Vulnerable Home Lab- Web Vulnerabilities

KALI LINUX

Warning- This Lab is intended for educational purposes

Introduction

This lab is designed to understand and exploit common web vulnerabilities. By working through this lab, you will gain hands-on experience with

- SQL Injection
- Cross-Site Scripting (XSS),
- Cross-Site Request Forgery (CSRF).

Key Learning Elements

1. The lab setup involves creating a simple web application using Flask and MySQL, which contains deliberately introduced vulnerabilities.
2. We will learn how to exploit these vulnerabilities and understand the potential impact of such security flaws in real-world applications.
3. We will also learn about best practices and techniques to mitigate these vulnerabilities and improve the security of web applications.

Vulnerable Lab Setup

- Tools Used

  - Flask-MySQLdb
  - Python 3.x
  - MySQL
  - Flask

Steps to Create the home lab

STEP 1 -: Install Required Packages:

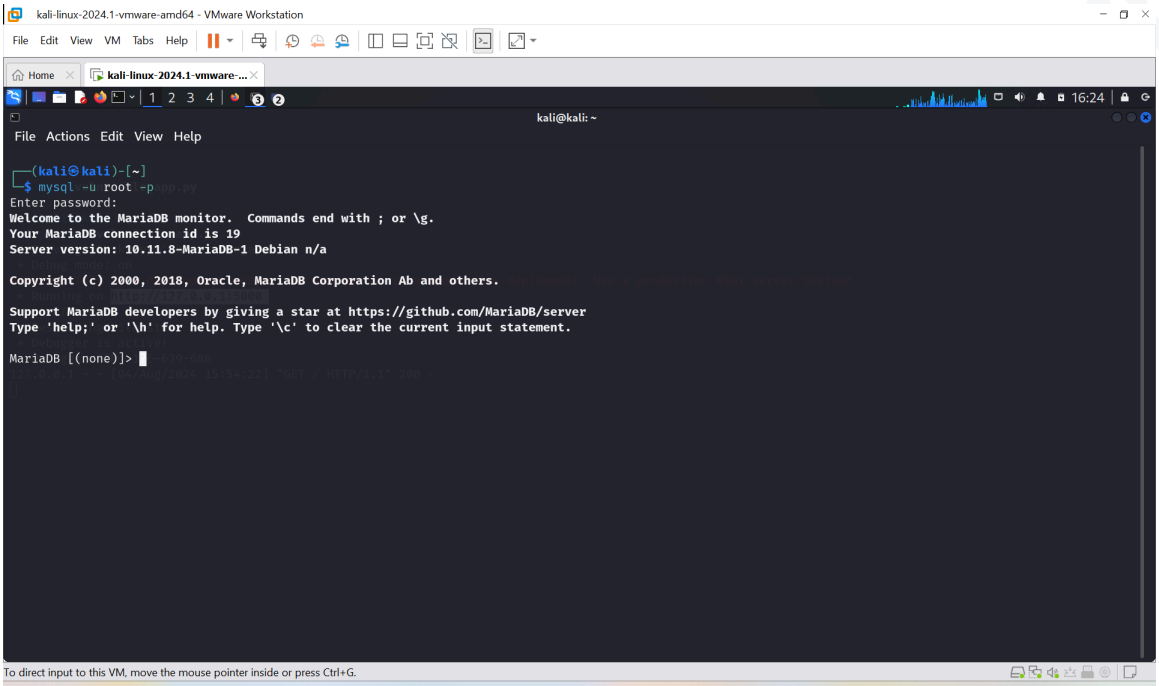
pip install Flask Flask-MySQLdb

STEP 2: Start MariaDB Sql server

sudo systemctl start mariadb (Note- Install the Sql Server first if not installed "sudo apt install mariadb-server")

STEP 3: Log into MySQL:

mysql -u root -p (Enter your MySQL root password when prompted.)



