CSRF notes

Eyad Islam El-Taher

October 17, 2025

Introduction

CSRF (Cross-Site Request Forgery) $\Longrightarrow$ A web security vulnerability that allows an attacker to induce users to perform actions that they do not intend to perform. It allows an attacker to partly circumvent the same origin policy, which is designed to prevent different websites from interfering with each other.

CSRF vulnerabilities happen when

CSRF vulnerabilities happen when a web application relies solely on the automatic submission of a user's credentials (like session cookies) to authenticate a request, without requiring any other proof that the user actually intended to perform that action.

Key conditions that must be present for a CSRF attack to be possible:

- **Cookie-Based Session Handling:** The application uses session cookies to identify the user. The browser automatically sends these cookies with every request to the domain.
- **No Unpredictable Tokens:** The application does not use CSRF tokens (unique, secret, and unpredictable values) to validate that the request came from a legitimate source on its own site. (CSRF tokens can be manipulate)
- **No Additional Confirmation:** The application does not require re-authentication (e.g., password) for sensitive actions.
- **Predictable Request Parameters:** The attacker can reliably guess the parameters needed for a state-changing request (e.g., changing an email, transferring funds).

CSRF vulnerabilities happen where

CSRF vulnerabilities happen in the web application's server-side code, specifically in its handling of state-changing HTTP requests.

- **User Account Settings:** Changing a password, updating an email address.
- **E-commerce Functions:** Adding items to a cart, applying discount coupons, making a purchase.
- **Financial Transactions:** Transferring funds between accounts.
- **Social Media & Content:** Posting a status, sending a message, liking a post.
- **Admin Functions:** Promoting a user to an admin, deleting user accounts.

How CSRF vulnerabilities effect on users

- **Unauthorized Financial Transactions:** An attacker could force a user to transfer money out of their bank or brokerage account without their knowledge.
- **Account Takeover:** An attacker can change the victim's email address and/or password, effectively locking them out of their own account and giving the attacker full control.
- **Data Theft and Privacy Breach:** While CSRF doesn't typically steal data directly, an attacker could change privacy settings to expose private data or use it as a stepping stone for further attacks.

- **Reputational Damage and Social Embarrassment:** An attacker could force a user to post embarrassing status updates, send offensive messages to friends, or delete important content from their social media profiles.
- **Fraudulent Purchases:** On an e-commerce site, an attacker could make the victim purchase items to be shipped to the attacker's address.

How CSRF Works:

- **Step 1:** The victim logs into a trusted site (e.g., good-website.com), which authenticates them and sets a session cookie in their browser.
- **Step 2:** The victim, in a different tab, visits a malicious site created by the attacker (evil-website.com).
- **Step 3:** The malicious site contains hidden HTML that automatically sends a request to the trusted site (good-website.com). This is the forged request.
- **Step 4:** The victim's browser, being loyal, sees a request going to good-website.com and automatically attaches the valid session cookie from Step 1.
- **Step 5:** The trusted site (good-website.com) receives the request with a valid session cookie. It thinks, "This is a legitimate request from my logged-in user!" and processes it.
- **Step 6:** The action is completed without the victim's knowledge or consent.

How to construct a CSRF attack:

The easiest way to construct a CSRF exploit is using the CSRF PoC generator that is built in to Burp Suite Professional:

- **Step 1:** Select a request anywhere in Burp Suite Professional that you want to test or exploit.
- **Step 2:** From the right-click context menu, select Engagement tools / Generate CSRF PoC.
- **Step 3:** Burp Suite will generate some HTML that will trigger the selected request (minus cookies, which will be added automatically by the victim's browser).
- **Step 4:** You can tweak various options in the CSRF PoC generator to fine-tune aspects of the attack. You might need to do this in some unusual situations to deal with quirky features of requests.
- **Step 5:** Copy the generated HTML into a web page, view it in a browser that is logged in to the vulnerable website, and test whether the intended request is issued successfully and the desired action occurs.

