Assignment 5

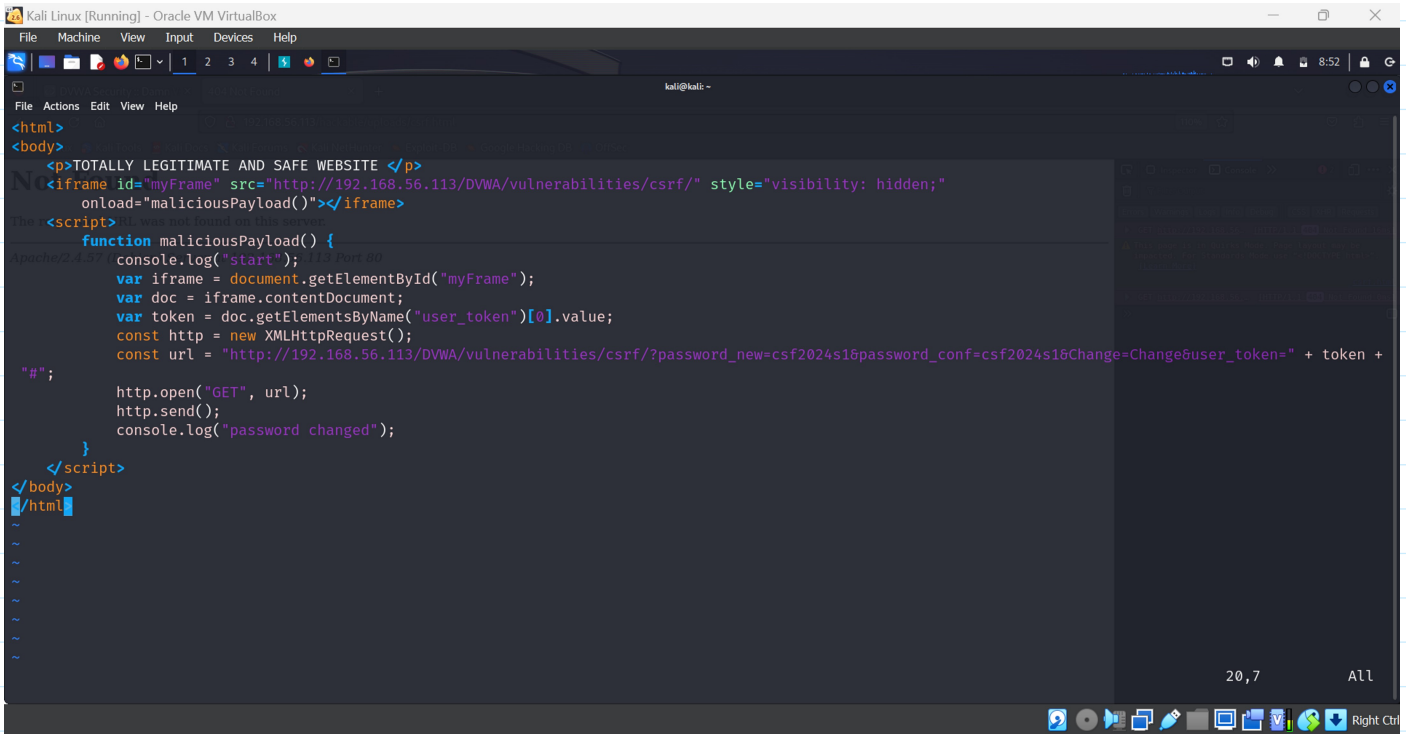
Wednesday, 15 May 2024 10:04 PM

Part 1 - Advanced Web Exploits

q1. [1 point] - When on the high-security setting of DVWA, a unique ANTI-CSRF token is created each time the password change page is accessed, as shown in the workshop. To launch a CSRF attack in this case, we first need to steal the token. Create an HTML (name it: 'csrf.html') file that can steal the token from the DVWA CSRF page http://[hacklabvm_ip]/DVWA/vulnerabilities/csrf and change the password to 'csf2024s1'. You can use the template here. Show the content of your csrf.html file.

[HINT] To properly craft the malicious HTML file, you will need to take note of the request sent by DVWA when the change password is triggered.

The html file used for csrf is shown below in the screenshot. The html file works when it is loaded into a browser, the hidden iframe triggers a script that sends a forged request to change the user's password associated with the provided token.

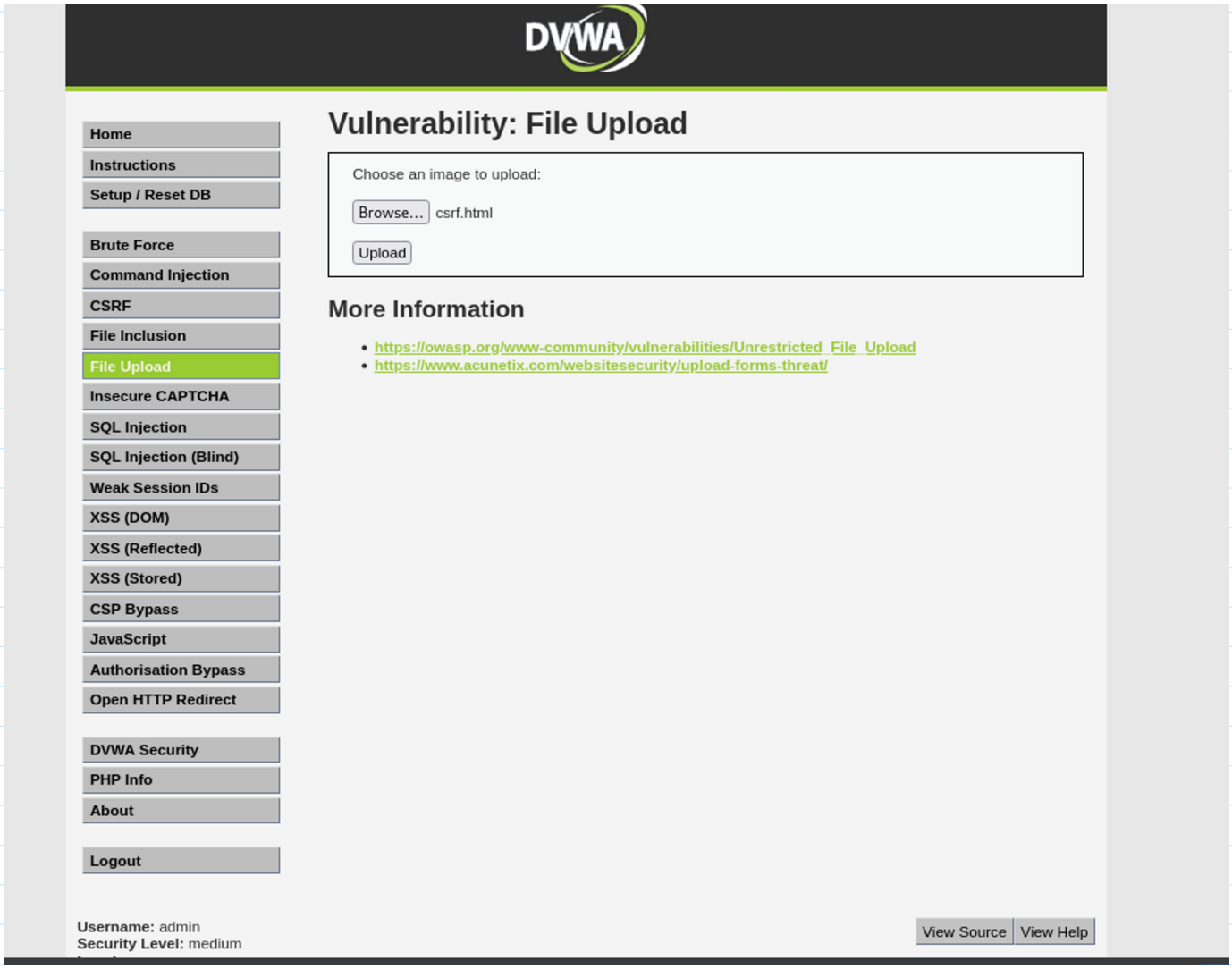


q2. [1 point] - Set the DVWA security level to MEDIUM. Upload the csrf.html file to the "hackable/uploads/" folder. Provide details of the steps you use to upload the HTML file with the security level set at medium.