STEP 4 : Create Database and Table

CREATE DATABASE vulnlab; # Database creation

USE vulnlab; #Using created Database

CREATE TABLE users (# Creation of Table

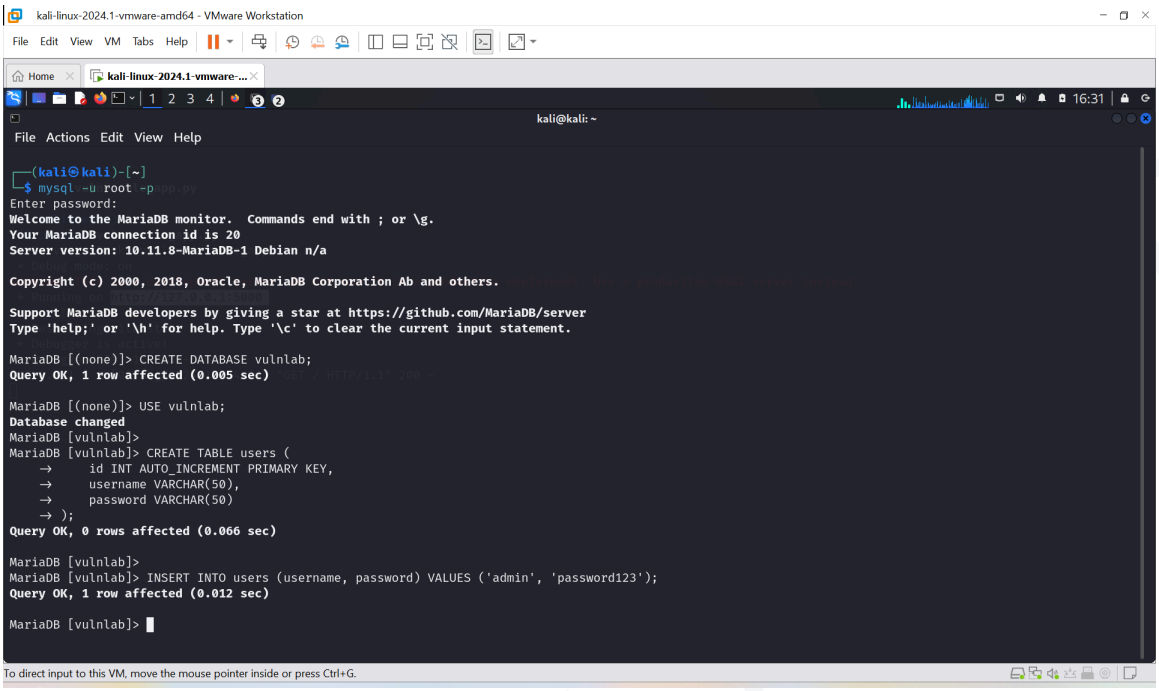
id INT AUTO_INCREMENT PRIMARY KEY,

username VARCHAR(50),

password VARCHAR(50)

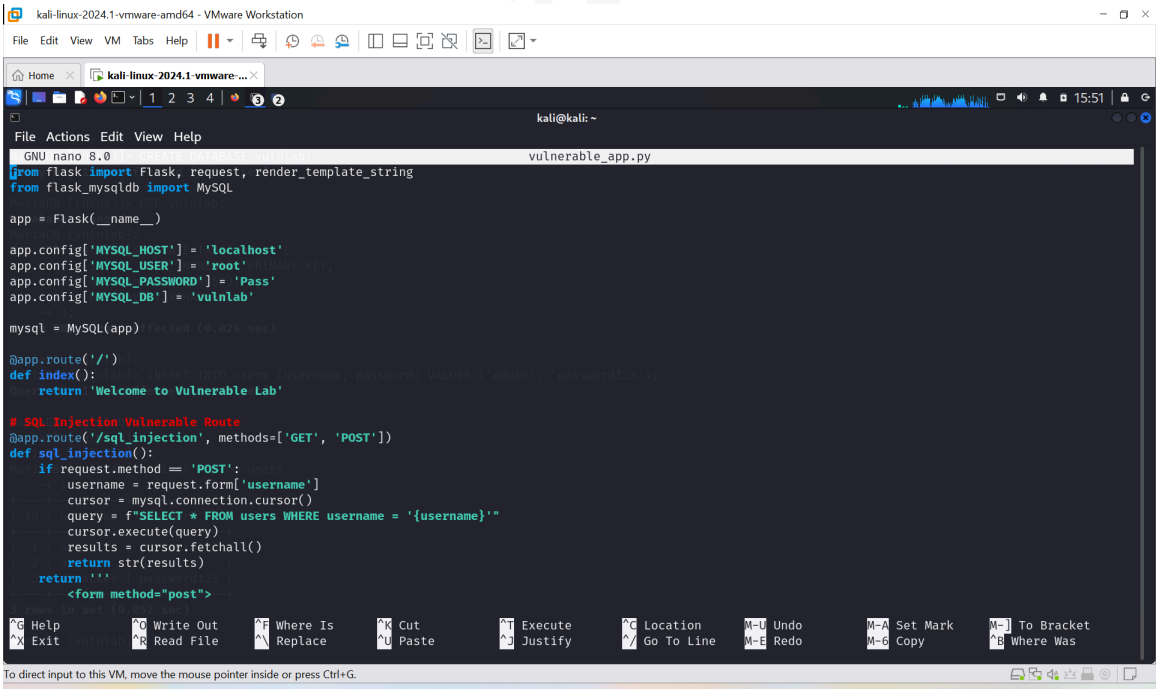
);

INSERT INTO users (username, password) VALUES ('admin', 'password123'); #Inserting into the Table



STEP 5 : Create Flask Application

nano Vulnerable_app.py



kali-linux-2024.1-vmware-amd64 - VMware Workstation

File Edit View VM Tabs Help

Home | kali-linux-2024.1-vmware-...

