Cross-Site Request Forgery (CSRF) Vulnerability Prevention

Cross-Site Request Forgery (CSRF) is a type of web security vulnerability that allows attackers to trick victims into executing unwanted actions on a web application in which they are currently authenticated. In simpler terms, it's a way for an attacker to make your web browser do something you didn't intend to do, even if you're logged into a trusted website.

Here's a case study of an actual CSRF attack:

Imagine you're the CEO of a company, and you have a trusted assistant who always carries out your orders without question. An attacker could trick you into giving your assistant a forged document that looks like a legitimate order from you. The assistant, trusting the document and your authority, would then execute the actions on the forged order, even though you never actually gave those instructions.

Similarly, in a CSRF attack, the attacker tricks the victim's web browser into sending a forged request to a trusted website, just like the forged document tricking the assistant. The website, seeing the request coming from the victim's browser and assuming it's legitimate, processes the request and performs the action, even though the victim never actually intended to do that.

Here are some of the things an attacker could do with a successful CSRF attack:

- Transfer money out of your bank account
- Change your password
- Post embarrassing content on your social media accounts
- Buy items on your online shopping cart

How CSRF attacks work:

There are two main ways that CSRF attacks can be carried out:

1. Tricking the victim into clicking on a malicious link: The attacker can send the victim a link that, when clicked, sends a forged request to the targeted website. The link could be embedded in an email, social media post, or even a seemingly harmless image.
2. Embedding malicious code on a website: The attacker can embed malicious code on a website that the victim is likely to visit. This code can then send a forged request to the targeted website without the victim's knowledge or consent.

How to prevent CSRF vulnerabilities:

Token-Based Mitigation: The synchronizer token pattern is one of the most popular and recommended methods to mitigate CSRF.

Use Built-In Or Existing CSRF Implementations for CSRF Protection

Since synchronizer token defenses are built into many frameworks, find out if your framework has CSRF protection available by default before you build a custom token generating system. For example, .NET can use built-in protection to add tokens to CSRF vulnerable resources. If you choose to use this protection, .NET makes you responsible for proper configuration (such as key management and token management).

Synchronizer Token Pattern

CSRF tokens should be generated on the server-side and they should be generated only once per user session or each request. Because the time range for an attacker to exploit the stolen tokens is minimal for per-request tokens, they are more secure than per-session tokens. However, using per-request tokens may result in usability concerns.

For example, the "Back" button browser capability can be hindered by a per-request token as the previous page may contain a token that is no longer valid. In this case, interaction with a previous page will result in a CSRF false positive security event on the server-side. If per-session token implementations occur after the initial generation of a token, the value is stored in the session and is used for each subsequent request until the session expires.

When a client issues a request, the server-side component must verify the existence and validity of the token in that request and compare it to the token found in the user session. The request should be rejected if that token was not found within the request or the value provided does not match the value within the user session. Additional actions such as logging the event as a potential CSRF attack in progress should also be considered.

CSRF tokens should be:

- Unique per user session.
- Secret
- Unpredictable (large random value generated by a secure method).

CSRF tokens prevent CSRF because without a CSRF token, an attacker cannot create valid requests to the backend server.

Transmissing CSRF Tokens in Synchronized Patterns

The CSRF token can be transmitted to the client as part of a response payload, such as a HTML or JSON response, then it can be transmitted back to the server as a hidden field on a form submission or via an AJAX request as a custom header value or part of a JSON payload. a CSRF token should not be transmitted in a cookie for synchronized patterns. A CSRF token must not be leaked in the server logs or in the URL. GET requests can potentially leak CSRF tokens at several locations, such as the browser history, log files, network utilities that log the first line of a HTTP request, and Referer headers if the protected site links to an external site

For example:

```
<form action="/transfer.do" method="post">
    <input type="hidden" name="CSRFToken" value="OWY4NmQwODE4ODRjN2Q2NTlhMmZlYWEwYzU1YWQwM
    TVhM2JmNGYxYjJiMGI4MjJjZDE1ZDZMGYwMGEwOA==">
    [...]
    </form>
```

Since requests with custom headers are automatically subject to the same-origin policy, it is more secure to insert the CSRF token in a custom HTTP request header via JavaScript than adding a CSRF token in the hidden field form parameter.

Double-submit cookie prevention:

A double-submit cookie token approach can be used if using a valid token on the server side is impossible. In this cookie-based session handling, when a user visits a website, the site generates a value that stores as a cookie on the user's device, apart from the cookie that serves as a session identifier.

When a legitimate request is submitted to the site, it must contain the same value as included in the cookie. The server then verifies this, and the request parameter is accepted if the values match.

Custom request header

A technique that is particularly effective for AJAX or API endpoints is the use of custom request headers. In this approach, JavaScript is used to add a custom header. Unfortunately, JavaScript can't make cross-origin requests with a custom header because of the SOP security restrictions.

This prevents sending a cross-domain request with custom headers, thereby eliminating the possibility of a CSRF attack.

Django:

Django is similarly easy to protect any form by a CSRF-Token using the snippet within the *<form></form>* tags.

```
{% csrf_token %}
```

To provide the token for use with JavaScript requests, retrieve it from its storage cookie and add it to the request.

```
var csrftoken = Cookies.get('csrftoken');
...
xhr.setRequestHeader("X-CSRFToken", csrftoken);
```

References:

OWASP CSRF Prevention Cheat Sheet:

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

PortSwigger Web Security Academy - CSRF Prevention:

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

Acunetix: CSRF Attacks: Anatomy, Prevention, and XSRF Tokens:

- https://www.acunetix.com/blog/articles/cross-site-request-forgery/