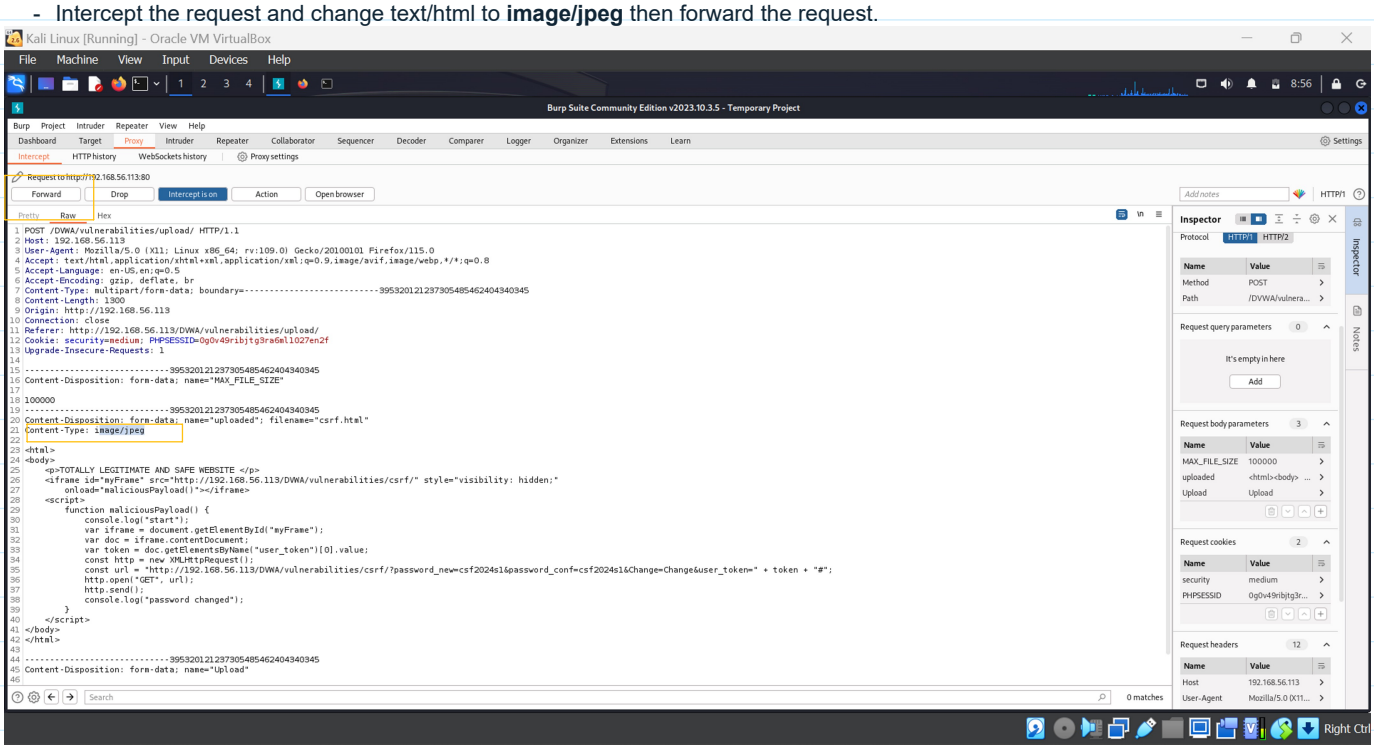
[HINT] Set the DVWA security level to MEDIUM first. At this level of security, you won't be able to upload the HTML file directly as only certain file types are allowed. You may need a proxy intervention.

I followed the following steps to upload the file onto the server.

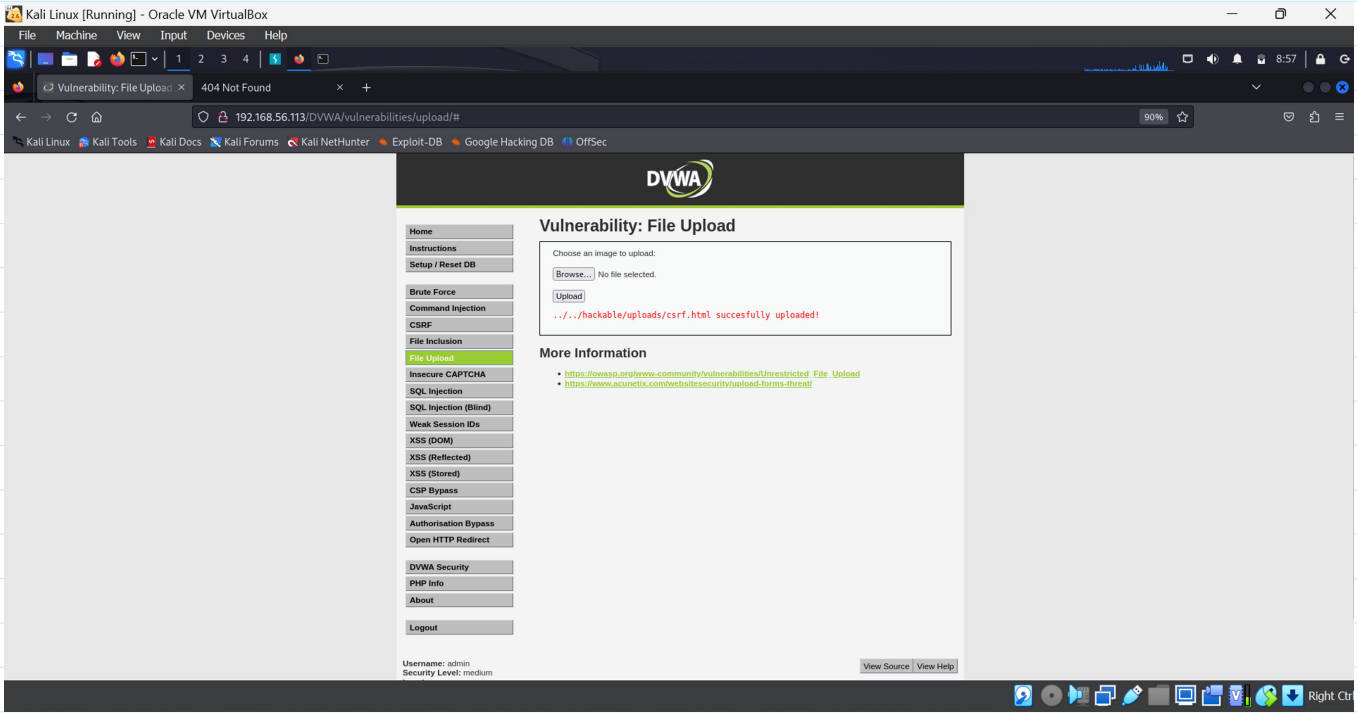
- Put safety to "Medium"
- Select the html file but in medium security setting, only images or jpeg formats are allowed, the html format is not accepted so before submitting the form, set up burp suite to intercept the request.



- Intercept the request and change text/html to **image/jpeg** then forward the request.



- After forwarding the request, the file upload DVWA page displays a confirmation of the upload as seen in the screenshot below.



q3. [1 point] - Show that after a user visit http://[hacklabvm_ip]hackable/uploads/csrf.htmlLinks to an external site. the password changes to 'csf2024s1'. Explain what happened.

[HINT] To trigger the malicious HTML, note the location where you uploaded your file. Also, consider changing the DVWA security back to high, as the anti-CSRF token is only used in high settings.

- Now to trigger the csrf attack, load the page http://192.168.56.113hackable/uploads/csrf.html as shown in the screenshot below.

