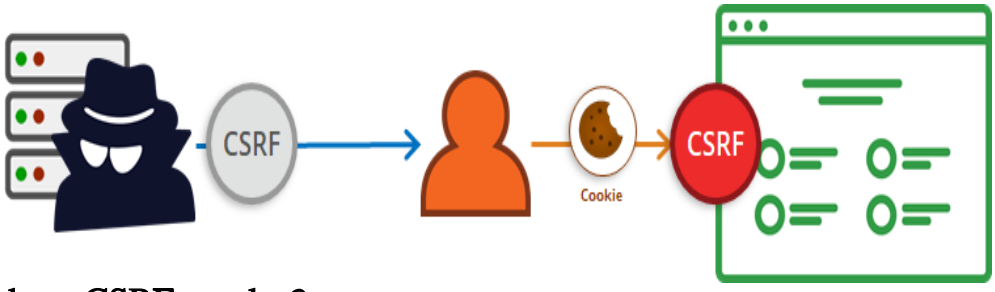
Cross-site Request Forengy – CSRF

I. What is CSRF attack ?

Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated. With a little help of social engineering (such as sending a link via email or chat), an attacker may trick the users of a web application into executing actions of the attacker's choosing. If the victim is a normal user, a successful CSRF attack can force the user to perform state changing requests like transferring funds, changing their email address, and so forth. If the victim is an administrative account, CSRF can compromise the entire web application.



II. How does CSRF work ?

For a CSRF attack to be possible, three key conditions must be in place:

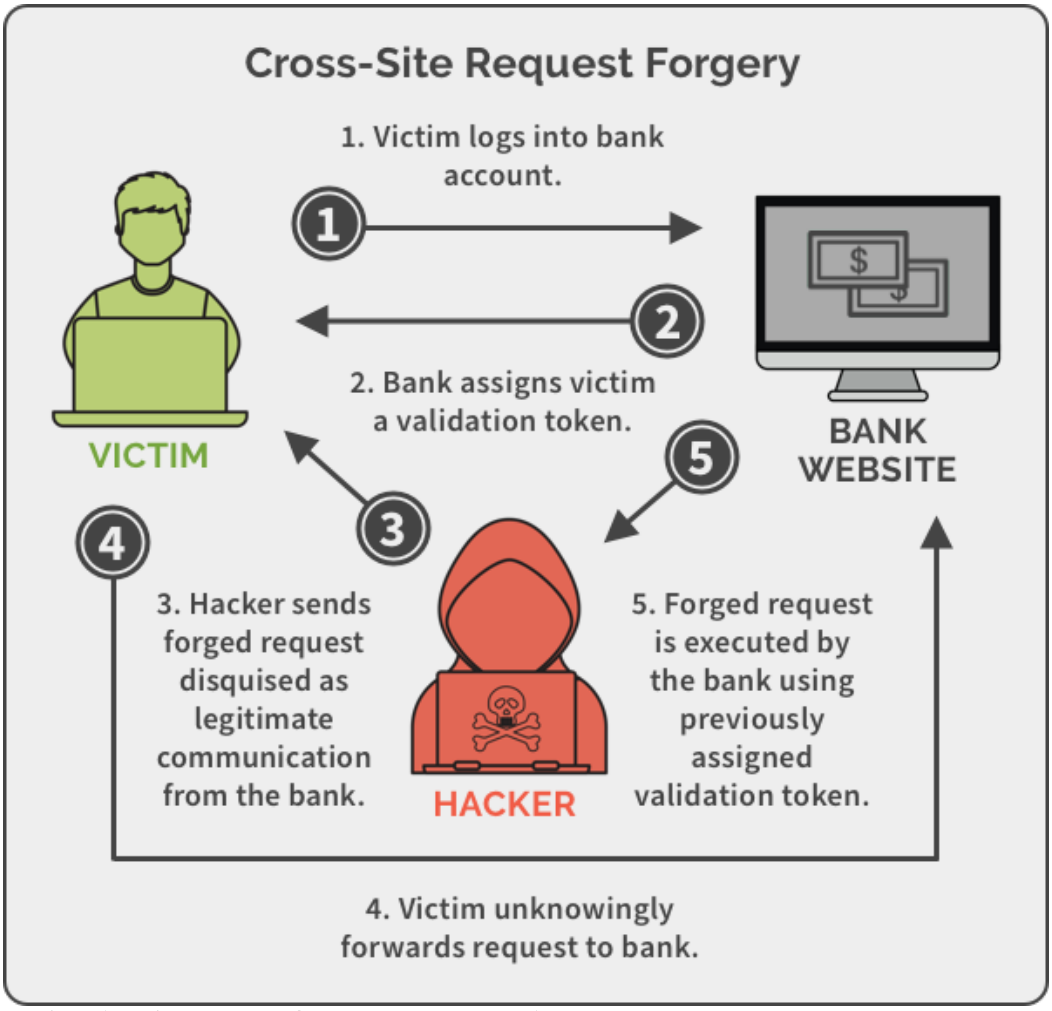
- **A relevant action:** There is an action within the application that the attacker has a reason to induce. This might be a privileged action (such as modifying permissions for other users) or any action on user-specific data (such as changing the user's own password).
- **Cookie-based session handling:** Performing the action involves issuing one or more HTTP requests, and the application relies solely on session cookies to identify the user who has made the requests. There is no other mechanism in place for tracking sessions or validating user requests.
- **No unpredictable request parameters:** The requests that perform the action do not contain any parameters whose values the attacker cannot determine or guess. For example, when causing a user to change their password, the function is not vulnerable if an attacker needs to know the value of the existing password.

