**CSRF**

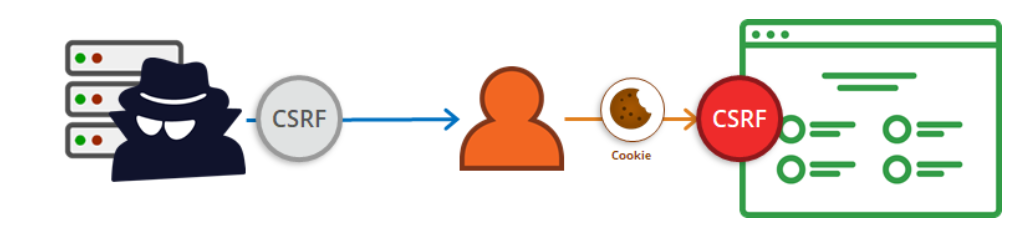
**Cross Site Resource Forgery –** It is an attack that forces the **authenticated** user to submit a request to a web application against which it is authenticated.

A CSRF exploits the vulnerability in a web application if it cannot differentiate between a request generated by user by his consent or not.

Unlike [cross-site scripting](https://en.wikipedia.org/wiki/Cross-site_scripting) (XSS), which exploits the trust a user has for a particular site, CSRF exploits the trust that a site has in a user's browser.

An attacker aim is to force the user to submit a state changing request.

* Submit or delete a record.
* Submit a transaction.
* Purchasing a product.
* Changing password.
* Sending message.



**Why Web application can’t differentiate between legitimate request and malicious request from authenticated user?**

* A general property of web browsers is that they will automatically and invisibly include any cookies used by a given domain in any web request sent to that domain.
* Once a user is authenticated for a web app, many cookies are created. These cookies are included automatically in any web request made by a browser.
* In the event that a user is tricked into inadvertently submitting a request through their browser these automatically included cookies will cause the forged request to appear real to the web server and it will perform any appropriately requested actions including returning data, manipulating session state, or making changes to the victim's account.

**How Websites vulnerable for attack**:

Social engineering platforms are often used by attackers to launch a CSRF attack.

* They trick the victim into clicking a URL that contains maliciously crafted, unauthorized request for a particular web application.
* Once user click, the user’s browser then send this request to a targeted web app.
* If user is in **active** session with targeted web app, the web app treats this new request as an authorized request submitted by user.
* Thus, attacker succeeds in exploiting the web application CSRF vulnerability.

**Example**:

**Bob** has an online banking account on **samplebank.com**. He regularly visits this site to conduct transactions with his friend **Alice**. Bob is unaware that samplebank.com is vulnerable to CSRF attacks. Meanwhile, an attacker aims to transfer $5,000 from Bob’s account by exploiting this vulnerability.

To successfully launch attack:

* The attacker must build an exploit URL.
* The attacker must also trick Bob into clicking URL.
* Bob needs to have an **active** session with sampleBank.com

Let’s suppose bank application has GET request for money transfer request. If Bob wants to transfer money to Alice, the actual URL will be: *GET https://samplebank.com/onlinebanking/transfer?amount=500&accountNumber=213367 HTTP/1.1*

Now maliciously crafted URL will be like: *https://samplebank.com/onlinebanking/transfer?amount=500&accountNumber=425654.*

Now how the attacker can trick Bob into loading malicious URL:

* Including malicious HTML elements onto forms.
* Placing malicious URL on pages that are often access by user while logged in
* Sending malicious URL through mail.

The following is an example of a disguised URL:

*<img src  = “https://samplebank.com/onlinebanking/transfer?amount=5000&accountNumber=425654” width=“0” height= “0”>*

Consider the scenario that includes an image tag in an attacker-crafted email to Bob. Upon receiving it, Bob’s browser application opens this URL automatically—without human intervention. As a result, without Bob’s permission, a malicious request is sent to the online banking application. If Bob has an active session with samplebank.com, the application would treat this as an authorized amount transfer request coming from Bob. It would then transfer the amount to the account specified by an attacker.

**Limitation to CSRF attack:**

* The success of a CSRF attack depends on a user’s session with a vulnerable application. The attack will only be successful if the user is in an **active** session with the vulnerable application.
* An attacker must find a valid URL to maliciously craft. The URL needs to have a state-changing effect on the target application.
* An attacker also needs to find the right values for the URL parameters. Otherwise, the target application might reject the malicious request.

**How application can prevents CSRF?**

1. **By using CSRF tokens:**

* To defeat a CSRF attack, applications need a way to determine if the HTTP request is **legitimately** generated via the application’s user interface.
* The best way to achieve this is through an **anti**-**CSRF token**. A CSRF token is a secure random token (e.g., synchronizer token or challenge token) that is used to prevent CSRF attacks.
* An [anti-CSRF token](https://www.netsparker.com/blog/web-security/protecting-website-using-anti-csrf-token/) is a type of server-side CSRF protection. It is a random string that is only known to the user’s browser and the web application. The [anti-CSRF token](https://www.netsparker.com/web-vulnerability-scanner/vulnerabilities/cookie-values-used-in-anti-csrf-token/) is usually stored inside a session variable. On a page, it is typically in a hidden field that is sent with the request.
* The token needs to be unique per user session and should be of large random value to make it difficult to guess.
* These tokens are inserted within hidden parameters of HTML forms related to critical server-side operations. They are then sent to client browsers.
* It is the application team’s responsibility to identify which server-side operations are sensitive in nature. The CSRF tokens must be a part of the HTML form
* The easiest way to add a non-predictable parameter is to use a **secure hash function** (e.g., SHA-2) to hash the user’s session ID. To ensure randomness, the tokens must be generated by a cryptographically secure random number generator.
* Whenever a user invokes these critical operations, a request generated by the browser must include the associated CSRF token. This will be used by the application server to verify the legitimacy of the end-user request. The application server rejects the request if the CSRF token fails to match the test.

1. **By using Same-site Flag cookies**

The [SameSite flag in cookies](https://www.netsparker.com/blog/web-security/same-site-cookie-attribute-prevent-cross-site-request-forgery/) is a relatively new method of preventing CSRF attacks and improving web application security. Let’s assume that https://attacker.com/ could send a POST request to https://example.com/ together with a session cookie. This session cookie is unique for every user, so the web application uses it to distinguish users and to determine if they are logged in.

If the session cookie is marked as a **SameSite** cookie, it is only sent along with requests that originate from the same domain. Therefore, when https://example.com/index.php wants to make a POST request to https://example.com/post\_comment.php, it is allowed. However, https://attacker.com/ can’t send POST requests to https://example.com/post\_comment.php, since the session cookie originates from a different domain, so it is not sent along with the request.

**Why POST request is less vulnerable to CSRF than GET?**

Tricking a victim into sending a POST request may be slightly more difficult. With a GET request, the attacker only needs the victim to send a URL with all the necessary information. In the case of POST, a request body must be appended to the request. However, an attacker can design a malicious website to include JavaScript that causes the user’s browser to send an unsolicited POST request as soon as the page loads.

**References**:

* <https://www.netsparker.com/blog/web-security/csrf-cross-site-request-forgery/>
* <https://www.acunetix.com/websitesecurity/csrf-attacks/>
* <https://www.synopsys.com/glossary/what-is-csrf.html>
* <https://medium.com/@pandey.ritwik/implementing-csrf-protection-in-java-based-application-using-owasp-1d62466654da>

consumes = MediaType.APPLICATION\_JSON\_VALUE, produces = MediaType.APPLICATION\_JSON\_VALUE