15:52

kali@kali: ~

File Actions Edit View Help

GNU nano 8.0 vulnerable_app.py

```
# SQL Injection Vulnerable Route
@app.route('/sql_injection', methods=['GET', 'POST'])
def sql_injection():
    if request.method == 'POST':
        username = request.form['username']
        cursor = mysql.connection.cursor()
        query = f"SELECT * FROM users WHERE username = '{username}'"
        cursor.execute(query)
        results = cursor.fetchall()
        return str(results)
    return '''
        <form method="post">
            Username: <input type="text" name="username"><br>
            <input type="submit" value="Submit">
        </form>
    '''

# XSS Vulnerable Route
@app.route('/xss', methods=['GET', 'POST'])
def xss():
    if request.method == 'POST':
        comment = request.form['comment']
        return f"Your comment: {comment}"
    return '''
        <form method="post">
            Comment: <input type="text" name="comment"><br>
            <input type="submit" value="Submit">
        </form>
```

^G Help ^O Write Out ^F Where Is ^K Cut ^T Execute ^C Location M-U Undo M-A Set Mark M-] To Bracket
^X Exit ^R Read File ^\ Replace ^U Paste ^J Justify ^/ Go To Line M-E Redo M-6 Copy ^B Where Was

To direct input to this VM, move the mouse pointer inside or press Ctrl+G.

kali-linux-2024.1-vmware-amd64 - VMware Workstation

File Edit View VM Tabs Help

Home | kali-linux-2024.1-vmware-...

15:53

kali@kali: ~

File Actions Edit View Help

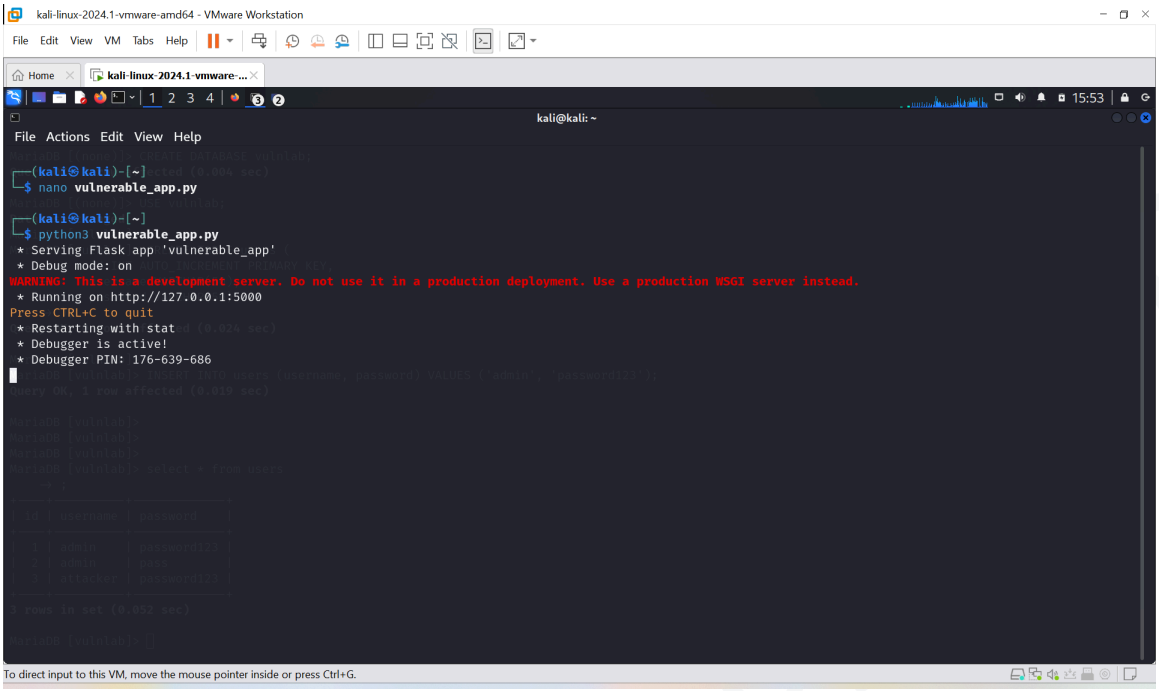
GNU nano 8.0 vulnerable_app.py

```
    return '''
        <form method="post">
            Comment: <input type="text" name="comment"><br>
            <input type="submit" value="Submit">
        </form>
    '''

# CSRF Vulnerable Route
@app.route('/csrf', methods=['GET', 'POST'])
def csrf():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        cursor = mysql.connection.cursor()
        cursor.execute("INSERT INTO users (username, password) VALUES (%s, %s)", (username, password))
        mysql.connection.commit()
        return 'User added!'
    return '''
        <form method="post">
            Username: <input type="text" name="username"><br>
            Password: <input type="password" name="password"><br>
            <input type="submit" value="Submit">
        </form>
    '''

if __name__ == '__main__':
    app.run(debug=True)
```

