**Synchronizer Token Pattern**

CSRF tokens can be generated once per user session or for each request (more secure).

**How should CSRF tokens be transmitted?**

**From server to client:**

**Server should transmit the token to client within a hidden field** of an HTML form that is submitted as a request parameter using the POST method:

<input type="hidden" name="csrf-token" value="CIwNZNlR4XbisJF39I8yWnWX9wX4WFoz" />

This field should be placed as early as possible within the HTML document, ideally before any locations where user-controllable data is embedded within the HTML. This mitigates against various attacks in which an attacker can use crafted data to capture parts of the HTML contents.

**From cliemt to server:**

**Placing the token into the URL** query string (GET request) is less safe because the query string:

* Is logged in various locations on the client and server side
* Is liable to be transmitted to third parties within the HTTP Referer header
* can be displayed on-screen within the user's browser

**Transmitingt CSRF tokens within a custom request header** is good because browsers do not normally allow custom headers to be sent cross-domain.

CSRF tokens **should not be transmitted (upstream) only within cookies** because cookies will be automatically sent to the server.

**Double Submit**: CSRF-Token in Cookies + Header (or request parameter) => no server state

## **Login CSRF**

For example, if an attacker uses CSRF to assume an authenticated identity of a target victim on a shopping website using the attacker's account, and the victim then enters their credit card information, an attacker may be able to purchase items using the victim's stored card details. Login CSRF can be mitigated by creating pre-sessions (sessions before a user is authenticated) and including tokens in login form. The session should be destroyed and a new one should be made to avoid [session fixation attacks](http://www.acrossecurity.com/papers/session_fixation.pdf).

**For testing:**

- Check if any state-changing actions are implemented using GET methods

- Check if CSRF-Token exists for sensitive POST actions

- Check if SameSite exists?

- Check if CSRF-Token is validated server-side by removing (the whole or just its value) or trying invalid CSRF-Token

**Validation of CSRF token depends on request method (x)**

Some applications correctly validate the token when the request uses the POST method but skip the validation when the GET method is used.

=> Check if it is possible to perform the action with GET with/without valid/invalid CSRF-Token

**CSRF token is not tied to the user session (x)**

Some applications do not validate that the token belongs to the user who is making the request. Instead, the application maintains a global pool of tokens that it has issued and accepts any token that appears in this pool.

=> Log in to the application using our own account, obtain a valid token, and then feed that token to the victim user in the CSRF attack.

**CSRF token is tied to a non-session cookie (x)**

Some applications do tie the CSRF token to a cookie, but not to the same cookie that is used to track sessions:

POST /email/change HTTP/1.1

Host: vulnerable-website.com

Content-Type: application/x-www-form-urlencoded

Content-Length: 68

Cookie: session=pSJYSScWKpmC60LpFOAHKixuFuM4uXWF; **csrfKey=rZHCnSzEp8dbI6atzagGoSYyqJqTz5dv**

**csrf=RhV7yQDO0xcq9gLEah2WVbmuFqyOq7tY**&email=wiener@normal-user.com

If the web site contains any behavior that allows us to set a cookie in a victim's browser, then an attack is possible. We can log in to the application using our own account, obtain a valid token and associated cookie, leverage the cookie-setting behavior to place their cookie into the victim's browser, and feed our token to the victim in the CSRF attack.

Replace the <script> block in PoC with:

<img src="https://web-security-academy.net/?search=test%0d%0aSet-Cookie:%20csrfKey=YOUR-KEY%3b%20**SameSite=None**" onerror="document.forms[0].submit()">

**CSRF token is simply duplicated in a cookie (double submit) (x)**

POST /email/change HTTP/1.1

Host: vulnerable-website.com

Content-Type: application/x-www-form-urlencoded

Content-Length: 68

Cookie: session=1DQGdzYbOJQzLP7460tfyiv3do7MjyPw; **csrf=R8ov2YBfTYmzFyjit8o2hKBuoIjXXVpa**

**csrf=R8ov2YBfTYmzFyjit8o2hKBuoIjXXVpa**&email=wiener@normal-user.com

We can perform a CSRF attack if the web site contains any cookie setting functionality. Here, we don’t need to obtain a valid token of our own, we can simply invent a token (perhaps in the required format, if that is being checked), leverage the cookie-setting behavior to place our cookie into the victim's browser, and feed our token to the victim in our CSRF attack.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Defending against CSRF with Referer/Origin header (as defense in depth)**

The server checks the Referer/Origin header to see if the request comes from another domain. This is less effective than CSRF tokens because Referer header can be manipulated by the attacker.

## **Validation of Referer depends on header being present**

Some applications validate the Referer header when it is present in requests but skip the validation if the header is omitted.

