Lab: SQL Injection and XSS

Jaiden Shoel, May 16th, 2025

Introduction and Materials

For this lab, we explored two of the most common vulnerabilities in web security which are SQL Injections and Cross-Site Scripting (XSS). We put this into practice through our UW HusKey Password Manager running on localhost. The main goal of this lab was to understand how attackers could exploit user input fields to bypass authentication, steal data, or manipulate application behavior along with implementing and thinking about ways to patch and mitigate them . We not only demonstrated successful offensive techniques, but also applied some secure coding practices to patch those vulnerabilities.

Tools Used:

- Localhost HusKey Manager (PHP-based)
- OWASP ZAP: For automatic scanning and vulnerability alerts
- VS Code:
  - For editing and patching source code
- Chrome/Incognito: To simulate victim behavior and to mitigate through XSS
- Docker

Steps to Reproduce

First Attack: Log in as Admin without using their password (.5 pt)

1. On the login page, in the username field, I entered: ' OR username='admin' –
2. In the password field, I typed any dummy value (literally anything like asfjksajfkalf)
3. This still bypassed password verification and logged me in as the admin account due to the application executing raw SQL

Video Link: https://drive.google.com/file/d/1DgzAxBxWET0_9j3Jvxr3BEhflpIfGN9C/view?usp=sharing

Second Attack: Stored XSS – Fake Pop-up Prompt (2 pt)

1. Went to the the Vaults tab.
2. Navigated to add vault.
3. Entered the following malicious script into the input field:
   a. <script>prompt("Please re-enter your master password");</script>
4. Submitted the form and visited the vault page
5. The malicious script triggered automatically everytime showing a phishing-style prompt. I actually sort of stumped myself for a minutes because couldn't escape the prompt at all. Until I did some research and learned that visiting the site in an incognito tab allowed me to mitigate it and fix it.

Video Links:

- https://drive.google.com/file/d/1EkK1JLkEyTtg0Bye4xPi9F7VpiIwd6Pk/view?usp=sharing (I entered the script in the add vault input field. Just didn't capture that on video).
- https://drive.google.com/file/d/1a7EHQWQvf5c6F4ME2TFHM71tXcaBxZhO/view?usp=sharing
- https://drive.google.com/file/d/1zBvhkhpoqgRMlfvnuGgSK2uWgnKP8Iui/view?usp=sharing

Third Attack or rather Mitigation: Patch the login vulnerability (2 pt)

1. Replaced the vulnerable SQL code in login.php that allowed the first attack to happen:
   a. $sql = "SELECT * FROM users WHERE username = '$username' AND password = '$password'";
2. With this secure version through the looking up stuff via the internet and ChatGPT:

$stmt = $conn->prepare("SELECT * FROM users WHERE username = ? AND password = ? AND approved = 1");

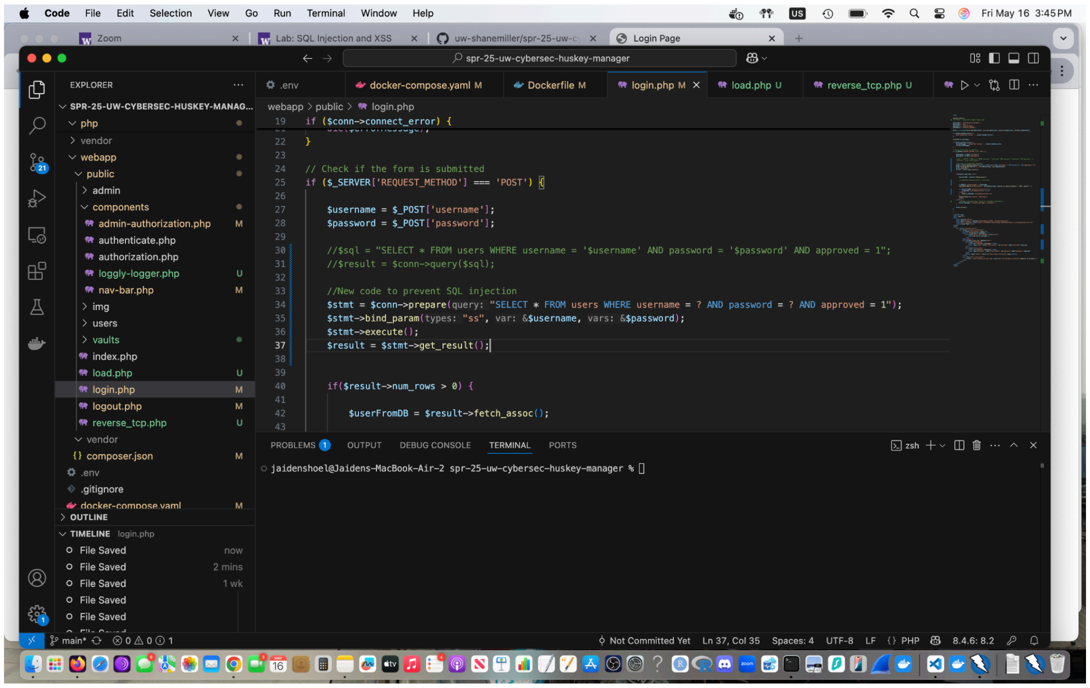
$stmt->bind_param("ss", $username, $password);

