International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 08 Issue: 10 | Oct - 2024 SJIF Rating: 8.448 ISSN: 2582-3930

Enhancing Web Security: Token-Based Solutions to Prevent CSRF Attacks

Prof. K. N. Hande, Mehak Khan, Mansi Bhujade

1 Professor. k. N. Hande, Department of Computer Science and Engineering, Priyadarshini Bhagwati College of Engineering, Nagpur, Maharashtra, India

2 Mehak Khan, Department of Computer Science and Engineering, Priyadarshini Bhagwati College of Engineering, Nagpur, Maharashtra, India

3 Mansi Bhujade, Department of Computer Science and Engineering, Priyadarshini Bhagwati College of Engineering, Nagpur, Maharashtra, India

---

Abstract - A Cross-Site Request Forgery (CSRF) is a security vulnerability in web applications where an attacker tricks a user into performing unintended actions on a different website without their knowledge. subverts the user's trust in a single site and causes the user to perform actions that he or she did not intend to take on that domain. When logged in, an attacker may act even when the user does not intend for them to do so, such as making illegal transactions, modifying user configurations, or deleting information. As discussed in an understanding of Cross-Site Request Forgery: Attacks and Countermeasures, a CSRF attack is dangerous because it takes advantage of the trust the victim has placed in the web application, making it difficult to detect. The This paper explores CSRF attacks by examining their operational principles, common tactics used by attackers, and potential prevention techniques, such as token implementation, Same-site cookie policies, and secure coding practices. As the dependency on web applications continues to grow, the need to encourage and enforce proper security practices against CSRF attacks becomes increasingly important.

***Key Words*:** Cross-Site Request Forgery, Web security vulnerability, Anti-CSRF tokens, Defensive coding against CSRF Session validation, Forgery of requests

1.INTRODUCTION

Cross-Site Request Forgery (CSRF) is a web application security flaw that exploits the trust users have in a specific website. In this type of attack, an attacker tricks a logged-in user into performing actions on the website that they did not intend to, or causes the user to unknowingly perform actions. These actions could range from changing user profile settings to making a bank transfer. The attacker takes advantage of the fact that the user is already logged in and bypasses necessary security steps. This makes CSRF difficult to detect, as it appears to be legitimate activity by an authorized user. With the rise of web-based applications, it is crucial for users and developers to understand how CSRF works and the methods to prevent it.

2. Working of CSRF

Cross-Site Request Forgery (CSRF) works by taking advantage of the trust that web applications place in user requests. have in a user's browser. The attack begins when a user authenticates on a trusted website, such as a banking platform, where they log in with their credentials. Upon successful login, the server establishes a session, typically storing session information in cookies. As the user continues to navigate the site, they remain logged in, allowing them to perform various actions without needing to re-enter their credentials. Meanwhile, an attacker crafts a malicious link or webpage that includes a request designed to execute an action on the trusted site, such as initiating a money transfer or altering account settings. When the user inadvertently clicks this link or visits the malicious page—often through social engineering tactics—they unwittingly send a forged request to the trusted application. Because The user's browser automatically attaches the session cookies to the request., the web application processes the action as if it were a legitimate request from the authenticated user. This process occurs without any visible indication to the user, making CSRF attacks particularly insidious, as they can lead to unauthorized transactions or data modifications without the victim's knowledge.

2.1Methodology of CSRF Attacks

The methodology of Cross-Site Request Forgery (CSRF) attacks involves several distinct phases that exploit the inherent trust between a user and a web application. Initially, the attack commences when a user logs into a secure web application, such as a financial institution. During this login process, the server authenticates the user and establishes a session, typically maintained through cookies stored in the user's web browser.

Once authenticated, the user retains an active session, allowing them to execute various actions without re-entering their credentials. Meanwhile, an attacker prepares a malicious webpage or an email that contains a forged request, which might be designed to perform sensitive actions on the trusted site, such as transferring funds or modifying account settings.

To enhance the effectiveness of the attack, the perpetrator often employs social engineering tactics. This may include crafting phishing emails that entice the user to click a seemingly harmless link or embedding the exploit within a legitimate-looking webpage. When the victim clicks the link or interacts with the malicious content, the attack is triggered.

At this point, the attacker's code sends a forged HTTP request (either GET or POST) to the targeted web application. Because The user's browser automatically sends the relevant session cookies along with the request..the trusted application assumes that the action is being performed by the authenticated user. As a result, the request is processed without any indication of malicious intent.

