# Cross-Site Request Forgery (CSRF) and Cross-Site Scripting (XSS)

Logan Brooks

IT241

Web security is one of the most critical aspects of modern application development, given the increasing sophistication of cyber threats. Among the various vulnerabilities that web applications face, Cross-Site Request Forgery (CSRF) and Cross-Site Scripting (XSS) are two particularly dangerous and prevalent threats. Both attacks exploit the inherent trust between a user's browser and the website they are interacting with, but they operate in distinct ways and require different approaches for mitigation. Web developers need to be aware of potential vulnerabilities, as well as how to prevent them to ensure their web applications are not prone to cross-site attacks.

## Cross-Site Request Forgery (CSRF)?

Cross-Site Request Forgery (CSRF) is an attack that tricks a victim's browser into making an unwanted request to a web application where the victim is authenticated. Unlike other attacks like XSS, CSRF does not attempt to directly inject malicious content into a site. Instead, it exploits the trust a website has in the user’s browser and performs an action on behalf of the victim without their knowledge or consent. The attacker essentially "forces" the victim's browser to submit requests that appear legitimate but are, in fact, malicious. A CSRF attack can perform any action that requires a user's authentication, such as changing account settings, transferring funds, or altering administrative privileges. Depending on the user that is affected by an attack, it can have drastic consequences. For example, an administrator being successfully targeted by a CSRF attack could result in the entire application becoming irreparably damaged. An Important aspect to note about CSRF attacks is that the attacker does not receive any data from the action they triggered, which makes it harder to detect.

A real-world example of this type of attack was a 2008 uTorrent exploit, which used the fact that a web console accessed at localhost:8080 allowed actions to be executed through a GET request.

### How Does CSRF Work?

In a typical CSRF attack, the attacker forges a request to the vulnerable website, embedding it in a link, image, or form that they send to the victim. If the victim is logged into the website, clicking the malicious link or submitting the form causes the browser to send the forged request with the user's session cookies. The server, unable to distinguish the forged request from a legitimate one, processes the action, and the attack succeeds. For example, if a user is logged into their online banking system, an attacker could trick them into clicking a link that transfers money from the user's account to the attacker's account. The browser would automatically include the session cookies, and the bank would process the request as if the user had made it themselves.

### Vulnerabilities

CSRF attacks typically exploit cookie-based session management Web applications rely on cookies to maintain a user's session after they authenticate. Since browsers automatically send session cookies with each HTTP request to the same domain, the attacker can use this behavior to craft requests that carry the victim’s authentication credentials. The attack requires at least one HTTP request to the server and is successful as long as the website has no additional mechanism for validating requests, such as a token or origin header. If there are predictable request parameters (such as an ID or username), these can be guessed by the attacker to execute the attack.

### Prevention

The most effective way to prevent CSRF attacks is by implementing CSRF tokens. These are unique, unpredictable values generated by the server and included in requests that require a state change. The server checks if the token received with the request matches the one associated with the user’s session. If the token does not match, the request is rejected. Though this is the most common and effective way of mitigating CSRF attacks, it can be bypassed if not configured properly. Some applications fail to verify tokens in certain cases (e.g., when a token is missing), which allows attackers to bypass the protection. If an application uses a shared pool of tokens rather than associating them with a specific user session, attackers can impersonate any user by using a valid token from the pool. If the CSRF token is copied into cookies, an attacker could manipulate the cookies to bypass the token validation mechanism.

Another method of prevention is by using the SameSite cookie attribute, which can restrict when cookies are sent with cross-site requests. When properly configured, this prevents cookies from being sent when requests originate from different sites, thus mitigating CSRF. Many modern APIs require custom headers to ensure that requests originate from a legitimate source. For example, by requiring an `X-CSRF-Token` header, the application can check whether the header is present and valid before processing the request. Avoid Using GET Requests for State-Change Operations. GET requests should only be used for retrieving data and not for operations that modify state (e.g., transferring money, changing account details). By using POST or other methods, the attack surface for CSRF is reduced. For particularly sensitive actions requiring user verification—such as CAPTCHA or multi-factor authentication (MFA)—adds an additional layer of defense that can thwart CSRF attacks.

## Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) is a vulnerability that allows attackers to inject malicious scripts into websites. These scripts are then executed in the browser of any user who visits the compromised website. XSS attacks can have serious consequences, including compromising sensitive data, stealing session cookies, redirecting users to malicious sites, or installing malware. XSS essentially breaks the Same Origin Policy, which is designed to separate the data of different websites to prevent such attacks.

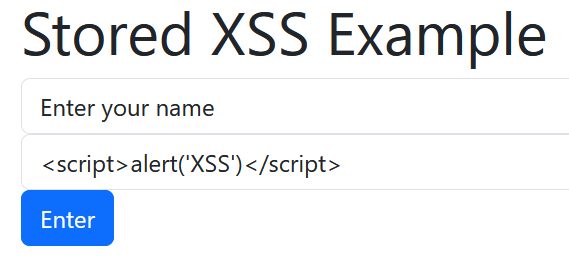
### How Does XSS Work?

XSS attacks occur when an application includes untrusted input from users in its output without properly validating or encoding it. This allows the attacker to inject malicious JavaScript into the page, which gets executed when another user visits the page. The attacker can use various means to inject this script, such as submitting malicious content through forms, URLs, or even via cookies.

There are two main types of XSS attacks, the first of which being reflected XSS. In this type of attack, the malicious script is immediately reflected by the server in its response. The attacker sends a URL with a malicious script embedded, and when the user clicks the link, the script is executed. Reflected XSS typically occurs in search boxes or URL parameters and requires user interaction to execute the attack.

The next type of attack, stored XSS, is more dangerous because the injected script is stored on the server (e.g., in a database) and is executed every time a user visits the affected page. Attackers often exploit forums, comment sections, and other user-driven content forms to inject malicious scripts into the databases.

In this example, the attacker enters their malicious code into an input that is submitted to a database. If the attacker manages to get a user to visit the page with the data entry containing this code, it will run on the user’s local browser.

A screenshot of a computer program

Description automatically generated

### Prevention

The most effective way to mitigate XSS vulnerabilities is by output encoding. This is done by making sure that any user input that is included in HTML, JavaScript, or other code contexts is treated as data rather than executable code, leaving no room for malicious script injections. All user inputs should be validated and sanitized to prevent malicious code from being submitted. This can include checks for unexpected characters or patterns that may indicate malicious code. In addition to these methods, custom HTTP headers like Content-Type and X-Content-Type-Options ensure that browsers interpret the response correctly, reducing the likelihood that user input will be interpreted as code. If you are using a modern framework such as Angular or React, it may come with built-in protections against XSS, such as automatic encoding or auto-escaping. It's important to understand these protections and ensure they are correctly configured, as they cannot be relied on to protect your web application from all attacks. Finally, a strong Content Security Policy (CSP) restricts the resources (like scripts) that can be executed on a webpage, reducing the attack surface for XSS. CSP helps prevent the execution of malicious scripts by blocking unauthorized scripts from running.

### Testing for XSS

Testing for XSS involves submitting various inputs to identify vulnerable points in the application. Tools such as OWASP ZAP (and their security cheat sheet), Burp Suite and Nikto can be used to scan for vulnerabilities. Testing should include checking both server-side inputs (e.g., form submissions) and client-side inputs (e.g., cookies or JavaScript events). DOM-based XSS is particularly difficult to detect because it involves client-side manipulation of data, so reviewing JavaScript code is essential.

CSRF and XSS are two of the most dangerous vulnerabilities in web security, but they can be mitigated with a combination of secure coding practices, robust session management, and proper input validation. By understanding how these attacks work and implementing best practices like CSRF tokens, output encoding, and input sanitization, web developers can significantly reduce the risk of these attacks. However, continuous testing, awareness, and vigilance are crucial in the ever-evolving landscape of web security.

<https://brightsec.com/blog/cross-site-request-forgery-csrf/>

<https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html>

[https://portswigger.net/web-security/cross-site-scripting - what-is-cross-site-scripting-xss](https://portswigger.net/web-security/cross-site-scripting#what-is-cross-site-scripting-xss)

<https://owasp.org/www-community/attacks/xss/>

<https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html>

<https://nvd.nist.gov/vuln/detail/CVE-2008-6586>