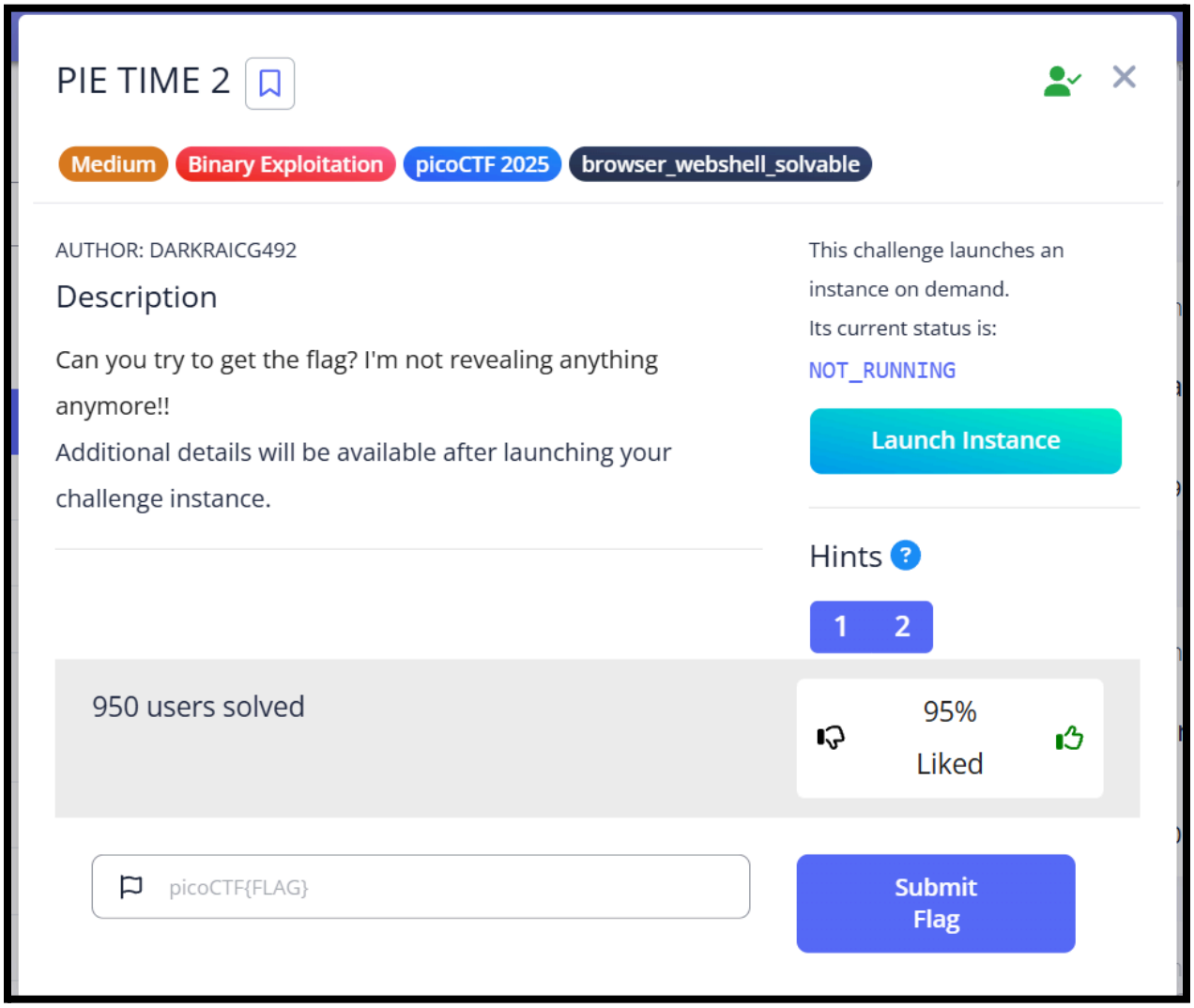
Written by Billie
PIE TIME 2 from PicoCTF https://play.picoctf.org/practice/challenge/491



The whole objective of this challenge is to jump into a “win” function that contains our flag. We can see the addresses of all the functions in GDB.