How to deliver a CSRF exploit:

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 website that they control, and then induce victims to visit that website. This might be done by feeding the user a link to the website, via an email or social media message. Or if the attack is placed into a popular website (for example, in a user comment), they might just wait for users to visit the website.

Note that some simple CSRF exploits employ the GET method and can be fully self-contained with a single URL on the vulnerable website. 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.

Common defences against CSRF:

- CSRF tokens
- SameSite cookies
- Referer-based validation

CSRF tokens bypass

A CSRF token is a unique, secret, and unpredictable value that is generated by the server-side application and shared with the client. When attempting to perform a sensitive action, such as submitting a form, the client must include the correct CSRF token in the request. This makes it very difficult for an attacker to construct a valid request on behalf of the victim.

1. **If CSRF token depends on request method**
   Some applications correctly validate the token when the request uses the POST method but skip the validation when the GET method is used.
   - From the right-click context menu, select change request method $\Longrightarrow$ to switch between POST and GET methods
   - From the right-click context menu, select Engagement tools / Generate CSRF PoC.

2. **If CSRF token depends on token being present**
   Some applications correctly validate the token when it is present but skip the validation if the token is omitted.
   if the POSR request body like this

   ```
   email=saaaaadfds%40gmail.com&csrf=iHMm0Q4MbEGvSbjwPWCRmKtPfwvyXDFv
   ```

   remove the CSRF token

   ```
                          email=saaaaadfds%40gmail.com
   ```

3. **If CSRF token is not tied to the user session**
   Some applications do not validate that the token belongs to the same session as 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.

   In this situation, the attacker can log in to the application using their own account, obtain a valid token, and then feed that token to the victim user in their CSRF attack.

4. **If CSRF token is tied to a non-session cookie**
   Some applications do tie the CSRF token to a cookie, but not to the same cookie that is used to track sessions. This can easily occur when an application employs two different frameworks, one for session handling and one for CSRF protection, which are not integrated together

   ```
   POST /email/change HTTP/1.1
   Host: vulnerable-website.com
   Content-Type: application/x-www-form-urlencoded
   Content-Length: 68
   Cookie: session=pSJYSScWKpmC60LpFOAHKixuFuM4uXWF; csrfKey=rZHCnSzEp8dbI6
   atzagGoSYyqJqTz5dv

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

   We have to see if the CSRF token is tied to CSRF cookie NOT the session cookie
   Step 1: Check if the CSRF token is tied to CSRF cookie
   - Sumbit an invalid CSRF token
   - Sumbit a valid CSRF token but from another user (from private window)

   Step 2: Check if the CSRF token is tied to CSRF cookie NOT the session cookie
   - Sumbit a valid CSRF token and CSRF cookie from another user to see if the session cookie also tied to them or not

   if not this is good news that means that the session handling system and the csrf defense mechanism are not tied together so what we need in order to exploit this vulnerability is to be able to inject an http cookie called csrf key and inject our own
   The idea is to make a malicious website that inject our own csrf cookie into the user browser

the CSRF POS

```
<html>
<!-- CSRF PoC - generated by Burp Suite Professional -->
        <body>
                <form action="https://website.nett\/my-account/change-ema
                il" method="POST">

                        <input type="hidden" name="email" value="testma
                        il310&#64;normal&#45;user&#46;net" />

                        <input type="hidden" name="csrf" value="QHBtrtc
                        T4hitwv8Q6I2DVEYLwBJnVSpn" />

                        <input type="submit" value="Submit request" />
                </form>

<img src="https://website.net/?search=sdfdsfdsf%0D%0ASet-Cookie%3A+cs
rfKey%3Dk6iDX1lTVk0e0CuPbGPAamOUVhBZOHNX" onerror="document.forms[0].subm
it()">
</body>
</html>
```

So when the user open the link his browser will set the hacker's csrfkey cookie and by using the hacker CSRF token also $\Longrightarrow$ the attacker is able to change the email

5. **If CSRF token is simply duplicated in a cookie**
Some applications do not maintain any server-side record of tokens that have been issued, but instead duplicate each token within a cookie and a request parameter. When the subsequent request is validated, the application simply verifies that the token submitted in the request parameter matches the value submitted in the cookie. This is sometimes called the "double submit" defense against CSRF