  INSERT SCREENSHOT HERE - NOT attempted

- To verify that the password has been changed, login using the new password

  INSERT SCREENSHOT HERE - Not attempted

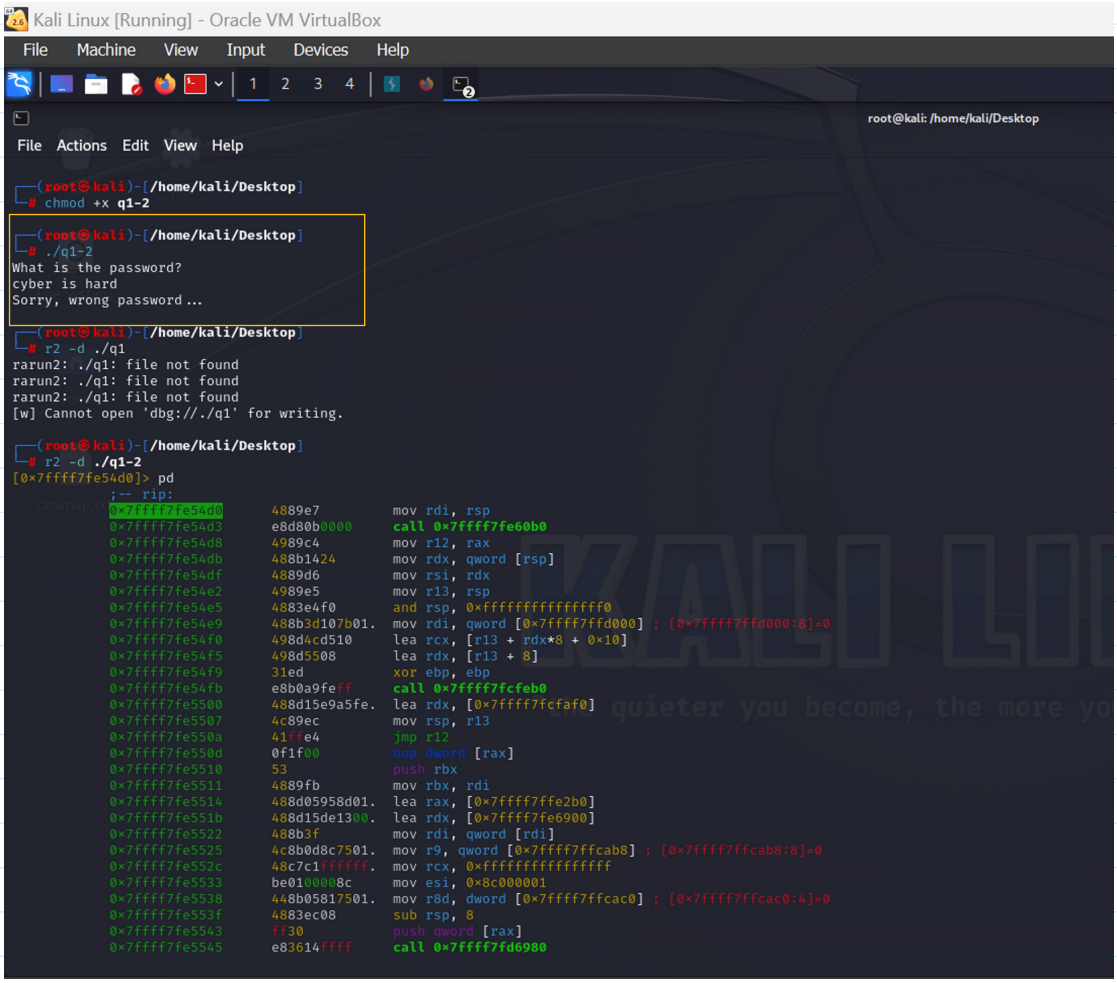
Part 2 - Digital Forensics

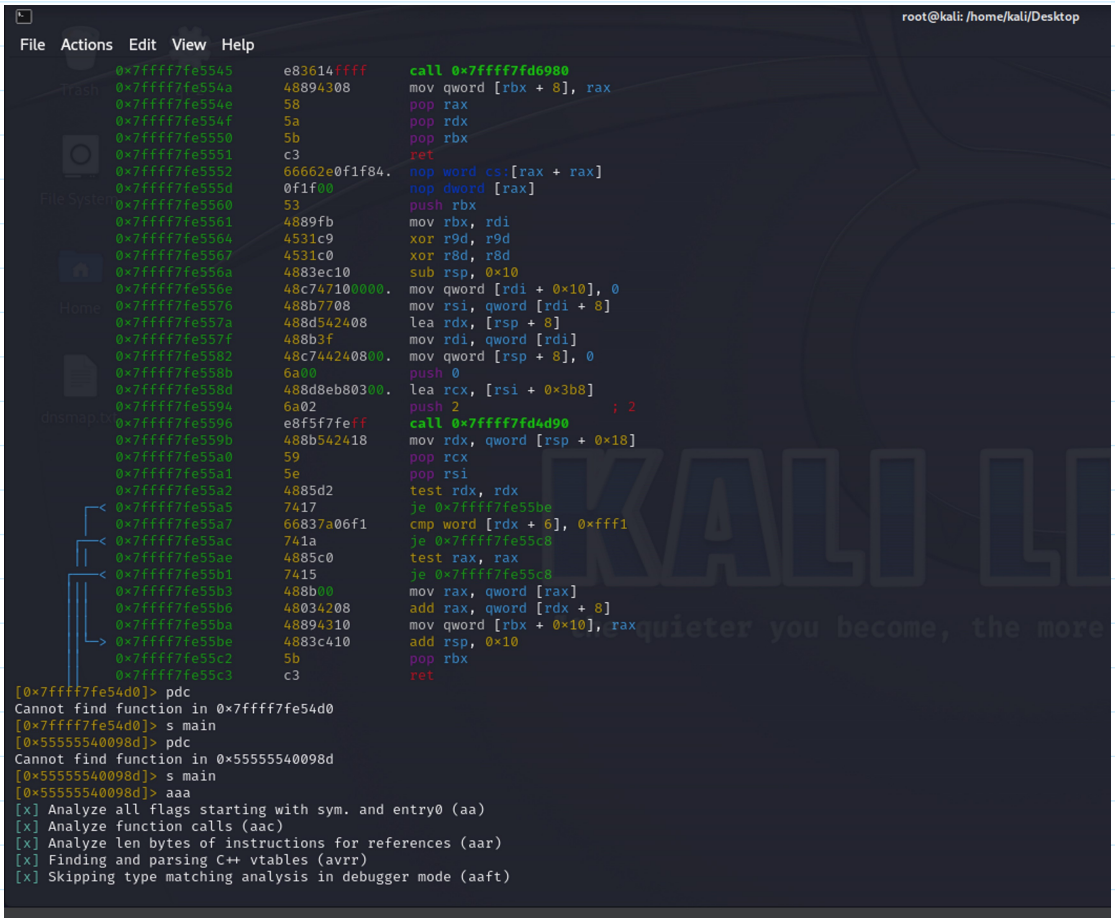
q1. [2 points] Reversing (1)

- **Download this binary Download binary. You can run it as "./q1" in Linux**
- **You ARE NOT allowed to patch this program**
- **Use Ghidra, Cutter, or Radare2 (or something else) to decompile and deduce the password required for revealing the secret.**
- **Get the program to print the secret.**

First, I started by executing the file to observe what I may need to do. Observing from the screenshot below, it appears that we need to look for some sort of input validation as the user input is being compared with some sort of stored parameter, I will be using Radare2 for this.

Then to view more detail about the stored parameter, I decided to use the pd command to disassemble the code as shown in the screenshot below.





