ELF文件延迟绑定过程与获取外部动态库函数对应的plt表项地址

0x0 前言

在之前做的一些项目中对ELF文件中的函数进行Hook后要调用外部动态链接库函数，因此需要获取外部动态链接库的函数对应在plt表中的地址。对于ELF文件内部函数，其偏移地址可以直接从文件的二进制数据中获取，因为此类函数的偏移都已经写在了文件中；但是对于外部动态库的函数，则没办法直接获取，因为动态库采用了动态链接、延迟绑定技术，只有在第一次调用该函数，该函数的真实地址才会被写入文件中，在此之前记录该函数地址的数据仅为无意义的占位符。

而该问题的答案并没能直接在网上直接搜索得到，因此在思索得到结果后，写下该博客进行记录。

0x1 ELF文件结构简述

若要解决获取函数地址的问题，那么自然需要对ELF文件结构有所了解。

但是在这里，就不对ELF文件进行特别详细的解析，因为对于文件结构，都是做好了规范的死东西，可以从任何一个资料上获取每个字段、每个数据的意义。因此，这里就简要说明一些值得注意的东西。

0x10 ELF文件两种视图

ELF文件格式规范，将ELF文件分成共享目标文件(shared object file)、可执行文件(executable file)和可重定位文件(relocatable file)。ELF文件的作用也就是用于构建动态链接库或可执行文件，主要体现在链接过程；以及可执行文件用于运行程序，动态链接库参与程序的运行，体现在运行过程，其中动态链接库则属于共享目标文件。根据该ELF文件的特性，ELF文件则可分成两种视图(view)，链接视图(Linking View)与运行视图(Execution View)。

对于这两种视图，我觉得这样理解会更加容易。链接视图则是ELF文件存储在磁盘时的结构，而运行视图则是ELF文件载入内存后，在内存中的结构。这与PE文件是类似的，PE文件在磁盘与在内存中的结构并不完全相同。

由 图1-1 ELF文件的两种视图，可以看到两种视图的区别。两种视图仅在节区上发生了变化，由section变成了segment。segment是由一个或者多个类型(sh_type数据)相同的section组成。那为什么这样做呢？当文件从磁盘载入到内存的时候，会根据这部分代码的属性，也就是该部分代码具有读写执行中的哪些权限，分别进行载入处理，若把相同属性的节区组织到一起同时载入，加载器的工作量则减少了，一定程度提高程序载入效率。