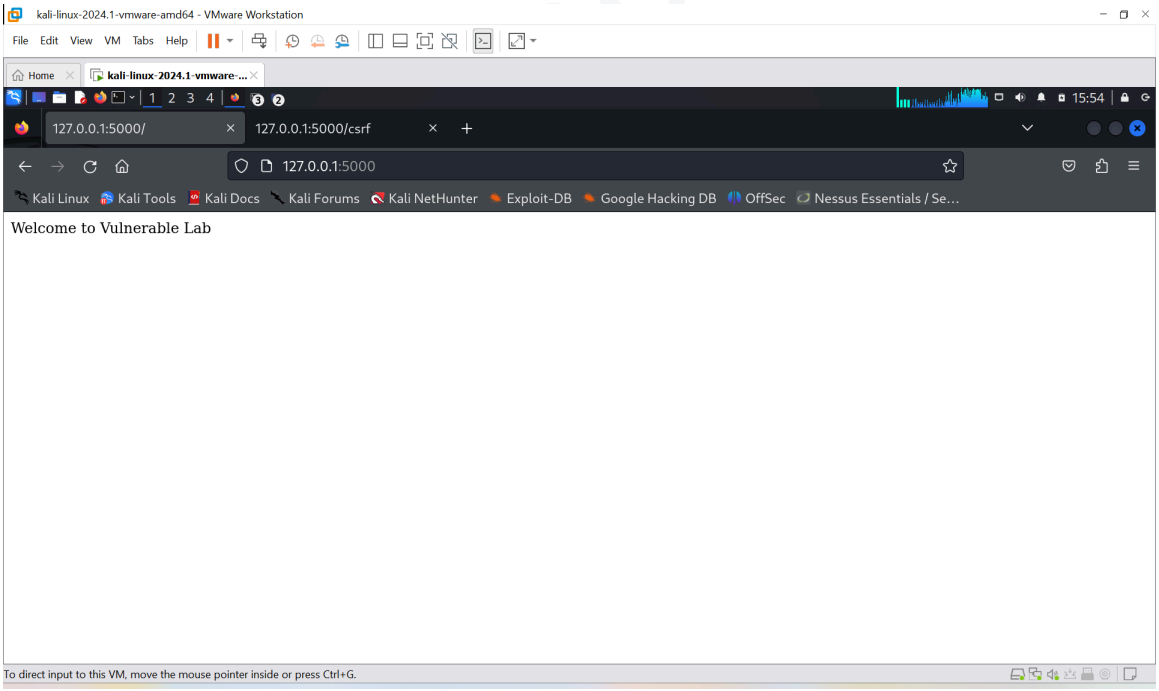
^G Help ^O Write Out ^F Where Is ^K Cut ^T Execute ^C Location M-U Undo M-A Set Mark M-] To Bracket
^X Exit ^R Read File ^\ Replace ^U Paste ^J Justify ^/ Go To Line M-E Redo M-6 Copy ^B Where Was

To direct input to this VM, move the mouse pointer inside or press Ctrl+G.

STEP 6 : Run the Application on the Python Server



STEP 6 : Go to the *http://127.0.0.1:5000/*



Vulnerability Walkthroughs

1. SQL Injection

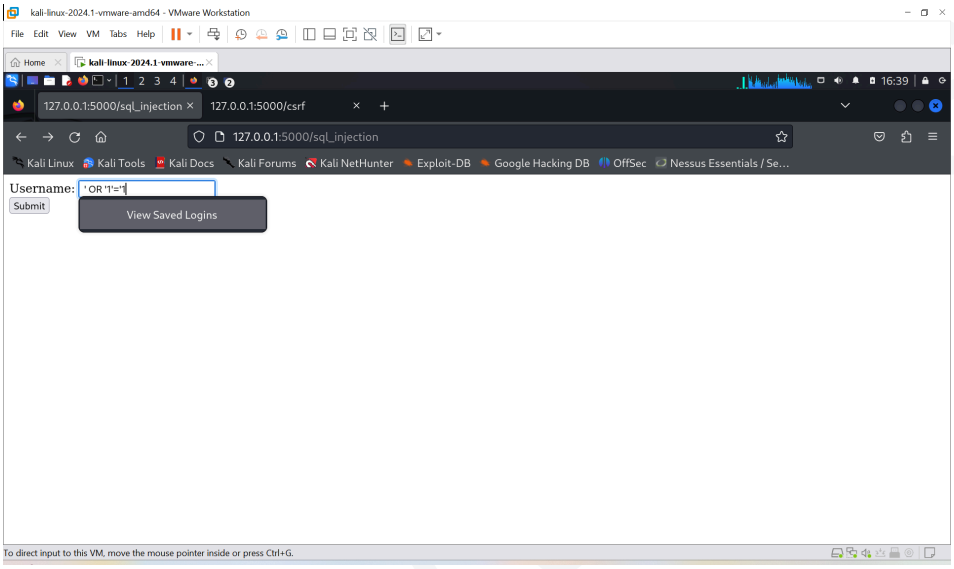
SQL Injection allows an attacker to manipulate SQL queries by injecting malicious SQL code.

