Cross-Site Scripting (XSS) Vulnerability Analysis and Mitigation

Prepared By

jrdhame@ilstu.edu & lcaculo@ilstu.edu

Introduction & Purpose

Given the growing reliance on web applications as core components of the modern digital world, maintaining a proactive stance toward common security vulnerabilities is essential. Among these vulnerabilities, Cross-Site Scripting (XSS) remains one of the most common and consequential. XSS occurs when an attacker is able to inject malicious client-side scripts, typically JavaScript, into a webpage viewed by other users. When successfully executed, XSS attacks can result in data alteration, session hijacking, and the damage and stealing of sensitive information.

The objective of this project was to bridge the gap between theoretical knowledge of web application security and practical application. Specifically, the project aimed to develop hands-on experience in the comprehensive process of identifying, exploiting, and mitigating XSS vulnerabilities across multiple environments.

The project's scope was extensive and included:

- Lab Setup: Using Burp Suite and the bWAPP (Bee-box Web Application) platform in a Windows 11 virtual machine on Proxmox provided by Illinois State University.
- Exploitation: Recognizing and carrying out effective attacks against two main types of XSS: Stored, and Reflected.
- Mitigation: Putting countermeasures in place and testing them, such as a strong content security policy, context-specific output encoding, and server-side input validation.

Burp Suite Community Edition was used for traffic interception, manipulation, and payload testing, while bWAPP running on XAMPP provided a structured vulnerable environment. The remainder of this report outlines the methods used, the mitigation strategies implemented, and a technical analysis of the project’s findings.

Technical Implementation

The project followed a structured, hands-on penetration testing process, from environment setup to creating payloads that could bypass weak or missing security controls.

Lab Environment and Tool Configuration

The project environment was constructed to simulate a real-world vulnerable application setup:

- Web Server: To host the bWAPP application, XAMPP was installed and configured, offering the required Apache and MySQL services.

- Target Application: With multiple adjustable levels of difficulty for XSS testing, bWAPP was used as the purposefully vulnerable target.
- Interception Proxy: Between the browser and the bWAPP application, Burp Suite Community Edition was configured as an HTTP proxy. This made it possible to modify payload requests, intercept all web traffic, and analyze server responses, all of which are essential for finding and taking advantage of online vulnerabilities.

Exploitation of XSS Variants

For each of the two types of vulnerabilities, a different approach was needed for exploitation:

Reflected XXS

Reflected XSS occurs when a malicious script is injected through an HTTP request, and the application immediately includes that payload in its response.

- Procedure: We located and selected the reflected XSS challenge in bWAPP with the security level set to low. After clicking “hack” on the bWAPP, we entered a simple test payload (firstname = “Hello” and lastname = “World”), and we intercepted the resulting GET request in Burp Suite. By reviewing the server’s response, we identified the injection point and confirmed that the input was being reflected directly into the HTML body without proper encoding.
- Proof-of-Concept Payload: Instead of using a JavaScript alert, we used a simple test payload that replaced the “firstname” and “lastname” fields with the values “test” and “test” respectively. The application reflected this input directly into the returned HTML without encoding, confirming that user-controlled data was being inserted into the page unsafely.
- Payload Execution: As soon as the crafted request was submitted, the modified values appeared in the returned webpage. This immediate reflection demonstrated how an attacker could manipulate page content or insert a malicious script using the same injection point.

Stored XXS

Stored XSS is the most serious form of XSS because the injected payload is permanently stored on the server and delivered to every user who accesses the affected page.

- Procedure: We located and selected the stored XSS challenge in bWAPP with the security level set to low. After clicking “hack” on the bWAPP, we targeted the blog entry input field. Instead of typing the payload directly into the webpage, we intercepted the POST

request in Burp Suite, modified the blog entry value to include our script, and then resent the request. Burp Suite confirmed that the modified input was being stored in the server's data store.

- Proof-of-Concept Payload: The payload we inserted into the intercepted request replaced the original blog entry text with the script "<script>alert('Stored XSS');</script>", allowing us to test whether the application would store and later execute this code.
- Payload Execution: When we returned to the page and submitted a new blog entry, the stored payload was loaded and executed by the browser, triggering a pop-up displaying the message "Stored XSS." This confirmed that the script had been successfully saved on the backend and executed for any user viewing the page.