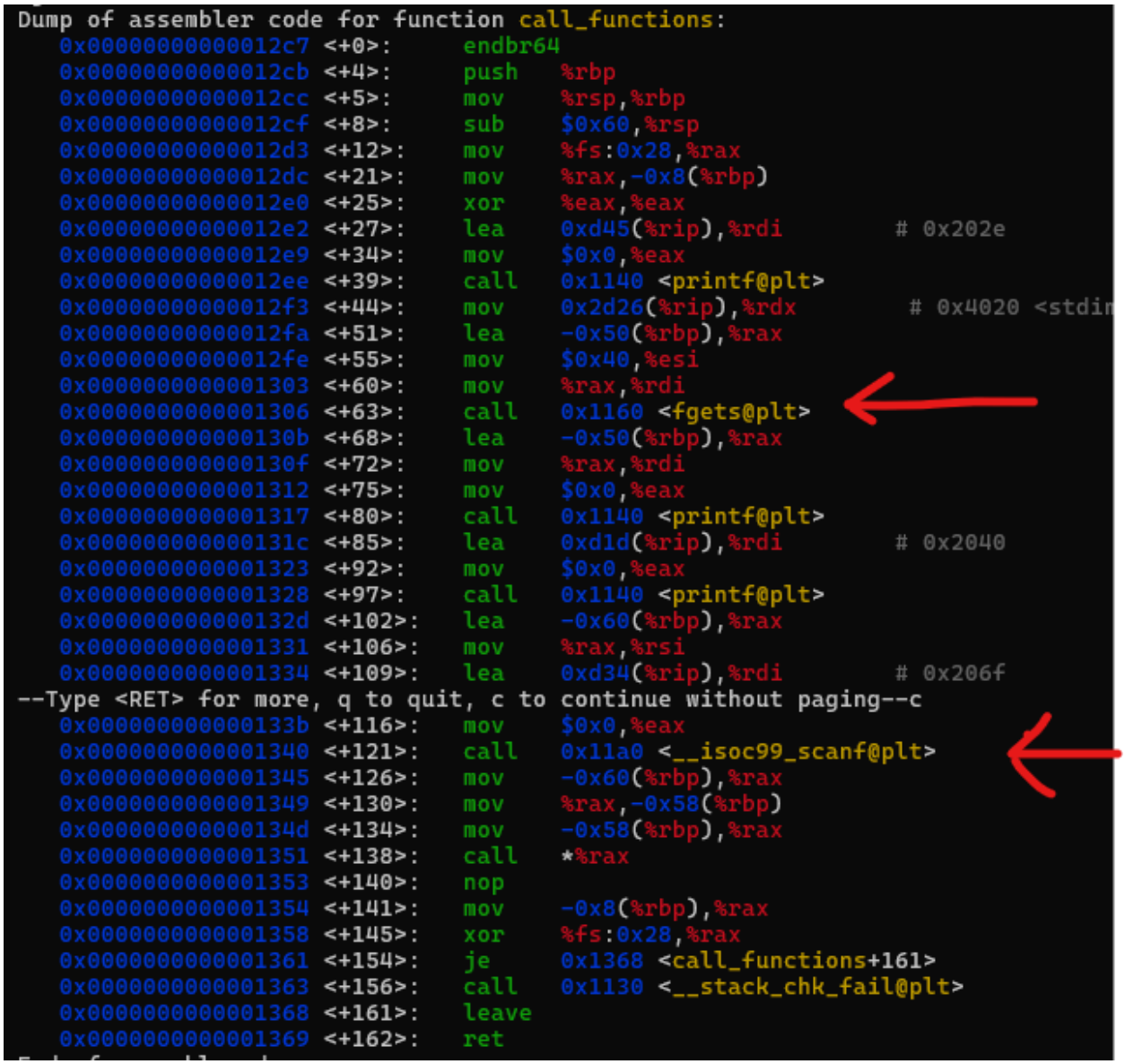
PIE stands for Positional Independent Executable. Which means that everytime we run this executable, it will change the functions addresses. That means the address of the win function changes every time we run the program.

