Comprehensive Analysis of Cross-Site Request Forgery (CSRF) Vulnerabilities in PortSwigger Labs

Executive Summary

This report details the methodologies and findings from a series of penetration tests conducted on PortSwigger's Web Security Academy labs, specifically focusing on various forms of Cross-Site Request Forgery (CSRF) vulnerabilities. The objective was to systematically identify, exploit, and document these vulnerabilities, providing insights into their impact and the techniques employed for their exploitation. This analysis is presented from the perspective of a seasoned penetration tester, emphasizing practical approaches and the strategic thinking involved in navigating complex CSRF scenarios.

Introduction to Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF), also known as XSRF, Sea-Surf, or Session Riding, is a type of malicious exploit of a website where unauthorized commands are transmitted from a user that the web application trusts. Unlike Cross-Site Scripting (XSS), which exploits the trust a user has in a particular site, CSRF exploits the trust that a site has in a user's browser. A successful CSRF attack can force an end-user to execute unwanted actions on a web application in which they're currently authenticated. This can include changing email addresses, transferring funds, or even changing passwords, all without the user's knowledge or consent. CSRF attacks often leverage social engineering to trick victims into clicking a malicious link or visiting a compromised website.

Lab 1: CSRF where token is tied to non-session cookie

Objective

This lab presented a scenario where a web application implemented a CSRF token, but its validation mechanism was flawed, specifically by tying the token to a non-session cookie. The objective was to exploit this weakness to bypass the CSRF protection and execute an unauthorized action, such as changing a user's email address.

Methodology

Initial analysis of the application's functionality, particularly the email change feature, revealed the presence of two fields involved in CSRF protection. Attempts to modify either of these fields independently resulted in an

invalid CSRF token error. This indicated that both fields were part of the CSRF protection mechanism and that a simple manipulation of one field was insufficient.

The critical insight came from considering the possibility that the CSRF token, despite being present, might not be strictly tied to the user's active session. If a valid CSRF token could be obtained from another user or a different context and then reused, the application would be vulnerable. This hypothesis formed the basis of the exploitation strategy.

To test this, a search functionality within the application was identified as a potential injection point. Intercepting the search request using Burp Suite revealed an interesting behavior in the response headers. Specifically, the `Set-Cookie` header for `LastSearchTerm` was observed. This cookie was being set with the value of the search term, and importantly, it included the `Secure` and `HttpOnly` flags. This suggested a potential for injecting arbitrary cookie values through the search functionality.

The breakthrough involved crafting a search query that would inject a valid `csrfKey` cookie into the user's browser. The goal was to make the server respond with a `Set-Cookie` header that included a `csrfKey` with a value controlled by the attacker, and crucially, with `SameSite=None` to ensure it could be sent cross-site. The crafted search query looked like this:

`/?search=123; Secure; HttpOnly%0d%0aSet-Cookie: csrfKey=gUGRH16AyA7mYB4Ref66U9ca2AnaEDbt; SameSite=None`

This payload leverages a carriage return and newline (`%0d%0a`) to inject a new `Set-Cookie` header. When this search query was submitted, the server responded by setting the `csrfKey` cookie with the attacker-controlled value and the `SameSite=None` attribute. This effectively bypassed the CSRF protection, as the attacker could now control the `csrfKey` cookie in the victim's browser.

With the ability to control the `csrfKey` cookie, the final step was to craft a malicious HTML page (Proof of Concept - PoC) that would perform the unauthorized action. This PoC included a form targeting the email change functionality, pre-filled with an attacker-controlled email address and the now-controlled `csrfKey`. An `<img>` tag with an `onerror` event was used to automatically submit the form when the image failed to load, ensuring the attack was silent and automatic.

```
<html>
<!-- CSRF PoC - generated by Burp Suite Professional -->
<body>
<form action="https://[LAB_ID].web-security-academy.net/my-account/change-email" method="POST">
<input type="hidden" name="email" value="attacker-email@example.com" />
<input type="hidden" name="csrf" value="e3d8CWsAtD32K6lZ8cCU8KGdiZLlgohM" />
</form>
<img src="https://[LAB_ID].web-security-academy.net/?search=test%0d%0aSet-Cookie: csrfKey=6WzE4NNsOISlxCZ9ghQw43W1gPWe4MxH; SameSite=None" onerror="document.forms[0].submit()" />
</body>
</html>
```

By enticing a victim to visit this malicious page, the form would be submitted with the valid `csrfKey` (injected via the `Set-Cookie` header), leading to the unauthorized email change. This successfully solved the lab.

Impact

This vulnerability demonstrates a critical flaw in CSRF protection mechanisms where the token is not adequately bound to the user's session. By manipulating non-session cookies, an attacker can bypass CSRF defenses, leading to unauthorized actions being performed on behalf of the victim. The impact can range from account compromise (e.g., changing email/password) to financial fraud or data manipulation, depending on the sensitive actions available within the application.