Justification & Analysis

The exploitation stage demonstrated why XSS is still a serious risk in web applications. The application was processing user-controlled input without appropriate validation and returning it straight to the page without encoding, as shown by both reflected and stored XSS. This demonstrated that the program made the assumption that all user input was secure, which is uncommon in real-world settings. Attackers can exploit this assumption to run scripts on users' browsers, alter page content, or steal session data. The mitigation phase was created to use a layered defense to address these vulnerabilities, making sure that several defenses cooperate to lower risk.

Input Validation

Because it stops malicious content from ever entering the system, server-side input validation was selected as the first layer. The application must accept a script or unsafe characters as legitimate input for both reflected and stored XSS. We lessen the possibility that any malicious code will enter the database or be reflected back to users by verifying input on the server to make sure that only permitted characters and data formats are processed. Server-side validation is an essential control for real-world applications because, in contrast to client-side validation, it cannot be bypassed with straightforward browser changes or interception tools like Burp Suite. Furthermore, since attackers can frequently come up with creative ways to get around blacklists, whitelisting, explicitly defining what is acceptable, is more effective than blacklisting known bad characters.

Output Encoding

Even with input validation, no program can ensure that all stored data is secure in every situation. As a second line of defense, output encoding converts potentially hazardous characters into safe representations prior to the browser rendering them. For instance, changing < and > to

“<” and “>” guarantees that user input is shown as text instead of being run as code. This is especially crucial for stored XSS, since a malicious payload may stay in the database and eventually impact several users. Additionally, context-specific encoding makes sure that scripts cannot execute in different areas of the page, like inside JavaScript blocks or HTML attributes. Therefore, even in the event that a malicious payload is able to get into the system, output encoding lowers the risk of XSS and supports input validation.

Content Security Policy

Content Security Policy (CSP) was implemented as the final layer of defense at the browser level to guard the application. Even if a malicious script bypasses input validation and output encoding for some reason, CSP regulates what scripts will be executed by the browser. By enforcing rules, such as allowing only scripts from the application's domain and blocking inline scripts, CSP reduces the impact of injected code. It also covers some attack scenarios that input validation or encoding alone may not cover, such as complex, multi-step exploits, or scripts loaded from attacker controlled domains. Including CSP exemplifies the principle of defense-in-depth, stating that no layer is entrusted fully, and the system is much safer in case some layer fails.

Effectiveness

Re-testing the application after the implementation of these controls showed that mitigations were effective. The attempts to inject the same payloads from the exploitation phase were either rejected by server-side input validation or appeared as harmless text due to output encoding. Stored XSS payloads were no longer executed in users' browsers, and reflected XSS was also rendered harmless. Finally, CSP acted as a last line of defense wherein, even if a payload somehow managed to bypass all other defenses, it would still be prevented from executing within the browser. This testing validated the role of layered defenses and confirmed that the mitigation strategy undertaken by the project successfully addressed the vulnerabilities identified.

Overall Justification

Input validation, output encoding, and CSP together provide defense in depth at three different points: at the point of entry, at render time, and at browser execution time. Each layer addresses a different aspect of the XSS threat and makes it significantly harder for an attacker to succeed. This approach reinforces best practices in web security by treating all input as hostile and applying multiple safeguards, rather than relying on a single control that could fail. It also highlights the value of practical, hands-on testing for understanding vulnerabilities and ensuring that mitigations are effective in real-world scenarios.

Conclusion

This project successfully demonstrated the risks and practical implications of XSS in web applications. By exploiting both reflected and stored XSS vulnerabilities in a controlled environment, we gained firsthand experience in identifying injection points, crafting payloads, and observing how unvalidated input can be manipulated to execute scripts in a user's browser. The hands-on process reinforced the theoretical understanding of XSS and highlighted the different levels of risk associated with reflected versus stored vulnerabilities.

The mitigation phase revealed the importance of a layered, defense-in-depth approach to security. Server-side input validation effectively prevented malicious content from entering the application, output encoding neutralized potentially unsafe data before it was rendered, and a restrictive CSP provided a final layer of security at the browser level. Re-testing the vulnerabilities confirmed that these measures collectively protected the application, demonstrating that multiple complementary controls are essential for effective web security.

Overall, this project emphasized the need to treat all user input as potentially hostile and to implement multiple layers of defense to reduce the likelihood of exploitation. Beyond technical mitigation, the experience highlighted the value of practical testing in reinforcing security concepts and preparing for real-world application security challenges. Future work could extend this methodology to additional types of XSS, as well as other common web vulnerabilities, further strengthening an organization's security posture.