The delivery mechanisms for cross-site request forgery attacks are essentially the same as for reflected XSS. Typically, the attacker will place the malicious HTML onto a web site that they control, and then induce victims to visit that web site. This might be done by feeding the user a link to the web site, via an email or social media message. Or if the attack is placed into a popular web site (for example, in a user comment), they might just wait for users to visit the web site.

Note that some simple CSRF exploits employ the GET method and can be fully self-contained with a single URL on the vulnerable web site. In this situation, the attacker may not need to employ an external site, and can directly feed victims a malicious URL on the

vulnerable domain. In the preceding example, if the request to change email address can be performed with the GET method, then a self-contained attack would look like this:

<img src="https://Bankvulnerable-website.com/transfer?amount=1000000&account=Quyenson">



III. What is the impact of a CSRF attack?

In a successful CSRF attack, the attacker causes the victim user to carry out an action unintentionally. For example, this might be to change the email address on their account, to change their password, or to make a funds transfer. Depending on the nature of the action, the attacker might be able to gain full control over the user's account. If the compromised user has a privileged role within the application, then the attacker might be able to take full control of all the application's data and functionality.

IV. How to prevent XSS attack ?

Protecting against CSRF (commonly pronounced "sea-surf") requires two things: ensuring that GET requests are side-effect free, and ensuring that non-GET requests can only be originated from your client-side code.

REST

Representation State Transfer is a series of design principles that assign certain types of action (view, create, delete, update) to different HTTP V. Following REST-ful designs will keep your code clean and help your site scale. Moreover, REST insists that GET requests are used only to view resources.

Keeping your GET requests side-effect free will limit the harm that can be done by maliciously crafted URLs–an attacker will have to work much harder to generate harmful POST requests.

Anti-Forgery Tokens

Even when edit actions are restricted to non-GET requests, you are not entirely protected. POST requests can still be sent to your site from scripts and pages hosted on other domains. In order to ensure that you only handle valid HTTP requests you need to include a **secret** and **unique** token with each HTTP response, and have the server verify that token when it is passed back in subsequent requests that use the POST method (or any other method except GET, in fact.)

This is called an **anti-forgery** token. Each time your server renders a page that performs sensitive actions, it should write out an anti-forgery token in a hidden HTML form field. This token must be included with form submissions, or AJAX calls. The server should validate the token when it is returned in subsequent requests, and reject any calls with missing or invalid tokens.

Anti-forgery tokens are typically (strongly) random numbers that are stored in a cookie or on the server as they are written out to the hidden field. The server will compare the token attached to the inbound request with the value stored in the cookie. If the values are identical, the server will accept the valid HTTP request.

Most modern frameworks include functions to make adding anti-forgery tokens fairly straightforward. For example in python:

```
<form action="." method="post">
  {% csrf_token %}
</form>
```

Ensure Cookies are sent with the SameSite Cookie Attribute

The Google Chrome team added a new attribute to the `Set-Cookie` header to help prevent CSRF, and it quickly became supported by the other browser vendors. The `Same-Site` cookie attribute allows developers to instruct browsers to control whether cookies are sent along with the request initiated by third-party domains.

Setting a Same-Site attribute to a cookie is quite simple:

```
Set-Cookie: CookieName=CookieValue; SameSite=Lax;

Set-Cookie: CookieName=CookieValue; SameSite=Strict;
```

A value of `Strict` will mean than any request initiated by a third-party domain to your domain will have any cookies stripped by the browser. This is the most secure setting, since it prevents malicious sites attempting to perform harmful actions under a user's session.

A value of `Lax` permits GET request from a third-party domain to your domain to have cookies attached - but only GET requests. With this setting a user will not have to sign in again to your site if the follow a link from another site (say, Google search results). This makes for a friendlier user-experience - but make sure your GET requests are side-effect free!

Include Addition Authentication for Sensitive Actions

Many sites require a secondary authentication step, or require re-confirmation of login details when the user performs a sensitive action. (Think of a typical password reset page – usually the user will have to specify their old password before setting a new password.) Not only does this protect users who may accidentally leave themselves logged in on publicly accessible computers, but it also greatly reduces the possibility of CSRF attacks.

Reference:

Websecurity Academy: https://portswigger.net/web-security/csrf

Hackplaning: https://www.hacksplaining.com/prevention/csrf

OWASP: https://owasp.org/www-community/attacks/csrf