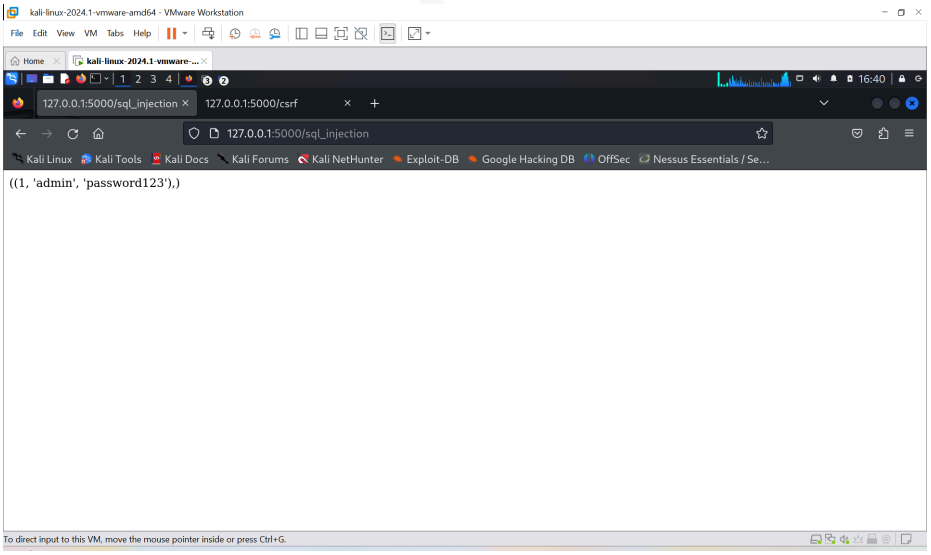
Steps to Exploit:

1. Navigate to http://127.0.0.1:5000/sql_injection.
2. Enter the following payload in the username field: ' OR '1'='1.
3. Submit the form.

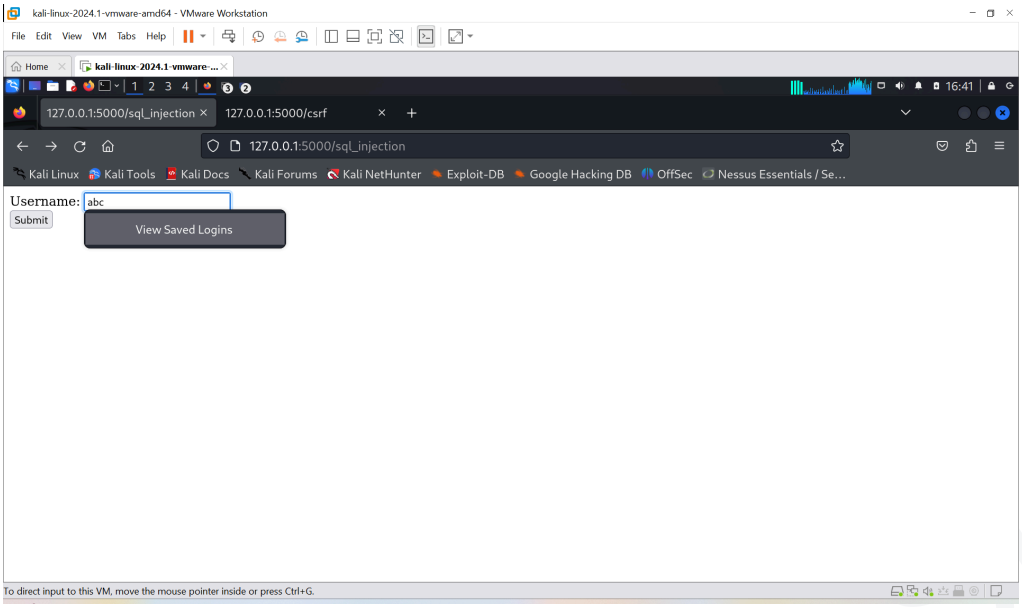
PoC:

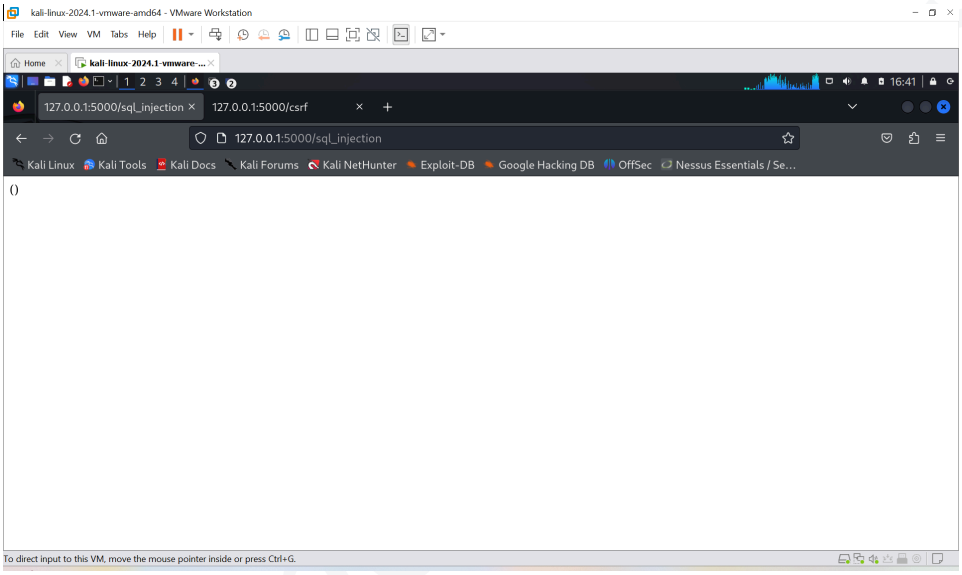
- Username: ' OR '1'='1
- The response will show all users in the database due to the injected SQL query.





