Cross-site request forgery (CSRF)

What is CSRF?

Cross-site request forgery (also known as CSRF) is 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.

When testing for a CSRF, the following 3 conditions must be in place:

- **A relevant action:** There is some sort of function or action on the site/app that an attacker can gain something of value by forcing another user to execute. This could be something like a password or email change function.
- **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.

CSRF vulnerability with no defenses

This example contains an email field that is vulnerable to CSRF, and has no protective measures in place.

To exploit, we make us of burp pro’s CSRF PoC generator. Change the email to whatever you desire, open the tool, copy html and upload to the exploit server.

When a user goes on your site, the code will be executed and their email changed.

CSRF where token validation depends on request method

Sometimes CSRF token validation is done only on post requests, as the program expects to be receiving data in the form of post. To alter our payload to a get request, we simply capture the request, and add in a query string `email?evil@evil.com`when we create our burp pro CSRF PoC.

CSRF where token validation depends on token being present

Another possibility is that the site skips validation all together if no CSRF token is supplied, meaning we can simply delete it from the request before making our PoC.

Capture a request, change email to whatever desired, delete the CSRF token entirely and create our CSRF PoC using burp pro.

CSRF token is not tied to the user session

Some websites do not link CSRF tokens to the users who generated them, they just store all CSRF tokens in a global pool. This means we can generate a CSRF token on our own account and use it to perform a malicious action on another user.

We need to capture a request containing a CSRF token, copy the token and then drop the request as CSRF tokens are 1-time use only. We then paste it into our CSRF PoC and host.

CSRF where token is tied to non-session cookie

In a variation on the preceding vulnerability, 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

This can be seen if there is both a CSRF token and a CSRF key cookie.

We can test to see if it is vulnerable to this by copying the CSRF key and CSRF token from one account and using them in a request on another user we have access to. If the site accepts these, we know it is vulnerable to this type of attack.

To exploit:

Step 1: Copy CSRF Token and CSRF Key

Step 2: Capture request from the secondary account

Step 3: Swap key & Token, generate PoC file.

Step 4: Enter the payload:

```
<img src="https://ac491f3a1f909b20c0a83d43005a0036.web-security-academy.
net/?
search=test%0d%0aSet-Cookie:%
20csrfKey=zHzpblOwhT2xGEYSSvvqSUDMlb9EE6hG"
onerror="document.forms[0].submit()">
```

To break this payload down,

On the first line we have a link to the vulnerable site.

The second line contains a querystring of search taht we simply set to test, and then call the Set-Cookie function to set the csrfKey cookie equal to the one we saved from step 1.

Third line submits the form.

If all is well, we should have been able to edit the users email.

CSRF where token is duplicated in cookie

This is a common technique used to try to prevent CSRF as it requires less serverside overhead. Instead of storing cookies server side to compare against what is sent in a request, it simply ensure the csrf token and csrf cookie are identical.

The payload is identical to the previous example.

CSRF where Referer validation 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.

In this situation, an attacker can craft their CSRF exploit in a way that causes the victim user's browser to drop the `Referer` header in the resulting request. There are various ways to achieve this, but the easiest is using a META tag within the HTML page that hosts the CSRF attack:

`<meta name="referrer" content="never">`

To exploit we simply add this tag to the CSRF PoC file.

CSRF with broken Referer validation

Some applications validate the `Referer` header in a naive way that can be bypassed. For example, if the application validates that the domain in the `Referer` starts with the expected value, then the attacker can place this as a subdomain of their own domain:
`http://vulnerable-website.com.attacker-website.com/csrf-attack`

Likewise, if the application simply validates that the `Referer` contains its own domain name, then the attacker can place the required value elsewhere in the URL:
`http://attacker-website.com/csrf-attack?vulnerable-website.com`

In the lab, we can observe there is a vulnerability by injecting an arbitrary string at the beginning of the referrer parameter followed by a ? to turn the original value into a query string. As the website still accepts this request, we know this vulnerability may be possible to exploit.

To exploit it, we a CSRF PoC like normal and alter the History.pushState() value

```
history.pushState("", "", "/?your-lab-id.web-security-academy.net")
```