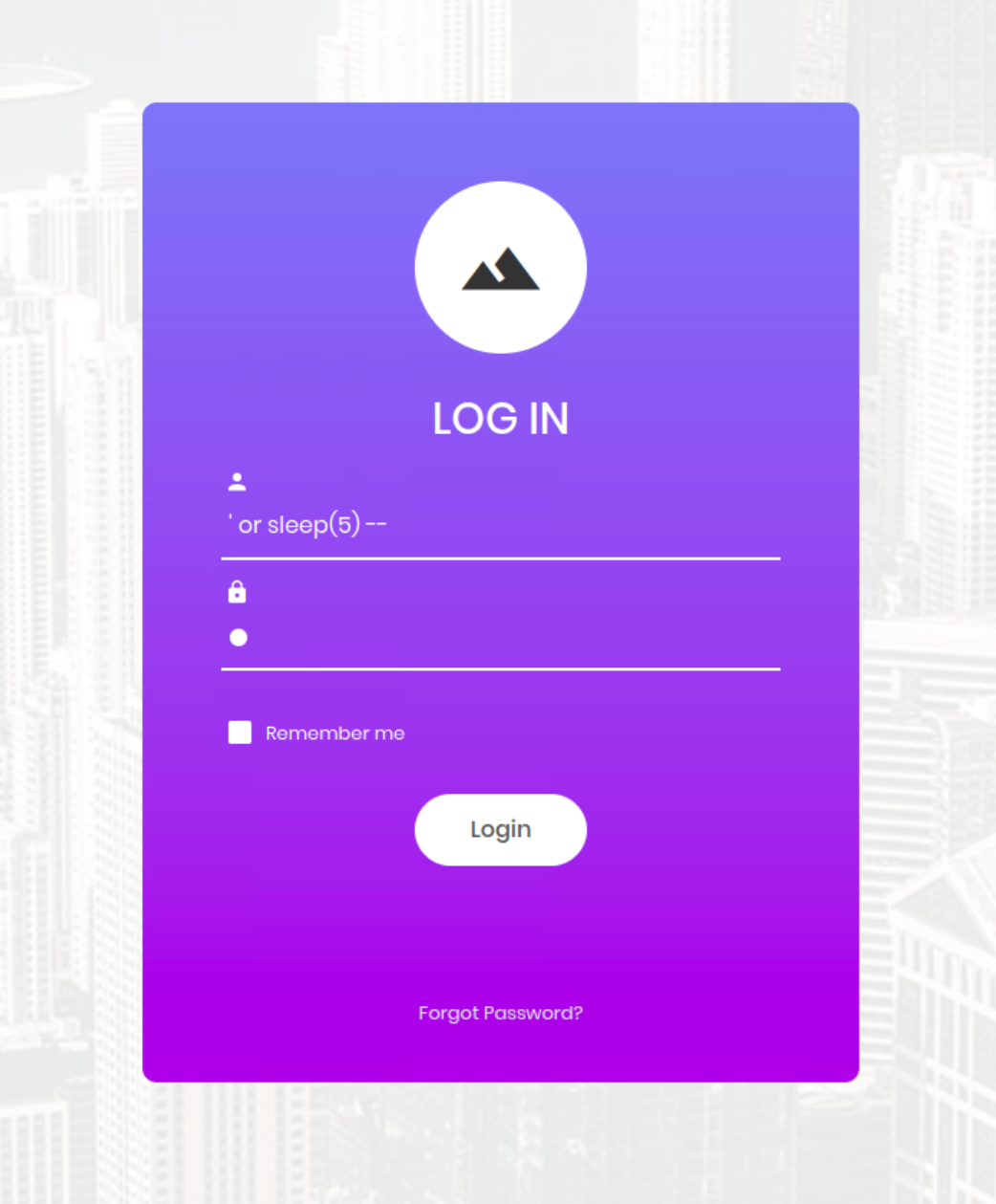


Figure 1.8.1.3.3.1: An attempt to get the application to ‘sleep’ for 5

seconds but unsuccessful as the wrong comment character is used



Figure 1.8.1.3.3.2: A successful attempt to get the application to ‘sleep’ for 5 seconds

