William Hupp
CIS5930

Final Write-Up

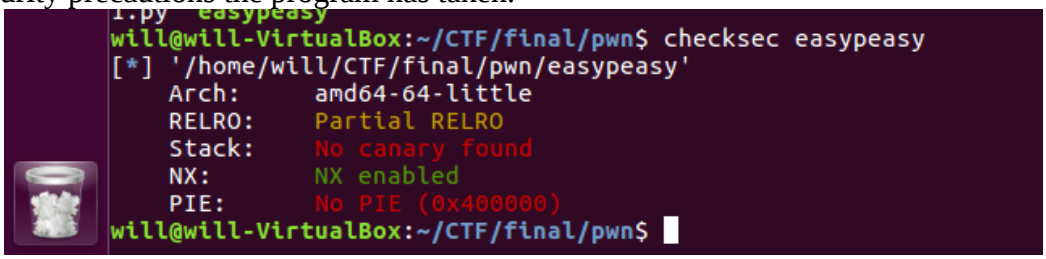
Team: betterThanWoodie

Pwn1: easypeasy

nc pwn.n0l3ptr.com 10981

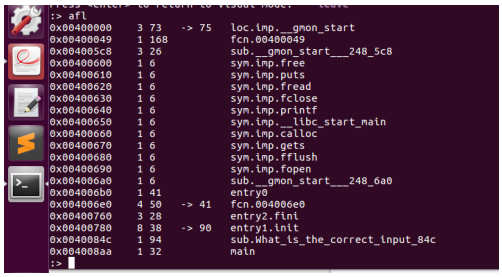
Solution:

First, I downloaded the file ‘easypeasy’. I then proceeded to run ‘checksec’ on the program to see what kind of security precautions the program has taken.

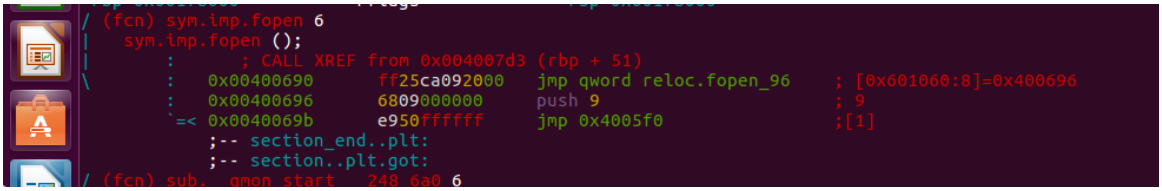


NX is enabled so this is not a shell code injection. Next, I run strings on the program and see the strings “./flag” and “No flag found!:(“. Therefore, the program prints the flag. When running the program, with ‘nc pwn.n0l3ptr.com 10981’, I am prompted for the correct input.

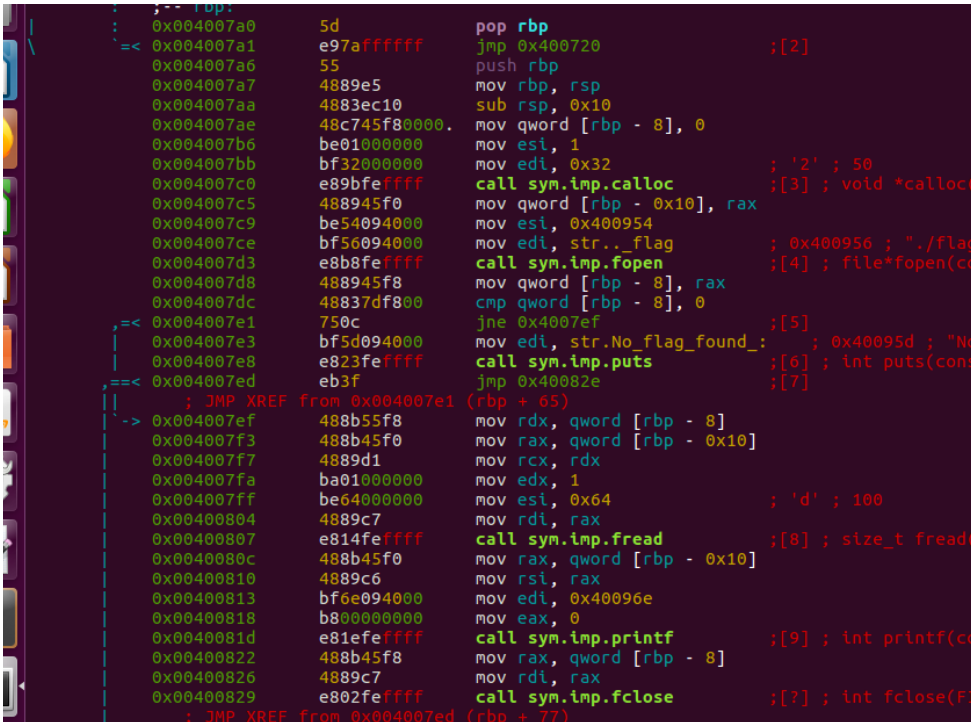
The program just prints back whatever I type. I run the file command and see ‘easypeasy’ is a 64 bit ELF stripped binary. Therefore, registers are 8 bytes. Next, I analyze the program with Radare. I type ‘aaaa’ into Radare to analyze the binary and then ‘afl’ to view functions.



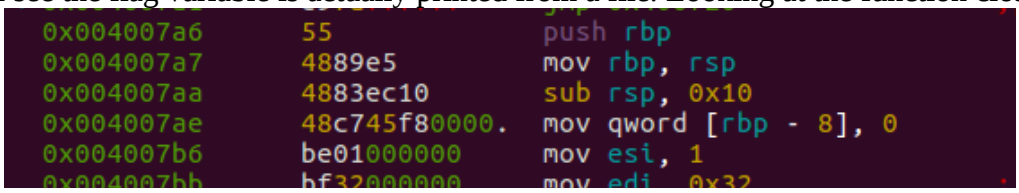
Fopen seems useful, so I seek to fopen with ‘s 0x00400690’ and then type V for visual mode and ‘p’ a few times to view instructions.



Here we can see an xref from 0x4007d3. This means that location is calling fopen(). So we navigate to that instruction.



Here we can see the flag variable is actually printed from a file. Looking at the function closer.



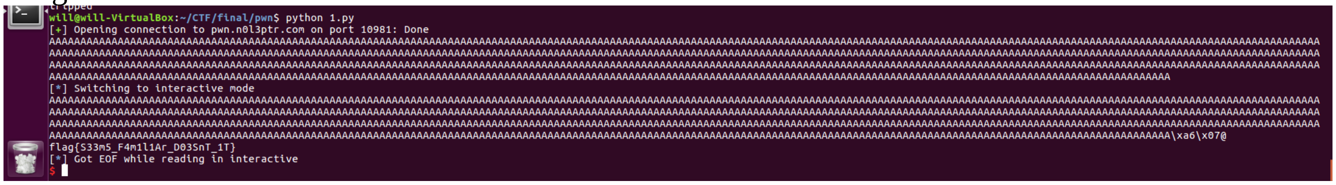
This looks like the start of a function, however it is not labeled as one in Radare. To me, 0x4007a6 looks like the start of a printFlag function. Since the main execution of the program doesn't print the flag, I must overwrite some return address to 0x4007a6 to print the flag. Now we need to find the vulnerable buffer. Looking at main

We see that main calls What_is_the_correct_input and that function has a gets function. The parameter for the gets function is ‘local_310h’ and that is our vulnerable buffer. That buffer is stored at rbp-0x310 and that is the stack space allocated for the function What_is_the_correct_input. We have to overwrite 0x310 bytes of buffer to overwrite the return address that was pushed when What_is_the_correct_input is called. However, 0x310 takes us to the beginning of rbp. We also have to overwrite rbp which is 8 bytes. Therefore we must write 0x318 bytes before we can overwrite the return address. To do so I wrote a python script attached as ‘pwn1.py’

```
1  from pwn import *
2  r = remote('pwn.n0l3ptr.com', 10981)
3  a=r.recvuntil("? ")
4
5  A= "A"
6  for i in range(0,791):
7      A = A + "A"
8
9  r.sendline(A+ p32(0x4007a6, endian="little"))
10 print A
11 r.interactive()
```

I import pwntools and connect to the server using pwntools. I then receive input up until I am prompted to enter the correct input. What I enter in as input will go to the vulnerable buffer in the gets function. I then create a string of 792 As (0x318 = 792). This fills up the buffer and rbp to the return address. This works because 1 character = 1 byte. I then pack the address of the supposed printFlag() function into raw bytes so the program can interpret the return address. I send the string of As and raw bytes of the supposed printflag function (using pack()) to overwrite the return address of What_is_the_correct_input with the address of printflag(). I then use r.interactive() to interact with the

program and receive the response from the server (ie. the flag). After running the program I get the flag.



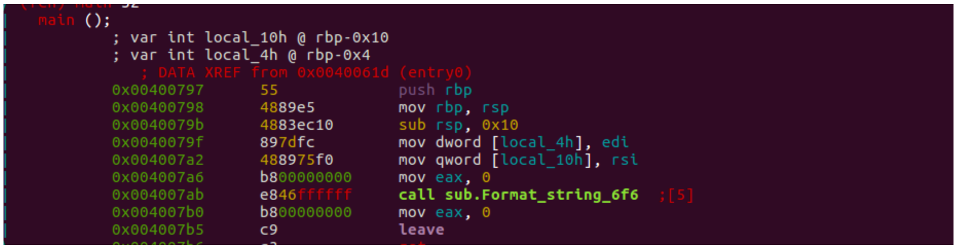
flag{S33m5_F4m1l1Ar_D03SnT_1T}

Pwn2: med

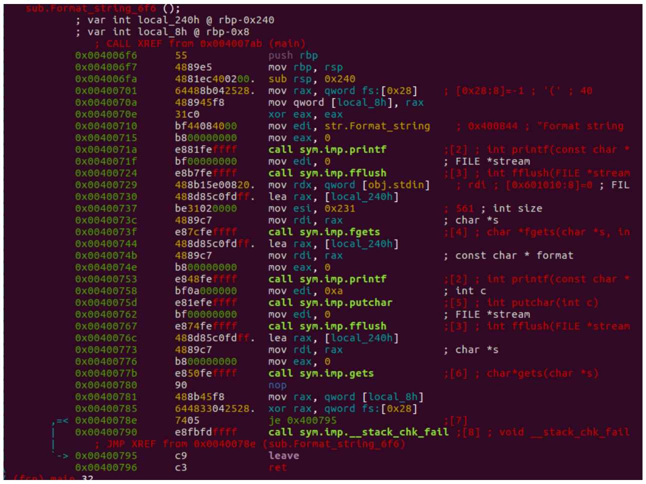
nc pwn.n0l3ptr.com 10982

Solution:

After unzipping the file, we see there are two files ‘libc-2.23.so’ and ‘med’. Running file on ‘med’ we see that med is a 64 bit ELF files with a stripped binary. Therefore, registers are 8 bytes. Running checksec on med, we see there is a full RELRO, canary, and NX is enabled so we will not be injecting shell code. ASLR is also enabled. After running ‘./med’ the user is prompted for a format string. This makes me believe this is a format string vulnerability. Fortunately, Sean did a format string vulnerability question in class and I copied his script. From Sean’s script, I know we will have to find the buffer location relative to ebp. I analyze the file in Radare. First, after tpying ‘afl’ I see a function called ‘libc_start_main’. In main, I see a call to Format_string.



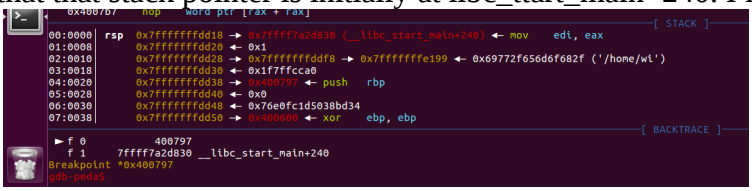
In Format_String



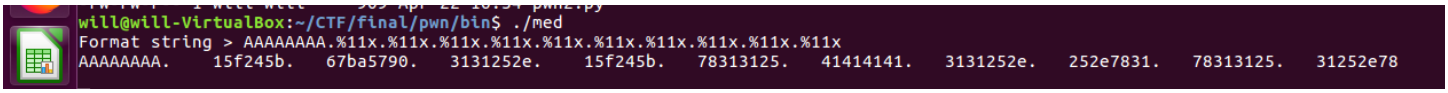
We see the function space is allocated to be 0x240 bytes. This happens to also be the buffer size for the fgets() function (local_240h is returned from the function call). 240h=rbp-0x240. Next, I will have to find the location of libc_start_main relative to the stack pointer. We know libc is aways called at the

start of main. We need to find the difference between the stack pointer when libc is called, to the stack pointer when fgets is called and the string format vulnerability is executed. First, the stack space for main is 0x10 and the stack space for Format_String is 0x240. That brings us to esp-0x250. 8 bytes of rbp are pushed on the stack in main and another 8 bytes of rbp are pushed on the stack in Format_String. Finally The 8 bytes of the return address from libc_start_main are pushed onto the stack. That leaves the difference between the stack pointer when libc.so is called and the stack pointer during the format string vulnerability to be 0x268.

It is important to note that that stack pointer is initially at libc_ttart_main+240. I found this using GDB.



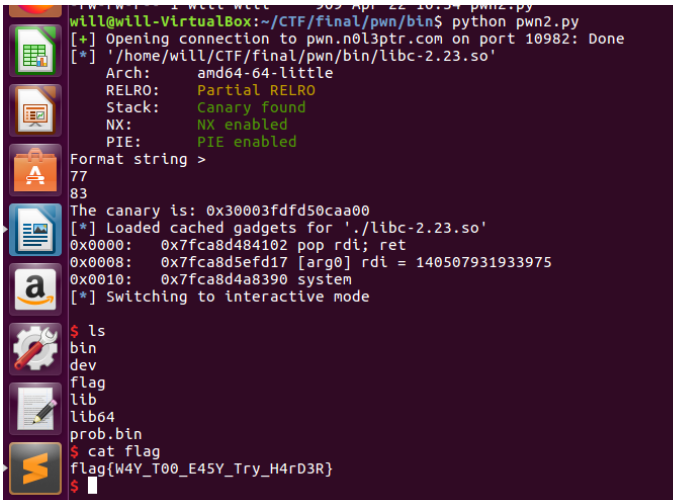
Next, I need to find which parameter for the the string format vulnerability is the buffer. According to the notes, I should input the following into the buffer to figure out which parameter the buffer is.



Here we can see ‘AAAAAAAA’ was entered as input into the buffer, and the hex value 0x41414141 is in the 6th %11x slot. Therefore, the buffer is the 6th parameter because A = 0x41. I now have enough information to write my script.

The code is attached ‘pw2.py’. This is mostly the same script Sean wrote in class. The offset, libc_start_offset, libc_base, and libc are the main differences. First, I set the context of the system. Then, I connect to the server and receive until the program prompts for input. I create a libc ELF file

using the provided libc.so file. I set offset=0x240 and lib_c_start_offset=0x268 based off of the previous calculations. I also set param_buffer to 0x6 because the buffer is the 6th parameter of the string format vulnerability. We also know there is a canary based off checksec. The canary is located 8 bytes before the saved ebp, so it is offset - 0x8. Next, I find the parameters needed to find the canary and libc_start_main using the buffer's parameter location (I divide by 8 because each entry in libc_start_main takes 8 bytes). Starting on line 19, I leak the canary and libc_start using the calculated libc_start and canary parameters. We need these values to spawn a shell using the string format vulnerability. A canary cannot be overwritten without knowing its value. Next, I use the leaked libc address to find the libc_base. Since libc_address = libc_start_main- 240, the base would be libc_address + 240. To get the actual base address of libc, we must subtract the offset of of the provided libc.so file (taken from the ELF object). I update libc_address to the be the base. Next, I create a payload and fill the buffer with 'A's. Then I use hexspeak to pad the stored rbp value. I create a ROP object of the calculated value of libc_start_main (libc_address) to return the location of /bin/sh. Only though finding the correct location of libc_start_main (stored in libc_address), can we find the shell location. Next, I overwrite Format_String's return address to be the libc_start_main shell. I send the padding of 'A's to fill the buffer, followed by the canary, the padding for rbp, and finally the return address of the shell. After running the program I spawned into a shell.



I interact with the shell and see there is a flag. I use cat to print the flag.
flag{W4Y_T00_E45Y_Try_H4rD3R}

Web1: Easy

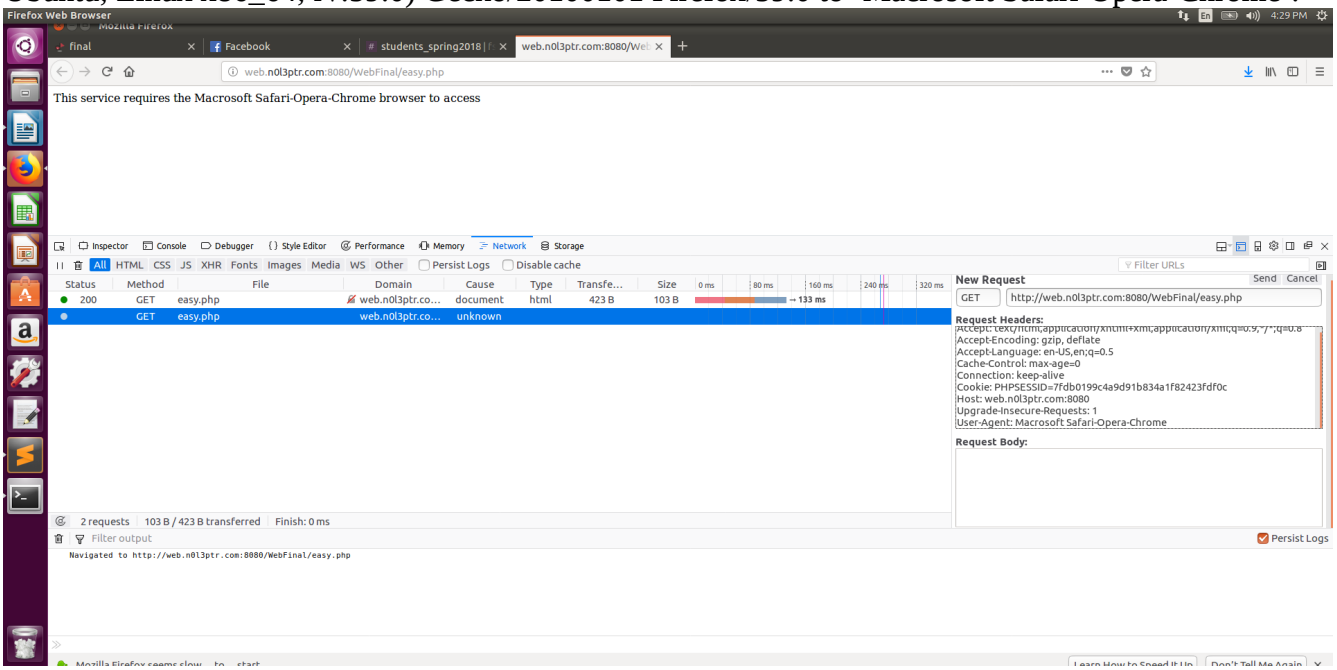
You know what to do - Start up everyone's favorite web client, put your thinking cap on, and do some math.

http://web.n0l3ptr.com:8080/WebFinal/easy.php

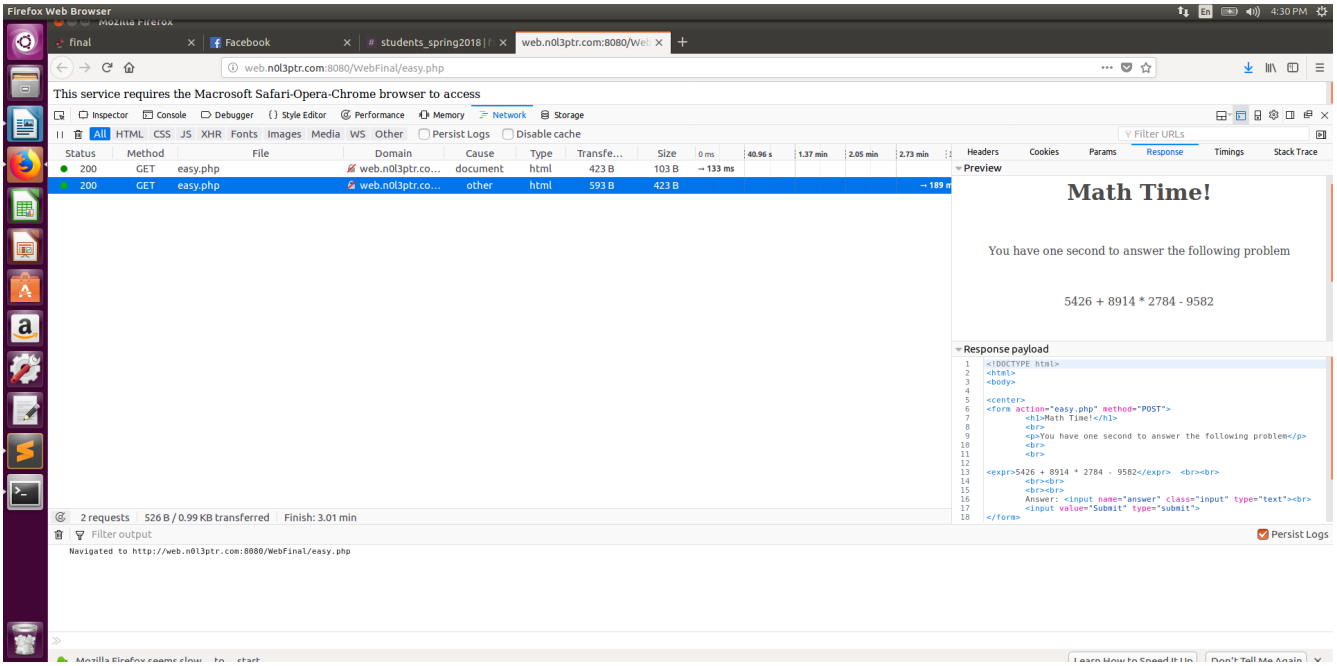
Solution:

First, I analyze the website. It is a blank website that states 'This service requires the Macrosoft Safari-Opera-Chrome browser to access'. Therefore, I must make the server think I am on the Macrosoft Safari-Opera-Chrome browser. I do this by editing the User-Agent. On the website, I hit ctrl-shift-I to open up network, refresh the page to record the easy.php file in the network tab, then I click on

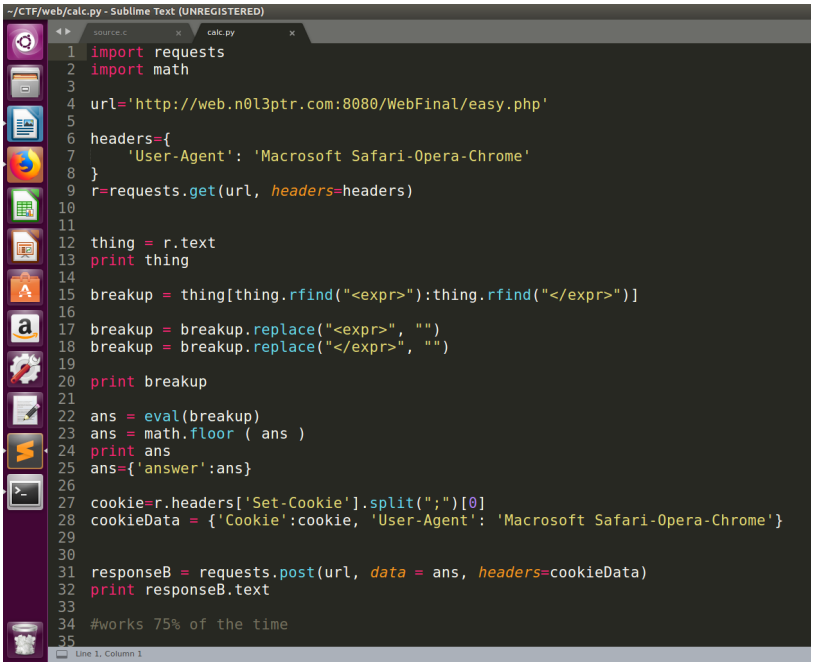
easy.php. Under headers, I click ‘edit and resend’ and change the User-Agent from Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:59.0) Gecko/20100101 Firefox/59.0 to ‘Macrosoft Safari-Opera-Chrome’.



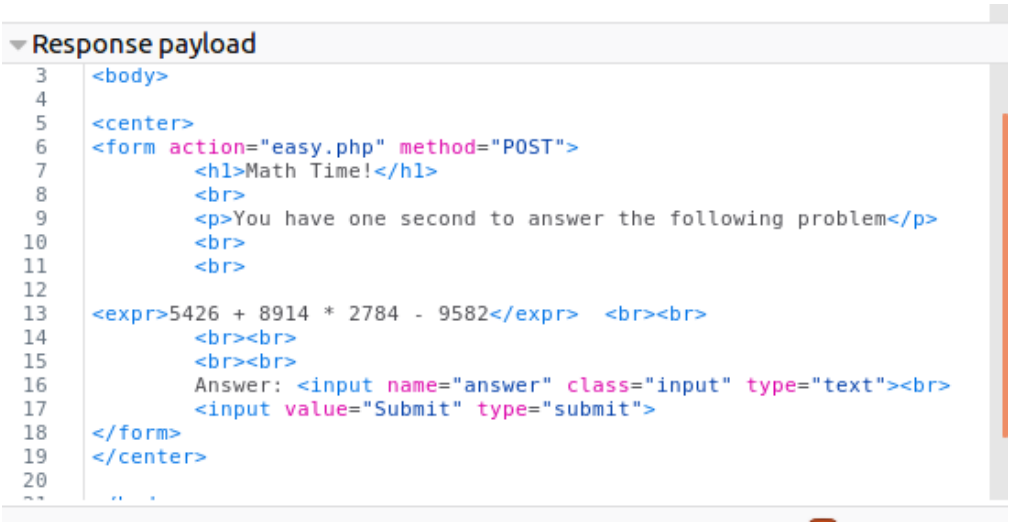
After sending that request, a new easy.php file is generated. By clicking on that file, and then clicking response, we see the website through the hands of a new User-Agent.



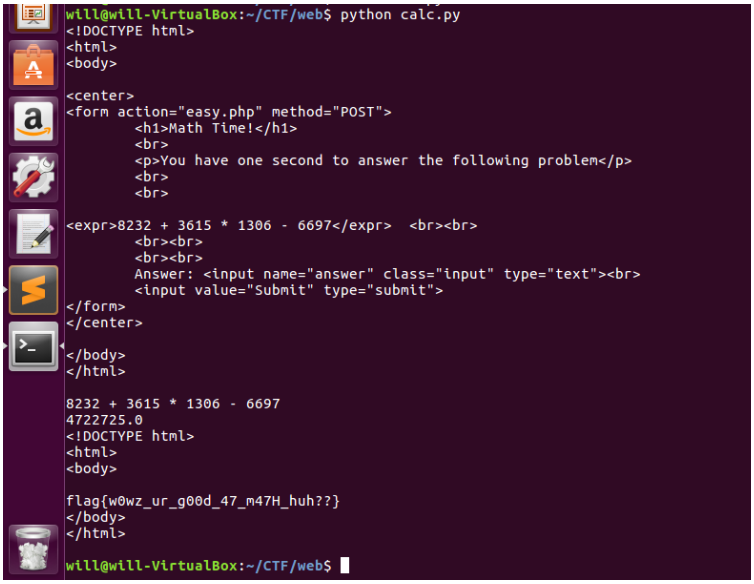
Here we see we have to solve a math question in 1 second and we are provided the source code for the website. However, the equation changes every visit to the website. This is similar to what we did in WebI, so I will alter my script from that homework. My script is attached as ‘calc.py’



First I import the request library, so I can interact with the website. Then, I import the math library. I set the url to the target wesite and create a a headers variable where the User-Agent is set to Macrosoft Safari-Opera-Chrome. I then use requests.get to retrieve the information from the calculator website using the correct User-Agent. Analyzing the HTML of the website, I see the following.



The math expression is stored between <expr> and </expr>. So, in my script, I parse the data between those two string, eliminate <expr> and </expr>, and am left with the string of the math expression. I then use the eval function from math to evaluate the string and floor the result as suggested. I proceed to set the answer variable to be the result of the expression. I then store the cookie information from the website into the variable cookie. I then set the headers of User-Agent and Cookies to Macrosoft Safari-Opera-Chrome and the websites cookies respectively using the cookieData variable. I then send the response back to the server with the requests.post function. The result of the expression is used as data for the post, and the cookieData variable is used as the header for the post with the appropriate cookie and user agent. I then print the response. After running my script I get the flag.



flag{w0wz_ur_g00d_47_m47H_huh??}

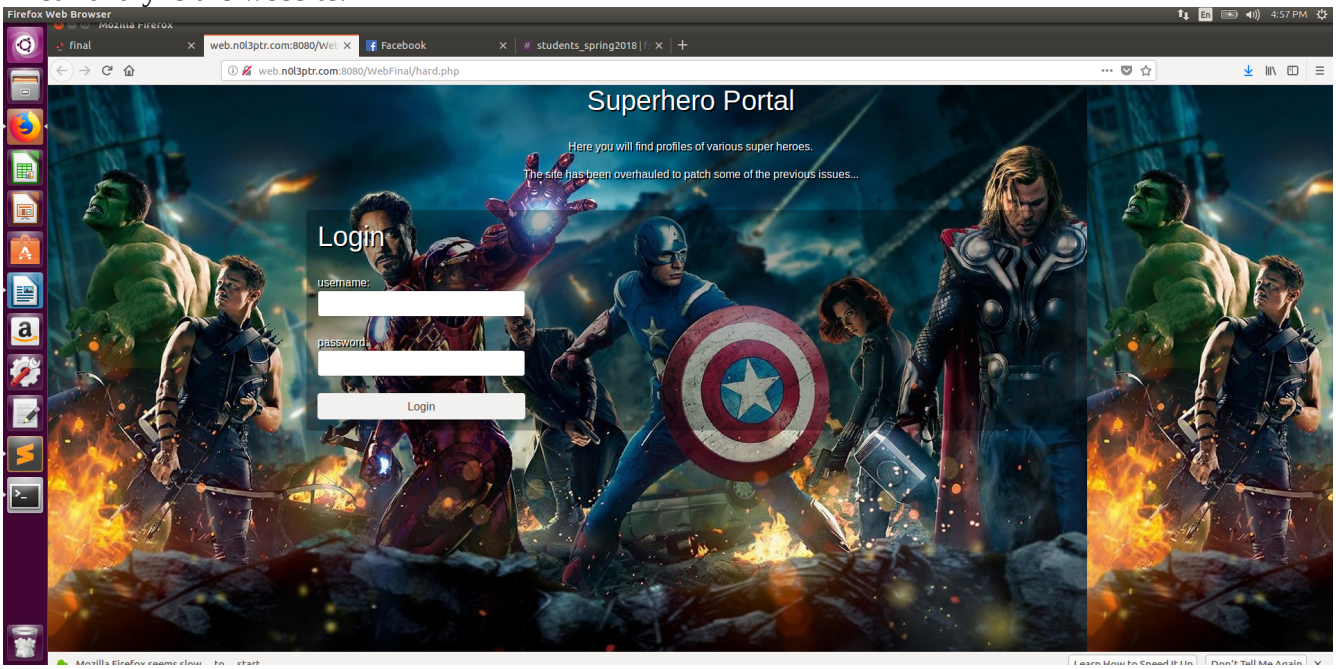
Web3: Hard

The superhero gang is back and this time they fixed some of the issues with their previous site. Can you still break into Ironman's account?

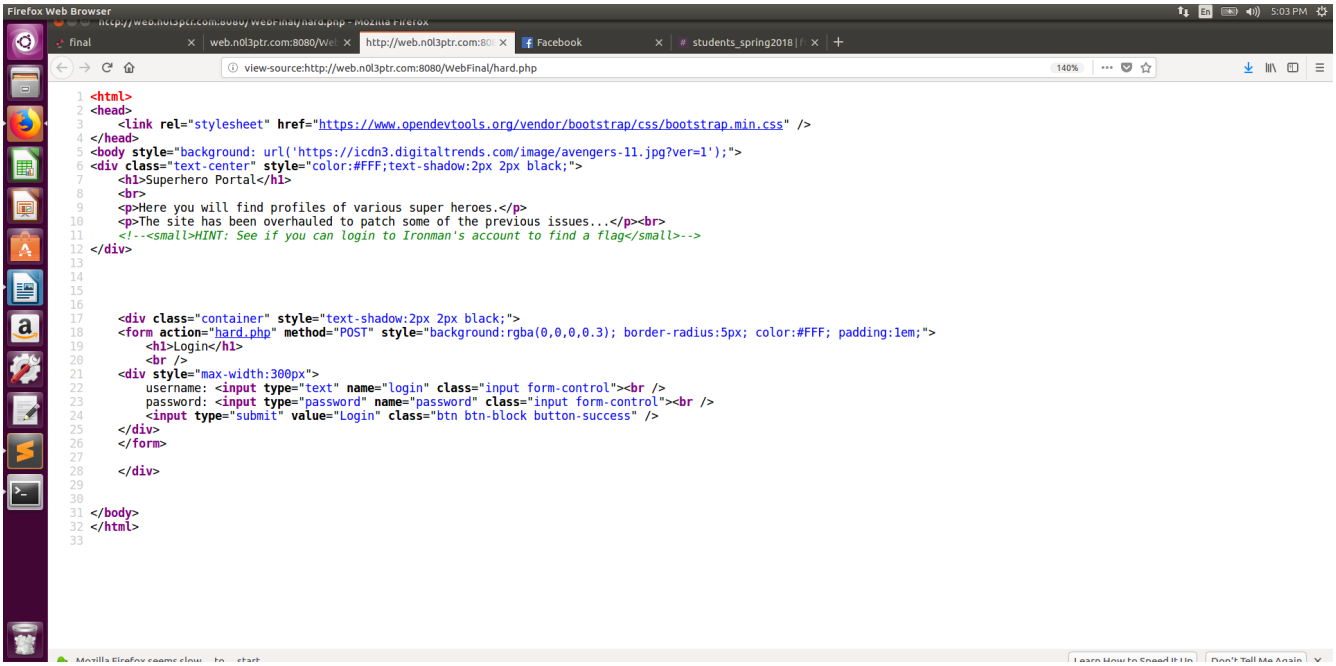
http://web.n0l3ptr.com:8080/WebFinal/hard.php

Solution:

First I analyze the website.



This is the same website as the sqlmap problem from WebII. Next I analyze the source code.



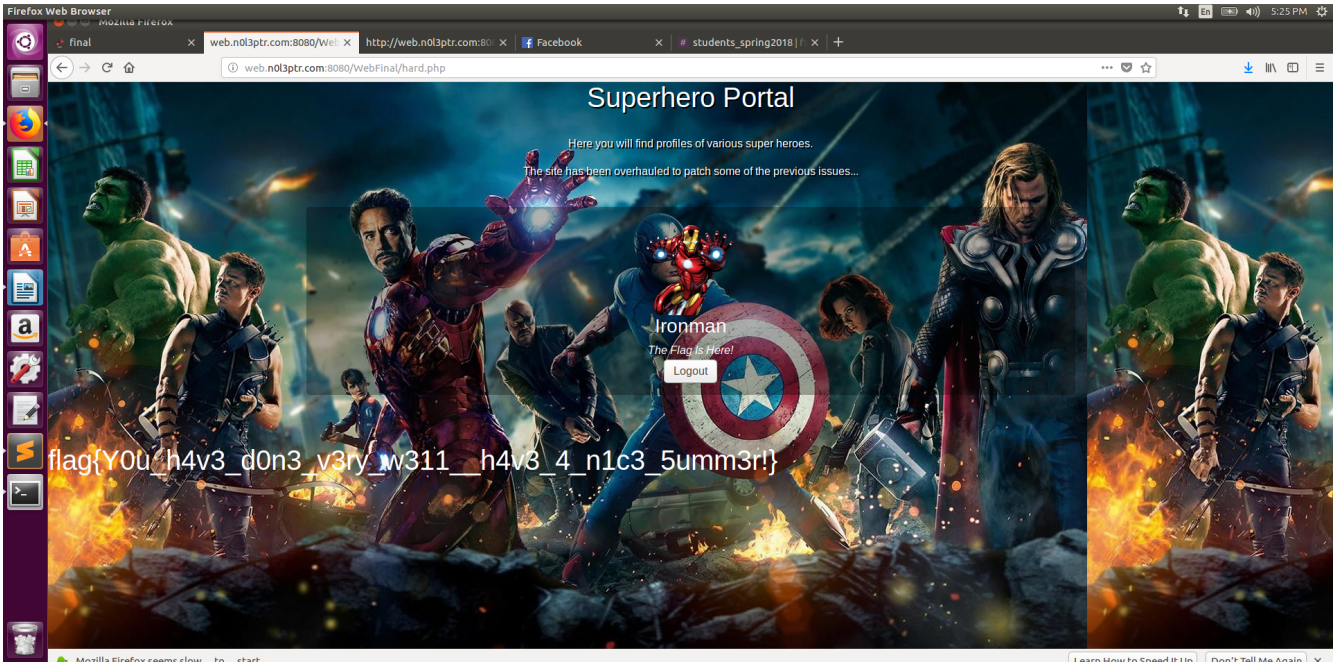
Here we can see that we must again login to Ironman's account. The website is using method=POST. The two fields displayed on the website are titled login and password. However, this time we don't have the SQL statements. I know I will have to use sqlmap to sql inject into the website's database and retrieve Ironman's login as I did in the previous super hero problem. I run the command 'python sqlmap.py -u 'http://web.n0l3ptr.com:8080/WebFinal/hard.php?login=Ironman&password=test' --method=POST --sql-shell' I used user=Ironman&password=test because we know we must login to the Ironman account, so the login must exist. I had to specify the method of the login with –method=POST. I used the –sql-shell parameter as recommended by the notes. Sql-shell invoked the built in SQL interpreter, so I can interact with the injected websites SQL. After running the command, I inject into the default location and spawn a sql shell where I can interact with the database through executing SQL commands.

However, SELECT * FROM heroes does not query the tables anymore like it did in the last homework. SELECT * returns an empty query as well. That means the table of heroes and users must be hidden. So, I use 'SELECT table_name FROM information_schema.tables' to get all of the tables., including hidden ones, from the database. This is a handy select statement I learned in databases that actually

came in handy. After executing that command we see there is a table titled ‘secret’. This looks interesting.

To view the secret table I ‘SELECT * FROM secret’ (this is SQL statement to view the contents of a table).

On the last line we can see Ironman’s account information. According to the SQL statement from the challenge in WebII, the password is stored right after the username. Therefore, ‘nucl34r_r34c70r’ is the password to the Ironman account. Entering in these credentials we get the flag.



flag{Y0u_h4v3_d0n3_v3ry_w311__h4v3_4_n1c3_5umm3r!}

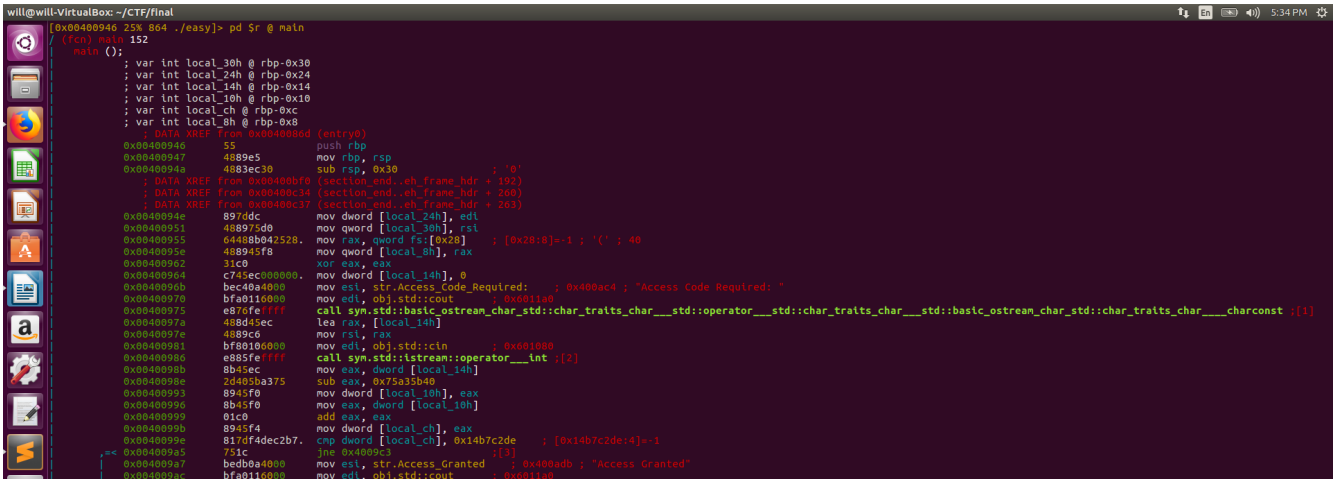
RE1: Access Code

Can you break the access code?

Flag format: "flag{" + accesscode + "}"
EX: flag{1234567890}

Solution:

First I downloaded the ‘easy’ file. I run strings and see ‘Access Granted’ but no flag. However, the access code is the flag, so I need to reach Access Granted. I run file on the program and see it is 64 bit ELF not stripped binary. I open the program in Radare, type ‘aaaa’ to analyze binaries and ‘afl’ to see functions. I see nothing interesting, so I seek to main with ‘s main’. I open visual mode with V and hit ‘p’ a few times to to view instructions. In main I notice the following,



Here we can see Access Granted is being printed after “cmp dword [local_ch], 0x14b7c2de”, “jne 0x4009c3” That means local_ch at this point must be the user input. Looking at the function a little closer we can see what is happening.

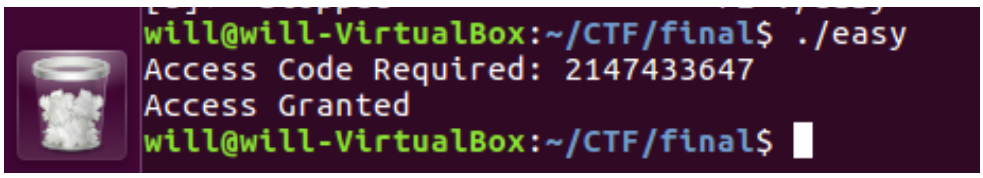
```
0x0040097e      4889c6          mov rsi, rax
0x00400981      bf80106000      mov edi, obj.std::cin       ; 0x601080
0x00400986      e885feffff      call sym.std::istream::operator___int ;[2]
0x0040098b      8b45ec          mov eax, dword [local_14h]
0x0040098e      2d405ba375      sub eax, 0x75a35b40
0x00400993      8945f0          mov dword [local_10h], eax
0x00400996      8b45f0          mov eax, dword [local_10h]
0x00400999      01c0            add eax, eax
0x0040099b      8945f4          mov dword [local_ch], eax
0x0040099e      817df4dec2b7.   cmp dword [local_ch], 0x14b7c2de    ; [0x14b7c2de:4]=-1
0x004009a5      751c            jne 0x4009c3                ;[3]
0x004009a7      bedb0a4000      mov esi, str.Access_Granted    ; 0x400adb ; "Access Granted"
0x004009ac      bfa0116000      mov edi, obj.std::cout      ; 0x6011a0
```

The return value of std::cin (the user entered value) is being stored in eax at 0x40098b. Then 0x75a35b40 is being subtracted from user input the line after. Then at 0x400999 the result of the subtraction is being added to itself (multiplied by two). The result of that is being compared to 0x14b7c2de. Cmp takes the two arguments and subtracts them, and if they are equal, the result is 0. We must get the result of the instruction at 0x400999 to be equal to 0x14b7c2de for the flag to print and the jump around “Acess Granted” not to be taken. We can calculate the password through a little algebra and working backwards.

2(input-0x75a35b40) = 0x14b7c2de
input = (0x14b7c2de / 2) + 0x75a35b40

Running this through python,
input = 2147433647

We have solved the input.



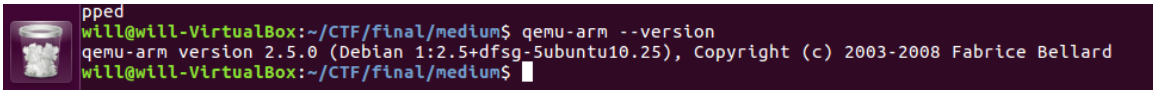
flag{2147433647}

RE2: Learn 2 Run

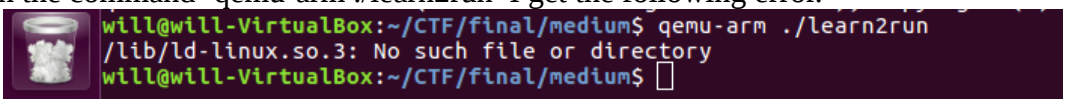
Keep on running...

Solution:

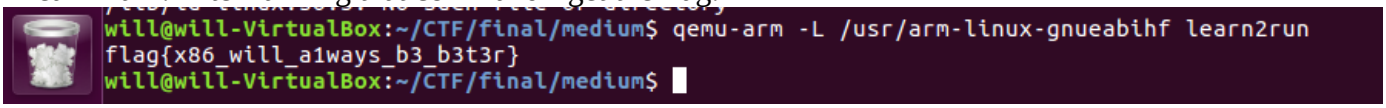
First I downloaded the two files from medium.zip. ‘medium.txt.enc’ is encoded text and ‘learn2run’ is a program. I ran file on ‘learn2run’ and was informed learn2run is ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV). Since this is an ARM file, we will have to use qem-arm. This is similar to the qemu problem from the reversing II homework titled ‘Run Me’, as hinted by the corresponding names. I used my write-up for that as reference. After I installed qemu with the instructions on the slides, I for some reason still have have version 2.5.0.



When I run the command ‘qemu-arm ./learn2run’ I get the following error.



After Googling the error, I found a github that told me to use the following commands to fix the problem. First, run ‘sudo apt-get install gcc-arm-linux-gnueabihf libc6-dev-armhf-cross qemu’. After I did that I had to use the following command to run qemu, ‘qemu-arm -L /usr/arm-linux-gnueabihf learn2run’. After running that command I get the flag.



flag{x86_will_a1ways_b3_b3t3r}

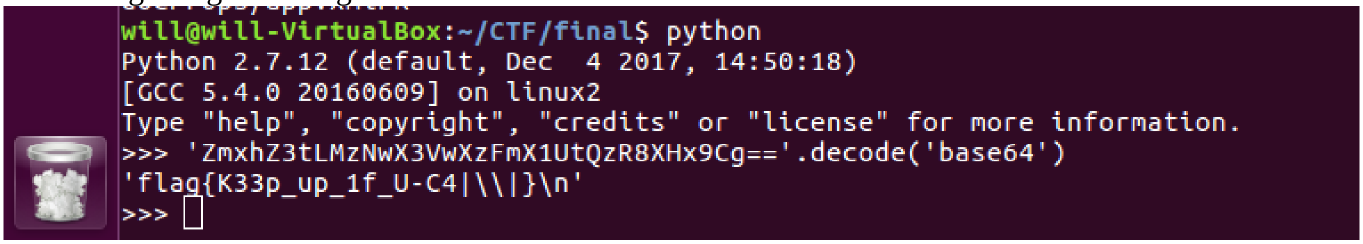
Forensics1: Anti-Forensics

VILE is troubled by your recent success. They are deploying advanced anti-forensics techniques to thwart your future endeavours. We grabbed a draft of their new anti-forensics guide. See if you can find any flags in it.

Solution:

First I downloaded the file ‘VILEGuideForEvadingAgents-DRAFT.docx’. I then ran strings on it.

Here we can see what appears to be a Base64 encoded string because it ends in '=='. I base64 decoded this string and got the flag.



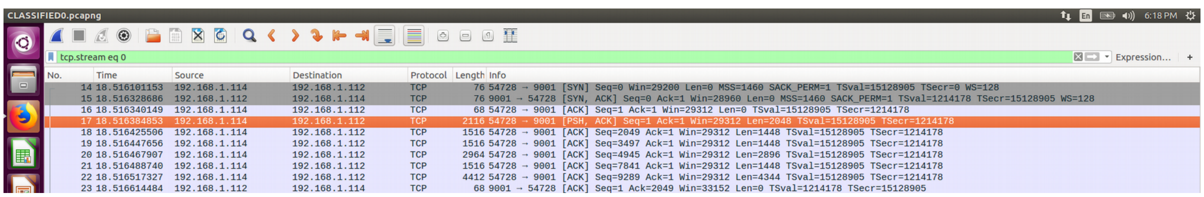
flag{K33p_up_1f_U-C4|\\|}

Forensics2: Double-Dealing Diva

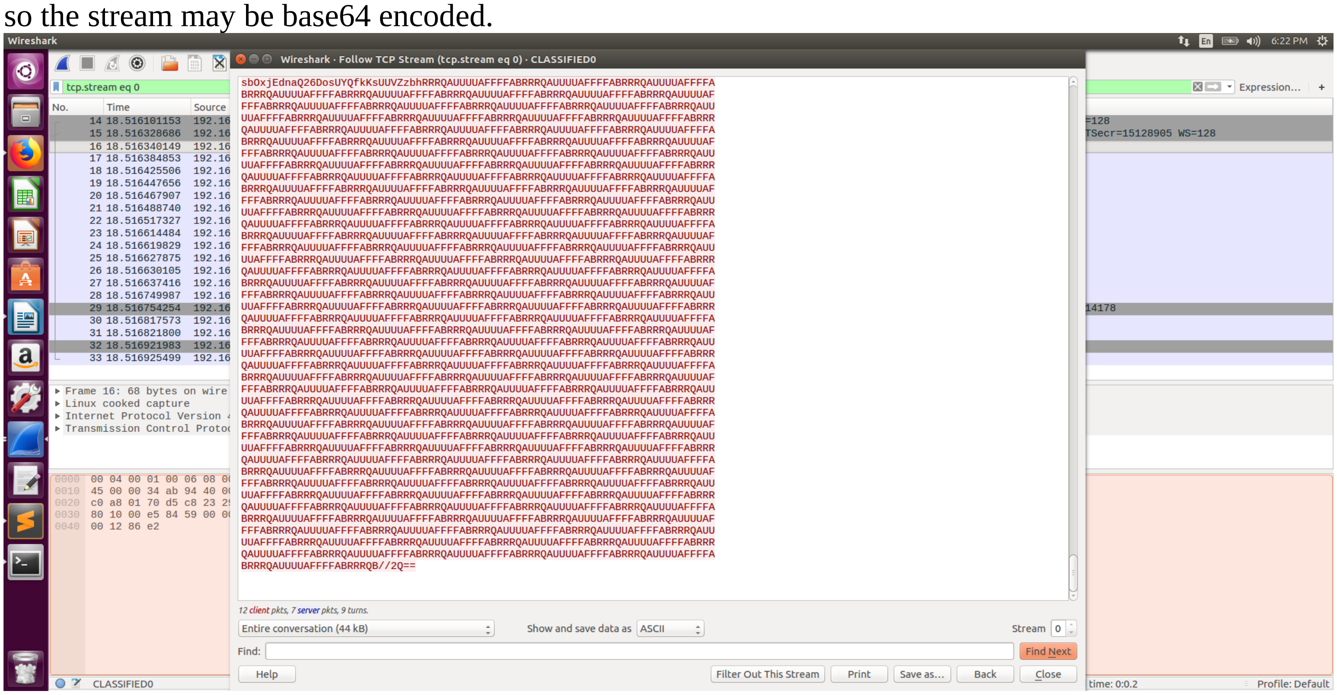
We have intercepted some communications between two computers on VILE's network. They are employing advanced techniques to keep us from understanding their communications. What are they saying to each other?

Solution:

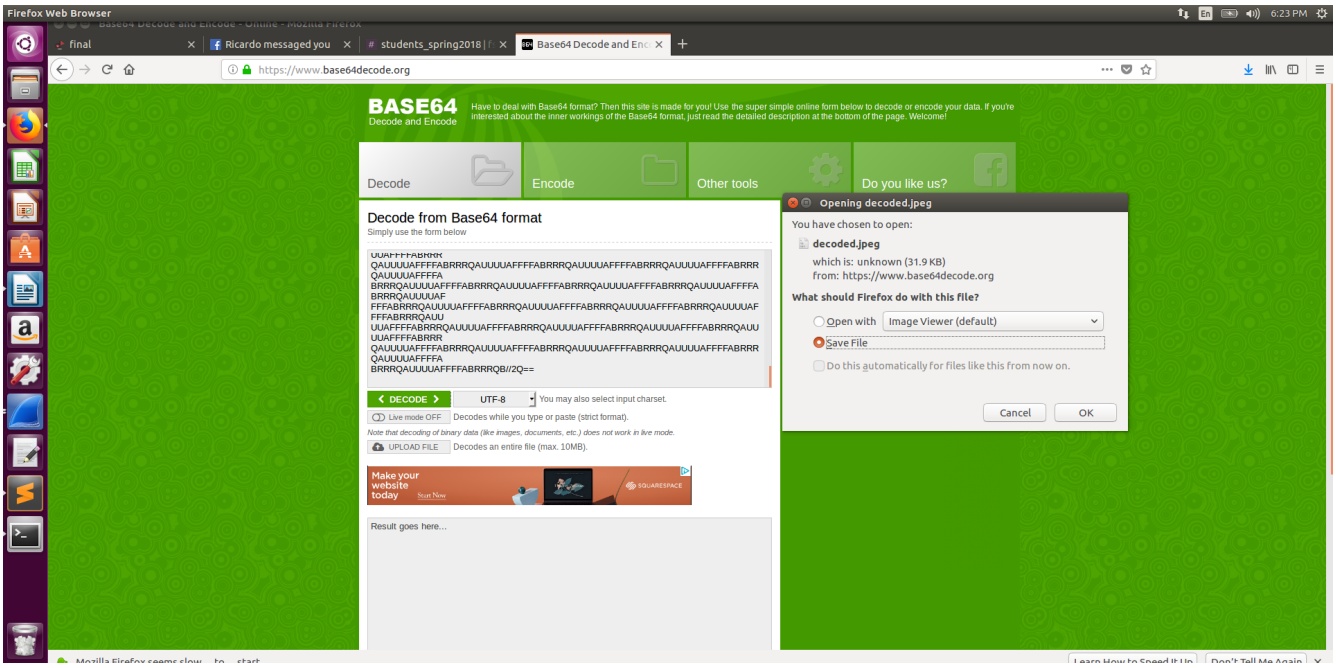
First, I downloaded the communications 'CLASSIFIED0.pcapng' and 'CLASSIFIED.pcapng'. I opened up 'CLASSIFIED0.pcapng' in wireshark to analyze communications. Since we are looking for communication between two computers on the same network, the source and destination should be similar addresses. I find streams with source '192.168.1.114' and destination '192.168.1.112'. I pick one



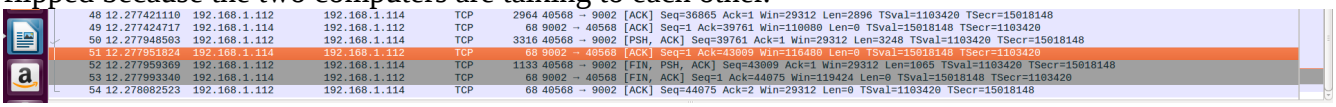
I right click, >Follow>TCP Stream to view the TCP stream. I then notice the last two characters are ==, so the stream may be base64 encoded.



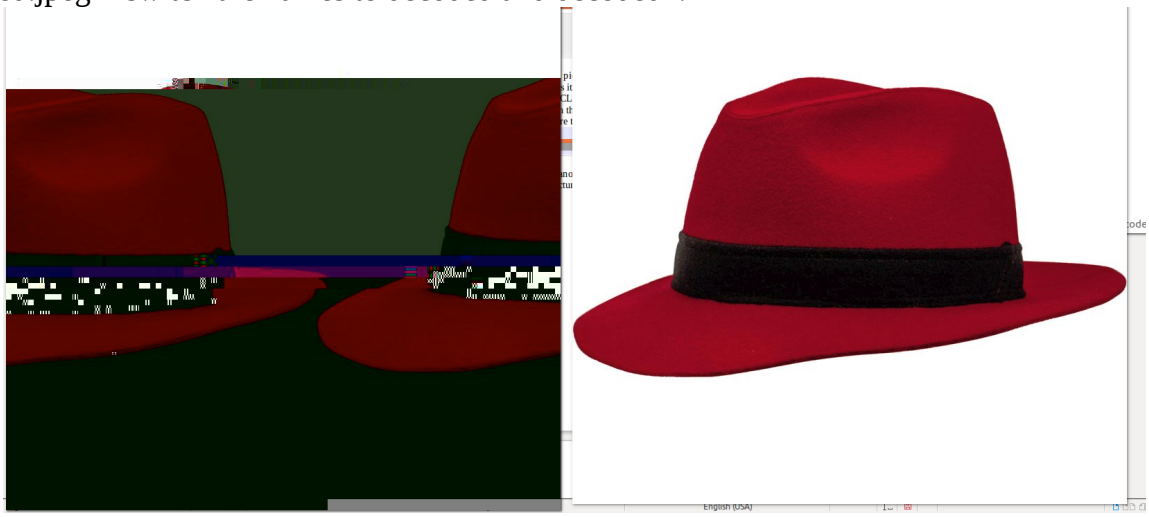
I copy all of the text and run it through a base64 decoder found at https://www.base64decode.org/.



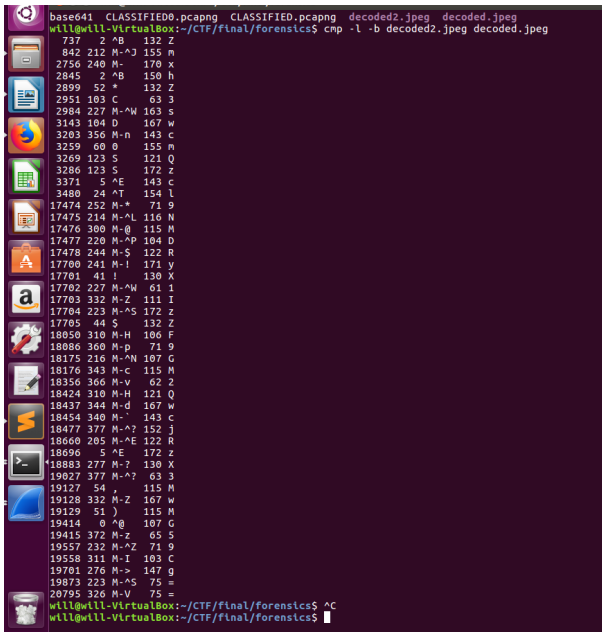
This downloads a jpeg file and it is a picture of a fedora. I run strings on this file and don't find anything interesting and file confirms its a jpeg. I then remember there are two communications captures. I repeat the steps I did for CLASSIFIED0.pcapng with CLASSIFIED.pcapng. In wireshark I find a TCP stream communicating on the same network, and this time the source and destination are flipped because the two computers are talking to each other.



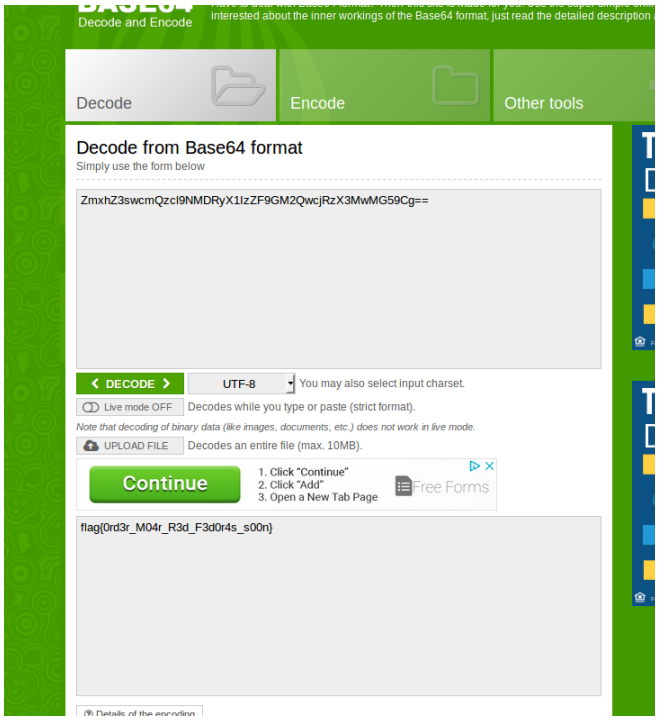
I follow this TCP stream, and I find another base64 encoded string. After decoding the string, it downloads me another picture of a fedora. However, this picture is blurry. Both pictures are called 'decoded.jpeg' I switch the names to decoded and decoded2.



This reminds me of an old forensics problem where the flag is stored in the difference between the two pictures. I start by just comparing the two pictures with ‘cmp decoded.jpeg decoded2.jpeg’ but this doesnt give anything interesting. I then look at the man page of cmp. I notice the -b flag prints differing bytes which could be useful because the pictures are so similar. When that didn’t work I tried more flags. This time I added -l to my -b flag because -l outputs differing byte values. ‘cmp -l -b decoded2.jpeg decoded.jpeg’ gives the following output:



Here we can see the last column looks like a base64 encoded string because it ends in ‘==’. I type the last column into a base64 decoder and find the flag.



flag{0rd3r_M04r_R3d_F3d0r4s_s00n}

Forensics3: Itinerary

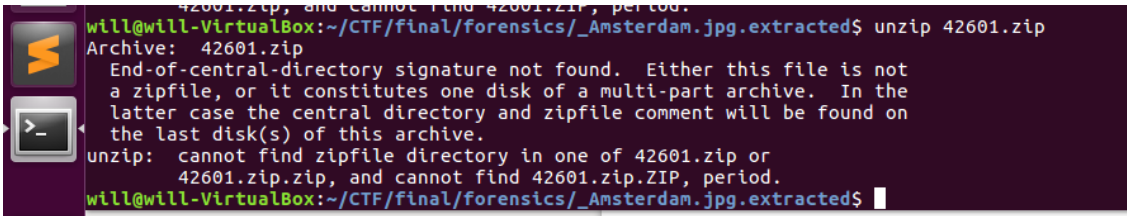
Our hackers retrieved a file that's supposed to hold the suspect's itinerary. However we can't seem to find anything useful. We know her itinerary is loaded up with something, can you find it?

The file is available here: https://drive.google.com/open?id=1JGf9UVcZ2R-Lh4WRX4Neq2QmPF4-4f3T

Solution:

I downloaded the file and ran ‘file’ on it to confirm it was in fact a JPG. It is a JPG. I then tried running strings and unzipping it to no avail. I viewed the hexdump and even the source code on Google Drive, but still I couldn’t find anything. After I had given up, I came back and looked at the slides and noticed “When all else fails, we carve”. Then, I remembered our problem Dank Memes and Broken Dreams from our homework was similar to this one, so I pulled up the writeup. Just as I did in the past assignment, I used ‘binwalk -e Amsterdam.jpg’. This created a directory called _Amsterdam.jpg.extracted which contained the files ‘42601.zip’, ‘album’, and ‘itinerary.txt’. I ran file on the zip file and realized that is indeed a zip archive. I ran file on ‘album’ and saw it was data. I also ran strings and saw ‘Milan’ nested in other strings. I opened up ‘itinerary.txt’

Here we can see the hint to Scrub date metadata from photos and Encrypt .zip file “Milan”. I then proceeded to try to unzip ‘42601.zip’ but no matter what I tried, I kept getting this error



So I Googled the error and found out the zip file is probably encrypted. This means the hint doesn’t mean I need to encrypt some zip file called Milan, but the file ‘42601.zip’ was encrypted using “Milan”. Since ‘./album’ contains “Milan” in strings, I assume this is the encrypted data. Milan could be used as the key for an XOR encryption, since substitution makes no sense in this case. However, XOR encryptions only use 1 byte keys and “Milan” is 5 bytes. That means ‘M’ is used for the first byte key, ‘i’ for the second, and it keeps repeating until the file has been decrypted. To do this I wrote a script attached as ‘foresncis3.py’

```
file = open('./album', 'rb')
file=file.read()
byte= bytearray(file)

leng=len(byte)
for i in range(leng):
    if i % 5 ==0:
        byte[i] = byte[i] ^ ord('M')
    if i % 5 ==1:
        byte[i] = byte[i] ^ ord('i')
    if i % 5 ==2:
        byte[i] = byte[i] ^ ord('l')
    if i % 5 ==3:
        byte[i] = byte[i] ^ ord('a')
    if i % 5 ==4:
        byte[i] = byte[i] ^ ord('n')

output=open('output.zip', 'wb')
output.write(byte)
```

First I open up the ‘./album’ file in binary mode to read. I then read the bytes and store them in a byte array called byte. I then iterate over the byte array. I xor byte 1 at index 0 with ‘M’, then xor byte 2 at index 1 with ‘i’, and so on repeating using each character of “Milan” as the key every 5 bytes (‘M’ will also xor byte 6 at index 5). I then create a new zip file called ‘output.zip’ and I write the decoded binary to it. Now, when I ‘unzip output.zip’ I get images.

```
will@will-VirtualBox:~/CTF/final/forensics/_Amsterdam.jpg.extracted$ python 2.py
will@will-VirtualBox:~/CTF/final/forensics/_Amsterdam.jpg.extracted$ unzip output.zip
Archive:  output.zip
  inflating: Antarctica.jpg
  inflating: Belize.jpg
  inflating: Berlin.jpg
  inflating: Carolina.jpg
  inflating: China.jpg
  inflating: Greenland.jpg
  inflating: Kiev.jpg
  inflating: Mekong.jpg
  inflating: RedSea.jpg
  inflating: TheBlues.jpg
will@will-VirtualBox:~/CTF/final/forensics/_Amsterdam.jpg.extracted$
```

Immediately I remember the hint to view the metadata. To view metadata I use exiftool. Scrolling through the meta data I notice something interesting,

```
Artist                          : fla
Y Cb Cr Positioning             : Centered
Exif Version                    : 0230
Create Date                     : 2018:01:01 01:01:01
```

I notice the artists category for each picture is three letters and one of them starts with fla. That means the artists entries put together the flag. Also, the create date is repeating ones on fla. This could mean that the create date signifies the order in which the artist values are ordered in the flag. Looking through the rest of the image’s metadata, this pattern matches up. Putting the artists in order by create date we get flag{bTB2aW5nX3YxMGw0NzEwbjUK}. However, this doesn’t look like a flag. Flags are always written in 3L1T3 text. I tried looking up the MD5 hash of the body of the flag to no avail. Although I did not see = at the end of the string, I figured I would still try base64 because the = is optional.

```
>>> 'bTB2aW5nX3YxMGw0NzEwbjUK'.decode('base64')
'm0ving_v10l4710n5\n'
>>>
```

And we found the 3LIT3 text. m0ving_v10l471n5 is the flag body.

flag{m0ving_v10l471n5}