Note -If we don't do SQL injection and just enter Random username it shows nothing





Mitigation:

- Use parameterized queries or prepared statements to prevent SQL injection.

2. Cross-Site Scripting (XSS)

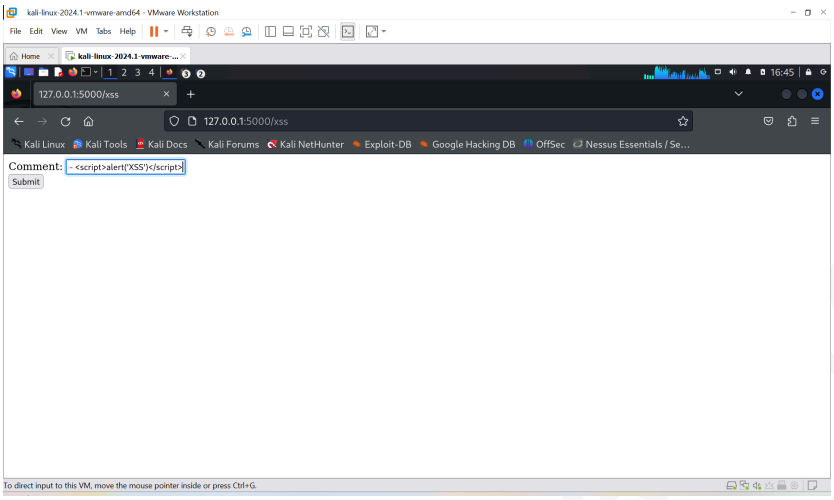
XSS allows an attacker to inject malicious scripts into webpages viewed by other users

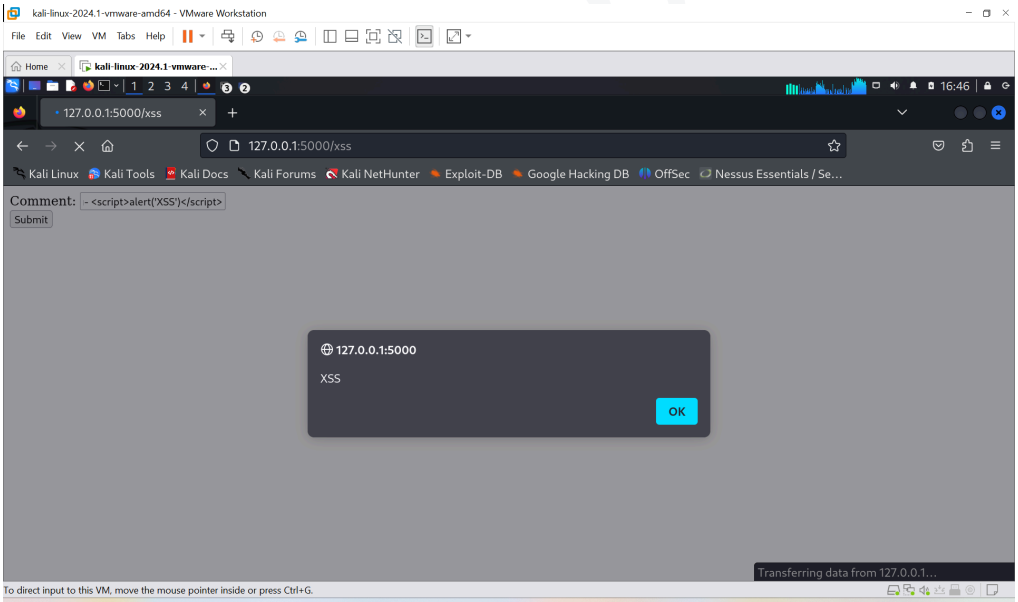
Steps to Exploit:

1. Navigate to http://127.0.0.1:5000/xss.
2. Enter the following payload in the comment field: <script>alert('XSS')</script>.
3. Submit the form.

PoC:

- Comment: <script>alert('XSS')</script>
- An alert box will pop up with the message 'XSS'.