##### 1.8.1.3.4 Gaining Access

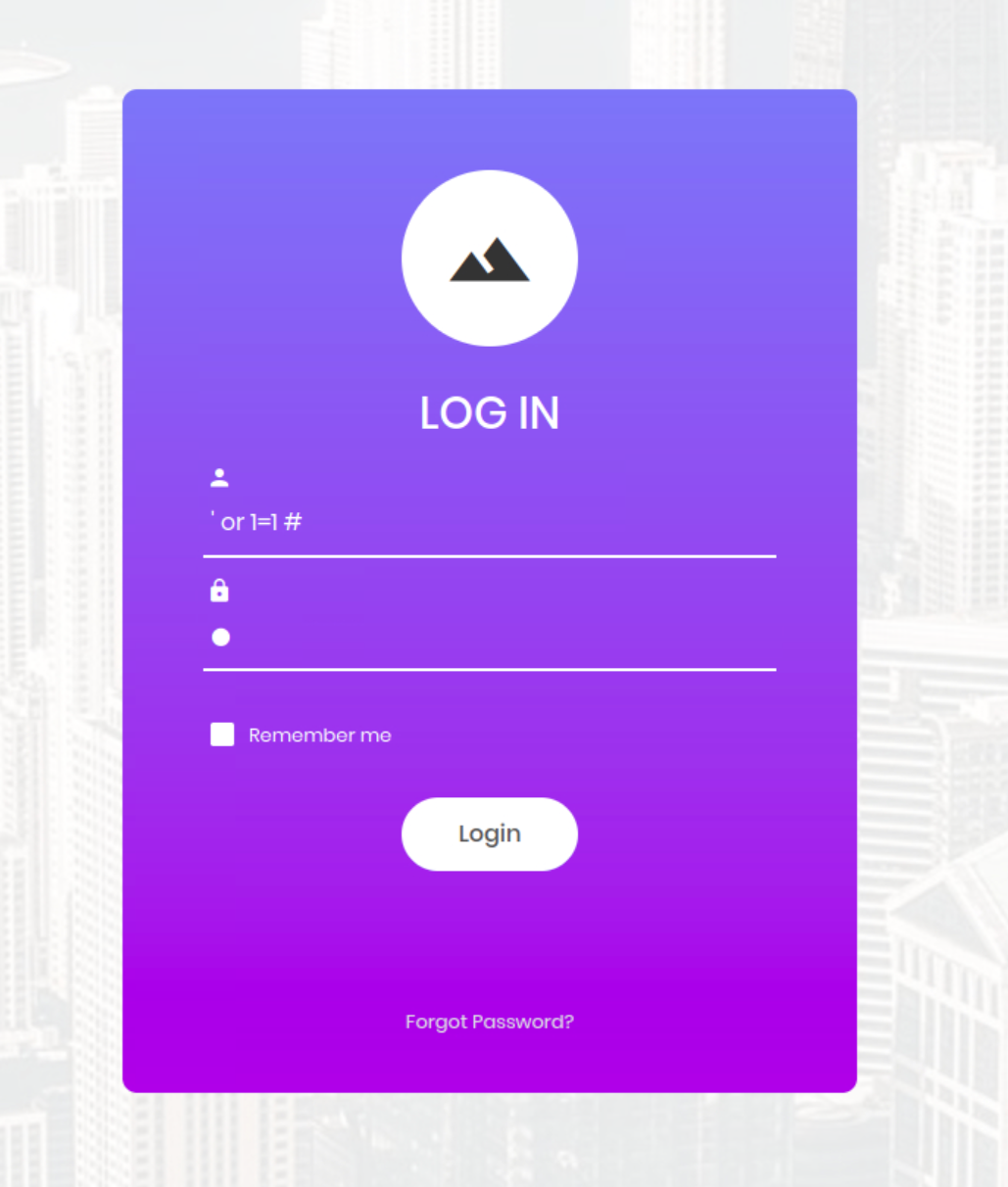


Figure 1.8.1.3.4.1: The statement used to gain access to the application

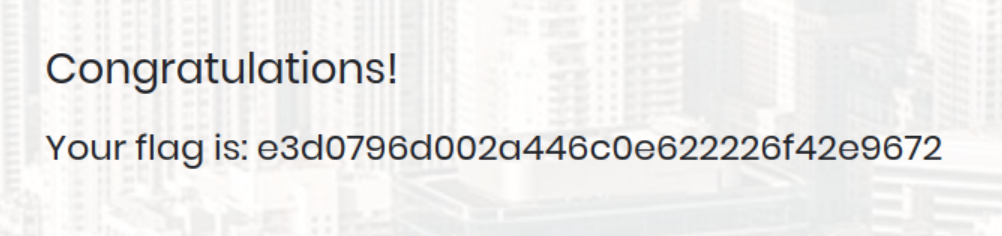


Figure 1.8.1.3.4.2: The flag, demonstrating the access to the application was granted

