Lab 8: Cross-Site Scripting (XSS)

Shivali Mate

I. Introduction

THIS laboratory exercise examined Cross-Site Scripting (XSS) using the WebGoat 2023.8 training application to demonstrate how unencoded or improperly handled user input can be executed by a browser. The tasks guided a learner through reflected and DOM-based XSS scenarios, emphasizing both the practical steps to identify vulnerable input/sinks and the real-world consequences such as session theft, forged requests, and phishing amplification. The exercises were designed to connect conceptual definitions and threat models with hands-on exploitation and browser debugging so that the learner could both prove exploitability and reason about appropriate mitigations.

II. Methodology

A. Concept and definition of XSS

The lab began by defining Cross-Site Scripting: any situation where attacker-controlled HTML/JS is returned to a browser and executed because input was not properly encoded or sanitized.

```
alert(document.cookie)
```

These were used to confirm that the application reflected input to the client, establishing the attack surface and whether the payload traversed the server.

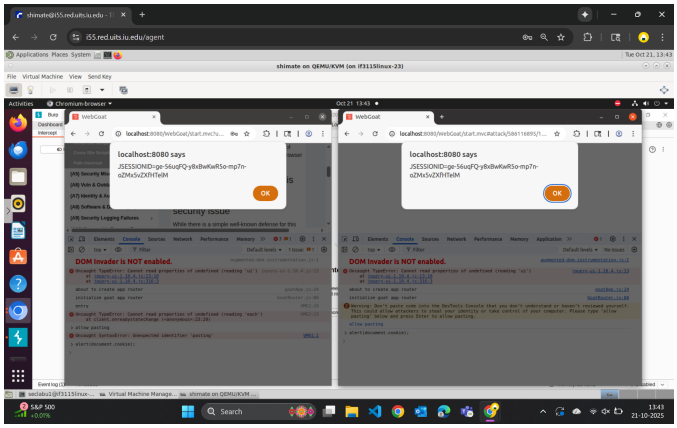

Fig. 1. Cookies displayed across tabs after executing XSS test script

B. Common sinks, impact, and XSS types

Likely sinks (search fields, form inputs, error messages, hidden fields, message boards, headers) were prioritized for testing because any echoed user data can become executable. The exercise linked XSS outcomes (cookie theft, forged requests, UI injection, redirects) to real risks and clarified types: reflected (link-triggered), DOM (client-only), and stored (persisted), guiding which delivery and encoding methods to use.

C. Reflected XSS scenario

A reflected attack flow was demonstrated: craft a URL, victim visits it, script runs, and data or actions can be exfiltrated. A minimal payload was injected into a vulnerable input.

```
<script>alert(23)</script>
```

The confirmed DOM inspection and page output is displayed in Fig. 2; payloads placed in URLs or routes were URL-encoded when needed to keep links valid.

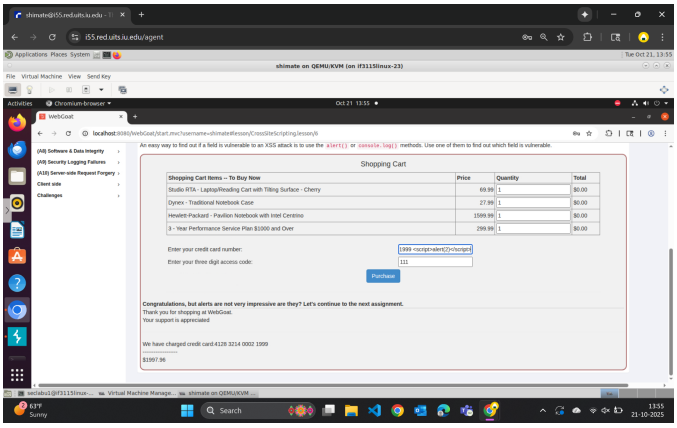

Fig. 2. Injected DOM and purchase confirmation reflecting XSS payload

D. Self-XSS vs reflected and DOM-based XSS

The lab distinguished self-XSS from a true reflected exploit that runs from a crafted link, and emphasized that DOM XSS is client-side only and won't touch the server. The approach was to inspect client code for uses of location.hash, innerHTML, document.write, or unsanitized templating to find where route or fragment parameters are reflected.

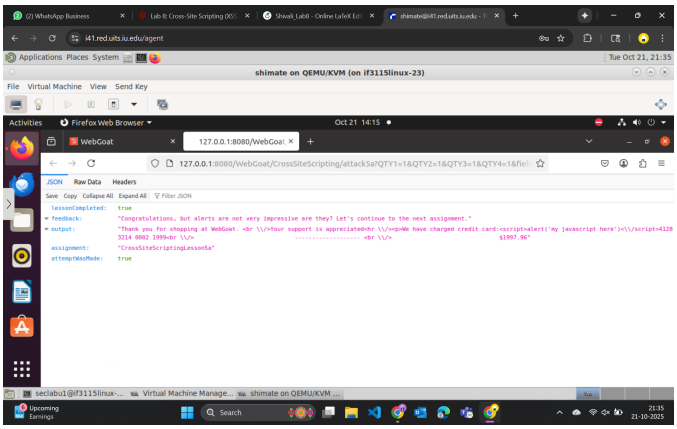

Fig. 3. Example of a crafted reflected XSS URL triggering script execution.

E. Locate vulnerable client routes and handlers

Client-side files i.e. router code were examined to find routes that accept parameters and insert them into the DOM; once a base route like start.mvc#test/ was found, fragment-based URLs were crafted to test reflection. This combined static code reading with dynamic navigation to validate whether the handler wrote parameters into an unsafe sink.