Crucially, the execution of a CSRF attack often goes unnoticed by the victim, who may not realize that unauthorized actions have been taken until they later review their account. This characteristic makes CSRF a particularly insidious threat, emphasizing the need for robust preventive measures to protect users from such vulnerabilities.

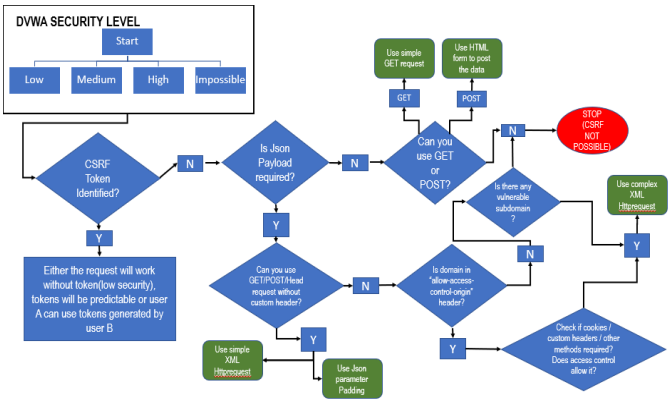

Fig-1: Cross Site Request Forgery Architecture

The flowchart outlines a methodical approach for assessing the feasibility of executing a Cross-Site Request Forgery (CSRF) attack on a web application. The steps are based on various security conditions that the application may have in place. Below is a simplified explanation of the flow:

2.2DVWA Security Level

The process starts by identifying the security level of the web application, which can range from Low, Medium, High, to Impossible. Each level increases the difficulty of exploiting a CSRF vulnerability, with "Impossible" being fully secure.

The next step evaluates whether a CSRF token is implemented, as the presence of this token is a key defense mechanism. If a token is present, the flow determines if it can be exploited, which may occur if the token is predictable or improperly managed. If no CSRF token is found, the analysis shifts to examining the request method and payload. If a JSON payload is not required, simpler methods such as GET or POST requests can be used to conduct the attack. However, if JSON is required, the complexity increases, potentially necessitating advanced techniques like JSON parameter padding or complex request formatting.

It checks if there is a vulnerable subdomain or if the target domain's Cross-Origin Resource Sharing (CORS) policy allows for cross-origin requests. If vulnerabilities in subdomains or CORS settings are found, they may facilitate the attack. Finally, in highly secure environments where more straightforward methods are blocked, the attacker may need to employ advanced techniques, such as using custom headers or manipulating XML Http Requests. Ultimately, the flowchart provides a clear decision-making process to assess the likelihood of a successful CSRF attack, based on the web application's security configuration, highlighting key factors like CSRF token usage, request handling, and cross-origin policies.

2.3Impact of CSRF Attacks

Cross-Site Request Forgery (CSRF) attacks can have severe consequences for both individual users and the web applications they interact with. The primary impact of a CSRF attack lies in its ability to exploit the trust a web application places in an authenticated user. When an attacker successfully performs a CSRF attack, they can force the user's browser to send unauthorized requests to a website where the user is already authenticated, often without the user's knowledge. These unauthorized actions can range from making financial transactions to changing account settings or passwords.

For end-users, CSRF attacks can result in significant financial and personal damage. If, for example, a CSRF attack targets an online banking platform, the attacker may be able to transfer funds from the victim's account without their consent. Similarly, in e-commerce platforms, an attacker could place orders or modify the victim's shopping cart. The violation of personal privacy is another major concern, as CSRF can be used to change user account details, potentially exposing sensitive information to malicious actors.

From the perspective of web applications, the damage caused by CSRF attacks extends beyond user exploitation. A successful attack can undermine the trust between the application and its users is essential for maintaining user engagement, especially on platforms that manage sensitive information, such as financial services and healthcare systems., or social media. When a CSRF vulnerability is exploited, it can lead to reputational damage, loss of user confidence, and in extreme cases, legal liabilities, especially if the platform fails to safeguard user information according to industry standards or regulatory frameworks like the General Data Protection Regulation (GDPR).

In addition, CSRF attacks can disrupt business operations by causing unauthorized changes to an application's settings, corrupting user data, or triggering unintended administrative actions. Attackers might use CSRF to escalate privileges or modify security settings, potentially leaving the web application exposed to further vulnerabilities. Consequently, addressing CSRF vulnerabilities is critical not only for protecting individual users but also for preserving the integrity and security of the entire web application ecosystem.