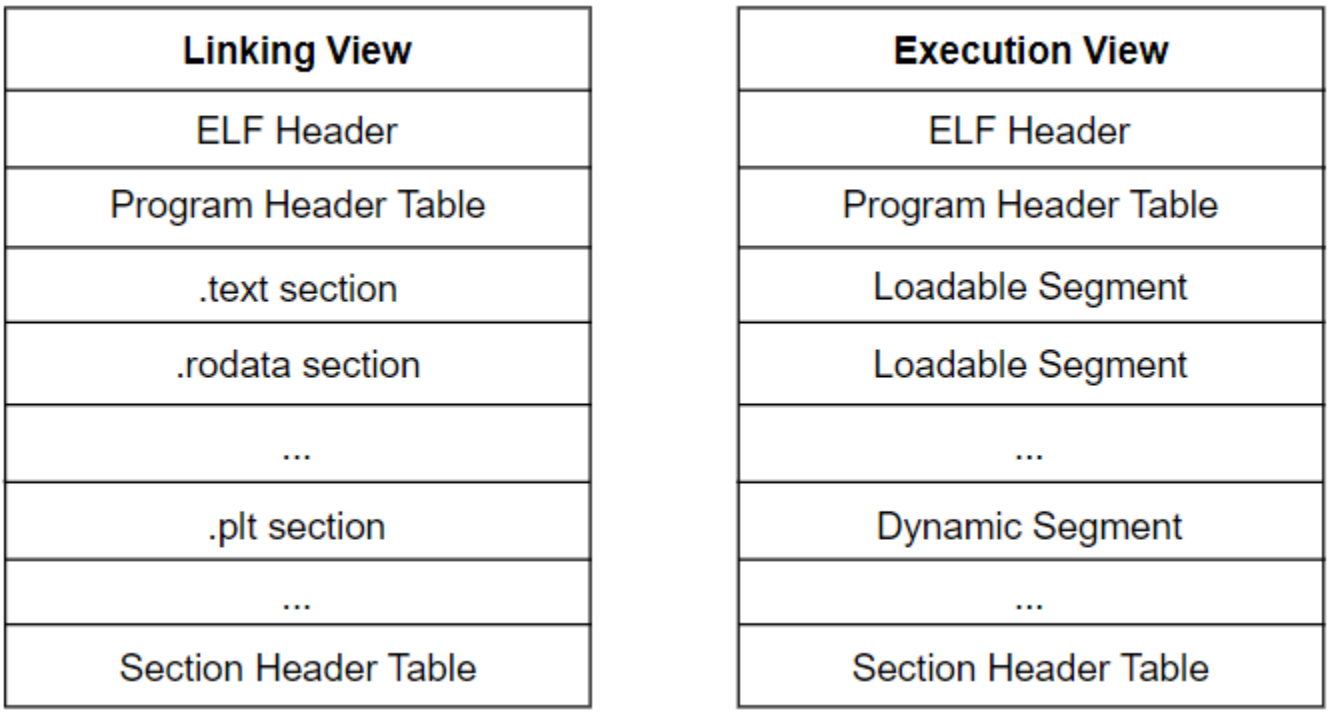

图 1-1 ELF文件的两种视图

0x11 节

接下来将简述Hook程序所使用到的节，及使用的原因。

(1) .shstrtab节。要获取多个节区的数据时，需要确定当前遍历到的节头对应的是什么节区，因此需要比较节头的保存的节的名字。对于.shstrtab节来说，其对应的节头在节头表位置固定，即在位于节头表的末尾。该节区保存了所有节的名字，并且保存的顺序与节对应的节头在节头表的顺序相同。因此需要利用.shstrtab节区的数据来遍历节头表。

(2) .dynsym节。该节保存了与动态链接相关的导入导出函数相关的信息，例如导出函数的起始地址以及函数名字符串在.dynstr节的偏移等。可以通过该节寻找到导出函数的地址，当Hook动态链接库的函数时，就可以通过该节区获取目标导出函数。

(3) .dynstr节。该节保存了动态符号的字符串表，配合.dynsym节遍历查找目标导出函数。

(4) .plt与.got节。这两节主要是获取plt表以及got表的起始地址，在Hook程序中需要计算外部导入函数的相对虚拟地址时将会使用到plt表的起始地址，而在调用外部导入函数时则需要将got表的起始地址保存到ebx中，才能正确调用。

(5) .rel.plt节。该节保存了与plt表的符号的重定位信息，因为ELF文件的延迟绑定机制，无法直接获取外部导入函数的起始地址，因此需要通过重定位表来定位外部导入函数其在plt表的索引，再通过call targetfunc@plt则可以实现对外部导入函数的调用。

在对.rel.plt节的表述中，就已经透露了获取外部动态库的函数地址的方法。下面会详细的说明

0x12 延迟绑定

ELF文件为了实现动态链接，使用了两个表。Got表(Global Offset Table)全称为全局偏移表；plt表(Procedure Linkage Table)全称为过程链接表。这两表在ELF文件进行动态链接与延迟重定位的过程中起到关键的作用。在

介绍got表与plt表之前先简述何为动态链接与延迟重定位。

通过将程序使用的模块从程序中拆分出来形成独立的模块，在程序运行并需要使用这些独立的模块时，再将这些模块与程序链接在一起。通过动态链接技术，这些模块导入内存一次便可以供多个程序共享，从而减小了内存的开销，这些模块也被称为动态链接库。该技术的出现就解决了静态链接导致的程序与内存臃肿的问题。但动态链接技术也导致了新的问题，若程序在运行时需要使用动态链接库，那么就会对整个动态链接库进行重定位操作，但是程序并不会在此刻使用到动态链接库的全部数据，因此会导致不必要的性能开销，同时程序运行时会链接大量的动态库，这就会导致程序运行变慢。为了解决这个问题，就提出了延迟重定位技术，该技术又称为延迟绑定。其原理就是通过额外的代码与数据与动态链接库中需要重定位的数据建立联系，那么程序需要哪个数据，就仅对该数据进行重定位，由此解决了不必要的性能开销导致程序运行速度下降的问题。