* Drop Referer header using (inside html tags):<meta name="referrer" content="never">

or <meta name="referrer" content="no-referrer">

## **Validation of Referer can be circumvented (x)**

If the application validates that the domain in the Referer starts with the expected value, then we can place this as a subdomain of our own malicious domain:

http://vulnerable-website.com.attacker-website.com/csrf-attack

If the application simply validates that the Referer contains its own domain name, then we can place the required value elsewhere in the URL:

http://attacker-website.com/csrf-attack?vulnerable-website.com

Many browsers strip the query string from the Referer header by default => add Referrer-Policy: unsafe-url Header to the page containing the exploit

To cause the Referer header in the generated request to contain the URL of the target site in the query string, edit the Java-Script:

history.pushState("", "", "/?vulnerable-website.com")

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Defending against CSRF with SameSite cookies (as defense in depth)**

A site is defined as the top-level domain (TLD), usually something like .com or .net, plus one additional level of the domain name.

Diagram

Description automatically generated

Graphical user interface

Description automatically generated with medium confidence

If the SameSite attribute is set to Strict, then the browser will not include the cookie in any requests that originate from another site.If the SameSite attribute is set to Lax, then the browser will include the cookie in requests that originate from another site but only if two conditions are met:

* The request uses the GET method.
* The request resulted from a top-level navigation by the user, such as clicking a link. Other requests, such as those initiated by scripts, will not include the cookie.

When setting a cookie with SameSite=None, the website must also include the Secure attribute, which ensures that the cookie is only sent in encrypted messages over HTTPS:

Set-Cookie: trackingId=0F8tgdOhi9ynR1M9wa3ODa; SameSite=None; Secure

## **Bypassing SameSite Lax restrictions using GET requests (x)**

Servers aren't always fussy about whether they receive a GET or POST request to a given endpoint. If they also use Lax restrictions for their session cookies, we may still be able to perform a [CSRF attack](https://portswigger.net/web-security/csrf) by eliciting a GET request, as long as the request involves a top-level navigation:

<script>

document.location = 'https://vulnerable-website.com/account/transfer-payment?recipient=hacker&amount=1000000';

</script>

Even if an ordinary GET request isn't allowed, some frameworks provide ways of overriding the method specified in the request line. For example, Symfony supports the \_method parameter in forms, which takes precedence over the normal method GET:

<form action="https://vulnerable-website.com/account/transfer-payment" method="GET">

**<input type="hidden" name="\_method" value="POST">**

<input type="hidden" name="recipient" value="hacker">

<input type="hidden" name="amount" value="1000000">

</form>

## **Bypassing SameSite restrictions using on-site gadgets (x)**

If a cookie is set with the SameSite=Strict attribute, browsers won't include it in any cross-site requests. But we may be able to get around this limitation if we can find a gadget that results in a secondary request within the same site. One possible gadget is a client-side redirect that dynamically constructs the redirection target using attacker-controllable input like URL parameters. Most importantly, this is a same-site request and will include all cookies related to the site, regardless of any restrictions that are in place. If we can manipulate this gadget to elicit a malicious secondary **GET** request, this can enable us to bypass any SameSite cookie restrictions completely.

**Bypassing SameSite restrictions via vulnerable sibling domains (x)**

It's essential to keep in mind that a request can still be same-site even if it's issued cross-origin.

Find a vulnerable sibling domain (i.e., search for “web-security-academy.net” in all resources).

Notice that responses to requests for resources like script and image files contain an Access-Control-Allow-Origin header, which reveals a sibling domain.

## **Bypassing SameSite Lax restrictions with newly issued cookies (x)**

Cookies with Lax SameSite restrictions aren't normally sent in any cross-site POST requests, but there are some exceptions: if Lax is applied by default (Chrome), top-level POST requests are possible for the first 120 seconds.

This two-minute window does not apply when Lax is explicitly set.

If we can find a gadget on the site that enables us to force the victim to be issued a new session cookie, you can refresh their cookie before following up with the main attack. For example, completing an OAuth-based login flow may result in a new session each time. We can trigger the cookie refresh from a new tab, so the browser doesn't leave the page before we're able to deliver the final attack. A minor snag with this approach is that browsers block popup tabs unless they're opened via a manual interaction. For example, the following popup will be blocked by the browser by default:

window.open('https://vulnerable-website.com/login/sso');

To get around this, you can wrap the statement in an onclick event handler as follows:

<script>

window.onclick = () => {

window.open('https://YOUR-LAB-ID.web-security-academy.net/social-login');

setTimeout(changeEmail, 5000);

}

function changeEmail() {

document.forms[0].submit();

}

</script>

This way, the window.open() method is only invoked when the user clicks somewhere on the page.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

### **User Interaction Based CSRF Defense**

* Re-Authentication
* One-time Token
* CAPTCHA