Name: Abdul Basit Rizwan
ID: DHC-251
Field: Cyber Security
Task:3

Task 1: Basic Penetration Testing – Report

Objective:
Test the login functionality of the Juice Shop application using browser-based testing to identify SQL injection vulnerabilities and verify HTTPS usage.

Step 1: Prepare Environment

- Opened the Juice Shop project in VS Code.
- Started the server: `npm start` → running at `http://localhost:3000/`.
- Opened the browser to access the login page.

Step 2: Test Input Validation

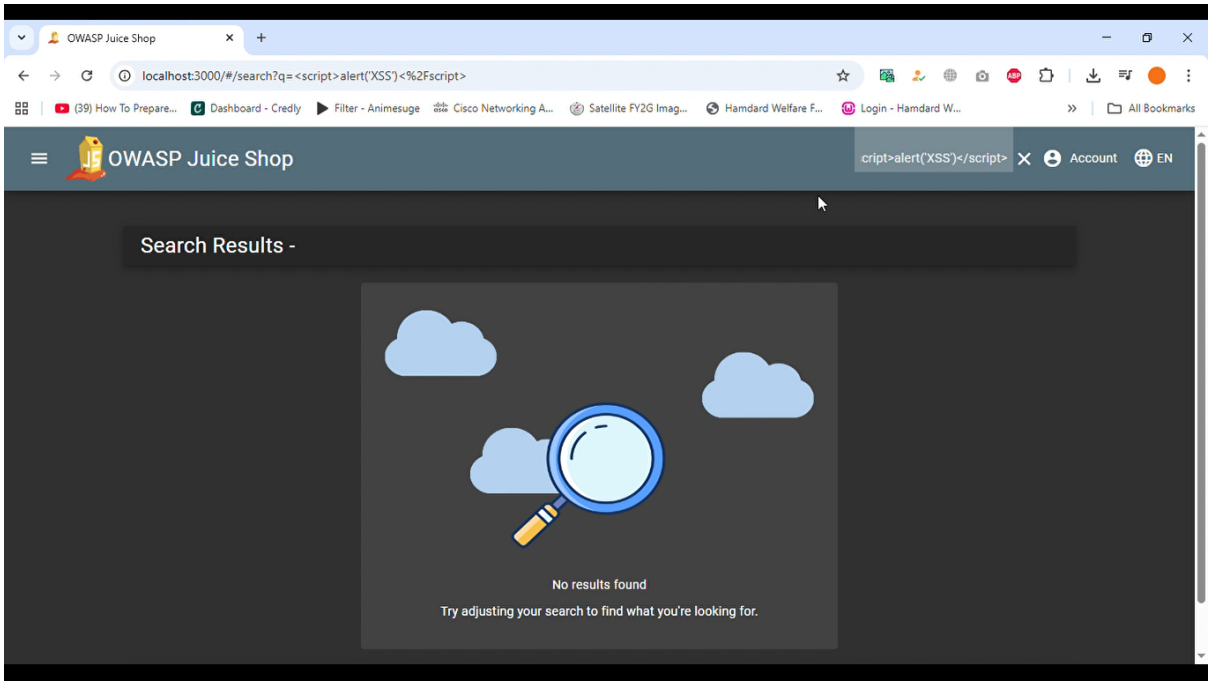
- Attempted to test login with invalid email formats.
- Example input: `' OR '1'='1'--` in the email field.
- Observation:
  - Juice Shop allows `' OR '1'='1'--` and similar inputs.
  - Entering password `123` successfully logged in.
  - Profile name shown: `admin@juice-sh.p`.

Conclusion:

- The login route is vulnerable to SQL injection **by design**.
- No matter how the local `login.ts` file is modified, the Juice Shop intentionally allows this login for the learning challenge.

Step 3: Browser-Based Testing for Pop-Ups

- Pasted test scripts in the search bar or login fields to check for alert pop-ups (simulating XSS).
- Observation:
  - No pop-up occurred.
  - Juice Shop's current input fields do not execute JavaScript in the browser.



Step 4: SQL Injection Testing

- Entered the following in the login form:

Email Password

' OR 1=1 -- 123

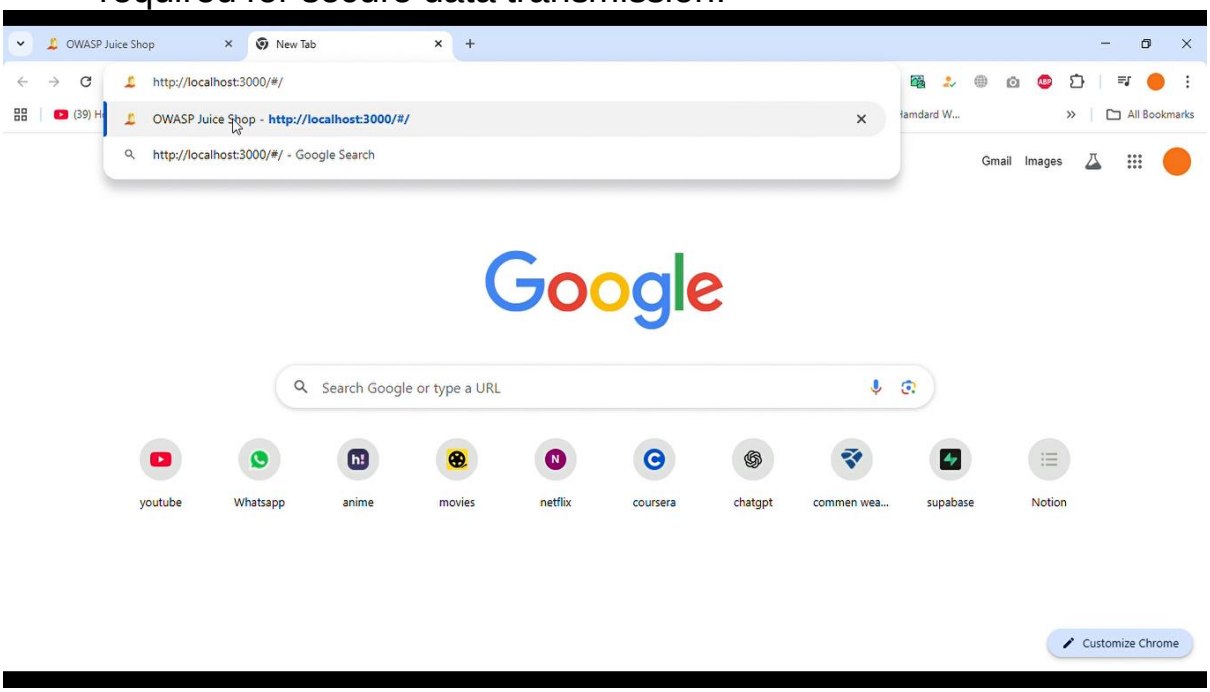
- Result: Successfully logged in as **admin@juice-sh.p**.

Explanation:

- This demonstrates SQL injection vulnerability.
- The vulnerability exists **intentionally** for learning purposes.

- Even with secure `login.ts` logic, the challenge environment is designed to let this attack succeed.

Step 5: Check HTTPS

- Current app URL: `http://localhost:3000/#/`.
- Observation:
  - The app runs over **HTTP**, not HTTPS.
  - Common in local development; in production, HTTPS is required for secure data transmission.
- 

Step 7: Conclusion

1. The Juice Shop login is **vulnerable to SQL injection** as part of the challenge.
2. SQL injection attempts successfully allowed admin access.
3. Browser-based XSS attempts did not trigger pop-ups.
4. The app runs over HTTP locally; HTTPS is required in production.
5. In a **real-world application**, SQL injection can be prevented using:

- Parameterized queries
- Input validation and sanitization
- Normalized emails
- Secure password hashing

Sure! Here's a **clean, updated report for Task 2** without the extra test line:

Task 2: Set Up Basic Logging

Objective

The purpose of this task is to implement **basic logging** in the OWASP Juice Shop application. Logging helps in:

- Monitoring application activities.
- Detecting security issues or errors.
- Maintaining a record of events for auditing purposes.

We used the **Winston library** to log important information both to the console and to a file.

Steps Performed

Step 1: Install Winston

In the VS Code terminal, we installed the Winston library:

```
npm install winston
```

- This adds Winston as a dependency in the project.
- Winston allows logging to the console, files, or other transports.

Step 2: Add Logging in app.ts

1. Imported Winston at the top of app.ts:

```
import winston from 'winston';
```

2. Created a logger instance:

```
const logger = winston.createLogger({
  transports: [
    new winston.transports.Console(),
    new winston.transports.File({ filename: 'security.log' })
  ]
});
```

- **Console transport** prints logs to the terminal.
- **File transport** writes logs to security.log for later reference.

3. Added a log entry when the application starts:

```
logger.info('Application started');
```

4. Added error logging in the catch block:

```
app().catch(err => {
  logger.error(`Application failed to start: ${err}`);
  throw err;
});
```

Step 3: Run the Application

- Started the Juice Shop app in VS Code terminal:

```
npm start
```

- The server started successfully, and Winston printed logs to the console.

Testing the Logging

1. **Check terminal output:**
   After starting the app, we saw multiple info: messages such as:

```
info: Detected Node.js version v22.13.0 (OK)
info: Server listening on port 3000
```

info: Restored 2-star loginAdminChallenge (Login Admin)

- This confirms **console logging is working**.

2. **Check `security.log` file:**
    - Opened the project folder and found `security.log`.
    - Verified it contained the same entries as the terminal.

Results

- Logging was successfully implemented using **Winston**.
- Logs are printed to the terminal and stored in `security.log`.
- The setup allows future logging of errors, warnings, and application events.

Got it! Here's a **ready-to-submit Task 3 checklist** for your OWASP Juice Shop project. You can save this as `security-checklist.md` in your project folder.

Security Checklist for OWASP Juice Shop

Objective

This checklist ensures that the application follows **basic security best practices**, reducing the risk of attacks and protecting user data.

Checklist

✅ 1. Validate all inputs

- All user inputs should be properly validated and sanitized.
- Helps prevent attacks such as **XSS (Cross-Site Scripting)** and **SQL Injection**.

✅ 2. Use HTTPS for data transmission

- All communication between client and server should be encrypted using HTTPS.
- Protects sensitive data like **login credentials** from interception.

✅ 3. Hash and salt passwords

- Passwords should be stored using a strong hashing algorithm like **bcrypt**.
- Each password should have a unique **salt** to enhance security.

✅ 4. Optional best practices

- Enable **logging for suspicious activity** (we implemented this in Task 2 with Winston).
- Use **secure HTTP headers** (e.g., Helmet.js) to protect against common web attacks.

Result

- This checklist serves as a **reference for secure coding and testing**.
- Ensures that basic security practices are considered in the development and testing of OWASP Juice Shop.