Secure Programming
CSRF Lab

CSRF tokens, cookie SameSite settings, and POST requests vs. GET all help defend against CSRF attacks. There are other defences, but to understand them better, let's launch a CSRF attack against Coffeeshop user Bob.

In Firefox, if a cookie does not specify the SameSite attribute, Firefox treats it as SameSite=Lax by default, which means cookies are only sent in same-site requests and top-level navigations, not in cross-site requests unless explicitly set to SameSite=None. So unless we specify this in our cookie settings, the CSRF attack won't work. We need to first perform the following steps before starting the exercise:

1. Edit "\coffeeshop\vagrant\coffeeshopsite\coffeeshopsite\settings.py". Change:

```
SESSION_COOKIE_SAMESITE = None
SESSION_COOKIE_SECURE = False
```

to

```
SESSION_COOKIE_SAMESITE = 'None'
SESSION_COOKIE_SECURE = True
```

None without the quotes omits the SameSite attribute, selecting the browser's defaults. 'None' with the quotes sets SameSite to None.

2. Start a separate instance of Coffeeshop running through the development SSL server. From inside the Coffeeshop VM, run:

```
cd /vagrant/coffeeshopsite
python3 manage.py runsslserver 0:8100
```

3. Once the ssl server is running, you can access the coffeeshop site at:

```
https://127.0.0.1:8100
```

and log in as Bob from there. **Don't leave out the https**, or it will default to HTTP.

(NOTE: We use HTTPS to ensure the browser will still include the user's session cookie during the attack. Otherwise, Firefox would block the cookie on HTTP, and the CSRF attack would fail—not because the site is secure, but because the browser refused to send the cookie.

Launching the CSRF Attack

If an attacker wants to gain access to a victim's account, one option is to get the victim to follow a change email link. If the attacker crafts the link to change the email to their address, they can visit the site, click on the Forgot password link, and enter their email address. A password reset link will be emailed to them.

Our Coffeeshop application is vulnerable to this in Firefox because the change email URL does not need a CSRF token (it has the @csrf_exempt), and there is no SameSite attribute on the session ID cookie. In Firefox, this means it defaults to None.

Attack Scenario

Visit the Coffeeshop site in Firefox (not Chrome) at https://127.0.0.1:8100 and log in as bob.

Imagine Bob receives a link to http://127.0.0.1:8081/youhavewonssl/ in an email promising prize money. Suppose he clicks on it. Visit this URL now in Firefox and look at the code (right-click and "view source code"). It exploits the fact that the URL https://127.0.0.1:8100/changeemail is not CSRF-protected. Ignore the Test CSRF button for now.

The HTML contains a hidden form that calls this URL with the Evil Hacker's email address. The URL requires the current email address to be sent along with the new one, so the page prompts the victim for this. Enter Bob's email address bob@bob.com and click Submit. The Submit button puts the victim's typed email address into the hidden form and submits it.

You will notice two things. Visit https://127.0.0.1:8100 and click My Account. You will see that the email address has been changed. Use this opportunity to change it back to bob@bob.com.

The other thing you will notice is that after clicking Submit, a page is displayed confirming Bob's password has been changed. This is likely to alert Bob to the fact his account has been hacked. Most likely, he will immediately change his email address back (unless Evil Hacker is very quick in resetting the password).

Hiding the Response

Ideally, as a hacker, we would like to hide the response page. Open the HTML template: `csthirdparty/vagrant/csthirdpartysite/csthirdparty/templates/csthirdparty/youhavewon.html`

in a text editor. Go to the JavaScript function sendemail(). This submits the form synchronously with form.submit().

Because it is synchronous, the result replaces the current page. We can capture the output and suppress it by replacing this with an Ajax call. The code is already in the function, commented out.

Modified JavaScript

Uncomment the Ajax call and comment out form.submit(). Your function should look like the following:

```
function sendemail() {
    var enteredemail = document.getElementById('enteredemail').value;
    var form = document.getElementById('changeemailform');
    document.getElementById('old_email').setAttribute('value', enteredemail);
    //form.submit()

    $.ajax({
        type: "POST",
          url: "https://127.0.0.1:8100/changeemail",
          data: $('#changeemailform').serialize(), // serializes the form's elements.
        xhrFields: {
            withCredentials: true
        },
       success: function(html){
            alert("Thank you.  We will contact you shortly")
        },
        crossDomain: true
       });
}
```

Restart Apache in the CSThirdparty VM and reload the page. Now enter Bob's email again and click Submit. This time, you will see a more deceptive response, and Bob's email has still been changed.

To finish the exercise, set Bob's password back to bob@bob.com.

Preventing CSRF

There are several ways we can prevent CSRF. We will implement 2 ways but it would be best to include all for a defence-in-depth approach.

1. **Cookie settings:** The first fix is to simply change the cookie settings in the settings.py file. We will set the SameSite setting to Lax. SameSite=Lax helps prevent CSRF by ensuring cookies are not sent automatically with cross-site requests, except for top-level navigations initiated by user actions like following a link. In the case of the attack on Bob, the request will be blocked because it is a cross-site request.

2. **Csrf_exempt**: In the views.py file, the changeemail function was set to csrf_exempt, which is dangerous because it disables CSRF protection for the Change Email page. We can easily solve this by changing this:

```
@login_required
@require_http_methods(["GET", "POST"])
@csrf_exempt  # this is a bad idea - it is to demo
def changeemail(request):
```

To this:

```
@login_required
@require_http_methods(["GET", "POST"])
@csrf_protect
def changeemail(request):
```

Note: You will need to import the csrf_protect function at the top of views.py:

from django.views.decorators.csrf import csrf_protect

Once you have made the code changes, restart the CoffeeShop Apache server for the code changes to be updated.

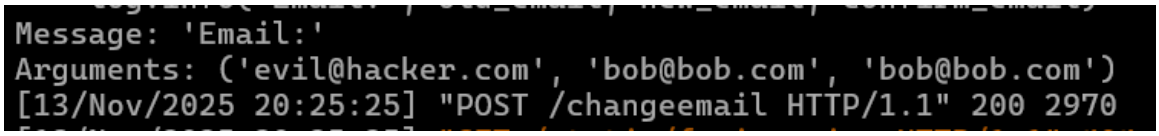
Other safeguards

- Always ensure any forms have the {% csrf_token %} added, otherwise they can be vulnerable to CSRF.
- As an additional defence, check the Origin or Referer header on state-changing POST requests. Reject requests where the origin is not your site.

Re-run the attack

Repeat the attack as before. Nothing should appear to happen on the youhavewonssl.html page. Confirm the attack failed by going to the CoffeeShop site and checking Bob's email address (should be unchanged). Also, if you compare the ssl logs (CSThirdparty shell):

Before the CSRF controls were added:



After the controls were added: