SQL Injection Attack & Prevention System in PHP

ANKIT SRIVASTAVA (212IS003)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA, SURATHKAL, INDIA

Abstract. SQL injection attacks constitute a serious security risk to Web applications because they offer attackers free access to the databases supporting the apps and the highly sensitive information stored in such databases. Despite the fact that researchers and practitioners have presented many techniques to address the SQL injection problem, existing approaches either fail to address the entire breadth of the problem or have constraints that preclude their use and acceptance. In this article, we will look at PHP strategies and other ways for preventing SQL from the injection, methods for detecting SQL assaults, forms of SQL injection, causes of SQL injection via receiving and posting, and SQL vulnerability protection technologies.

Keywords: SQL Injection Attacks, PHP, Database Security, SQL Injection Prevention

1. Introduction

In today's technology-driven society, website apps play an important role in daily living. Websites are used for a variety of activities, including online shopping, banking, and talking with friends. Databases are frequently used on the backend of websites to store user data. Because such files include sensitive information such as passwords, credit card numbers, and social security numbers, hostile hackers frequently target them. One of the major flaws that ignite the hackers to compromise the database information is SQL injection. It has been identified as one of the most severe risks to Web applications. SQL injection vulnerabilities in web applications may allow an attacker to obtain total access to its supporting databases. Since these databases usually contain sensitive client or user information, the associated security violations can include identity theft, confidential information loss, and fraud. In rare situations, a SQL injection vulnerability can be used by attackers to seize the power of and harm the system that serves the Web application.

1.1 Problem Statement

SQL injections lead to the theft of critical information. Furthermore, it has the potential to be disastrous for any business, government, or institution. Such occurrences can jeopardize the company's operations and image, as well as result in large fines imposed by data protection regulations. In order to prevent such kind of highly devastating attack, we have discussed some crucial prevention and detection methodology in the upcoming section.

1.2 Motivation

SQL injection is a form of insertion attack that may be defined as a harmful tactic that attacks the website's SQL-based application software by injecting malicious SQL statements or exploiting faulty

input. It does not need high-level implementation, resulting in perhaps one of the deadliest assaults. SQL injection attacks are constantly at the top of the attackers' priority list. SQL injection attacks accounted for nearly two-thirds (65.1 percent) of all attacks against software applications between 2017 and 2019. Looking at the above records, the prevention of such deadly activity should be the top priority for us, thus we have tried to provide some way to get rid of it.

1.3 Scope

SQL Injection is a methodology for trying to persuade an application to execute SQL statements that were not anticipated. The attacker's goal is to get around the security screening and get illegal access using the results of the modified SQL query. This type of attack is devastating to the majority of online applications. Such dangers can be avoided with cautious planning and execution. Now that we are aware of such flaws, we are implementing the appropriate validation tests to avoid any form of SQL injection. Once we will be able to prevent such attacks, most of the web applications would be less prone to attack, at least from their database respective. There can be more such methods to prevent such attacks, so we can dive deep into this to find the best-suited and strong protection wall against all such injection attacks.

1.4 Objectives

This proposal's goal is rather obvious. To begin, we discovered a common cause and vulnerability in the system, such as insufficient validation of user input. To address this issue, we suggested a set of coding principles that encourage defensive coding approaches such as encoding user input and validation. A thorough and systematic implementation of these strategies is an excellent solution for preventing SQL injection issues. However, in practice, the execution of such strategies is human-based and hence prone to mistakes. Furthermore, repairing older codebases that may include SQL injection vulnerabilities may be a time-consuming and labour-intensive operation.

1.5 Organization of the Report

The remainder of this work is structured as follows: Section 2 provides a detailed literature review of prior work on this topic. Section 3 identifies and provides the most significant aspect of this work, which includes several ways of dealing with SQL injection attacks. Section 4 outlines all of the experimental results, which are then followed by the Conclusion and Reference.

2. Related Work

According to an assessment of countless hacking incidents, as operating security systems are enhanced and security protection software solutions become more widely available, security breaches directly triggered by operating system vulnerabilities reduce every year, while WEB application system vulnerabilities increase in usage. PHP has become the dominant language for constructing all types of portal websites and Web application programs due to its simplicity and excellent development performance.

In 2009, 15th IEEE Pacific Rim International Symposium on Dependable Computing paper, Nuno Antunes and Marco Vieira from the University of Coimbra proposed a comparison between two major techniques Penetration Testing and Static Code Analysis for SQL Injection Detection. In order to understand the strengths and limitations of these tactics, they used a variety of commercial and open-source tools to uncover vulnerabilities in a collection of susceptible services. Static code analysers, according to the research, reveal more SQL Injection vulnerabilities than penetration testing methodologies. Another significant discovery is that tools that utilize the same detection method frequently detect distinct vulnerabilities. Finally, many tools have insufficient coverage and have a large rate of false positives, making them inappropriate for programmers.

At the IEEE international symposium on secure software engineering,2006, William G.J. Halfond, Jeremy Viegas, and Alessandro Orso from the College of Computing, Georgia Institute of Technology proposed an in-depth analysis of the many forms of SQL injection attacks known to date. Their paper contain descriptions and illustrations of how each form of attack may be carried out. They also provided an assessment of existing SQL injection detection and prevention methodologies, as well as the benefits and drawbacks of each strategy in dealing with a wide spectrum of SQL injection threats.

At International Journal on Computer Science and Engineering (IJCSE), Nikita Patel, Fahim Mohammed and Santosh Soni proposed a paper titled "SQL Injection Attacks: Techniques and Protection Mechanisms". This document presents challenges relating to information leaking via SQL injection attacks, as well as protection methods.

During 3rd International Conference on Computer Science and Information Technology in 2010, Lambert Ntagwabira, from Central South University (CHINA) P.O.BOX 410083, Department of Information Science and Engineering, China and Song Lin Kang from Central South University (CHINA) P.O.BOX 410083, Department of Information Science and Engineering, China proposed a paper titled "Use of Query tokenization to detect and prevent SQL injection attacks". Their study objective was to develop a mechanism for identifying and preventing SQL injection attacks by analysing whether user inputs impact the intended output of the query. They demonstrated a technique for detecting SQL injection attacks that makes use of query tokenization through the QueryParser function. In most cases, when performing SQL injection, the attacker should use a space, single quotes, or double dashes in his input. Tokenization is accomplished by detecting a space, single quotation mark, or double dash, and all strings before each symbol are combined to form a token. Following the creation of tokens, they are all combined to form an array, with each token being a member of the array.

In 2nd International Conference on Computer Science and Application Engineering, October 2018 Article No.: 187, Haiyan Zhang, Agricultural informatics, Hebei North University, Zhangjiakou, China

and Xiao Zhang, Medical informatics, Hebei North University, Zhangjiakou, China proposed a paper titled “SQL Injection Attack Principles and Preventive Techniques for PHP Site”. Their study examines the motivations for SQL injection and does comprehensive research on common SQL injection attack techniques, using PHP as an example. This article presents SQL injection detection tools and how to avoid SQL injection vulnerabilities while developing WEB software code based on real penetration testing experience. This article provides in-depth technical assistance for SQL injection testing as well as a solid assurance for WEB information system SQL injection protection.

In 2014, Navdeep Kaur, Master’s Degree, M. Tech. in Software Systems, Guru Nanak Dev University, Amritsar and Parminder Kaur, Assistant Professor, Department of Computer Science \& Engineering, Guru Nanak Dev University, Amritsar published an article on “SQL Injection – Anatomy and Risk Mitigation “. The paper goes through SQL Injection, SQL Injection Anatomy, SQL Injection Mitigation, and GreenSQL in depth.

3. Proposed Approach

SQL Injection attack can be performed on any login page of a website to get access to the user’s account or to fetch all the data of the database. There can be many scripts designed to be implemented as a SQL Injection. There can be majorly two cases when an attacker is well versed in the username but not with the password and want to get access to a particular user. In this case, the attacker types the username and inserts a script in place of a password to successfully login into the user’s account and perform whatever desired to do. The second case is when an attacker is not familiar with a specific user’s credentials but he’s willing to fetch all user’s information from the database. In this case, the attacker put a similar script in the username and password column as well. When the malicious script is injected into a web application from an outside source, for example, an input field supplied by the web application to receive input from the end-user, this is referred to as code injection. This attack takes advantage of a lack of

reliable input/output data validation. The malicious code that was inserted runs as part of the program. A successful code injection attack may result in asset damage or undesired activity.

Different Types of SQL Injection Attack:

TYPE 1: The attacker is willing to gain access to a specific user's account.

Username: *Correct_UserName' OR '1=1*
Password: *(No Input)*

Query:

SELECT * FROM `users` WHERE username=' *Correct_UserName' OR '1=1* ' and password='';

In the above query, the password script is designed such that the password attribute returns a Boolean result based on Boolean expression such as password = ' ' or 1=1 which is always true as 1=1 returns a true value. With this method, an attacker is able to successfully login into the user's account.

TYPE 2: The attacker is willing to gain access to all user's accounts.

Username: *' or '1'='1*
Password: *' or '1'='1*

Query:

SELECT * FROM `users` WHERE username = ' *' or '1'='1* ' and password =
‘ *‘ or '1'='1* ‘ ;

In the above query, both the username and password script are designed such a way that both are equal and return a Boolean result based on Boolean expressions such as username = ‘ ‘ or 1=1 AND password = ‘ ‘ or 1=1 which is always true as 1=1 returns a true value. The WHERE clause returns true for every row, and as a result, it retrieves all database records, granting the attacker access to the application. With this method, an attacker is able to successfully fetch all the information from the user’s database.

Different Measures for SQL Injection Prevention:

Simple adjustments in server-side and client-side code can safeguard against SQL Injection attacks. Developers must be familiar with different forms of attacks and take precautions against them. Sanitization is required for all dynamic SQL statements. A single unprotected query can be devastating to the application, data, or database server.

1. Avoid Textbox on Login Page:

There should be a predefined choice for the user to enter the data. For Example, if the user needs to input only numeric values in the username, design a UI/UX such that the user can select numeric values instead of typing in a textbox.