2.4Real-World Examples of CSRF Exploits

1. Gmail CSRF Vulnerability (2007) -In 2008, Netflix experienced a CSRF vulnerability that allowed attackers to manipulate users' account settings. By exploiting the flaw, an attacker could change the victim's login email address and password, effectively locking the legitimate user out of their own account.
2. uTorrent CSRF Exploit (2008)- Another example occurred in 2008, targeting the popular BitTorrent client, uTorrent. A CSRF vulnerability in the web-based management interface of uTorrent allowed attackers to perform actions such as adding or removing torrent files from the user's download list. Attackers could inject malicious links into websites,

and when users clicked on them, unauthorized commands were sent to the uTorrent client.

3. ING Direct CSRF Incident (2010)- In 2010, a CSRF vulnerability was found in ING Direct's online banking platform in Canada. The flaw allowed attackers to initiate unauthorized money transfers from a user's account by tricking them into visiting a malicious webpage. By exploiting this vulnerability, attackers could automate the transfer of funds without the victim's knowledge. Given the sensitivity of financial information and transactions, this case underscored the critical need for financial institutions to implement strong anti-CSRF measures to protect their users from fraudulent activities.
4. GitHub CSRF Attack (2012)-In 2012, GitHub, a major code repository and collaboration platform, discovered a CSRF vulnerability that allowed attackers to add or delete SSH keys from users' accounts. By exploiting this flaw, an attacker could gain access to a user's repositories, potentially injecting malicious code or stealing private data. Given the large number of users and the sensitive nature of code repositories, this vulnerability posed a serious risk to GitHub's community.

2.5CSRF Tokens: The Primary Defense

Cross-Site Request Forgery (CSRF) tokens are one of the most effective defenses against CSRF attacks. They work by embedding a unique, unpredictable value into each user session or form submission, preventing attackers from forging unauthorized requests on behalf of authenticated users. These tokens ensure that any incoming request to a server contains a valid, session-specific token, which an attacker, without access to the user's session, cannot replicate.

When a user submits a request, the application generates a CSRF token and associates it with that specific user's session or form. The token is then included as a hidden field in forms or appended to URLs in GET or POST requests. The server validates When the server receives the request, it checks the token against the one stored with the user's session. If the tokens match, the request is deemed legitimate; otherwise, if the token is absent or incorrect, the request is rejected as a possible CSRF attack.

A major advantage of CSRF tokens is their uniqueness to each user session., making it extremely difficult for an attacker to predict or forge a valid token. Since CSRF tokens are generated and tied directly to the user's session, attackers cannot include the correct token in their malicious requests, even if they know the structure of the web application.

Another advantage is that CSRF tokens can be used in both synchronous and asynchronous requests (such as AJAX). For modern web applications, where many actions are performed without full-page reloads, implementing CSRF protection for APIs and AJAX calls is crucial to prevent attacks via background requests.

However, proper implementation of CSRF tokens is essential to their effectiveness. For instance, developers must ensure that tokens are truly random and not susceptible to prediction. Tokens should be securely stored and validated on the server side to prevent tampering. Additionally, developers must avoid common pitfalls, such as reusing tokens across sessions or failing to include tokens in all forms or state-changing requests. Inadequate implementation can leave applications vulnerable despite the presence of CSRF tokens.

3. Implementation of CSRF Prevention in Web Applications –

Effectively mitigating Cross-Site Request Forgery (CSRF) attacks in web applications. requires the correct implementation of multiple security measures. CSRF tokens, same-site cookies, and proper request validation are some of the most critical tools for developers to incorporate into their web applications to defend against these attacks.

3.1. CSRF Tokens Implementation

The primary method for preventing CSRF attacks is by utilizing of CSRF tokens. To implement CSRF tokens, web applications must Create a unique, random token for each user session or form. This token must be included in every form or state-changing request sent from the client to the server. For instance, during form submissions, the token can be added as a hidden field, while for AJAX requests, it can be placed in the request headers.

On the server side, the token is validated against the session-specific token stored on the server. If the token does not match or is missing, the request is rejected as a potential forgery. Many modern web development frameworks provide built-in mechanisms for automatically generating, embedding, and validating CSRF tokens. For example, in Django, developers only need to use the {% csrf_token %} tag in forms to embed the token and enable the middleware that verifies it on the server side. Similarly, Laravel automatically provides CSRF protection for any routes that use POST, PUT, DELETE, or PATCH requests, with tokens included in the forms by default.

3.2. Same-Site Cookie Attribute

Another essential approach to CSRF prevention is the Same Site attribute on cookies. When set to Same Site=Strict or Same Site=Lax, this attribute ensures that cookies will not be sent along with cross-origin requests. This prevents attackers from using malicious websites to exploit cookies tied to an authenticated session on another site. For example, if a user is logged into their bank's website and visits a malicious site, the cookies associated with the bank session will not be sent along with any requests from that external site, thus mitigating the potential for CSRF attacks.