##### 1.8.1.3.5 Gaining Administrator access

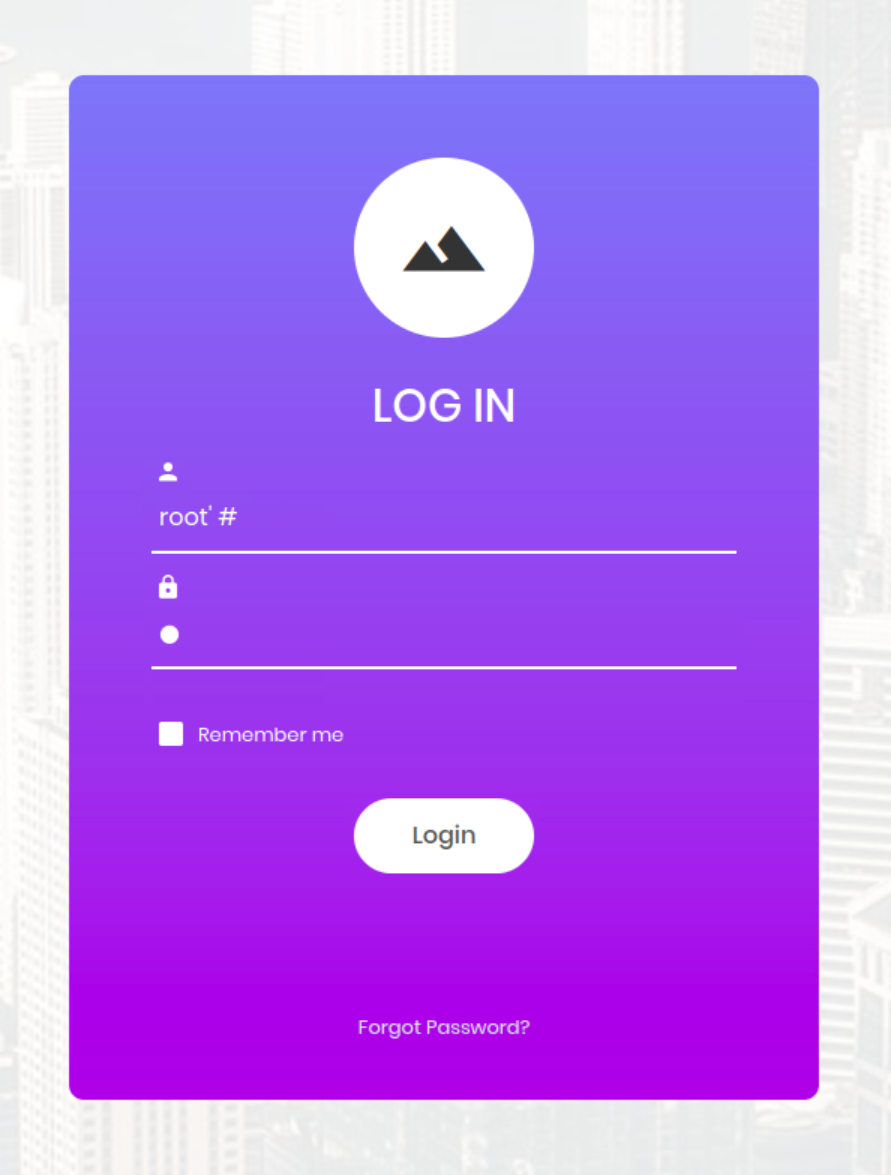


Figure 1.8.1.3.5.1: The failed attempt to gain administrator access using the ‘root’ username, implying

the administrator username is not ‘root’