```
POST /email/change HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 68
Cookie: session=1DQGdzYbOJQzLP7460tfyiv3do7MjyPw; csrf=R8ov2YBfTYmzFy

csrf=R8ov2YBfTYmzFy&email=wiener@normal-user.com
```

The attacker can again perform a CSRF attack if the website contains any cookie setting functionality. Here, the attacker doesn't need to obtain a valid token of their own. They simply invent a token (perhaps in the required format, if that is being checked), leverage the cookie-setting behavior to place their cookie into the victim's browser, and feed their token to the victim in their CSRF attack.

SameSite cookies bypass

SameSite is a browser security mechanism that determines when a website's cookies are included in requests originating from other websites. SameSite cookie restrictions provide partial protection against a variety of cross-site attacks, including CSRF, cross-site leaks, and some CORS exploits.

Since 2021, Chrome applies Lax SameSite restrictions by default if the website that issues the cookie doesn't explicitly set its own restriction level. This is a proposed standard, and we expect other major browsers to adopt this behavior in the future. As a result, it's essential to have solid grasp of how these restrictions work, as well as how they can potentially be bypassed, in order to thoroughly test for cross-site attack vectors

What is a site in the context of SameSite cookies?

In the context of SameSite cookie restrictions, a site is defined as the top-level domain (TLD), usually something like .com or .net, plus one additional level of the domain name. This is often referred to as the TLD+1.

When determining whether a request is same-site or not, the URL scheme is also taken into consideration. This means that a link from "http://app.example.com" to "https://app.example.com" is treated as cross-site by most browsers.

```
https://app.example.com
  |            |     |
  |            |     |
  |            |     |
scheme         +1   TLD
  |            |     |
  |            |     |
  --------------------
           |
           |
          SITE
```

NOTE: You may come across the term "effective top-level domain" (eTLD). This is just a way of accounting for the reserved multipart suffixes that are treated as top-level domains in practice, such as .co.uk.

What's the difference between a site and an origin?

The difference between a site and an origin is their scope; a site encompasses multiple domain names, whereas an origin only includes one. Although they're closely related, it's important not to use the terms interchangeably as conflating the two can have serious security implications.

Two URLs are considered to have the same origin if they share the exact same scheme, domain name, and port. Although note that the port is often inferred from the scheme.

```
 https://app.example.com
|                        |
|                        |
-------------------------
            |
            |
          origin
```

Request from	Request to	Same-site?	Same-origin?
`https://example.com`	`https://example.com`	Yes	Yes
`https://app.example.com`	`https://intranet.example.com`	Yes	No: mismatched domain nan
`https://example.com`	`https://example.com:8080`	Yes	No: mismatched port
`https://example.com`	`https://example.co.uk`	No: mismatched eTLD	No: mismatched domain nan
`https://example.com`	`http://example.com`	No: mismatched scheme	No: mismatched scheme

How does SameSite work?

SameSite works by enabling browsers and website owners to limit which cross-site requests, if any, should include specific cookies. This can help to reduce users' exposure to CSRF attacks, which induce the victim's browser to issue a request that triggers a harmful action on the vulnerable website. As these requests typically require a cookie associated with the victim's authenticated session, the attack will fail if the browser doesn't include this.

All major browsers currently support the following SameSite restriction levels:

- Strict
- Lax
- None

Developers can manually configure a restriction level for each cookie they set, giving them more control over when these cookies are used. To do this, they just have to include the SameSite attribute in the Set-Cookie response header, along with their preferred value:

```
Set-Cookie: session=0F8tgdOhi9ynR1M9wa3ODa; SameSite=Strict
```

NOTE: If the website issuing the cookie doesn't explicitly set a SameSite attribute, Chrome automatically applies Lax restrictions by default. This means that the cookie is only sent in cross-site requests that meet specific criteria, even though the developers never configured this behavior. As this is a proposed new standard, we expect other major browsers to adopt this behavior in future.

1. **Strict** If a cookie is set with the SameSite=Strict attribute, browsers will not send it in any cross-site requests. In simple terms, this means that if the target site for the request does not match the site currently shown in the browser's address bar, it will not include the cookie.

2. **Lax** Lax SameSite restrictions mean that browsers will send the cookie in cross-site requests, but only if both of the following conditions are met:

   - The request uses the GET method.
   - The request resulted from a top-level navigation by the user, such as clicking on a link.

   This means that the cookie is not included in cross-site POST requests, for example. As POST requests are generally used to perform actions that modify data or state (at least according to best practice), they are much more likely to be the target of CSRF attacks.

   Likewise, the cookie is not included in background requests, such as those initiated by scripts, iframes, or references to images and other resources.

3. **None** If a cookie is set with the SameSite=None attribute, this effectively disables SameSite restrictions altogether, regardless of the browser. As a result, browsers will send this cookie in all requests to the site that issued it, even those that were triggered by completely unrelated third-party sites.

   With the exception of Chrome, this is the default behavior used by major browsers if no SameSite attribute is provided when setting 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. Otherwise, browsers will reject the cookie and it won't be set.

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

General Testing Methodology for CSRF with SameSite Bypass

1. Reconnaissance Phase

Task	**Description**
Identify endpoints	Map all state-changing endpoints (email change, password reset, profile updates, etc.)
Authentication analysis	Document authentication mechanism and session management
CSRF token check	Verify if CSRF tokens are implemented and their predictability
Request flow mapping	Document the complete request flow for sensitive operations

2. Cookie Analysis

Check	Analysis Points
Set-Cookie headers	Examine login responses for cookie settings
SameSite attributes	• No SameSite specified → defaults to Lax • SameSite=Lax → GET requests with top-level navigation allowed • SameSite=Strict → more restrictive (requires same-site context) • SameSite=None → requires Secure flag
Cookie scope	Domain and path settings analysis

3. Endpoint Testing

Test	Procedure
HTTP method testing	Test if endpoints accept both GET and POST methods
Method override techniques	• `_method` parameter (Ruby, Laravel) • `X-HTTP-Method-Override` header • Framework-specific overrides
Parameter testing	Verify parameter acceptance in GET query strings vs POST body

4. Client-Side Redirect Analysis (for SameSite=Strict Bypass)

Step	Procedure
Identify redirect gadgets	• Find client-side redirects in application flow • Look for confirmation pages with automatic redirects • Identify JavaScript that handles navigation • Search for URL parameters that influence redirect targets
Analyze redirect logic	• Study JavaScript files for redirect mechanisms • Identify dynamic URL construction • Check for parameter injection points • Test path traversal in redirect parameters
Test redirect control	• Modify parameters to control redirect destination • Use path traversal (../) to access other endpoints • Verify cookies are sent in redirected request • Confirm same-origin context is maintained

5. SameSite Bypass Vectors

Vector	Implementation
Top-level navigation	• `window.location` • `<meta> refresh` • `<iframe>` with top navigation • Form submission with `target="_top"`
Client-side redirect chains	• Abuse application's own redirect mechanisms • Chain multiple client-side redirects • Use parameter injection to control final destination • Maintain same-site context through redirect flow
Browser-specific	Test different browser versions and behaviors

6. Exploit Crafting for SameSite=Strict Bypass

Step	Procedure
Chain vulnerabilities	• Combine open redirect with state-changing endpoints • Use path traversal in redirect parameters • Encode parameters to avoid syntax breaks • Test parameter normalization behavior
Construct exploit URL	• Start with vulnerable redirect endpoint • Inject path to target endpoint using ../ sequences • Append target parameters with proper encoding • Ensure final URL points to state-changing action
Delivery mechanism	• Use `<script>` tag with `document.location` • Implement via `<iframe>` with navigation • Chain through multiple redirect layers • Test in different browser contexts

Key Testing Checklist

Cookie Configuration Tests

Test Case	Check
No explicit SameSite attribute set	
SameSite=Lax with top-level navigation	
SameSite=Strict requiring same-site context	
Cookie without Secure flag on HTTPS sites	
Cross-domain cookie scope issues	

Endpoint Vulnerability Tests

Test Case	Check
GET requests accepted for state-changing operations	
Method overriding supported (`_method`, `X-HTTP-Method-Override`)	
No CSRF tokens or predictable tokens	
Parameters accepted via query string	
No Referer header validation	
Client-side redirects with user-controllable parameters	
Path traversal possible in redirect parameters	

Redirect Gadget Tests

Test Case	Check
Identify confirmation pages with automatic redirects	
Locate JavaScript files handling client-side navigation	
Test parameter control over redirect destinations	
Verify path traversal works in redirect parameters	
Confirm cookies are sent during client-side redirects	
Test URL normalization behavior	

Exploit Validation Tests

Test Case	Check
Proof-of-concept changes target user state	
Works with logged-in user session	
Bypasses SameSite=Strict restrictions	
Consistent across different browsers	
Maintains session through redirect chain	
Properly encodes special characters	

Common Redirect Gadget Locations

- Comment confirmation pages with automatic redirects
- Search result pages with "redirecting..." functionality
- Login/logout confirmation pages
- Form submission success pages
- Payment processing confirmation pages
- Any page with meta refresh or JavaScript timeout redirects

Some examples