2. Validate the User Data:

The validation procedure verifies if the type of input given by the user is permitted. Data validation ensures that the type, length, format, etc -

are correct. Only valid values will be handled. It aids in the suppression of any commands entered into the input string.

3. Use Escaping String:

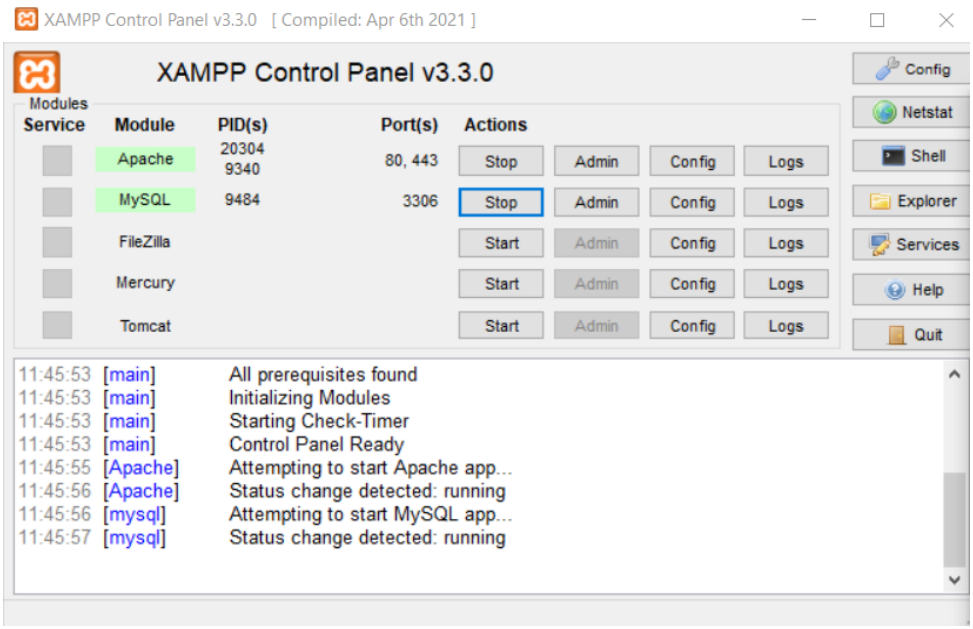
For developers, using character-escaping functions is a recommended practice. It is used anytime data from the client needs to be sent to a database. It aids in ensuring that the DBMS does not mix it up with the SQL statement given by the developer.

4. Experimental Results and Analysis

In order to perform the SQL Injection Demo Attack, there is some pre-requisite software & language that needs to be familiar/installed in the system. Below is the mentioned software and steps to start the activity.

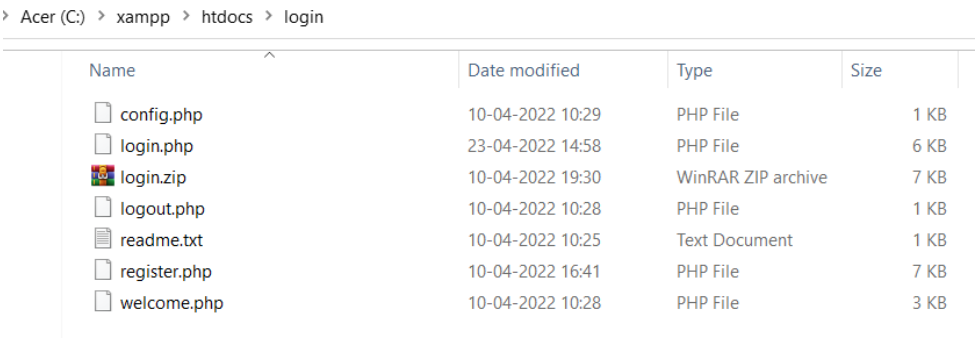
1. XAMPP Server Installation

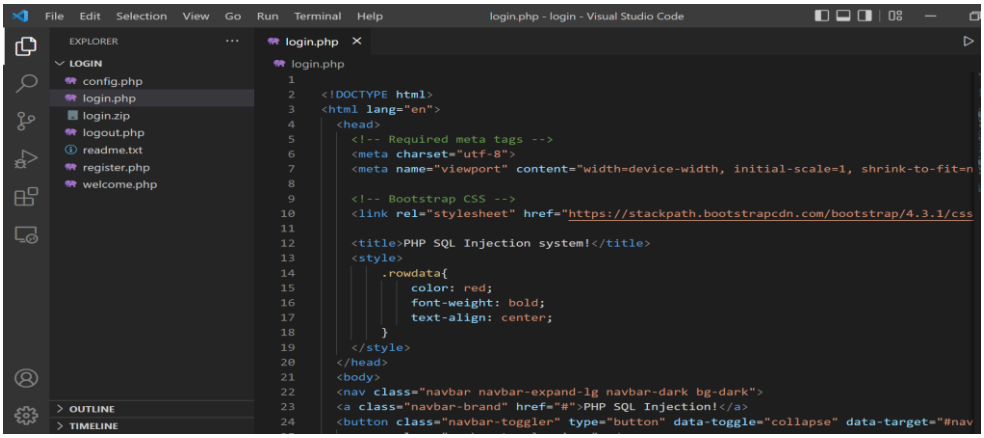
1.1. Open XAMPP Control Panel.
1.2. Start Apache and MySQL.



