

## Flags

1. h4ndY\_d4ndY\_sh311c0d3\_55c521fe
2. 3asY\_P3a5yb197d4e2
3. n0w\_w3r3\_ChaNg1ng\_r3tURn5a32b9368
4. th4t\_w4snt\_t00\_d1ff3r3nt\_r1ghT?\_d0b837aa
5. sl1pp3ry\_sh311c0d3\_0fb0e7da
6. arg5\_and\_r3turn55897b905
7. cAnAr135\_mU5t\_b3\_r4nd0m!\_bf34cd22
8. str1nG\_CH3353\_166b95b4

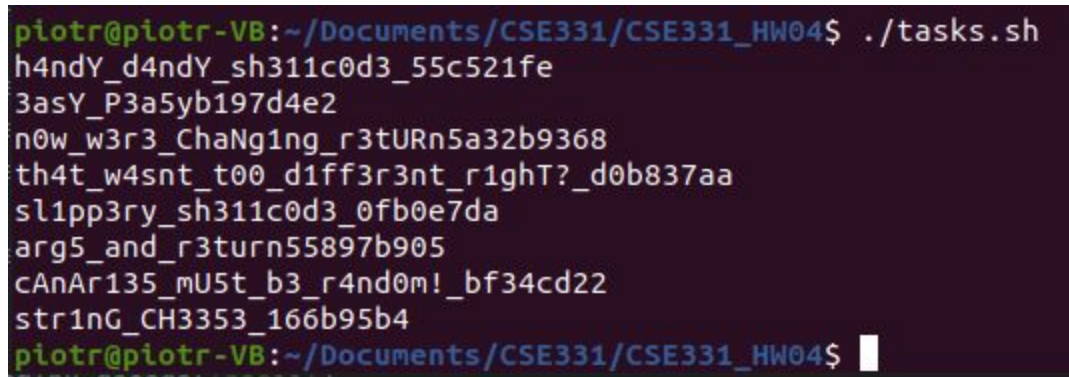
## Python Libraries Used

- Pwntools - ctf toolkit as recommended by the professor

## How to Run

1. In order to run each individual task do ./taskN.sh or to run all tasks at once do ./tasks.sh
2. If you would like to use the other homework server change the username in the ssh function at the top to 'cse3312'

## Proof of code working on my local machine (./tasks.sh)



```
piotr@piotr-VB:~/Documents/CSE331/CSE331_HW04$ ./tasks.sh
h4ndY_d4ndY_sh311c0d3_55c521fe
3asY_P3a5yb197d4e2
n0w_w3r3_ChaNg1ng_r3tURn5a32b9368
th4t_w4snt_t00_d1ff3r3nt_r1ghT?_d0b837aa
sl1pp3ry_sh311c0d3_0fb0e7da
arg5_and_r3turn55897b905
cAnAr135_mU5t_b3_r4nd0m!_bf34cd22
str1nG_CH3353_166b95b4
piotr@piotr-VB:~/Documents/CSE331/CSE331_HW04$
```

## Task 1

For this task I started off by downloading the c file from the server and looking through it. It became to me that all it was doing was taking in input and then running it so I then researched how to generate shellcode and conveniently I stumbled upon the shellcraft section in

pwntools and I used it to make shellcode to spawn a shell, convert it to assembly code and then used it as a payload so that when vuln asked for input I gave it the assembly code of the shellcode to spawn a shell. Once the shell was spawned I told it to read flag.txt and I printed out the output.

## **Task 2**

For this task I started off by downloading the c file from the server and upon inspection of the code it looked like they were using an unprotected strcpy which makes sense because the professor's hint was to overflow the buffer which means abusing this strcpy. What I did was make a string that was 150 characters long and I just sent it as input to vuln and once I did that the flag was printed out by vuln so I just printed it out.

## **Task 3**

For this task I started off by downloading the c file from the server and upon inspection of the code it looked like we were doing the same thing as task 2 except the flag wasn't going to be printed out for free. Thus I downloaded the executable as well and using radare2 I disassembled the executable into assembly code and then I found the address of the flag function which was 0x080485e6. Next I sent a random string of just 'a's and noticed that an error message was printed out that told me that the return address of 61616161 was invalid and so I decided to make a string in the form of 'aaaabbbbccccddd...' to figure out which part was the return address. Turns out it was 'tttt' and after replacing that with the address of the flag in big endian form and sending that to the program the program spit out the flag.

## **Task 4**

For this task I started off by downloading both the c file and the executable from the server and upon inspection it looked like we were doing the same thing as task 3 but when I tried doing the same thing it wouldn't tell me which part of the string to replace so I decided to write a program to try a bunch of offsets with the address of the flag function appended to it which I got from radare2 using the same method as task 3. This however did not work and I was a bit stumped as to what to do so I decided to randomly change the address of the flag function to the next address (from 00400767 to 00400768) and voila the program output the flag. I think what

happened with me moving the address to the next spot was that it would not update the base pointer which would let me properly place the address in but I'm not 100% sure.

### **Task 5**

For this task I started off by downloading both the c file and the executable from the server and upon inspection it looked like we were doing the same thing as task 1 except the program was introducing a random offset into the input that we were providing. The fact that the folder was called 'slippery-shellcode' kinda gave it away that we had to do something with a nop slide like we learned in lecture so I just included a nop slide that was 256 characters long to the same assembly shellcode that spawned a shell and tada I got the flag.

### **Task 6**

For this task I started off by downloading both the c file and the executable from the server and upon inspection it looked like we had to do the same thing as in task 2 except we also had to provide arguments to the flag function. I started off by doing some research and I found out that after the address to the flag function there is another address space that is 4 bytes long (I don't remember what for) and then there was space to put arguments so I formatted by payload in such a way that it skipped the 176 byte long buffer, the 12 bytes of padding that is introduced in the program, put in the address of the flag function, skipped the 4 bytes for the second address, and put in both arguments in the form of big endian hex values. I then sent that to the program and I got the flag.

### **Task 7**

For this task I started off by downloading both the c file and the executable from the server and upon inspection it looked like a canary was stored in the buffer and our job according to the hint was to guess it. I started off by writing code to guess the canary byte by byte since I just have to keep running the program with a new byte and if it doesn't crash then I guess a value for that byte's position and I can move on to the next position. Once I had the canary sorted out I thought I was done and I could just find the address of the flag function and put that into the payload but that was not the case since the second I tried running radare2 on the executable radare gave a warning to "run r2 with -e io.cache=true to fix relocations in disassembly." I

thought radare wasn't working correctly so I googled what this error message meant and I found out that this program was using position independent code which meant that it was loaded into random locations in the memory (read about it here:

<https://access.redhat.com/blogs/766093/posts/1975793> ). So this was quite the wildcard but I then found out that the first couple bytes of the address were what is randomized as per this link's intel:

<https://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries/> so I then put the appropriate padding and just ran the code in a loop with only the last 2 bytes of the address until it eventually got the flag.

### **Task 8**

For this task I started off by downloading both the c file and the executable from the server and upon inspection it looked like the flag was stored both on the stack and in the heap so I thought we should do some buffer overflow attack but the hint for this part from the professor was to use a format string so I looked into them primarily from this article

<https://kevinalmansa.github.io/application%20security/Format-Strings/> which taught me that to access stuff directly on the stack I can write a formatted string like “%i\$s” where I would increment the i until I would eventually get the flag. So that is what I did and I made a loop that would keep going until it would eventually find the flag on the stack using the formatted string payload.