```
<script>
    document.location = 'https://vulnerable-website.com/account/pay
    ment?recipient=hacker&amount=1000000';
</script>
```

```
 <html>
<body>
  <form action="https://vulnerable-website.com/post/comment/confirmation">
<input type="hidden" name="postId" value="&#47;my&#45;account&#47;change&#45
;email&#63;email&#61;efdfr&#64;normal&#45;user&#46;net&amp;submit&#61;1" />
<input type="submit" value="Submit request" />
  </form>
  <script>
```

```
document.location = "https://vulnerable-website.com/post/comment/confirmatio
n?postId=../my-account/change-email?email=afr%40normal-user.net%26submit=1";
  </script>
</body>
 </html>
```

```
<form action="https://vulnerable-website.com/account/payment" method="POST">
    <input type="hidden" name="_method" value="GET">
    <input type="hidden" name="recipient" value="hacker">
    <input type="hidden" name="amount" value="1000000">
</form>
```

Bypassing Referer-based CSRF Defenses

Some applications use the HTTP Referer header to defend against CSRF attacks, typically by verifying that requests originate from the application's own domain. This approach is generally less effective than proper CSRF token validation and can be bypassed using various techniques.

Understanding the Referer Header

Concept	Description
Purpose	Contains URL of webpage that linked to the requested resource
Browser behavior	Added automatically when users click links or submit forms
Privacy concerns	Often modified or withheld for privacy reasons
Common misspelling	Incorrectly spelled as "Referer" in HTTP specification

Referer Validation Bypass Techniques

1. Exploiting Absent Referer Header

Vulnerability	Exploitation Method
Weak validation	Applications validate Referer when present but skip validation when header is absent
Bypass method	Cause browser to omit Referer header in CSRF request
Implementation	Use `<meta name="referrer" content="never">` in HTML header

Example CSRF PoC with Referer Suppression:

```
<html>
  <head>
    <meta name="referrer" content="never">
  </head>
  <body>
    <form action="https://target.com/my-account/change-email" method="POST">
      <input type="hidden" name="email" value="attacker@evil.com">
      <input type="submit" value="Submit">
    </form>
    <script>
      history.pushState('', '', '/');
      document.forms[0].submit();
    </script>
  </body>
</html>
```

2. Circumventing Domain Validation Logic

Validation Flaw	Bypass Technique
Starts-with validation	Place target domain as subdomain of attacker domain
Contains validation	Include target domain in query string or path
Regex weaknesses	Exploit poor regular expression patterns

Subdomain Bypass Example:

```
http://target-domain.com.attacker-evil.com/csrf-poc
```

Query String Bypass Example:

```
http://attacker-evil.com/csrf-poc?target-domain.com
```

Browser Referrer Policy Challenges

Challenge	Solution
Query string stripping	Modern browsers often strip query strings from Referer by default
Override method	Set `Referrer-Policy: unsafe-url` header in exploit response
Header spelling	Note: "Referrer-Policy" uses correct spelling (double 'r')

Burp Suite Configuration for Referrer Policy

1. **Access Proxy Settings**
   - Go to `Proxy → Options` in Burp Suite
   - Scroll to `Match and Replace` section
2. **Create Replacement Rule**
   - Click `Add` to create new rule
   - Configure with these settings:
     - Type: `Response header`
     - Match: `Referrer-Policy`
     - Replace: `Referrer-Policy: unsafe-url`
3. **Activate Rule**
   - Ensure rule is checked/enabled
   - Click `OK` to save configuration

Now when your browser requests the PoC HTML, Burp will automatically add the correct header to the server's response.

Note $\Longrightarrow$ If you found the CSRF and you will make the POC make first the Referer using realserver then like burpsuite (the vuln web still just a query parameter)

```
Referer:https://burpsuite/?0ae30066047b8a858251259a00ed0054.web-secu
rity-academy.net
```

And the POC look like

```
<html>
  <!-- CSRF PoC - generated by Burp Suite Professional -->
  <body>
    <form action="https://0ae30066047b8a858251259a00ed0054.web-secur
    ity-academy.net/my-account/change-email" method="POST">
      <input type="hidden" name="email" value="ssf&#64;mdsail&#46;com"/>
      <input type="submit" value="Submit request" />
    </form>
    <script>
      history.pushState('', '', '/?0ae30066047b8a858251259a00ed0054.web-s
      ecurity-academy.net');
      document.forms[0].submit();
    </script>
  </body>
</html>
```

Note $\Longrightarrow$ the last parameter in method "history.pushState" is the query parameter which is the vuln website as you see

General Testing Methodology

1. Referer Header Analysis

Test	Procedure
Header presence check	Verify if application checks for Referer header existence
Validation logic	Determine how Referer domain is validated
Absent header handling	Test requests with Referer header completely removed

2. Validation Logic Testing

Test Case	Expected Result
Subdomain bypass	Application accepts `target.com.attacker.com` as valid
Query string inclusion	Application accepts Referer with target in query parameters
Path manipulation	Application accepts Referer with target in path
Protocol variations	Test HTTP vs HTTPS Referer handling

3. Exploit Development

Component	Implementation
Referer suppression	Use `<meta name="referrer" content="never">`
Domain spoofing	Craft URLs that appear to originate from target domain
Policy enforcement	Configure `Referrer-Policy: unsafe-url`
Delivery method	Choose appropriate delivery (iframe, form, script)

Testing Checklist for Referer-based CSRF Protection

Header Presence Tests

Test Case	Check
Application rejects requests with no Referer header	
Application accepts requests with missing Referer header	
Validation only occurs when Referer is present	

Validation Logic Tests

Test Case	Check
Subdomain bypass works (target.attacker.com)	
Query string bypass works (attacker.com?target.com)	
Path-based bypass works (attacker.com/target.com/)	
Protocol validation is properly implemented	
Exact domain matching required	

Browser Compatibility Tests

Test Case	Check
Referrer-Policy: unsafe-url forces full URL inclusion	
Meta tag referrer suppression works across browsers	
Query strings are preserved in Referer header	
Works in latest browser versions	

Defense Recommendations

- **Avoid Referer-based validation** - Use multiple layers of CSRF protection
- **If using Referer** - Implement strict exact-domain matching
- **Fail securely** - Reject requests with invalid/missing Referer when expected