2. Create different PHP files for login, logout, config, and register.

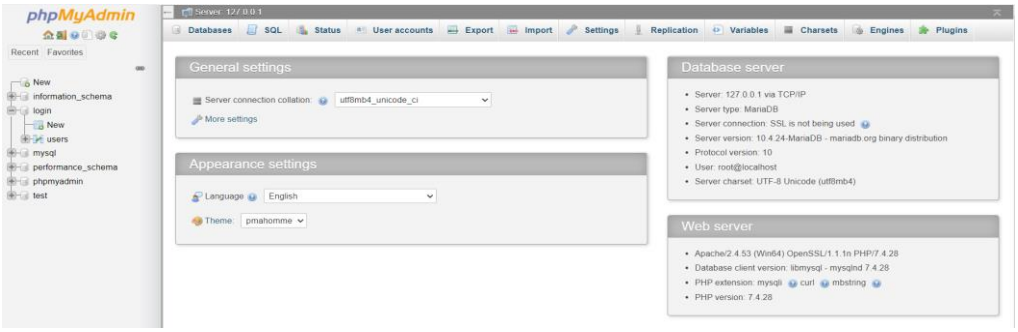
2.1. Create a PHP file in the location: C:\xampp\htdocs\login.
2.2. Open the folder in VS Code Editor.

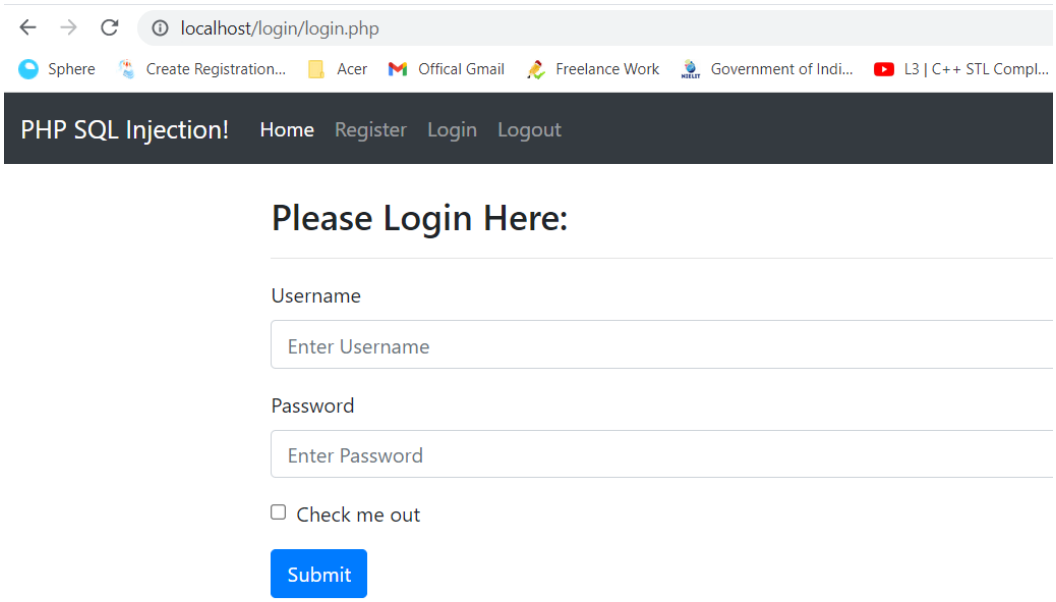




3. Open Localhost in the browser.

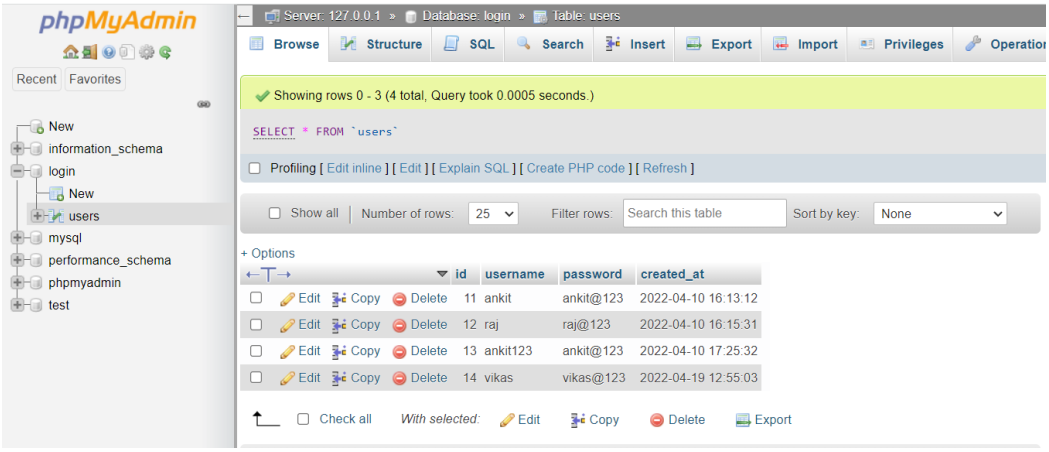
3.1. Open: http://localhost/login/login.php
3.2. Open: http://localhost/phpmyadmin/index.php