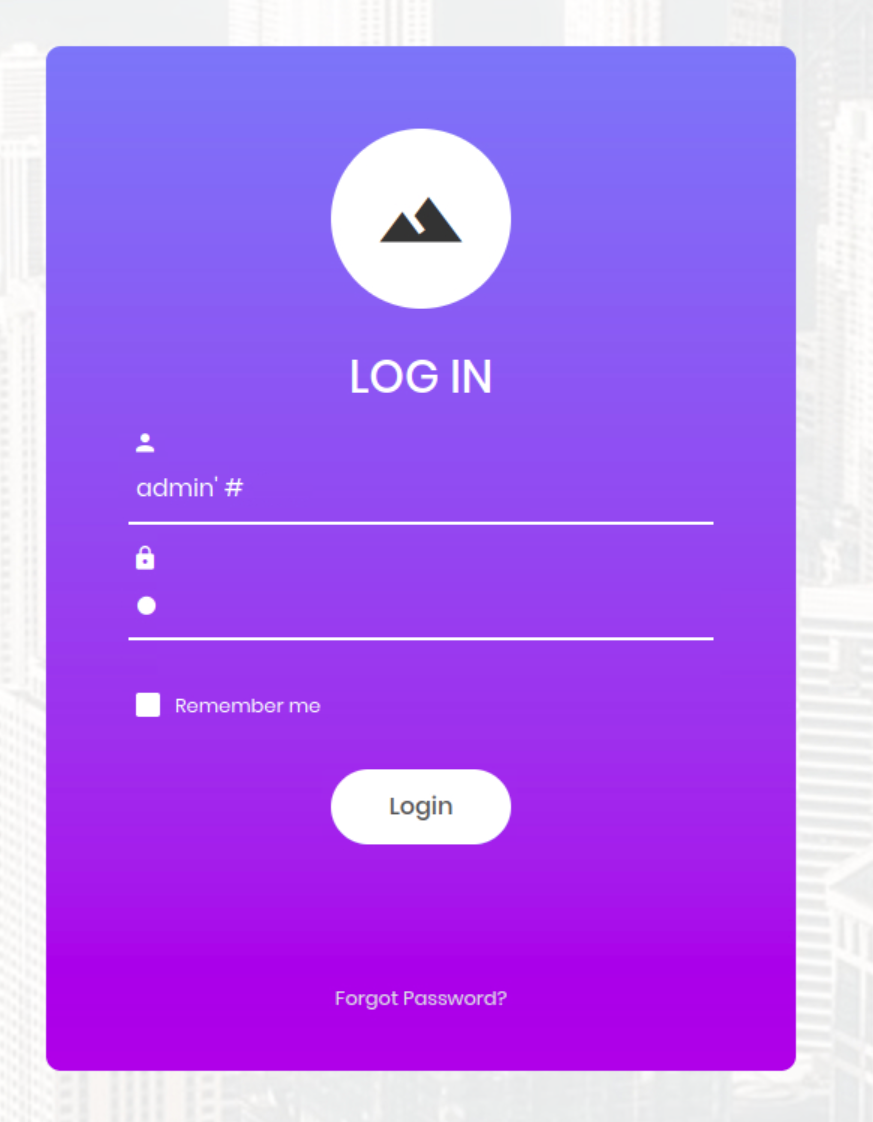


Figure 1.8.1.3.5.2: The successful attempt to gain administrator access using the ‘admin’

username, implying the administrator username is admin’

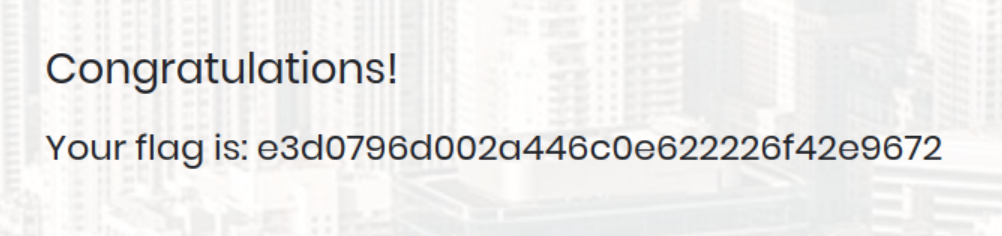


Figure 1.8.1.3.5.3: The flag, demonstrating the access to the application was granted

### 1.8.2 Code Rewrite

The code rewrite examples have all been written in PHP. This is due to the fact that the application is using an Apache server, on Debian, alongside MariaDB so it is likely that a LAMP (Linux, Apache, MariaDB and PHP) architecture is being used.

It is assumed that before the code snippet there is already a connection to the database.

The result itself is stored within the variable ‘$result’.