Observing the disassembled code in the screenshot below, the user input seems to be getting compared to the phrase " I Love Cyber Security".

```
[x] Skipping type matching analysis in debugger mode (aaft)
[x] Propagate noreturn information (aanr)
[x] Use -AA or aaaa to perform additional experimental analysis.
[0×55555540098d]> pdc
int main (int esi, int edx) {
    loc_0×55555540098d:
        // DATA XREF from entry0 @ 0×55555540071d
        push  (rbp)
        rbp = rsp
        rsp -= 0×420
        dword [var_414h] = edi // argc
        qword [var_420h] = rsi // argv
        rax = qword fs:[0×28]
        qword [var_8h] = rax
        eax = 0
        rdi = rip + str.What_is_the_password_ // 0×555555400cbf // "What is the password?"
        sym.imp.puts  ()
        // int puts("What is the password?")
        rdx = qword [reloc.stdin] // [0×555555602010:8]=0
        rax = var_410h
        esi = 0×400 // 1024
        rdi = rax
        sym.imp.fgets  ()
        // char *fgets("", -1, ?)
        rax = var_410h
        rsi = rip + 0×2ec // "\n"
        // 0×555555400cd5
        rdi = rax
        sym.imp.strtok  ()
        // char *strtok("", "\n")
        rax = var_410h
        esi = 0×d // 13
        rdi = rax
        sym.rot  () // sym.rot(0×0, 0×0)
        rax = var_410h
        rsi = rip + str.I_Love_Cyber_Security_ // 0×555555400cd7 // "I Love Cyber Security!"
        rdi = rax
        sym.imp.strcmp  ()
        // int strcmp("", "I Love Cyber Security!")
        var = eax & eax
        if  (var) goto loc_0×555555400a2b // likely

    loc_0×555555400a2b:
        // CODE XREF from main @ 0×555555400a1d
        rdi = rip + str.Sorry__wrong_password ...  // 0×555555400cee // "Sorry, wrong password ... "
        sym.imp.puts  ()
        // int puts("Sorry, wrong")
         // do {
    loc_0×555555400a37:
```

```
    loc_0×555555400a2b:
        // CODE XREF from main @ 0×555555400a1d
        rdi = rip + str.Sorry__wrong_password ...  // 0×555555400cee // "Sorry, wrong password ... "
        sym.imp.puts  ()
        // int puts("Sorry, wrong")
         // do {
    loc_0×555555400a37:
        // CODE XREF from main @ 0×555555400a29
        eax = 0
        rcx = qword [var_8h]
        rcx ^= qword fs:[0×28]
        if  (!var) goto loc_0×555555400a50 // unlikely
         // } while (?);
         // } while (?);
        }
        return eax;
    loc_0×555555400a50: // orphan
            // CODE XREF from main @ 0×555555400a49
            leav                    // rbp
            re

    loc_0×555555400a4b: // orphan
         sym.imp.__stack_chk_fail  ()

}
[0×55555540098d]>
```

Trying this phrase as the password, it does not appear to work. This indicates that the password may have undergone some sort of encryption.

```
}
[0×55555540098d]>
zsh: suspended  r2 -d ./q1-2

┌──(root㉿kali)-[/home/kali/Desktop]
└─# ./q1-2
What is the password?
I Love Cyber Security
Sorry, wrong password ...

┌──(root㉿kali)-[/home/kali/Desktop]
└─#
```

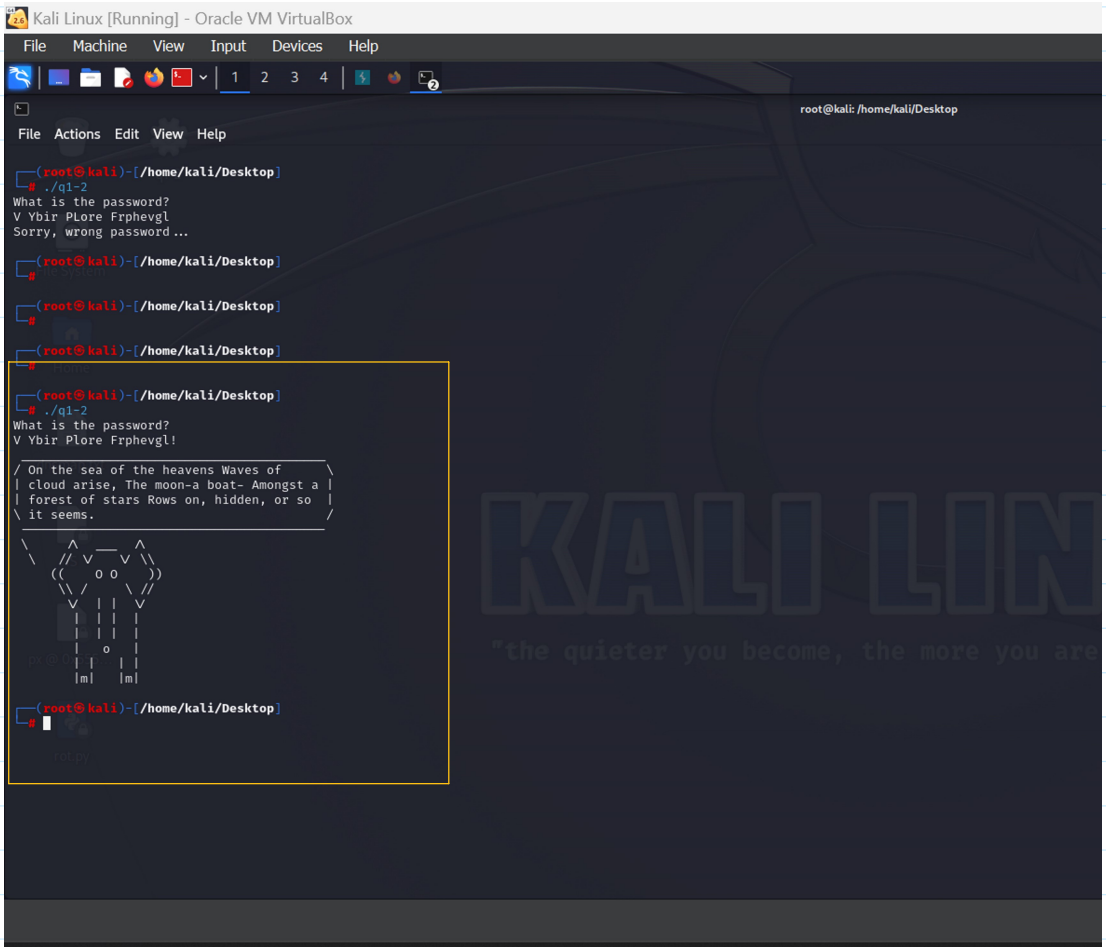
Taking a closer look in the decompiled code, it appears that the password is sent through a function called rot(). Rot is a simple letter substitution cipher.

```
sym.rot  () // sym.rot(0×0, 0×0)
rax = var_410h
rsi = rip + str.I_Love_Cyber_Security_ // 0×555555400cd7 // "I Love Cyber Security!"
rdi = rax
sym.imp.strcmp  ()
// int strcmp("", "I Love Cyber Security!")
```

There are a lot of ROT algorithms, lets assume that the code implements a ROT 13 which are common. ROT 13 replaces a letter with the 13th letter after it in the Latin alphabet. A simple python program such as one below can replicate this.

```
def rot13(text):
    result = ''
    for char in text:
        if 'A' <= char <= 'Z':
            result += chr((ord(char) - ord('A') + 13) % 26 + ord('A'))
        elif 'a' <= char <= 'z':
            result += chr((ord(char) - ord('a') + 13) % 26 + ord('a'))
        else:
            result += char
    return result
```

Running this script gives us the password which is "V Ybir Plorer Fpvcurel!". Now to verify, we use this as shown in the screenshot below. If this assumption was wrong, I would have used Ghidra to decompile and see more detail regarding the ROT algorithm used.



q2. [2 points] Reversing (2)

- **Download this binary Download binary. You can run it as "./q2" in Linux**
- **You ARE NOT allowed to patch this program**
- **Use Ghidra/Cutter/Radare2 to determine what you need to do outside of the program to get it to reveal the secret**
- **Get the program to print the secret**

Running the program to observe its behaviour, it appears that the program does not want to display the secret.

```
┌──(root㉿kali)-[/home/kali/Desktop]
└─# chmod +x q2-2.2

┌──(root㉿kali)-[/home/kali/Desktop]
└─# ./q2-2.2
Sorry no secret for you!

┌──(root㉿kali)-[/home/kali/Desktop]
└─#
```

Using Radare2 to disassemble the code, it appears that the yellow boxed code might be of interest.