$stmt->execute();

$result = $stmt->get_result();

3. Then tested the old injection payload ' OR 1=1 -- — login failed

Improved code:



```
if ($conn->connect_error) {
}

// Check if the form is submitted
if ($_SERVER['REQUEST_METHOD'] === 'POST') {

    $username = $_POST['username'];
    $password = $_POST['password'];

    //$sql = "SELECT * FROM users WHERE username = '$username' AND password = '$password' AND approved = 1";
    //$result = $conn->query($sql);

    //New code to prevent SQL injection
    $stmt = $conn->prepare(query: "SELECT * FROM users WHERE username = ? AND password = ? AND approved = 1");
    $stmt->bind_param(types: "ss", var: &$username, vars: &$password);
    $stmt->execute();
    $result = $stmt->get_result();


    if($result->num_rows > 0) {

        $userFromDB = $result->fetch_assoc();
```

```
jaidenshoel@Jaidens-MacBook-Air-2 spr-25-uw-cybersec-huskey-manager %
```

Video Link that shows it works:
https://drive.google.com/file/d/1p4Ooath7LB4dhMbQHSbq646UeYv822Ae/view?usp=sharing

Class Principles

1. **For each attack you executed, which of the CIA pillars were violated?**

For my first attack, the SQL Injection clearly violated both confidentiality and authentication. By manipulating the login query, I was able to bypass access controls and impersonate a user without needing valid credentials. This highlighted a failure in enforcing proper authentication checks on your HusKey Passwords Manager's website's end, and it compromised access to potentially sensitive user accounts. Additionally, in the case of the Stored XSS attack, the vulnerability violated both confidentiality and integrity. The malicious script injected into a user-visible field executed in another user's browser session, which could have been used to phish for passwords or other private data. It also compromised the trust and integrity of the application's interface by allowing my attacker-controlled content to run over and over on the page without an obvious escape. I ran into this specific attack sometimes as growing up and I never knew what it was called. Let alone how it was possible. But now that I do know ,thanks to this lab, it was very surreal to experience and perform.

2. **Why does our web application allow for such attacks to occur?**

I think the root cause of these attacks is due to the way our application handles and trusts user input. In the SQL Injection example, the login system directly embedded user-supplied input into the SQL query without using parameterized queries. Because of this, it allowed malicious input to alter the logic of the database command. Similarly, in the Stored XSS case, the application failed to sanitize output before rendering it in the browser. When unescaped user input is injected directly into HTML, it opens the door for arbitrary script execution. All in all, it seems these vulnerabilities both stem from a lack of proper validation and sanitization mechanisms on the server side, and from a design that assumes user input is safe by default.

3. **Which of the Security Principles provided by OWASP relate to the vulnerabilities outlined in this lab?**

This lab highlights several OWASP principles, but most prominently the principles of input validation and secure coding. For the SQL Injection attack, the failure to use prepared statements violated the principle of secure database access and allowed attackers to inject harmful SQL logic. Using things like parameterized queries would have prevented this by treating user input as data rather than executable code. In the case of XSS, the application violated the OWASP principle of output encoding by failing to sanitize user-provided content before injecting it into the page. Both issues also tie into the principle of least privilege, as these vulnerabilities allowed users to escalate their access or influence parts of the application that they should not be able to control in the first place. These examples show how critical it is to not only validate and sanitize inputs, but to also assume that all input is potentially malicious unless proven otherwise.

Hacker Mindset and Conclusion

Article That I found and read:
https://www.securityweek.com/millions-of-user-records-stolen-from-65-websites-via-sql-injection-attacks/

In this article, the author, Ionut Arghire, illustrated how the threat actor group ResumeLooters compromised over 65 websites using a combination of SQL injection and Cross-Site Scripting (XSS) attacks between November and December 2023. The group reportedly stole more than two million user records, including names, phone numbers, email addresses, and employment histories, primarily from recruitment and retail platforms. According to cybersecurity firm Group-IB, ResumeLooters relied heavily on publicly available penetration testing tools and open-source SQL injection scripts to carry out the campaign.

Additionally, The group’s tactics extended beyond database manipulation. In several cases, they embedded malicious XSS scripts in job search websites by creating fake employer profiles or uploading CVs (resumes) that contained injected JavaScript. These scripts triggered phishing prompts in the browser of the victim, including users with administrative access, and were designed to collect login credentials or escalate access within the system.

This attack was replicated very similarly in the lab. For example, we were able to use SQL injection to bypass authentication entirely and log into the password manager as an admin user without a valid password. Likewise, the stored XSS attack I performed allowed me to inject a malicious script into a user-facing field that executed in the browser of another user. In a real-world scenario, this could easily be adapted to deliver a phishing payload like the ones used by ResumeLooters, tricking users into revealing passwords or session tokens. Some food for though that I always think about is we are only conducting these attacks at a surface and basic level. I can only imagine what other advanced tricks and possibilities real threat actors employ to attack their targets, especially in these recent years where cyber defenses have gotten stronger overall.