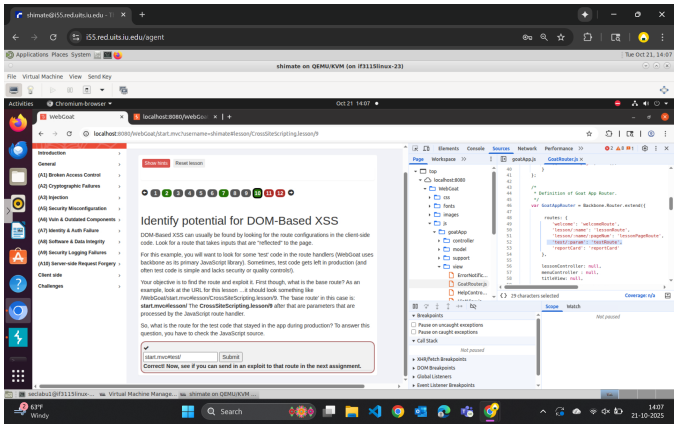

Fig. 4. Client-side route handler revealing vulnerable test route.

F. Exploiting DOM XSS

A proof-of-concept DOM exploit invoked a benign internal function via an injected script in the fragment, calling webgoat.customjs.phoneHome(), and success was confirmed by the expected console output (the random number).

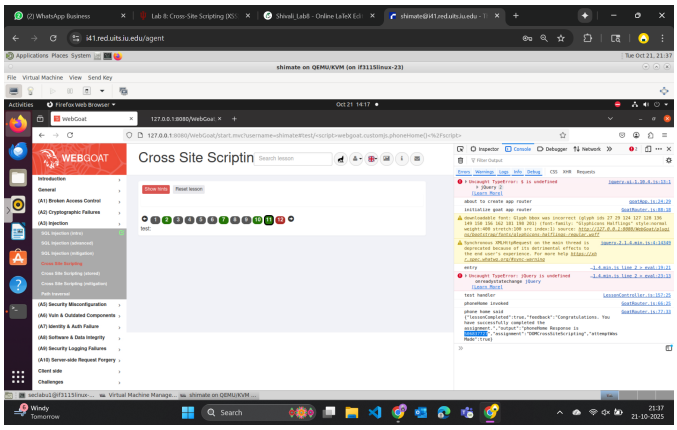

Fig. 5. Console output showing successful invocation of command

The lesson's final quiz was unavailable during the session; this was captured in a screenshot.

III. Tools and Learning Experience

The exercises used WebGoat as the vulnerable target, a local VM to contain the work, and browser developer tools like Chrome/Firefox Debugger and DOM inspector for runtime inspection; Dominic Breuker's toolkit and URL-encoding references were also consulted to build reproducible payloads. Suggested improvements are richer client-side flow visualization, built-in context-aware payload helpers, and exercises that mimic contemporary single-page frameworks and CSP configurations.

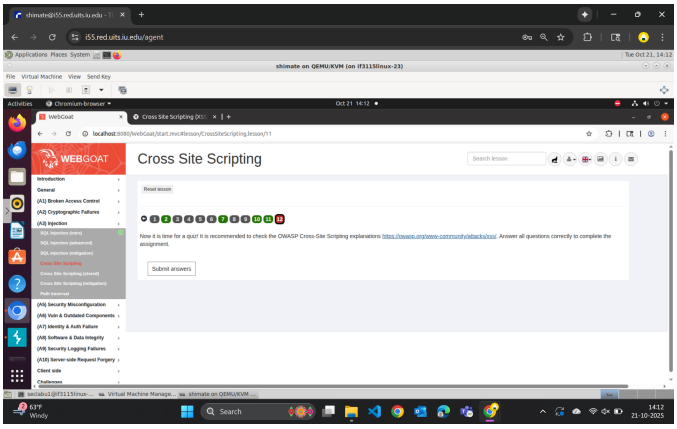

Fig. 6. Final lesson state showing quiz unavailability.

IV. Findings and Discussion

The lab reinforced that XSS remains a high-impact flaw: any user data reaching the browser without proper contextual encoding can execute as code, enabling cookie theft, forged requests, or UI manipulation; reflected, DOM, and stored XSS each require different delivery and mitigation approaches. In production, these translate to strict output encoding, framework escaping, CSP, and HTTP Only/same-site cookies to reduce impact. Remaining gaps for tooling and research are better detection of client-side router/template injection patterns and automated, context-aware tooling to reduce manual payload construction.

V. Conclusion

The laboratory exercises successfully demonstrated reflected and DOM-based XSS exploits in a controlled environment, and they highlighted the necessity of both server-side and client-side defenses. The hands-on workflow-identify injection surfaces, craft minimal payloads, and verify execution via DevTools, provided a concrete path for recognizing vulnerabilities and reasoning about mitigations. To improve training realism and learner productivity, future lab frameworks should include automatic context detection for payloads, integrated CSP simulation, and exercises that emulate modern single-page application routing and templating, thereby narrowing the gap between lab practice and production remediation.

References

[1] Cloudflare Learning. *Cross-site scripting (XSS)*. https://www.cloudflare.com/learning/security/threats/cross-site-scripting/
[2] Wikipedia. *Cross-site scripting*. https://en.wikipedia.org/wiki/Cross-site_scripting
[3] OWASP. *Cross-site Scripting (XSS)*. https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)
[4] MDN Web Docs. *Debugger*. https://developer.mozilla.org/en-US/docs/Tools/Debugger
[5] MDN Web Docs. *Debugger UI Tour*. https://developer.mozilla.org/en-US/docs/Tools/Debugger/UI_Tour
[6] Permadi. *URL Encoding tutorial*. https://www.permadi.com/tutorial/urlEncoding/
[7] R manuals. *URLencode*. https://stat.ethz.ch/R-manual/R-devel/library/utils/html/URLencode.html