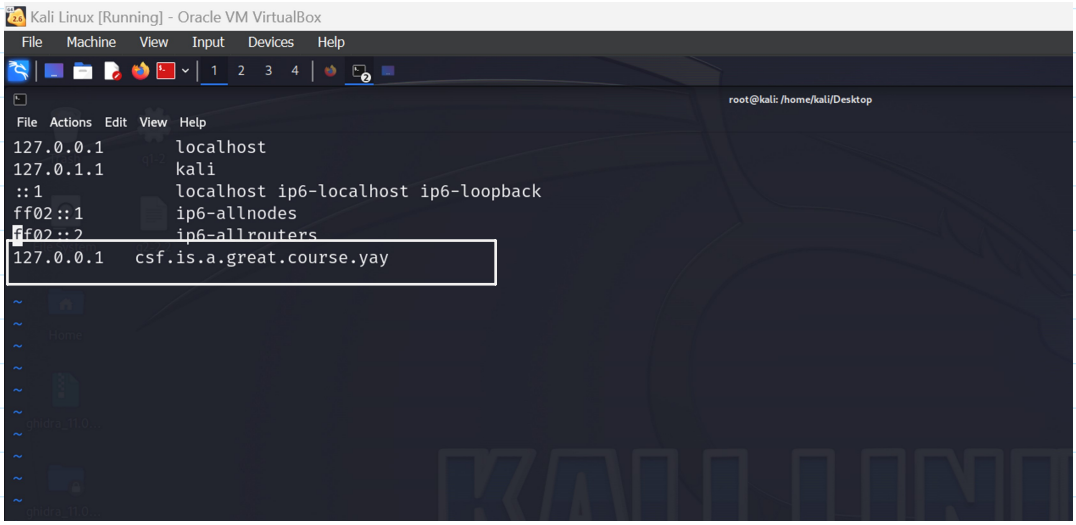
```
┌──(root㉿kali)-[/home/kali/Desktop]
└─# r2 -d ./q2-2.2

[0×7ffff7fe54d0]> s main
[0×555555400820]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Finding and parsing C++ vtables (avrr)
[x] Skipping type matching analysis in debugger mode (aaft)
[x] Propagate noreturn information (aanr)
[x] Use -AA or aaaa to perform additional experimental analysis.
[0×555555400820]> pdc
int main (int esi, int edx) {
    loc_0×555555400820:
        // DATA XREF from entry0 @ 0×55555540065d
        push  (rbp)
        rbp = rsp
        rsp -= 0×20
        dword [var_14h] = edi // argc
        qword [var_20h] = rsi // argv
        rdi = rip + str.csf.is.a.great.course.yay // 0×555555400a1f // "csf.is.a.great.course.yay"
        sym.imp.gethostbyname  ()
        qword [var_8h] = rax
        var = qword [var_8h] - 0
        if  (var) goto loc_0×555555400854 // unlikely
    loc_0×555555400854:
        // CODE XREF from main @ 0×555555400844
        eax = 0
        sym.print_secret  ()
         // do {
    loc_0×55555540085e:
        // CODE XREF from main @ 0×555555400852
        eax = 0
        leav      // rbp
        re
         // } while (?);
         // } while (?);
        }
        return eax;
}
[0×555555400820]>
```

Using Radare2 does not reveal enough information, using Ghidra we get the following decompiled code.

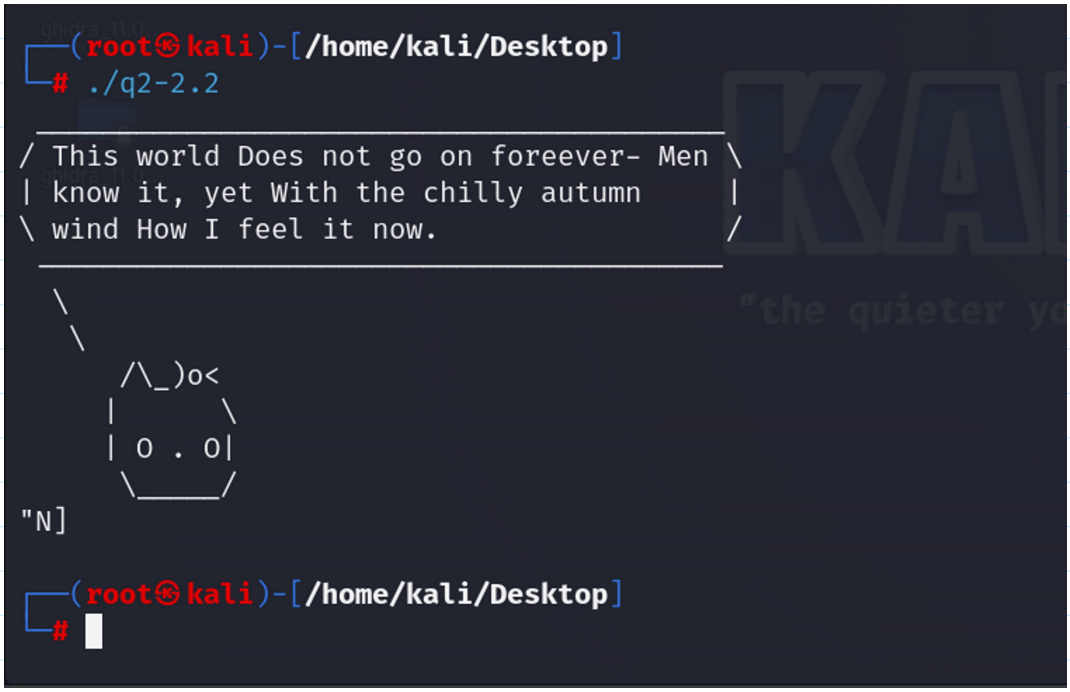
```
undefined8 main(void)
{
  hostent *phVar1;

  phVar1 = gethostbyname("csf.is.a.great.course.yay");
  if (phVar1 == (hostent *)0x0) {
    puts("Sorry no secret for you!");
  }
  else {
    print_secret();
  }
  return 0;
}
```

The above decompiled code reveals that to obtain the secret, we need to ensure that the hostname "csf.is.a.great.course.yay" resolves to a valid IP address. This can be done by adding this in the hosts file in the /etc/hosts directory.

The screenshot below shows the entry added into the hosts file. This will bypass the hostname resolution check.



Running the program again, it now resolves the hostname "csf.is.a.great.course.yay" to the IP address 127.0.0.1 and the function print_secret is called printing the secret as shown in the screenshot below.



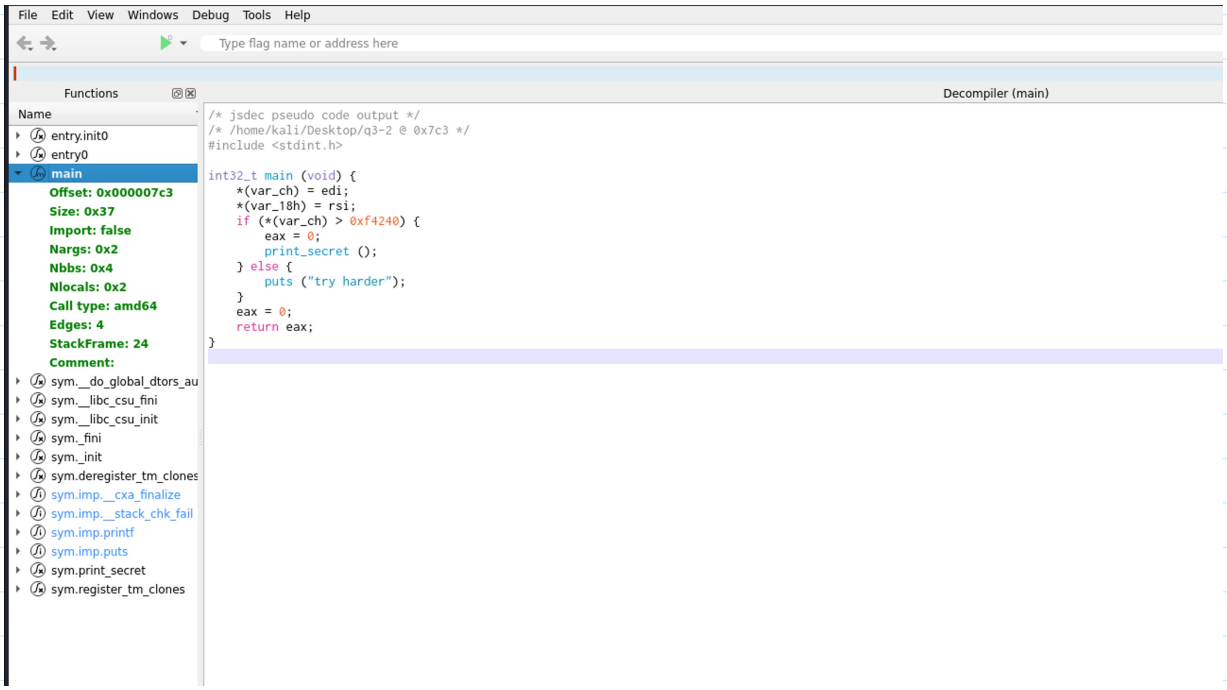
q3. [2 points] Reversing (3)

