WAPH-Web Application Programming and Hacking

Instructor: Dr. Phu Phung

Student

Name: Ian Cannon

Email: mailto:cannoni1@udayton.edu

Short-bio: Ian Cannon interests in Reinforcement Learning for Autonomous Control.

Ian's headshot

Repository Information

Respository's URL: https://github.com/Spiph/WebAppDev

This is a public repository for Ian Cannon to store all code from the course. The organization of this repository is as follows.

Lab 3

a. Database Setup and Management

I created `waph` and the non-root user `cannoni1`

database-account

Then I created an admin account

database-data

```
icannon@ACT3:~/code/cps/waph-cannoni1/labs/lab3$ sudo mysql -u root -p < database-account.sql
[sudo] password for icannon:
Enter password:
icannon@ACT3:~/code/cps/waph-cannoni1/labs/lab3$ mysql -u cannoni1 -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 81
Server version: 8.0.42-0ubuntu0.24.04.1 (Ubuntu)

Copyright (c) 2000, 2025, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases
    -> ^C
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| performance_schema |
| waph               |
+--------------------+
3 rows in set (0.01 sec)

mysql>
```

Here is the output

```
icannon@ACT3:~/code/cps/waph-cannoni1/labs/lab3$ mysql -u cannoni1 -p waph < database-data.sql
Enter password:
icannon@ACT3:~/code/cps/waph-cannoni1/labs/lab3$ mysql -u cannoni1 -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 103
Server version: 8.0.42-0ubuntu0.24.04.1 (Ubuntu)

Copyright (c) 2000, 2025, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> select * from users;
ERROR 1046 (3D000): No database selected
mysql> use waph;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> select * from users;
+----------+----------------------------------+
| username | password                         |
+----------+----------------------------------+
| admin    | 1a1dc91c907325c69271ddf0c944bc72 |
+----------+----------------------------------+
1 row in set (0.00 sec)

mysql>
```

And here is more output

b. A Simple (Insecure) Login System with PHP/MySQL

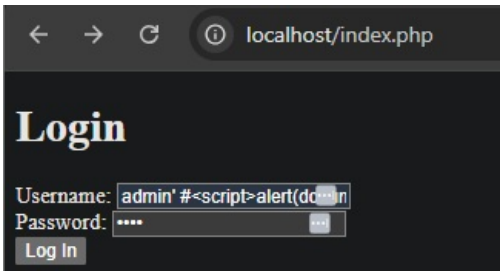
form

index

successful login

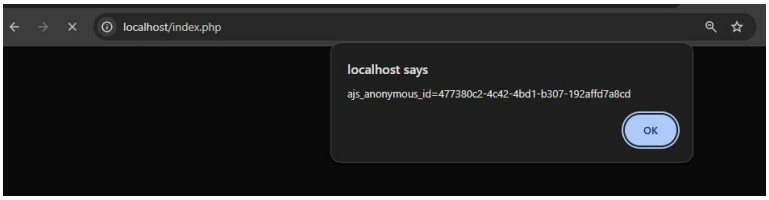
c. Performing XSS and SQL Injection Attacks

Following the instructions from the checkin, here is my attack: `admin' #<script>alert(document.cookie)</script>`



scripting attack

And the output from the attack



Attacked!

The SQL injection attack works because the application directly inserts user input into the database query without validation. The system concatenates the username and password into the SQL string, which allows an attacker to alter the query's logic.

Original Query Structure:

SQL

`SELECT * FROM users WHERE username='USER_INPUT' AND password = md5('PASSWORD_INPUT')` Attacker's Input: The payload used is admin' #.

Altered Query: When this input is inserted, the query sent to the database becomes:

```
SELECT * FROM users WHERE username='admin' #' AND password = md5('any_password')
```

How it Bypasses Authentication:

The single quote (') after admin closes the username value.

The hash symbol (#) acts as a comment character in MySQL, telling the database to ignore everything that follows it.

As a result, the `AND password = ...` part of the query is completely ignored. The database executes the simplified, valid query `SELECT * FROM users WHERE username='admin'`, finds the admin user, and logs the attacker in without a valid password.

Cross-site Scripting (XSS) Vulnerability Explanation The Cross-site Scripting (XSS) vulnerability exists because the application echoes user input directly back to the browser in the HTML response without proper sanitization.

When a user successfully logs in, the welcome message is generated by taking the username from the

$_POST variable and printing it to the page. Since the SQL injection attack allows malicious code to be part of the “username,” that code is then executed by the browser.

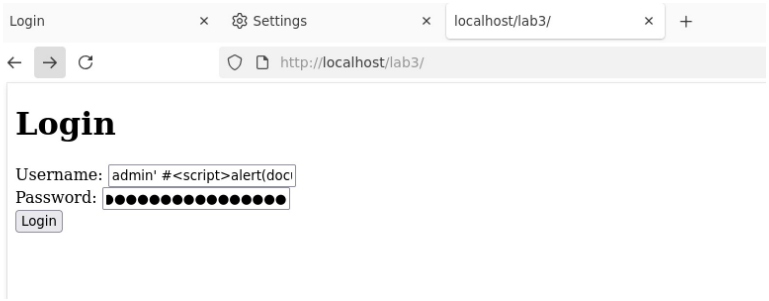
Vulnerable Code:

`<h2> Welcome <?php echo $_POST['username']; ?> !</h2>` Attacker’s Input: The full payload was `admin' #<script>alert(document.cookie)</script>`.

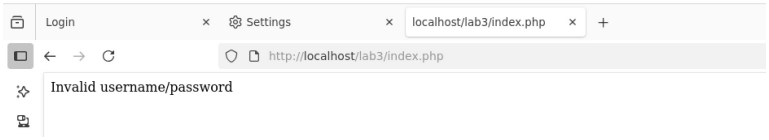
HTML Output: After the SQLi logs the attacker in, the server sends the following unsanitized HTML to the browser:

`<h2> Welcome admin' #<script>alert(document.cookie)</script> !</h2>` Browser Execution: The browser parses this HTML, sees the `<script>` tag, and executes the JavaScript inside it. This causes the alert box to appear, displaying the session cookie, as shown in the successful attack screenshot.

d. Prepared Statement Implementation



attempted injection



but it fails

Security Analysis Explanation of Prepared Statements Prepared statements prevent SQL injection by strictly separating the SQL query’s structure from the data it uses. The process works in two main stages:

Prepare: An SQL statement template containing placeholders (?) is sent to the database server. The database parses, compiles, and optimizes this query structure without any user input.

Execute: The user-supplied data is sent to the database separately. The database then binds this data to the pre-compiled template, treating it exclusively as data, not as executable code.

Because the query's logic is already compiled before the user's input is introduced, malicious commands like

' or # cannot alter the query's intent and are simply treated as part of the string to be matched.

Implementation of Output Sanitization To mitigate the Cross-site Scripting (XSS) risk, the output must be sanitized before being rendered in the browser. This is done using the htmlspecialchars() function, which converts characters that have special meaning in HTML (like < and >) into their HTML entity equivalents.

Revised Code: The welcome message is changed to sanitize the username.

`<h2> Welcome <?php echo htmlspecialchars($_POST['username']); ?> ! </h2>` Explanation: By wrapping the output in htmlspecialchars(), an input like