在编译过程中编译器是无法获知外部的符号存在于何处的，到了链接阶段，程序可以知道外部符号存在于哪个文件里，如果符号存在于可重定位文件中，即静态链接，链接过程中由于符号地址确定因此可以直接重定位；但如果在共享目标文件中，即动态链接，则因为链接阶段是无法修改编译得到的汇编指令，所以链接阶段无法进行重定位。因此为了解决动态链接重定位的问题，ELF文件由链接器生成了保存外部导入的函数的地址的数据段，与一小段用来获取外部导入的函数的地址的代码。而got表就是用来存放外部导入的函数的地址的数据表，即该表会把位置独立的地址重定向到绝对地址，plt表则是存放了用于获取外部导入的函数的地址的代码，即该表会把位置独立的函数调用重定向到绝对地址。

那么在ELF文件中，如果调用外部动态链接库中的函数程序会如何实现呢？

如 图1-2 ELF文件中外部函数的调用过程所示，当程序调用外部导入函数时，call的其实是该外部函数在其对应的plt表项的位置，然后再从其对应的plt表项中跳转到该函数对应的got表项位置，从got表获取该函数的地址后，就可以跳转至该函数的起始地址处了(因为动态链接库被载入内存后也会和载入该库的程序使用同一片虚拟内存)。ELF文件寻找外部函数起始地址的过程需要利用plt表以及got表。

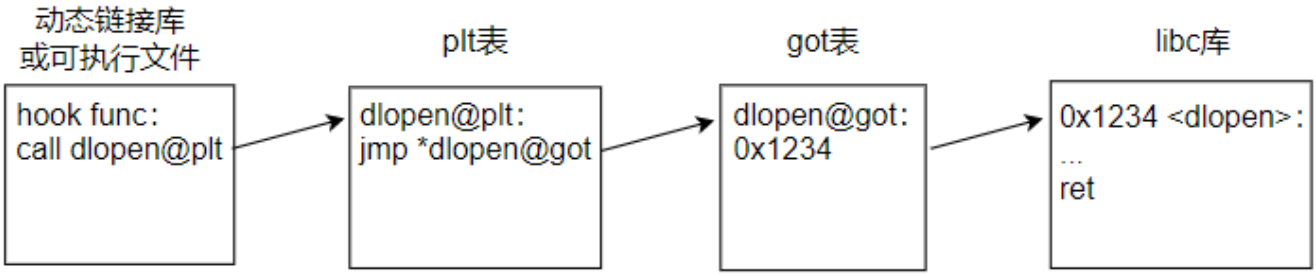

图 1-2 ELF文件中外部函数的调用过程

那么这就引出一个问题，plt表长啥样呢？怎么跳转到plt表了后就可以找到got表呢？

如 图1-3 Plt表结构所示，plt表是由plt表项组成的数组，每个plt表项大小为16字节，除了plt[0]项，其余的项结构都相同。对于plt[0]这一项来说，其第一条指令将got[1]处存放的地址入栈，第二条指令则是跳转到got[2]项中存放的地址执行。对于plt[n]项来说，其保存了某个函数链接时所需要的指令，从plt[1]项与got[3]开始，plt与got表就开始一一对应，即plt[1]对应got[3]，plt[n]对应got[n+2]，n>=1。Plt[n]的第一条指令会跳转到该plt表项对应的got表项处存放的地址执行，第二条指令入栈的值用来作为_dl_runtime_resolve()函数的参数。第三条指令则是跳转到.PLT[0]执行项里保存的两条指令。plt[n]中的前两条指令可以交换顺序，也就是可以先push，再jmp，因为push的数字会在jmp 到plt[0]之后才会使用到，那么got表则跳转回对应的plt的第三条指令而不是原来的第二条指令了。