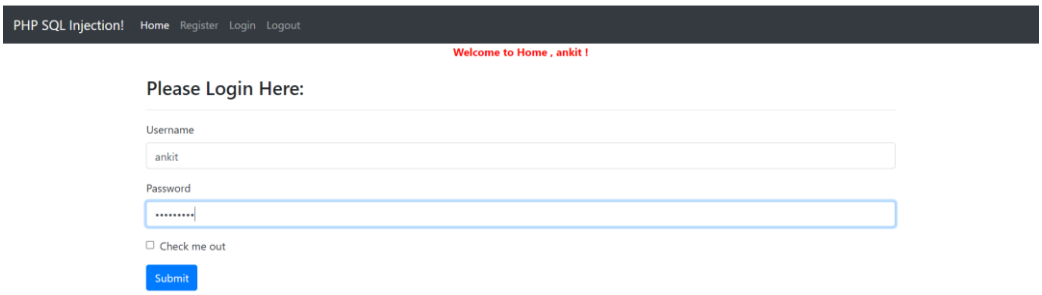


4. Create a Database and table in PHPMyAdmin.

4.1. Database Name: login
4.2. Table Name: users



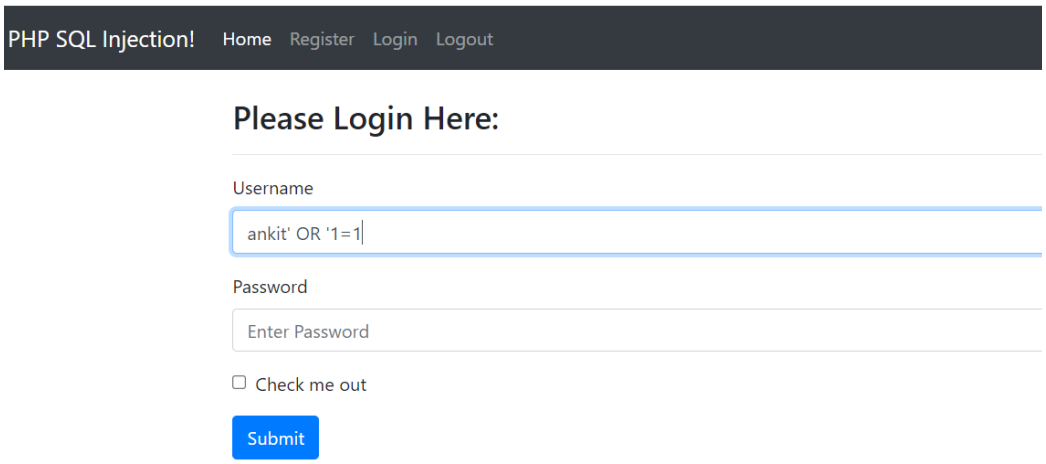
5. Register a user from the register page and then log in with valid credentials.



6. SQL Injection attack for a specific user.

SELECT * FROM `users` WHERE username='ankit' OR '1=1' and password='';

username : ankit' OR '1=1

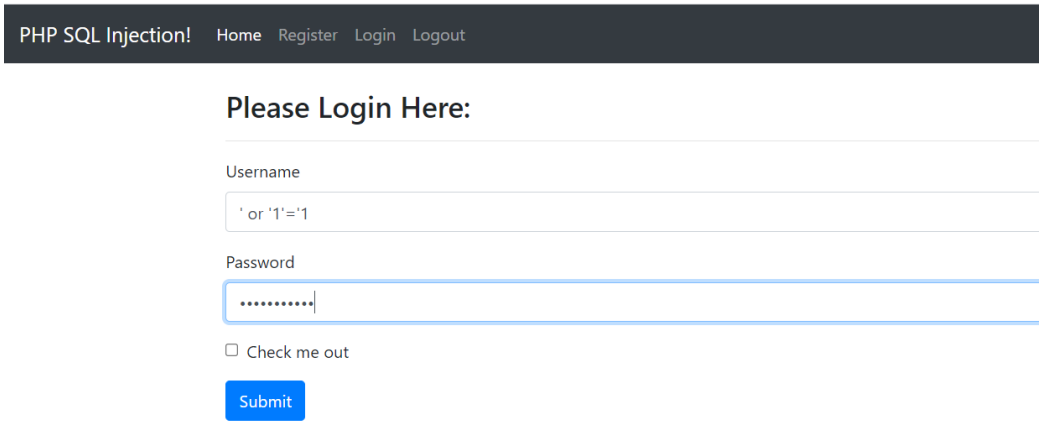


User Successfully Logged in!