#### 1.8.2.1 Input Sanitisation

——————————————————————————————————————————————————————————————————————————————

$username = $conn->real\_escape\_string($\_GET["username"]);

$password = $conn->real\_escape\_string($\_GET["password"]);

$select = "SELECT \* FROM users WHERE username='" . $username . "' AND

password='" . $password . "';";

$result = $conn->query($select);

——————————————————————————————————————————————————————————————————————————————

Figure 1.8.2.1.1: A code listing for using escaping functions to fix the current SQLi vulnerability

——————————————————————————————————————————————————————————————————————————————

$username = $\_GET["username"];

$password = $\_GET["password"];

// an example blacklist, this is not exhaustive and is case insensitive

$blacklist = array("#", "--", "or 1=1", "drop", "sleep", "union",

"insert");

$regex = "/";

for ($i = 0; $i < sizeof($blacklist)-1; $i++) {

$regex .= $blacklist[$i] . "|";

}

$regex .= $blacklist[sizeof($blacklist)-1] . "/i";

if (preg\_match($regex, $username)) {

echo "0 results";

return 0;

}

$select = "SELECT \* FROM users WHERE username='" . $username . "' AND

password='" . $password . "';";

$result = $conn->query($select);

——————————————————————————————————————————————————————————————————————————————

Figure 1.8.2.1.2: A code listing for using a blacklist to fix the current SQLi vulnerability

——————————————————————————————————————————————————————————————————————————————

$username = $conn->real\_escape\_string($\_GET["username"]);

$password = $conn->real\_escape\_string($\_GET["password"]);

// an example blacklist, this is not exhaustive and is case insensitive

$blacklist = array("#", "--", "or 1=1", "drop", "sleep", "union",

"insert");

$regex = "/";

for ($i = 0; $i < sizeof($blacklist)-1; $i++) {

$regex .= $blacklist[$i] . "|";

}

$regex .= $blacklist[sizeof($blacklist)-1] . "/i";

if (preg\_match($regex, $username)) {

echo "0 results";

return 0;

}

$select = "SELECT \* FROM users WHERE username='" . $username . "' AND

password='" . $password . "';";

$result = $conn->query($select);

——————————————————————————————————————————————————————————————————————————————

Figure 1.8.2.1.3: A code listing for using both escaping functions and a blacklist to fix the current SQLi vulnerability

#### 1.8.2.2 Parameterised Input

——————————————————————————————————————————————————————————————————————————————

$username = $\_GET["username"];

$password = $\_GET["password"];

$sql = $conn->prepare("SELECT \* FROM users WHERE username=? AND

password=?;");

$query = $sql->bind\_param('ss', $username, $password);

$sql->execute();

$result = $sql->get\_result();

——————————————————————————————————————————————————————————————————————————————

Figure 1.8.2.2.1: A code listing for using parameterised inputs to fix the current SQLi vulnerability

#### 1.8.2.3 Stored Procedures

——————————————————————————————————————————————————————————————————————————————

$username = $\_GET["username"];

$password = $\_GET["password"];

$conn->query("DROP PROCEDURE IF EXISTS getUser");

$procedure = "CREATE PROCEDURE getUser(IN username\_in VARCHAR(100), IN

password\_in VARCHAR(100)) SELECT \* FROM users WHERE username=password\_in

AND password=password\_in";

$conn->query($procedure);

$sql = "CALL getUser('" . $username . "', '" . $password . "')";

$result = $conn->query($sql);

——————————————————————————————————————————————————————————————————————————————

Figure 1.8.2.3.1: A code listing for using stored procedures to fix the current SQLi vulnerability

#### 1.8.2.4 Combined Techniques

——————————————————————————————————————————————————————————————————————————————

//escaping user input [[1.8.2.1.1](#bookmark=kix.rv0m5r6k791i)]

$username = $conn->real\_escape\_string($\_GET["username"]);

$password = $conn->real\_escape\_string($\_GET["password"]);

// blacklist [[1.8.2.1.2](#bookmark=id.si036v3e0ar)]

$blacklist = array("#", "--", "or 1=1", "drop", "sleep", "union",

"insert");

$regex = "/";

for ($i = 0; $i < sizeof($blacklist)-1; $i++) {

$regex .= $blacklist[$i] . "|";

}

$regex .= $blacklist[sizeof($blacklist)-1] . "/i";

if (preg\_match($regex, $username)) {

echo "0 results";

return 0;

}

// stored procedure [[1.8.2.3](#_heading=h.m2gdb1u9gbto)]

$conn->query("DROP PROCEDURE IF EXISTS getUser");

$procedure = "CREATE PROCEDURE getUser(IN username\_in VARCHAR(100), IN

password\_in VARCHAR(100)) SELECT \* FROM users WHERE username=password\_in

AND password=password\_in";