- **Download this binary Download binary. You can run it as "./q3" in Linux**
- **YOU SHOULD patch this program to get it to reveal the secret using Ghidra/Cutter/Radare2 (as per workshop, I think Cutter is easiest, but feel free to use any program).**
- **Use the modified (patched) program to print the secret**

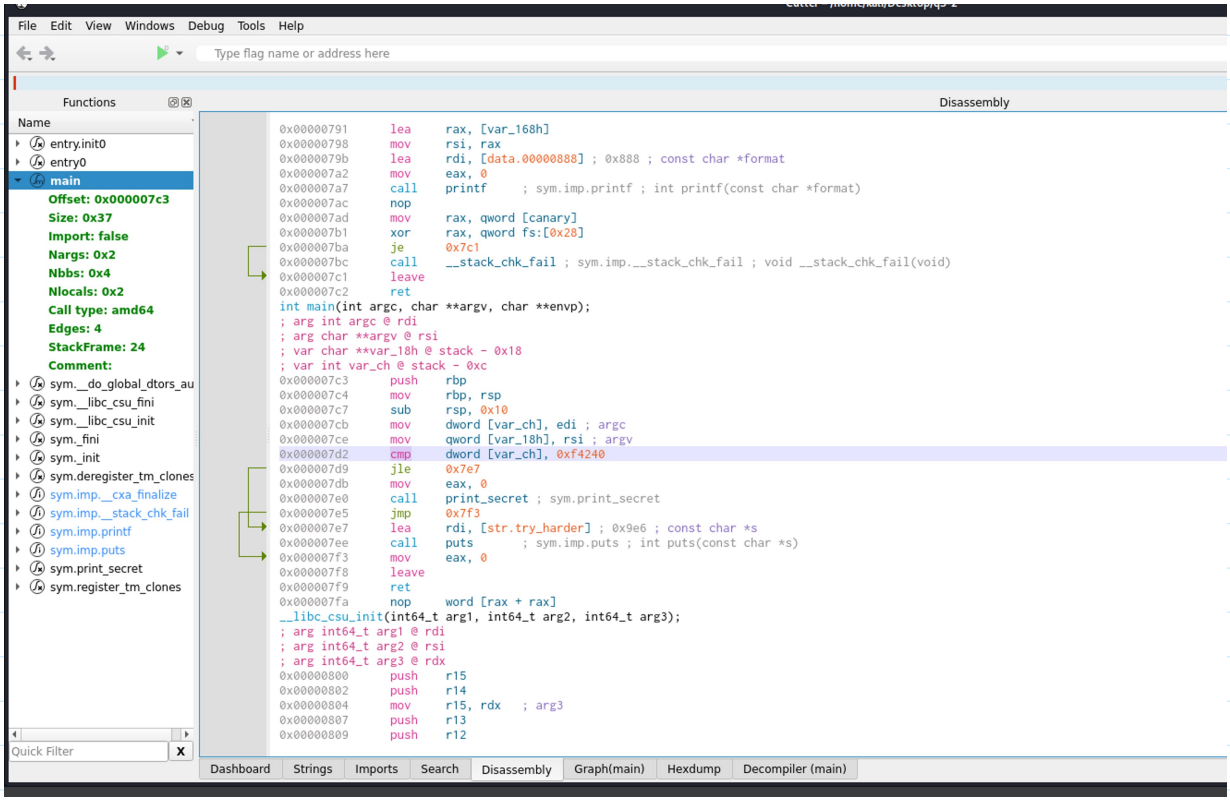
Running the program reveals nothing.

```
┌──(root㉿kali)-[/home/kali/Desktop]
└─# ls
Cutter-v2.3.4-Linux-x86_64.AppImage  ghidra_11.0.3_PUBLIC  ghidra_11.0.3_PUBLIC_20240410.zip  q1-2  q2-2.2  q3-2

┌──(root㉿kali)-[/home/kali/Desktop]
└─# ./q3-2
try harder

┌──(root㉿kali)-[/home/kali/Desktop]
└─#
```

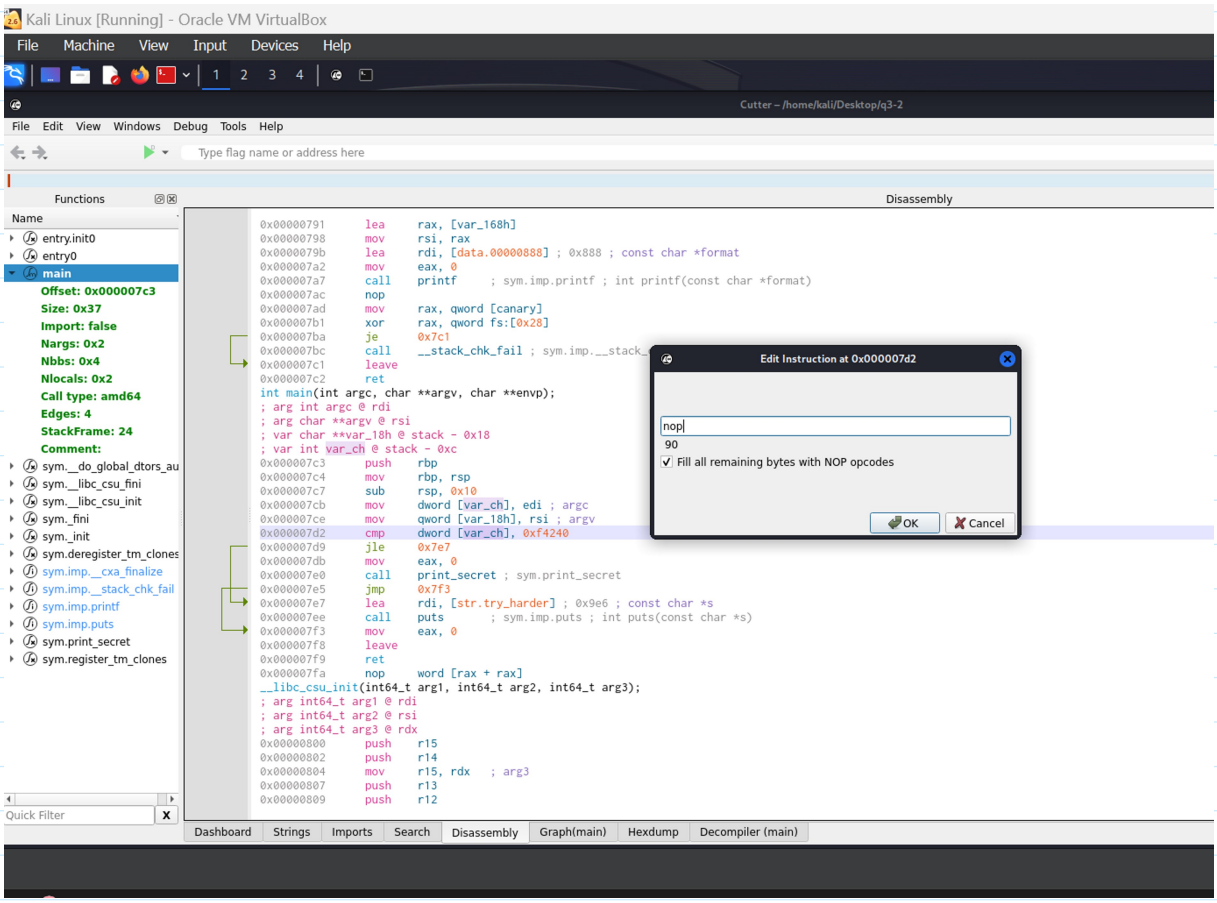
Then onto decompiling the program, It appears that the program compares the parameter with the hexadecimal value 0xf4241. If the parameter is less than the hexadecimal, a "try harder" message is printed, else it prints the secret.



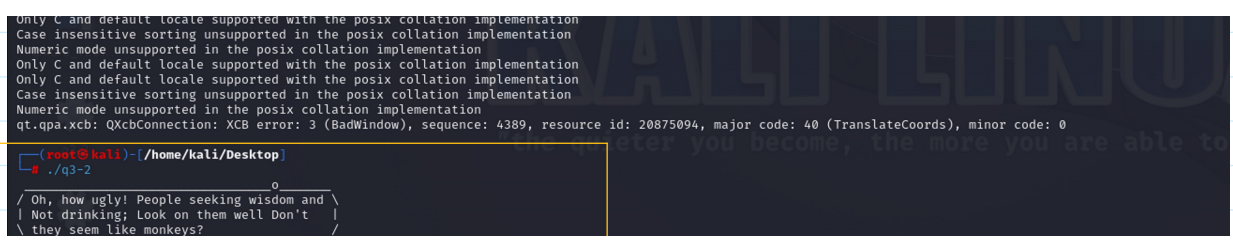
As suggested in the question, the binary needs to be patched to ensure that the conditions checking the parameter against the hexadecimal always evaluates and jumps to print the secret. We can do this by changing the comparison instruction using Cutter.



As shown in the screenshot below, we can change the line "cmp dword [var_ch], 0xf4240" to nop i.e. The instruction is then skipped.



Now that we have patched the condition, the condition will always evaluate to print the secret. This is shown in the screenshot below.



```
┌──(root㉿kali)-[/home/kali/Desktop]
└─# ./q3-2
 _____________________________o_____
/ Oh, how ugly! People seeking wisdom and \
| Not drinking; Look on them well Don't   |
\ they seem like monkeys?                 /
 -------------------------------------
       \   ^__^
        \  (@@)\_______
           (__)\       )\/\
               ||----w |
               ||     ||

┌──(root㉿kali)-[/home/kali/Desktop]
└─#
```

q4. [3 points] Matryoshka

- **Analyse this Hacktivist2-2.png downloadto get the secret!**

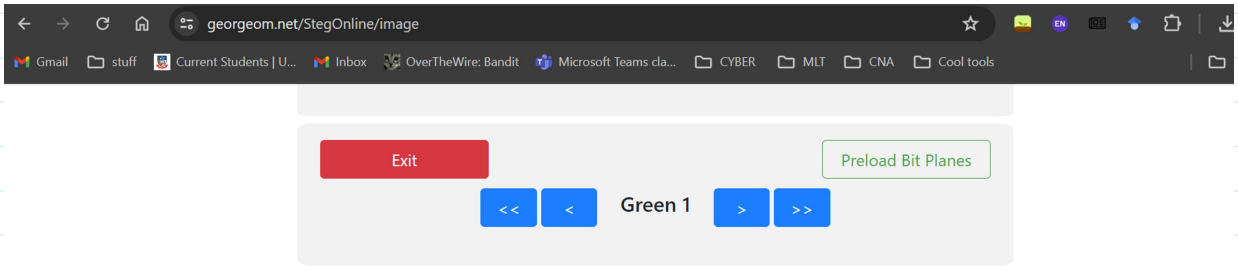
Observing the file, it appears to be a PNG file.

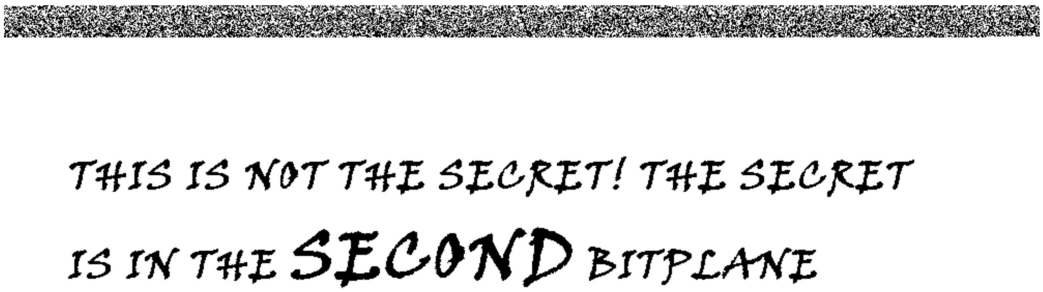
```
┌──(root㉿kali)-[/home/kali/Desktop]
└─# ls
Cutter-v2.3.4-Linux-x86_64.AppImage  ghidra_11.0.3_PUBLIC  ghidra_11.0.3_PUBLIC_20240410.zip  Hacktivist2-2.png  q1-2  q2-2.2  q3-2

┌──(root㉿kali)-[/home/kali/Desktop]
└─# strings Hacktivist2-2.png| more
IHDR
IDATx^
243$
K0lP
*z(?o[=
cN(>
2&      ;
Eq}a
_5&?
>8l[AlF
d>(_
:n.(V
y^yF
N2-n
?qzs%
tlf%
06&gN
:web'
BfA"
O'mA&
$jN*m
        O;^'
/5&e6
b#x^&
KqXI
}2oA
il@$
C@H/
8SuM
YOP9%/T
bTF@
        :@af,ZU
CbyR@/
@>WO
Ip#7=
Mr5BLv
>GQI
Yec\
[ymN
Zr</sU$T
RD$zJ
!'8Z
```

Looking deeper using binwalk, we can see that there is a PNG image and a Zlib compressed file.

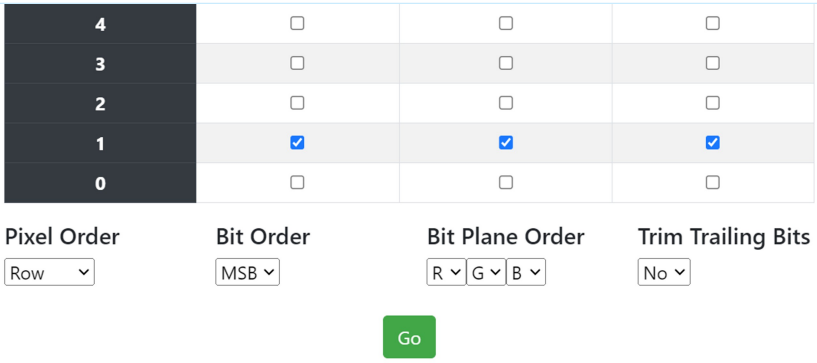
```
┌──(root㉿kali)-[/home/kali/Desktop]
└─#

┌──(root㉿kali)-[/home/kali/Desktop]
└─# binwalk Hacktivist2-2.png

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             PNG image, 1000 x 1359, 8-bit/color RGBA, non-interlaced
41            0x29            Zlib compressed data, compressed

┌──(root㉿kali)-[/home/kali/Desktop]
└─#
```

Using StegOnline and searching each plane, I found the below image inside the first green plane.





I then extract the bitplane and it appears to contain a zip file.

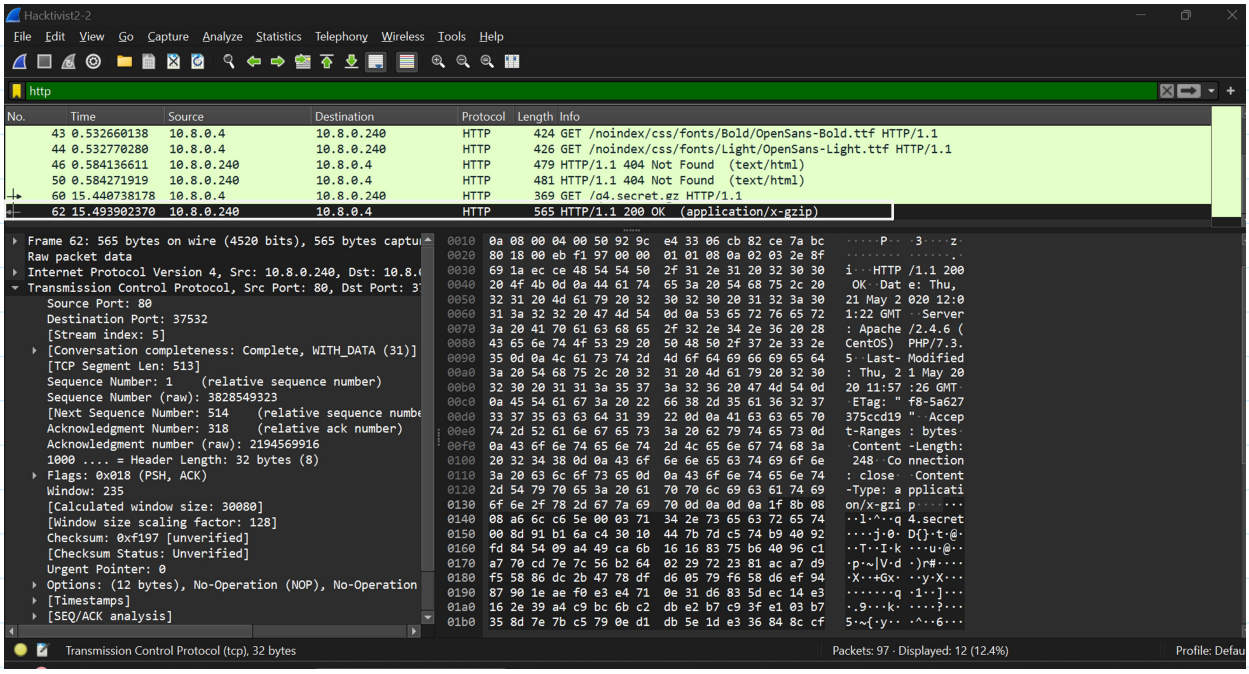


Downloading this and using the file command on it, we find that this is a pcapng type file.

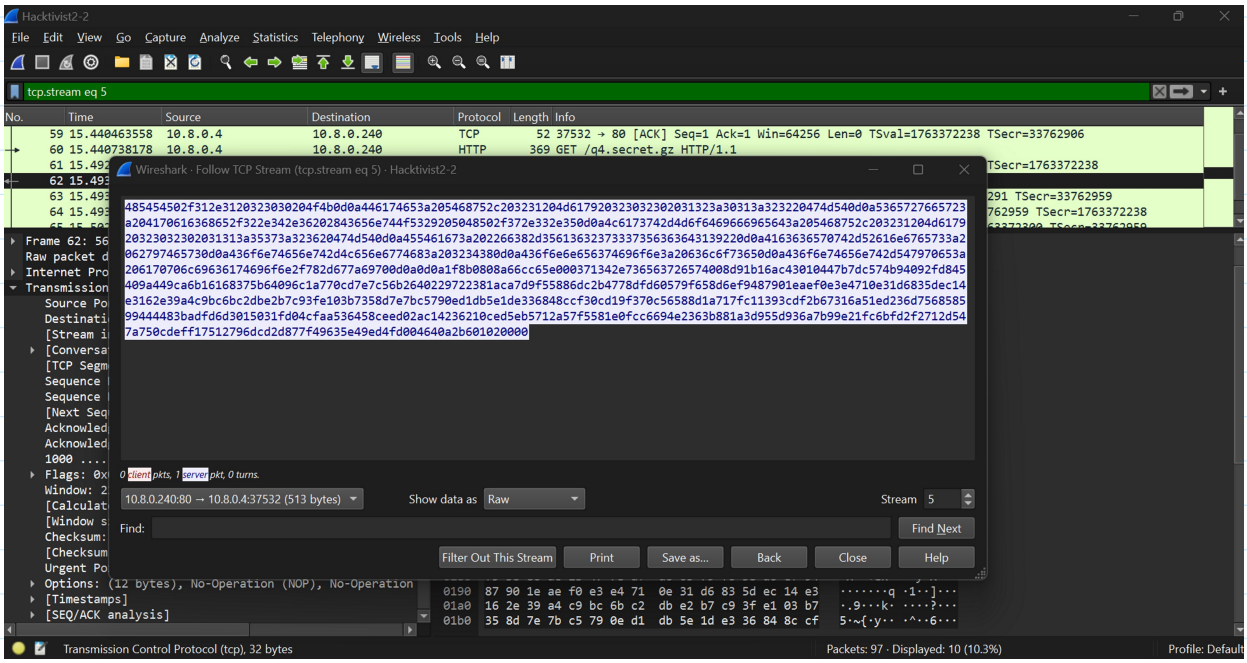
```
amna@DESKTOP-P15VFPV:/mnt/c/Users/honey/OneDrive/Documents/Desktop/Year 4/Cyber/Assignment5$ file Hacktiv
ist2-2
Hacktivist2-2: pcapng capture file - version 1.0
amna@DESKTOP-P15VFPV:/mnt/c/Users/honey/OneDrive/Documents/Desktop/Year 4/Cyber/Assignment5$
```

I then decided to use Wireshark to extract more information from this.

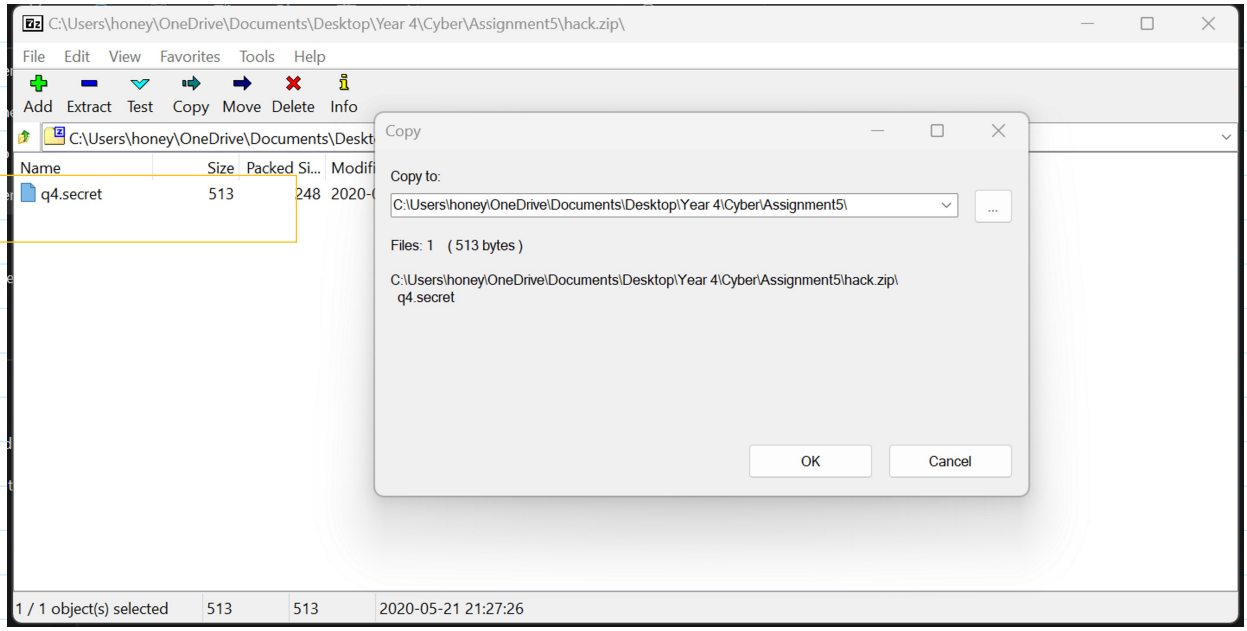
Opening the pcapng file in Wireshark and then filtering by http requests, I come across the "200 OK" request.



Then right clicking the request and then following the HTTP stream , we can see more raw data which we can export as a zip file.



Extracting this zip file again, we finally see the q4.secret file.



Then simply using the cat command displays the secret.

```
amna@DESKTOP-P15VFPV:/mnt/c/Users/honey/OneDrive/Documents/Desktop/Year 4/Cyber/Assignment5$ ls
Hacktivist2-2  hack.zip  q1-2  q4.secret
amna@DESKTOP-P15VFPV:/mnt/c/Users/honey/OneDrive/Documents/Desktop/Year 4/Cyber/Assignment5$ cat q4.secre
t
 ________________________________________
/ From the mountain's edge Will the      \
| drifting moon Emerge, I wonder? While I |
\ wait Night has fallen.                 /
 ----------------------------------------
  \     .    _  .
   \    |\_|/__/|
        / / \/ \  \
       /__|O||O|__ \
      |/_ \_/\_/ _\ |
      | | (____) | ||
      \/\___/\__/  //
      (_/        ||
       |         ||
       |         ||\
        \       //_/
         \_____//
        __ || __||
       (____(____)
amna@DESKTOP-P15VFPV:/mnt/c/Users/honey/OneDrive/Documents/Desktop/Year 4/Cyber/Assignment5$
```