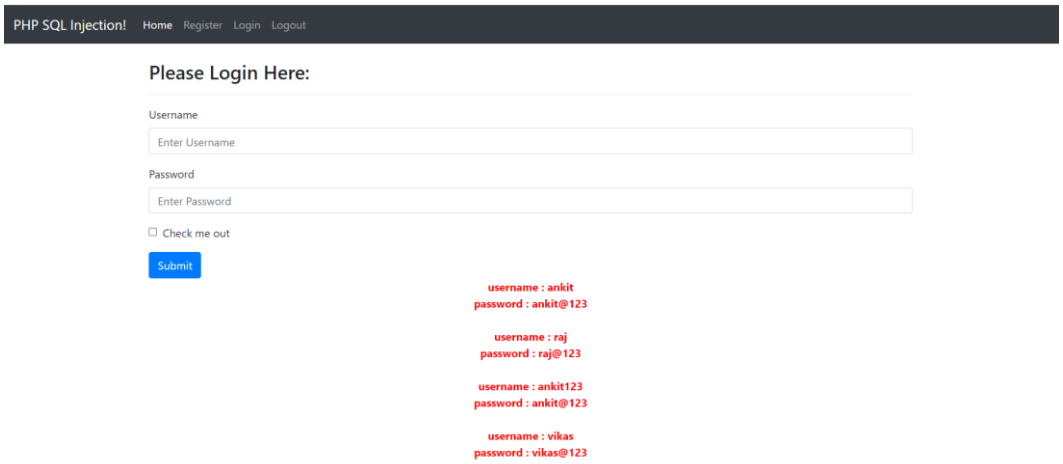
7. SQL Injection attack to fetch all data from the database.

SELECT * FROM `users` WHERE username = 'a' and password = '' or '1'='1';

password: ' or '1'='1



Fetch All User Data!

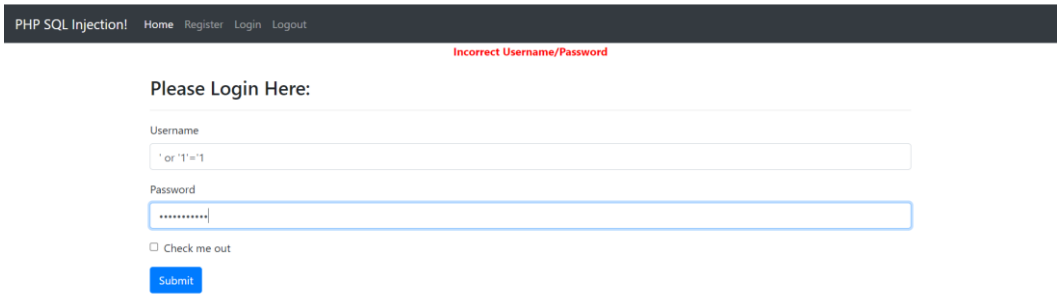


8. **Use of Escaping String to prevent SQL Injection.**

To prevent the SQL Injection Attack, we can apply various approaches. Here we have Escaping String to ignore the special characters. Before submitting a query to a Server-side, mysql real escape string() is used to escape special characters such as ",'n', and so on. The specified unescaped string is encapsulated, and the result is an escaped sql string. The length of the encoded or escaped sqlstring is returned by the mysql real escape string() function.

```
$uname = mysqli_real_escape_string($conn, $uname);//test or 1=1
$pass = mysqli_real_escape_string($conn, $pass);
```

The above-mentioned lines of code make sure that the web page doesn't send special characters in the query to execute in the database.



On applying the escape string, now we cannot able to fetch data using SQL Injection and it throws "Incorrect Username/Password" message as shown in the above snapshot.

5. Code & Algorithm

Config.php:

```
<?php

define('DB_SERVER', 'localhost');
define('DB_USERNAME', 'root');
define('DB_PASSWORD', '');
define('DB_NAME', 'login');

// Try connecting to the Database
$conn = mysqli_connect(DB_SERVER, DB_USERNAME, DB_PASSWORD, DB_NAME);

//Check the connection
if($conn == false){
```

```
    dir('Error: Cannot connect');
}

?>
```

login.php:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <!-- Required meta tags -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

    <!-- Bootstrap CSS -->
    <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css"
integrity="sha384-
ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"
crossorigin="anonymous">

    <title>PHP SQL Injection system!</title>
    <style>
        .rowdata{
            color: red;
            font-weight: bold;
            text-align: center;
        }
    </style>
  </head>
  <body>
  <nav class="navbar navbar-expand-lg navbar-dark bg-dark">
  <a class="navbar-brand" href="#">PHP SQL Injection!</a>
  <button class="navbar-toggler" type="button" data-toggle="collapse" data-
target="#navbarNavDropdown" aria-controls="navbarNavDropdown" aria-expanded="false"
aria-label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
  </button>
  <div class="collapse navbar-collapse" id="navbarNavDropdown">
  <ul class="navbar-nav">
```

