Lab 8.pdf

Sonal Chandrakant Hasbi

I. Introduction

This lab explores Cross-Site Scripting (XSS) in WebGoat 2023.8, covering reflected, stored, and DOM-based variants. I executed 12 progressively realistic tasks to understand how different contexts (HTML body, attributes, JavaScript strings, URLs, and DOM sinks) influence exploitability and how modern defenses (sanitization, contextual output encoding, and CSP) mitigate risk. The goal was to (1) successfully demonstrate XSS where intended, (2) articulate why each payload works, and (3) summarize effective mitigations

II. Tools & Environment

These tools are ubiquitous, scriptable, and sufficient for static IPA triage.

1) WebGoat 2023.8 for vulnerable scenarios and scoring.
2) Browser DevTools (Console/Network) to observe reflections, hash/query usage, and confirm payload execution.
3) URL encoding utilities (built-in browser encoding/quick cheats) to encode special characters when required.

III. Methodology

1. Checked if Cookies matched by opening another tab of webgoat on the browser and went to developer tools to type : alert(document.cookie); and verified that both the cookies were same

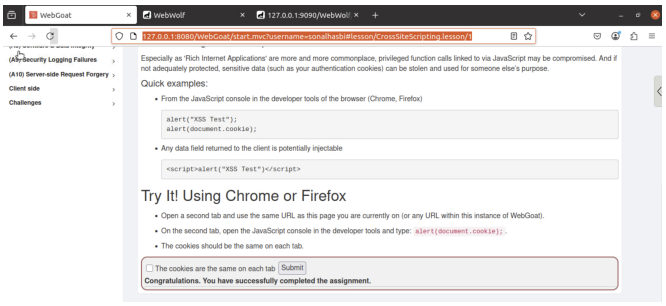

Fig. 1: Checked if cookies matched

2. Reflected XSS by using "><script>alert(1)</script> on the field along with the number mentioned - as shown in Fig. 2

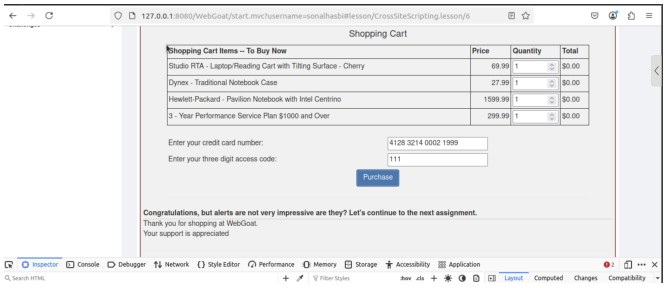

Fig. 2: Reflected XSS

3. Identified potential for DOM based XSS by going to the developer tools and checking all the route configurations on the debug section to get : start.mvc#test/ as the answer to the route and to crack the potential to send in an exploit through that route.

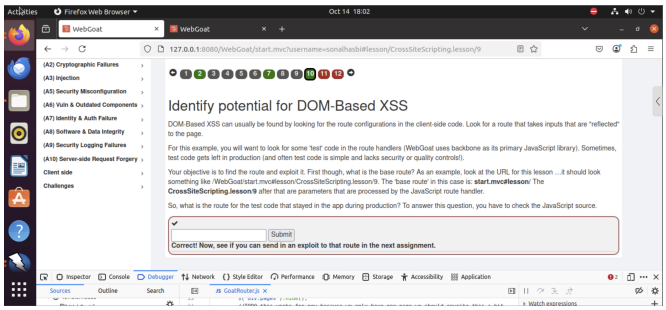

Fig. 3: potential DOM based XSS

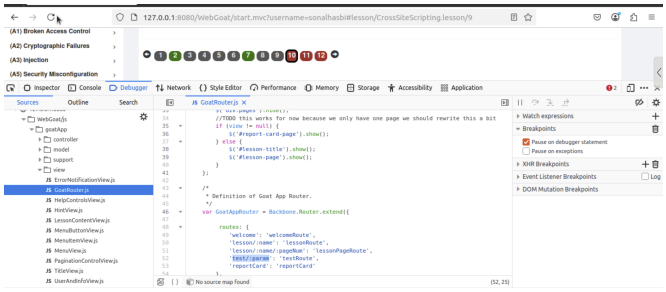

Fig. 4: Checking routes for potential DOM based XSS

3. DOM Based XSS - opened a new tab - used the previous results - start.mvc#test/ to add to the URL along with - webgoat.customjs.phoneHome() and then opened that to use developer tools to get the random number to crack the task.

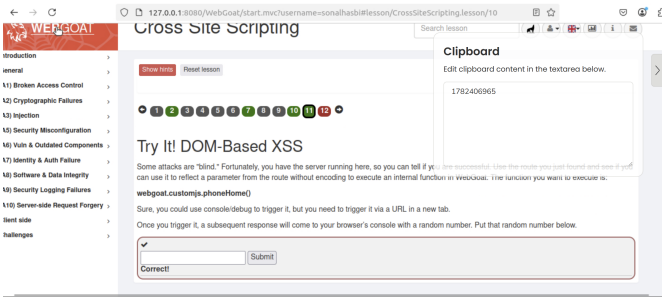

Fig. 5: DOM based XSS

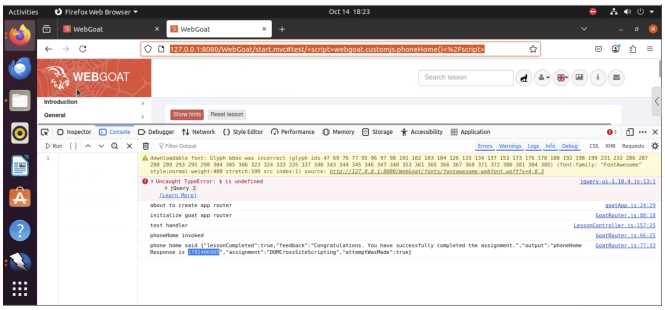

Fig. 6: DOM based XSS

IV. Key Findings

1) XSS is context-dependent: the same characters behave differently in HTML body vs. attributes vs. JS strings.
2) DOM XSS requires no server involvement and is common when developers use innerHTML with untrusted data.
3) Stored XSS poses the highest exposure since it automatically executes for every viewer.
4) Partial filters often fail; defense-in-depth is required.

V. Mitigations & "Advancing the State of the Art"

1) Always encode output per context (HTML, attribute, JS, CSS, URL). Template engines and security libraries should default to safe encoders.
2) Adopt safe DOM APIs (textContent, createElement, setAttribute), avoid innerHTML except with a trusted types policy.
3) Content Security Policy (CSP) with nonces/hashes and no unsafe-inline drastically reduces exploit surface.
4) Static analysis/taint tracking in CI can flag flows from sources (URL/search/hash) to dangerous sinks (innerHTML/eval).
5) Framework defaults (e.g., React's auto-escaping) help—ensure developers don't disable them.

VI. Conclusion

Through 12 tasks I demonstrated reflected, stored, and DOM XSS; learned how context controls exploitability; and validated that contextual encoding + safe DOM + CSP provides robust mitigation. These skills translate directly to real-world secure development and code review, where catching a single unsafe sink can prevent organization-wide compromise.

References

[1] Cloudflare Learning: Cross-Site Scripting (XSS)
[2] Wikipedia: Cross-site scripting
[3] OWASP: Cross-site Scripting (XSS)
[4] URL Encoding tutorials (Permadi; R URLencode)