Implementing Same Site cookies is a straightforward process. For example, in Express.js (a popular Node.js framework), developers can set this attribute using middleware like express-session to ensure that cookies are protected from cross-origin attacks.

3.3. Double-Submit Cookie Method

The double-submit cookie method is another effective technique for CSRF prevention. In this method, a CSRF token is set as a cookie in the user's browser, and this token is also submitted as a hidden field in forms or as a header in AJAX requests. When the server receives a request, it compares the token in the cookie with the token in the request. If both tokens match, match, the request is allowed; if not, it is rejected.

This method can be implemented without storing the token on the server, which makes it lighter in terms of resource usage. However, developers must ensure that the tokens are transmitted securely, over HTTPS, and that cookies are flagged as secure to prevent interception by attackers.

3.4. Validating Origin and Referrer Headers

Validating the Origin and Referrer headers is another way to defend against CSRF attacks. When a request is sent from a client to the server, the browser typically includes the Origin or Referrer header, which identifies the source of the request. By validating these headers to ensure that they match the expected domain, web applications can block requests originating from unauthorized sites. This technique is particularly effective when used in conjunction with CSRF tokens and same-site cookies.

While this method provides an additional layer of security, it is not foolproof. Some browsers, especially older ones, may not always send Origin or Referrer headers for all types of requests, and these headers can sometimes be manipulated. Therefore, this method should be used as a complementary defense rather than a standalone solution.

3.5. Framework-Specific Implementations

Most modern web development frameworks come with built-in support for CSRF protection. For example:

- **Django**: CSRF protection is enabled by default, and developers only need to include the CSRF token in forms using {% csrf_token %} in templates.
- **Laravel**: The framework automatically applies CSRF protection to all state-changing requests. Developers can disable it for specific routes if necessary but are encouraged to keep it active for maximum security.
- **Spring**: In the **Spring Security** framework for Java, CSRF protection can be enabled through simple configuration settings. The framework handles CSRF token generation and validation automatically, making it easy to implement.
- **ASP.NET Core**: Microsoft's framework also includes built-in CSRF protection, which can be configured in middleware and applied to specific forms or API endpoints.

3.6. Best Practices for CSRF Prevention

To ensure comprehensive protection against CSRF attacks, developers should adhere to a set of best practices:

- **Use HTTPS**: Always use HTTPS to prevent attackers from intercepting or modifying CSRF tokens in transit.
- **Generate Secure Tokens**: Ensure that CSRF tokens are long, random, and unique for each session or request to avoid predictability.
- **Protect All State-Changing Requests**: Apply CSRF protection to all forms and actions that modify the state of the application, including API endpoints and AJAX requests.
- **Regularly Test for CSRF Vulnerabilities**: Incorporate automated tools, such as OWASP ZAP or Burp Suite, to regularly scan web applications for CSRF vulnerabilities.

4.Result: -

4.1 Web Application for the showing the attacks

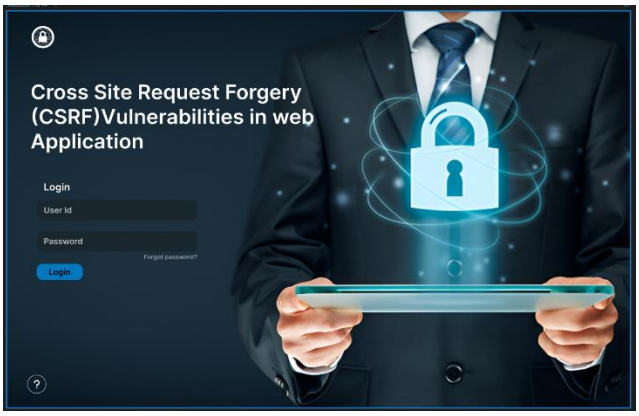

Fig 2 Web application for demonstrating the attacks.

Cross-Site Request Forgery (CSRF) takes advantage of the trust that a web application has in its users. places in a user's browser. In such attacks, a malicious website deceives the user's browser into sending unintended requests to another site where the user is already logged in. To reduce this threat, web applications can adopt a token-based approach in PHP. This method generates a unique token for each user session, embedding it in forms or URL parameters. When a request is made, the server checks the token's validity; if it's missing or incorrect, the request is denied. This ensures that only valid requests from authenticated users are processed, safeguarding the application from CSRF attacks.