Lab 2: CSRF where token is duplicated in cookie

Objective

This lab presented another variation of a flawed CSRF protection mechanism, where the CSRF token was duplicated in both a form field and a cookie. The objective was to identify how this duplication could be exploited to bypass the CSRF protection and execute an unauthorized action.

Methodology

Initial analysis of the email change functionality revealed that the CSRF token was present in two locations: a hidden form field and a cookie (specifically, `csrf-key`). Attempts to tamper with only one of these values resulted in an invalid token error, indicating that the application was indeed validating both. However, a crucial observation was made: if both the hidden form field token and the cookie token were changed to the *same* arbitrary, but valid-looking, value, the request would succeed. This suggested that the application was not validating the *randomness* or *uniqueness* of the token, but merely ensuring consistency between the two duplicated values.

This finding rendered the CSRF token effectively useless as a protective measure. Since the attacker could control both instances of the token (the one sent in the form and the one sent in the cookie), they could simply set them to an arbitrary, identical value and bypass the protection. The next step was to find an injection point to set the `csrf` cookie in the victim's browser. Similar to the previous lab, the search functionality was identified as a suitable vector due to its ability to reflect input into a `Set-Cookie` header.

The strategy involved using the search functionality to inject a `csrf` cookie with an attacker-controlled value. The crafted search query would look like this:

```
/?search=123; Secure; HttpOnly%0d%0aSet-Cookie: csrf=lol; SameSite=None
```

This payload would force the victim's browser to set a `csrf` cookie with the value `lol` and `SameSite=None`. Once this cookie was set, the attacker could then craft a malicious HTML page containing a form that would submit to the email change functionality. The hidden CSRF token field in this form would also be set to `lol`, ensuring consistency with the injected cookie.

```
 <html>
<!-- CSRF PoC - generated by Burp Suite Professional -->
<body>
<form action="https://[LAB_ID].web-security-academy.net/my-account/change-
email" method="POST">
<input type="hidden" name="email" value="lo@lo.com" />
<input type="hidden" name="csrf" value="lol" />
<input type="submit" value="Submit request" />
</form>
<img src="https://[LAB_ID].web-security-academy.net/?
search=123%3b%20Secure%3b%20HttpOnly%0d%0aSet-
Cookie:%20csrf=lol%3b%20SameSite=None"
onerror="document.forms[0].submit()">
</body>
</html>
```

By tricking a victim into visiting this malicious page, the form would be submitted with both the form field and cookie `csrf` values set to `lol`, bypassing the flawed CSRF protection and successfully changing the victim's email address. This successfully solved the lab.

Impact

This lab highlights a common implementation flaw in CSRF protection where the token's randomness or uniqueness is not properly enforced. When a token is duplicated and its validity is only checked by comparing the two instances, an attacker can easily bypass the protection by providing consistent, albeit arbitrary, values. The impact is similar to other CSRF vulnerabilities, allowing unauthorized actions to be performed on behalf of the victim.

Conclusion and Recommendations

The PortSwigger CSRF labs provide an excellent practical demonstration of various Cross-Site Request Forgery vulnerabilities and the nuances involved in their exploitation. These labs underscore that merely implementing a CSRF token is insufficient; the effectiveness of the protection hinges on its correct and secure implementation. As an experienced penetration tester, the key takeaways from these exercises are clear:

1. **Robust CSRF Token Implementation:** CSRF tokens must be truly random, unique per user session, and securely tied to the user's session. They should be

validated on the server-side for every state-changing request. Tokens should not be guessable or predictable.

2. **Strict Token Validation:** The application must rigorously validate the CSRF token, ensuring it matches the expected value and is associated with the current user's session. Any discrepancy should result in the rejection of the request.
3. **SameSite Cookie Attribute:** Implement the `SameSite` cookie attribute with appropriate values (`Lax` or `Strict`) to prevent browsers from sending cookies in cross-site requests. While not a complete defense against all CSRF attacks, it significantly reduces the attack surface.
4. **Referer Header Validation:** For critical actions, consider validating the `Referer` header to ensure that the request originated from the application's own domain. However, this should not be the sole defense, as the `Referer` header can sometimes be spoofed or absent.
5. **Double Submit Cookie Pattern (with caution):** While the labs demonstrated flaws with duplicated tokens, the double submit cookie pattern, when implemented correctly (i.e., the token in the form is a hash of the token in the cookie), can be an effective defense. However, careful implementation is crucial.
6. **User Education:** Educate users about the risks of clicking suspicious links and visiting untrusted websites, as social engineering is often a component of successful CSRF attacks.
7. **Continuous Security Testing:** Regular penetration testing, including dedicated CSRF testing, is crucial to identify and remediate these vulnerabilities. Automated tools can assist, but manual testing and creative bypass techniques are often necessary to uncover subtle flaws.

In conclusion, CSRF remains a significant threat to web applications. The PortSwigger labs serve as an invaluable resource for understanding the intricacies of these attacks and for developing the skills necessary to defend against them. A proactive and multi-layered security strategy, encompassing secure coding practices, robust token implementation, and continuous vigilance, is paramount in defending against CSRF.