```
    <li class="nav-item active">
      <a class="nav-link" href="#">Home <span class="sr-only">(current)</span></a>
    </li>
    <li class="nav-item">
      <a class="nav-link" href="register.php">Register</a>
    </li>
    <li class="nav-item">
      <a class="nav-link" href="login.php">Login</a>
    </li>
    <li class="nav-item">
      <a class="nav-link" href="logout.php">Logout</a>
    </li>
   </ul>
  </div>
</nav>

<div class="rowdata" >
<?php
$hostname = "localhost";
$username = "root";
$password = "";
$dbname = "login";
$conn = mysqli_connect($hostname, $username, $password, $dbname);
if(!$conn) {
  die("Unable to connect");
}
if($_POST) {
  $uname = $_POST["username"];
  $pass = $_POST["password"];
  //Making sure that SQL Injection doesn't work
  // $uname = mysqli_real_escape_string($conn, $uname);//test or 1=1
  // $pass = mysqli_real_escape_string($conn, $pass);
  $sql = "SELECT * FROM users WHERE username = '$uname' AND password = '$pass'";
  $result = mysqli_query($conn, $sql);
  if(mysqli_num_rows($result) == 1) {
       echo "Welcome to Home , ";
       while ($row = mysqli_fetch_row($result)) {
          printf ("%s !",$row[1]);
        }
  } else if(mysqli_num_rows($result) == 0){
    echo "Incorrect Username/Password";
  }
}
?>
```

```
</div>

<div class="container mt-4">
<h3>Please Login Here:</h3>
<hr>
<form action method="POST" autocomplete="off">
  <div class="form-group">
    <label for="exampleInputEmail1">Username</label>
    <input type="text" name="username" class="form-control" id="exampleInputEmail1"
aria-describedby="emailHelp" placeholder="Enter Username">
  </div>
  <div class="form-group">
    <label for="exampleInputPassword1">Password</label>
    <input type="password" name="password" class="form-control"
id="exampleInputPassword1" placeholder="Enter Password">
  </div>
  <div class="form-group form-check">
    <input type="checkbox" class="form-check-input" id="exampleCheck1">
    <label class="form-check-label" for="exampleCheck1">Check me out</label>
  </div>
  <button type="submit" class="btn btn-primary">Submit</button>
</form>
</div>
<div class="rowdata" >

<?php
$hostname = "localhost";
$username = "root";
$password = "";
$dbname = "login";
$conn = mysqli_connect($hostname, $username, $password, $dbname);
if(!$conn) {
  die("Unable to connect");
}
if($_POST) {
  $uname = $_POST["username"];
  $pass = $_POST["password"];
  //Making sure that SQL Injection doesn't work
  // $uname = mysqli_real_escape_string($conn, $uname);//test or 1=1
  // $pass = mysqli_real_escape_string($conn, $pass);
  $sql = "SELECT * FROM users WHERE username = '$uname' AND password = '$pass'";
  $result = mysqli_query($conn, $sql);
  if (mysqli_num_rows($result) > 1) {
       while ($row = mysqli_fetch_row($result)) {
```

```
            printf("username : ");
            printf ("\n %s ",$row[1]);
            echo "<br>";
            printf("password : ");
            printf ("\n %s ",$row[2]);
            echo "<br>";
            echo "<br>";
          }
    }
}
?>
</div>
</body>
</html>
```

logout.php:

```
<?php

session_start();
$_SESSION = array();
session_destroy();
header("location: login.php");

?>
```

register.php:

```
<?php
require_once "config.php";

$username = $password = $confirm_password = "";
$username_err = $password_err = $confirm_password_err = "";

if ($_SERVER['REQUEST_METHOD'] == "POST"){

    // Check if username is empty
    if(empty(trim($_POST["username"]))){
        $username_err = "Username cannot be blank";
    }
    else{
        $sql = "SELECT id FROM users WHERE username = ?";
        $stmt = mysqli_prepare($conn, $sql);
```

```
        if($stmt)
        {
            mysqli_stmt_bind_param($stmt, "s", $param_username);

            // Set the value of param username
            $param_username = trim($_POST['username']);

            // Try to execute this statement
            if(mysqli_stmt_execute($stmt)){
                mysqli_stmt_store_result($stmt);
                if(mysqli_stmt_num_rows($stmt) == 1)
                {
                    $username_err = "This username is already taken";
                }
                else{
                    $username = trim($_POST['username']);
                }
            }
            else{
                echo "Something went wrong";
            }
        }
    }

    mysqli_stmt_close($stmt);


// Check for password
if(empty(trim($_POST['password']))){
    $password_err = "Password cannot be blank";
}
elseif(strlen(trim($_POST['password'])) < 5){
    $password_err = "Password cannot be less than 5 characters";
}
else{
    $password = trim($_POST['password']);
}

// Check for confirm password field
if(trim($_POST['password']) !=  trim($_POST['confirm_password'])){
    $password_err = "Passwords should match";
}


// If there were no errors, go ahead and insert into the database
```

```
if(empty($username_err) && empty($password_err) && empty($confirm_password_err))
{
   $sql = "INSERT INTO users (username, password) VALUES (?, ?)";
   $stmt = mysqli_prepare($conn, $sql);
   if ($stmt)
   {
      mysqli_stmt_bind_param($stmt, "ss", $param_username, $param_password);

