Prevent CSRF, XSS and XSF attacks

Best practices to audit and configure web app again CSRF, XSS and XSF attacks

Introduction

Useful links :

- XSS attacks What is cross-site scripting (XSS) and how to prevent it? | Web Security Academy
- CSRF attacks What is CSRF (Cross-site request forgery)? Tutorial & Examples | Web Security Academy
- XSF attacks Framing Attacks and Cross-frame scripting explained
- CSP basics Content Security Policy (CSP) - HTTP | MDN
- X Frame options header X-Frame-Options - HTTP | MDN
- CSRF token Cross-Site Request Forgery Prevention - OWASP Cheat Sheet Series
- XSS, CSRF and CSP vulnerabilities (lab root-me) Challenges/Web - Client [Root Me : Hacking and Information Security learning platform]
- samesite cookie attribute Set-Cookie - HTTP | MDN

TL;DR : Use the security built-in your framework, and do not use custom injection of code. Enable the different securities integrated in your framework, such as CSRF token. Deny all iframe, or scope it to trusted domains if needed

CSRF attacks

Definition

SRF (Cross-Site Request Forgery) attacks involve tricking authenticated users into unknowingly performing actions on a web application. The attacker exploits the trust between the victim's browser and the application to execute unauthorized actions.

The classic example is a victim receiving an email containing an image tag that secretly initiates a fund transfer request to an attacker's website. When the victim's browser loads the image, it sends a request to the attacker's site with the victim's session cookie. As a result, funds are transferred from the victim's account to the attacker's account without the victim's knowledge or consent. This demonstrates how CSRF attacks exploit trust to perform unauthorized actions on web applications.

Security risks

The security risks related to CSRF attacks can be significant:

1. **Unauthorized actions**: CSRF attacks allow attackers to perform actions on behalf of the victim, potentially leading to unauthorized changes, data breaches, or financial loss.
2. **Bypassing authentication**: As CSRF attacks use the victim's authenticated session, they can bypass any authentication checks implemented by the web application, making it difficult to detect and prevent such attacks.
3. **Trust exploitation**: The attack leverages the trust relationship between the victim's browser and the web application, taking advantage of the fact that the web application treats the victim's requests as legitimate.
4. **Social engineering**: CSRF attacks often rely on social engineering techniques to deceive users into visiting malicious webpages or clicking on malicious links, making them more susceptible to exploitation.

How to prevent it

There is 2 solutions that can be used altogether to prevent CSRF attacks :

- **Implement anti-CSRF tokens**: Include a unique and random token in each HTML form or request that modifies state on the server. The token should be validated before processing the request, ensuring that it originated from the correct page and not an attacker.

> Anti-CSRF token are implemented in most of frameworks.
> But you often need to **enable them and enable the CSRF check**.

- **Define cookie's samesite attributes as** Lax **or** Strict
  - With the SameSite attribute set to Lax cookies are allowed to be sent with cross-site requests that are initiated by top-level navigation (for example, when a user clicks on a link). However, they are not sent for cross-site requests triggered by other means, such as through the use of an image tag or a form submission from another site.
  - When the SameSite attribute is set to Strict, cookies are not sent with any cross-site requests, regardless of how the request is initiated. They are only sent for requests originating from the same site or domain as the web application.

> **Do not use** `None` attribute as it enables the use of the cookie in cross site request, no matter the origin.

XSS attacks

Definition

XSS (Cross-Site Scripting) attacks are security vulnerabilities in web applications where an attacker injects malicious scripts into trusted websites, allowing them to execute arbitrary code in the victim's browser. This can lead to unauthorized access, data theft, cookie hijacking, and website defacement.

*Classic example : An attacker submits a comment on a forum with a malicious script embedded in it with

```
<script>
  //get user's cookie
</script>
```

When other users view the comment, the script gets executed in their browsers, giving the attacker access to their sensitive information or allowing them to perform actions on their behalf.*

Security risks

Security risks associated with XSS attacks include:

1. **Data theft**: Attackers can steal sensitive information, such as login credentials, personal data, or financial details, from users who unknowingly execute the malicious scripts.
2. **Cookie hijacking**: By injecting scripts, attackers can access or manipulate user cookies, leading to session hijacking or impersonation.
3. **Session riding**: Attackers can exploit XSS vulnerabilities to piggyback on authenticated sessions and perform actions on behalf of users, leading to unauthorized access and privilege escalation.
4. **Malware distribution**: Attackers can use XSS to deliver malware or malicious payloads to unsuspecting users, leading to system compromise or further attacks.
5. **Phishing attacks**: XSS vulnerabilities can be leveraged to create convincing phishing pages or pop-ups, tricking users into disclosing sensitive information or installing malicious software.

How to prevent it

There is 3 solutions that can be used altogether to prevent XSS attacks :

- **Input validation**: Validate and sanitize all user input on the server-side to ensure it does not contain malicious code or script tags.

> **All user's input, event file's name must be sanitized.**

- **Output encoding**: Encode user-generated content before displaying it on web pages to prevent the browser from interpreting it as executable code. Apply appropriate encoding based on the context in which the output is being used (e.g., HTML, JavaScript, CSS) to prevent script injection.

> **Always use your framework var injection with** `{}` which encode for you the variable content.
> Obviously, never inject your variables in `dangerouslySetInnerHTML` unless you want to inject js, that cannot be controlled by the user. But it is really **not recommended.**

- **Content Security Policy (CSP)**: Implement a robust CSP that restricts the types of content allowed to be loaded on a web page, limiting the potential sources of XSS attacks. Here, we inform the browser that the only source able to load js will be your website's subdmain.
  Adapt this header with your specifics needs :

```
Content-Security-Policy: script-src 'self' *.example.com;
```