Mitigation:

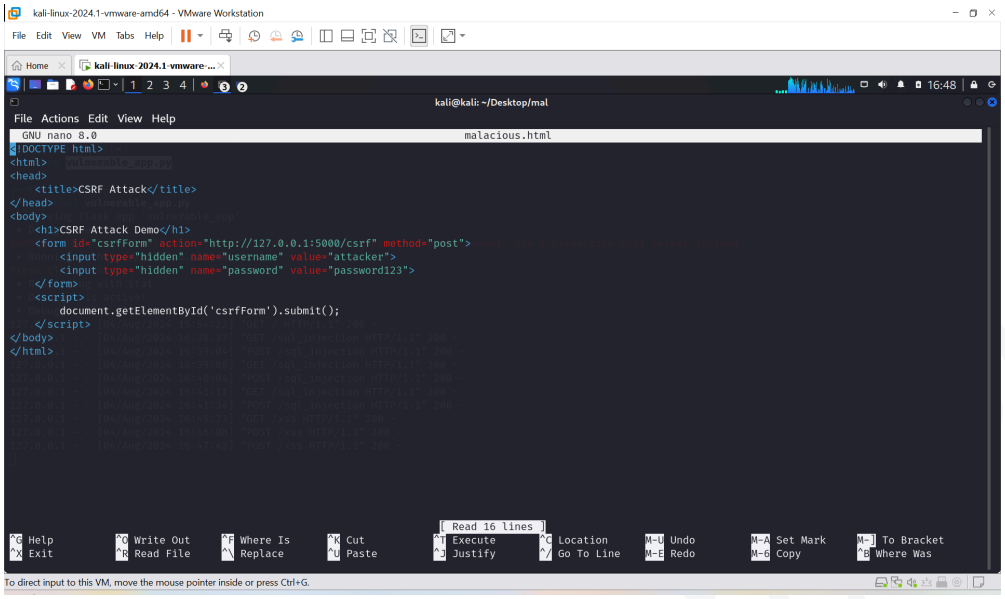
- Sanitize and encode user inputs before rendering them in HTML.

3. Cross-Site Request Forgery (CSRF)

CSRF allows an attacker to perform unauthorized actions on behalf of a logged-in user.

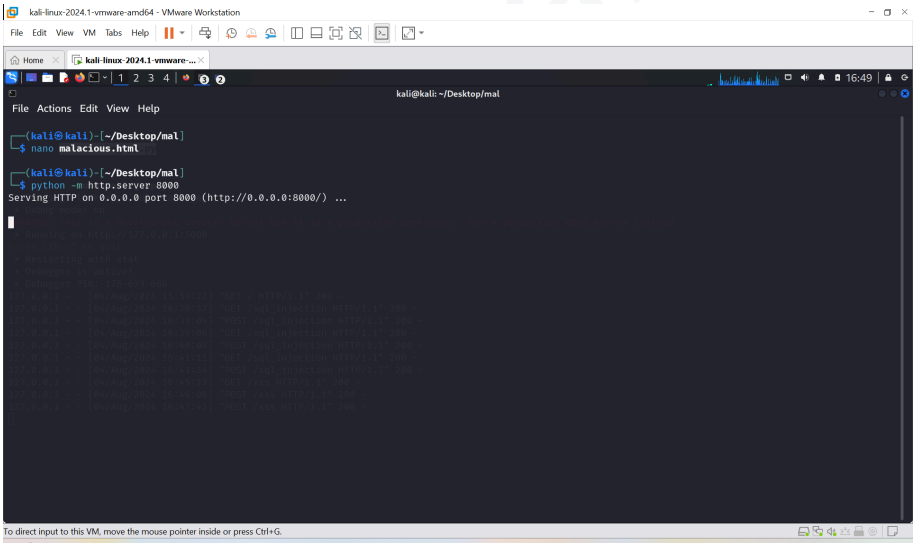
Steps to Exploit:

1. Create a malicious HTML file (malicious.html):

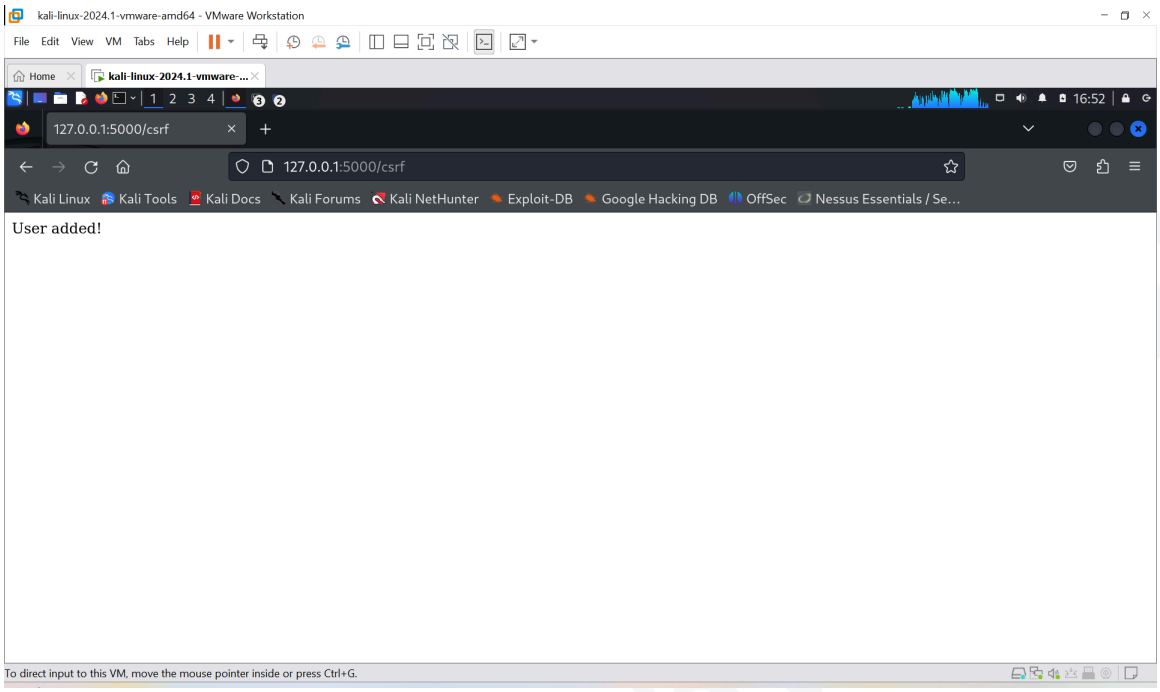


2. Host the file using Python's HTTP server:

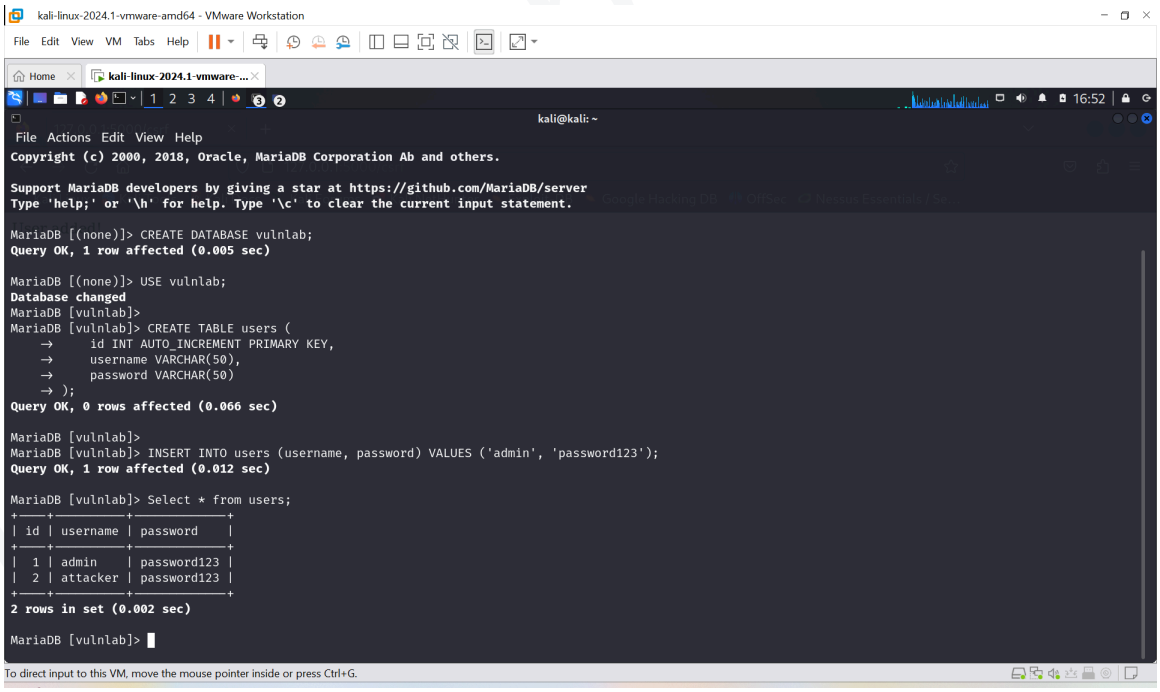
python -m http.server 8000



4. Open the malicious page in a browser: http://127.0.0.1:8000/malicious.html.



4. The form will be automatically submitted, adding a new user to the database.



PoC:

- Username: attacker
- Password: password123

Mitigation:

- Implement CSRF tokens to validate the origin of the requests.

Conclusion

1. Understanding Web Vulnerabilities:
    - SQL Injection: You learned how attackers can manipulate SQL queries to access or modify data within a database by injecting malicious SQL code.
    - Cross-Site Scripting (XSS): You saw how attackers can inject malicious scripts into web pages, which are then executed in the context of another user's browser, leading to data theft or session hijacking.
    - Cross-Site Request Forgery (CSRF): You explored how attackers can trick users into performing unintended actions on web applications where they are authenticated, leading to unauthorized actions.
2. Exploiting Vulnerabilities:
    - You performed practical exploitation of these vulnerabilities, observing the consequences and understanding the methods attackers use to leverage these weaknesses.
3. Mitigation Strategies:
    - SQL Injection: Implementing parameterized queries and prepared statements to prevent the inclusion of malicious code in SQL queries.
    - XSS: Sanitizing and encoding user inputs to prevent the execution of malicious scripts.
    - CSRF: Utilizing CSRF tokens to validate the origin of requests and ensure they come from trusted sources.
4. Best Practices:
    - Always validate and sanitize user inputs.
    - Implement proper authentication and authorization checks.
    - Use security libraries and frameworks that help prevent common vulnerabilities.