One thing that is good practice to always do, is to check for format string vulnerability. Every time a program asks for an input, try writing “%p” or any other format specifier in the buffer. If the program then writes random values to the console, that means that our format specifier has leaked some value from the stack. Essentially using format

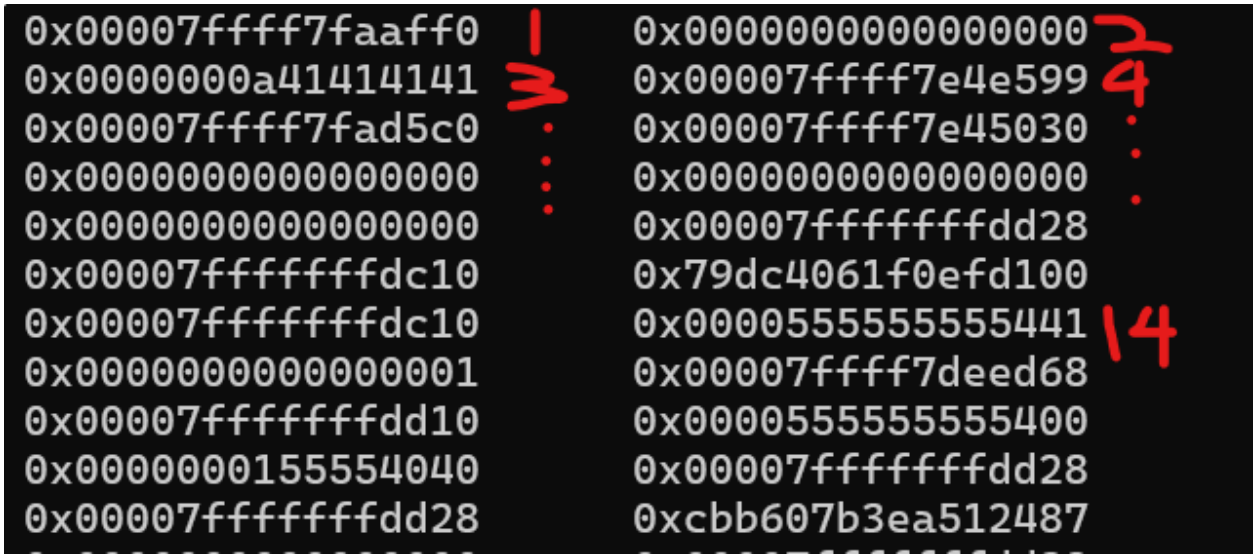
specifiers we can see what the stack looks like in run time. This is important because all the memory addresses are randomized between run times.

```
┌──(billie㉿Billie)-[/mnt/c/Users/Billie Bhaskara/Downloads/Pie 2]
└─$ ./vuln
Enter your name:%p%p%p%p%p%p
0x25702570257025700xfbad22880xa702570257025700x55fd6e0892ad0x40x7f097a103ff0
```

As you can see, I tried writing random amounts of “%p”s into the buffer and the program then proceeded to spout random addresses from the stack.



The program holds a function called “call_functions” where the program asks for our input two times. I put a breakpoint on the second input call (the second red arrow) because I want to see how the stack looks right before I put the second input.



Using the command “x/20gx $sp” which means “examine 20 of 64 bytes memory values from the stack”, we can see that in the 14th order from the start of the stack, we see the return address of the function into main which is main+65 (we have a +65 offset because we don't return to the beginning of main, rather to main but after we call “call_functions”).

```
0x000055555555543c <+60>:    call   0x5555555552c7 <call_functions>
0x0000555555555441 <+65>:    mov    $0x0,%eax
```

As you can see from this snippet from main, the address in the 14th order is the exact same address of the return point of the function in main.

```
0x000000000000136a  win
0x0000000000001400  main
```

The image above shows the GOT off the function win and main. Essentially all that PIE does is add a value to the GOT above. For example, say the function main has address “0x1” and win has value “0x2” in the GOT. If we run the program and the function mains address changed into “0x5”, that means there's a +4 offset from the original GOT, and we can infer that the function win is now located at address “0x6”

Now we have everything we need for the exploit. We know the GOT addresses of the function win and main, and we know the address of main+65 from the stack. This means in run time we can calculate the offset of the win function.

To do this, we have to type a bunch of “%p”s when the program asks for our name. This step is a little bit of trial and error. Theoretically, since the address is in the 14th order of the stack, this means we need to write 14 “%p”s to get the main+65 address, but due to my lack of knowledge, and the fact that GDB disables PIE and also adds debugging

symbol to the binary, there's just too many variable at play preventing me to just conclude that the main+65 address is going to pop from the 14th "%p".

One thing that I'm sure of is that the main+65 address really is going to be offsetted +65 from the original main address. Since this is part of the assembly code and not memory, GDB will not change anything regarding this.

Also note that 65 in hex is 41.

```
Enter your name:%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p|%p|%p%p%p%p%p
0x56a3c278b2a1(nil)0x56a3c278b2db0x7ffcbf19a7a00x7c0x7ffcbf1b52280x7f1e8b5e66a00x70257025702570250x7025702570250x702
57025702570250x7025702570250x7025702570250x7025702570250x7c70257c702570250x7025702570250x7ffc000a70250x56a3b3a9a1c00xd556d
e4f516de7000x7ffcbf19a8000x56a3b3a9a441(nil)0x7f1e8b41d0830x7f1e8b61d620|0x7ffcbf19a8f8|0x1000000000x56a3b3a9a4000x56a3b
3a9a4500xc39f02ab14103b950x56a3b3a9a1c0
```

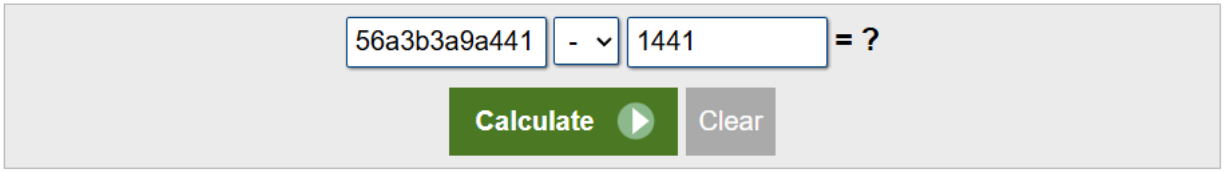
After inputting a bunch of "%p"s as expected the program outputs a bunch of memory from the stack. After reading the output carefully I found an address that ends with 41. This is relevant due to the fact (this is anecdotal, I don't know for sure) that most of the programs with PIE that I disassemble, the offset always leave out the last two digits with 0. For example, if we have a GOT address of 0x1011, usually the runtime address would be …XXX11 with the two "1"s still there.

You might wonder why 41 is relevant since the GOT address of main is 1400, but we realize that it is main+65, which is main+0x41 in hex. So that means the last two digits in hex would be 41. Exactly the address that we got above.

This means 0x56a3b3a9a441 is our main + 65 address. Now to get win, we have to do Win = 136a (the GOT address of win) + ([main+65]-0x1441(GOT address of main + 0x41))

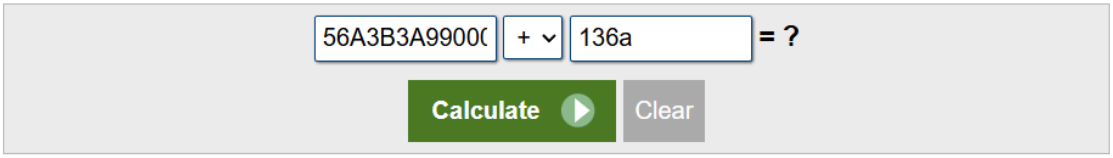
Hex value:
56a3b3a9a441 – 1441 = **56A3B3A99000**

Decimal value:
95261093897281 – 5185 = **95261093892096**



Hex value:
56A3B3A99000 + 136a = **56A3B3A9A36A**

Decimal value:
95261093892096 + 4970 = **95261093897066**



This means **56A3B3A9A36A** is our win address. We then input this to get our flag

```
 enter the address to jump to, ex => 0x12345: 0x56A3B3A9A36A
You won!
picoCTF{p13_5h0u1dn'7_134k_af46d901}
```