4.2. Token Generation technique

The image shows a web page titled "Cross-Site Request Forgery (CSRF) Vulnerability in Web Application," featuring a login form on the left side and the associated HTML code on the right side. The form includes fields for entering the

username and password. Additionally, the HTML code contains a hidden field labeled "csrf_token," which is included to protect against CSRF attacks. The page also includes a warning about the security risks posed by CSRF vulnerabilities.

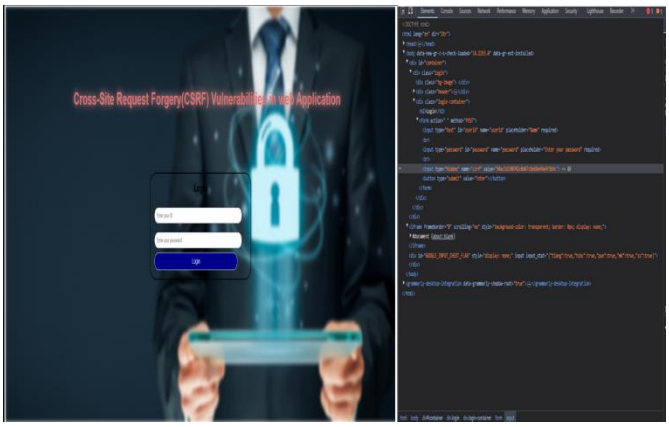

Fig: -3 Token Generation Technique

4.3. How CRSF Attacks works

The image displays a login form susceptible to Cross-Site Request Forgery (CSRF) attacks occur when an attacker could craft a malicious link designed to deceive users into clicking it, potentially granting unauthorized access to their accounts. To mitigate this risk, the form should implement a CSRF token to ensure secure authentication.

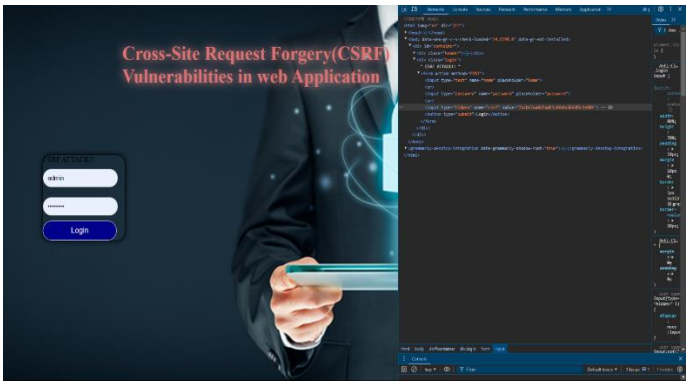

Fig: - 4 How CSRF Attacks happened

5. CONCLUSIONS

Cross-Site Request Forgery (CSRF) attacks present a serious risk to web applications. by exploiting the trust that a site has in a user's browser. These attacks can lead to unauthorized actions being executed on behalf of a user without their consent, potentially compromising sensitive data and user accounts.

To mitigate the risks associated with CSRF, developers should implement robust security measures. These include using anti-CSRF tokens, enforcing same-site cookie attributes, and employing user authentication mechanisms that can validate requests. Additionally, educating users about the risks of CSRF and promoting best practices for web security can further reduce the likelihood of such attacks.

In summary, while CSRF attacks can be detrimental to both users and web services, a proactive approach to security can effectively safeguard against these vulnerabilities, ensuring a safer online experience for all users.

6.ACKNOWLEDGEMENT

We I would like to extend my heartfelt thanks to everyone who played a role in the successful completion of this. work. My appreciation goes to our mentors and peers for their valuable insights and support throughout the research process. Special thanks to the resources and literature that provided foundational knowledge on the topic, enabling me to explore and understand the complexities of CSRF attacks. Your guidance and encouragement have been instrumental in enhancing my understanding and execution of this project. Thank you for being an integral part of my journey.

7. REFERENCES

1. OWASP Foundation. (2023). Cross-Site Request Forgery (CSRF).

2. W3C. (2021). Same Site Cookie Attribute.

3. Klein, A., & Hurst, S. (2020). Web Security for Developers: Building Secure Applications. O'Reilly Media.

4. Garfinkel, S. (2019). Web Application Security*: A* Beginner's Guide. McGraw-Hill Education.

5. Shah, D. (2022). Understanding CSRF Attacks: Prevention and Mitigation Strategies. Journal of Cybersecurity, 12(4), 45-58.

6. SANS Institute. (2020). *CSRF*: A Major Web Application Security Vulnerability.

7. Mowery, K., & Felten, E. W. (2019). Security and Privacy in Web Applications: Lessons Learned from the CSRF Attack Vector. IEEE Security & Privacy, 17(2), 56-63.