$conn->query($procedure);

// parameterised input [[1.8.2.2](#_heading=h.72brqwmgxhjy)]

$sql = $conn->prepare("CALL getUser(?, ?)");

$query = $sql->bind\_param('ss', $username, $password);

$sql->execute();

$result = $sql->get\_result();

——————————————————————————————————————————————————————————————————————————————

Figure 1.8.2.4.1: A code listing for the combined techniques to fix the current SQLi vulnerability

### 1.8.3 Glossary

* ***CIA*** - Confidentiality, Integrity, and Availability
* ***LAMP*** - Linux, Apache, MariaDB, and PHP
* ***MFA*** - Multi-Factor Authentication
* ***NIST*** - National Institute of Standards and Technology
* ***ORM*** - Object-Relational Mapping
* ***OWASP*** - Open Worldwide Application Security Project
* ***SDLC*** - Software Development Lifecycle
* ***SQLi*** - SQL Injection
* ***TDD*** - Test-Driven Development
* ***WAF*** - Web Application Firewall

## 1.9 Bibliography

1. Andress Jason. “Operating System Hardening.” Foundations of Information Security: A Straightforward Introduction, No Starch Press, 2019, pp. 146 - 150.
2. Cunningham, Chase. “Kill the password, limit the pain.” Cyber Warfare - Truth, Tactics, and Strategies: Strategic Concepts and Truths to Help You and Your Organization Survive on the Battleground of Cyber Warfare, Packt Publishing, 2020, pp. 244 - 251.
3. Forshaw, James . “SQL Injection.” Attacking Network Protocols: A Hacker's Guide to Capture, Analysis, and Exploitation, No Starch Press, 2017, pp. 228 - 229.
4. McDonald, Malcolm. “SQL Injection.” Web Security for Developers: Real Threats, Practical Defense, No Starch Press, 2020, pp. 50 - 56.
5. NIST. “DevSecOps | CSRC.” NIST Computer Security Resource Center, 21 October 2020, <https://csrc.nist.gov/projects/devsecops>. Accessed 13 March 2023.
6. OWASP. “A03 Injection.” A03 Injection - OWASP Top 10, 2021, <https://owasp.org/Top10/A03_2021-Injection/>. Accessed 10 March 2023.
7. Pollack, Ed. “Principle of Least Privilege & Login Security.” SQL Injection: Detection and prevention, SQLShack, 30 August 2019, <https://www.sqlshack.com/sql-injection-detection-and-prevention/>. Accessed 15 March 2023.
8. Positive Technologies. “The Pentest Framework Phases.” How to Prevent SQL Injection: Attacks and Defense Techniques - Tutorial and Best Practices, 2 August 2019, <https://www.ptsecurity.com/ww-en/analytics/knowledge-base/how-to-prevent-sql-injection-attacks/>. Accessed 15 March 2023.
9. RSI Security. “What is the NIST Penetration Testing Framework?” NIST Penetration Testing Framework, 24 August 2020, <https://blog.rsisecurity.com/what-is-the-nist-penetration-testing-framework/>. Accessed 13 March 2023.
10. SC Media. “How WAF protections can be bypassed.” How the latest SQL injection attacks threaten web application firewalls, 6 February 2023, <https://www.scmagazine.com/resource/data-security/how-the-latest-sql-injection-attacks-threaten-web-application-firewalls>. Accessed 15 March 2023.
11. Steele, Tom. “Building the SQL Injection Fuzzer.” Black Hat Go: Go Programming For Hackers and Pentesters, No Starch Press, 2020, pp. 193 - 196.
12. Tan, Shamane.“Secure coding and secure software development.” Building a Cyber Resilient Business: A Cyber Handbook for Executives and Boards, et al., Packt Publishing, Limited, 2022, pp. 137 - 139.
13. Weidman, Georgia. “Using SQLMap.” Penetration Testing: A Hands-On Introduction to Hacking, No Starch Press, 2014, pp. 321- 322.
14. W3Schools. “PHP mysqli real\_escape\_string() Function.” PHP Tutorial, <https://www.w3schools.com/php/func_mysqli_real_escape_string.asp>. Accessed 17 March 2023.
15. Yaworski, Peter. “Countermeasures Against SQLi.” Real-World Bug Hunting: A Field Guide to Web Hacking, No Starch Press, 2019, pp. 83 - 84.