      // Set these parameters
      $param_username = $username;
      $param_password = $password;

      // Try to execute the query
      if (mysqli_stmt_execute($stmt))
      {
         header("location: login.php");
      }
      else{
         echo "Something went wrong... cannot redirect!";
      }
   }
   mysqli_stmt_close($stmt);
}
mysqli_close($conn);
}

?>

<!doctype html>
<html lang="en">
  <head>
    <!-- Required meta tags -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

    <!-- Bootstrap CSS -->
    <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css"
integrity="sha384-
ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"
crossorigin="anonymous">

    <title>PHP login system!</title>
  </head>
```

```
 <body>
 <nav class="navbar navbar-expand-lg navbar-dark bg-dark">
 <a class="navbar-brand" href="#">PHP Login System</a>
 <button class="navbar-toggler" type="button" data-toggle="collapse" data-
target="#navbarNavDropdown" aria-controls="navbarNavDropdown" aria-expanded="false"
aria-label="Toggle navigation">
   <span class="navbar-toggler-icon"></span>
 </button>
 <div class="collapse navbar-collapse" id="navbarNavDropdown">
 <ul class="navbar-nav">
    <li class="nav-item active">
      <a class="nav-link" href="#">Home <span class="sr-only">(current)</span></a>
    </li>
    <li class="nav-item">
      <a class="nav-link" href="register.php">Register</a>
    </li>
    <li class="nav-item">
      <a class="nav-link" href="login.php">Login</a>
    </li>
    <li class="nav-item">
      <a class="nav-link" href="logout.php">Logout</a>
    </li>
  </ul>
 </div>
</nav>

<div class="container mt-4">
<h3>Please Register Here : </h3>
<hr>
<form action="" method="post">
 <div class="form-row">
   <div class="form-group col-md-6">
    <label for="inputEmail4">Username</label>
    <input type="text" class="form-control" name="username" id="inputEmail4"
placeholder="Email">
   </div>
   <div class="form-group col-md-6">
    <label for="inputPassword4">Password</label>
    <input type="password" class="form-control" name ="password" id="inputPassword4"
placeholder="Password">
   </div>
 </div>
 <div class="form-group">
    <label for="inputPassword4">Confirm Password</label>
```

```
    <input type="password" class="form-control" name ="confirm_password"
id="inputPassword" placeholder="Confirm Password">
   </div>
  <div class="form-group">
   <label for="inputAddress2">Address 2</label>
   <input type="text" class="form-control" id="inputAddress2" placeholder="Apartment,
studio, or floor">
  </div>
  <div class="form-row">
   <div class="form-group col-md-6">
    <label for="inputCity">City</label>
    <input type="text" class="form-control" id="inputCity">
   </div>
   <div class="form-group col-md-4">
    <label for="inputState">State</label>
    <select id="inputState" class="form-control">
     <option selected>Choose...</option>
     <option>...</option>
    </select>
   </div>
   <div class="form-group col-md-2">
    <label for="inputZip">Zip</label>
    <input type="text" class="form-control" id="inputZip">
   </div>
  </div>
  <div class="form-group">
   <div class="form-check">
    <input class="form-check-input" type="checkbox" id="gridCheck">
    <label class="form-check-label" for="gridCheck">
     Check me out
    </label>
   </div>
  </div>
  <button type="submit" class="btn btn-primary">Sign in</button>
</form>
</div>
</body>
</html>
```

6. Conclusion

One of the most infamous concerns is code injection attacks, particularly SQL injection attacks. Controlling harmful SQL code/script on the online application while ensuring end-user privacy remains a significant difficulty for the web developer. These challenges must be taken care of by web developers who are involved in the development of websites that use databases. This report illustrates how an attacker may use SQL injection to obtain private information from a database by exploiting a web application. Various defence techniques against SQL injection attacks are been offered.

References

1. Antunes N, Vieira M. "Comparing the effectiveness of penetration testing and static code analysis on the detection of sql injection vulnerabilities in web services," in 2009 15th IEEE Pacific Rim International Symposium on Dependable Computing. 2009;301-306.

2. Halfond WG, Viegas J, Orso A. "A classification of SQL-injection attacks and countermeasures," in Proceedings of the IEEE international symposium on secure software engineering. 2006;13-15.

3. Patel N, Mohammed F, Soni S. SQL injection attacks: techniques and protection mechanisms. International Journal on Computer Science and Engineering. 2011;3:199-203.

4. Ntagwabira L, Kang SL. "Use of query tokenization to detect and prevent SQL injection attacks," in 2010 3rd International Conference on Computer Science and Information Technology. 2010;438-440.

5. Zhang H, Zhang X. "SQL injection attack principles and preventive techniques for PHP site," in Proceedings of the 2nd International Conference on Computer Science and Application Engineering. 2018;1-9.

6. Kaur N, Kaur P. SQL injection–anatomy and risk mitigation. Cover Story What, Why and How of Software Security 7 Cover Story Developing Secure Software. 2014;9:27.