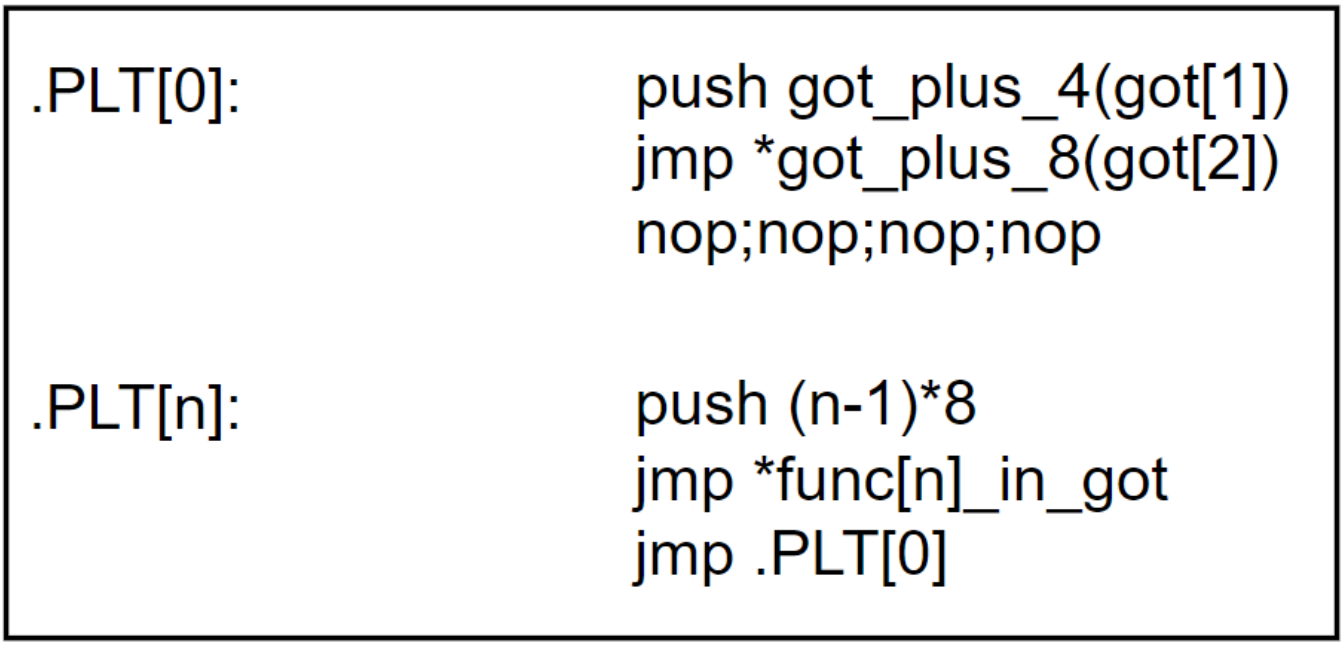

图 1-3 Plt表结构

那么问题又来了，那got表长啥样啊，got表又做了什么呢？

如 图1-4 Got表的结构所示。got表每个表项为4字节大小。got表的第一项保存了.dynamic节区的地址，第二项保存了link_map链表描述符的地址，该链表保存了程序需要用到的所有动态链接库的信息，例如库的名字以及地址等，通过遍历比较库的名字确定是否是目标函数所存在的库。动态链接器也将使用link_map链表描述符的地址进行符号解析。got[1]项保存的地址在plt[0]中被入栈，同样是作为_dl_runtime_resolve()函数的参数。第三项保存的则是_dl_runtime_resolve函数的地址，在plt[0]项的第二条指令会跳转到got[2]保存的地址去执行_dl_runtime_resolve函数。

当执行完_dl_runtime_resolve()函数后，目标函数的真实地址将被找到，并且会将该地址写入目标函数对应的got表项中。下一次再次调用目标函数时则将直接跳转到got表存放的地址执行函数，不用再进行重定位。因此可以看到图1-4中在延迟绑定之后，数据变成了函数地址，而不是plt[n]的第二条指令的地址了。

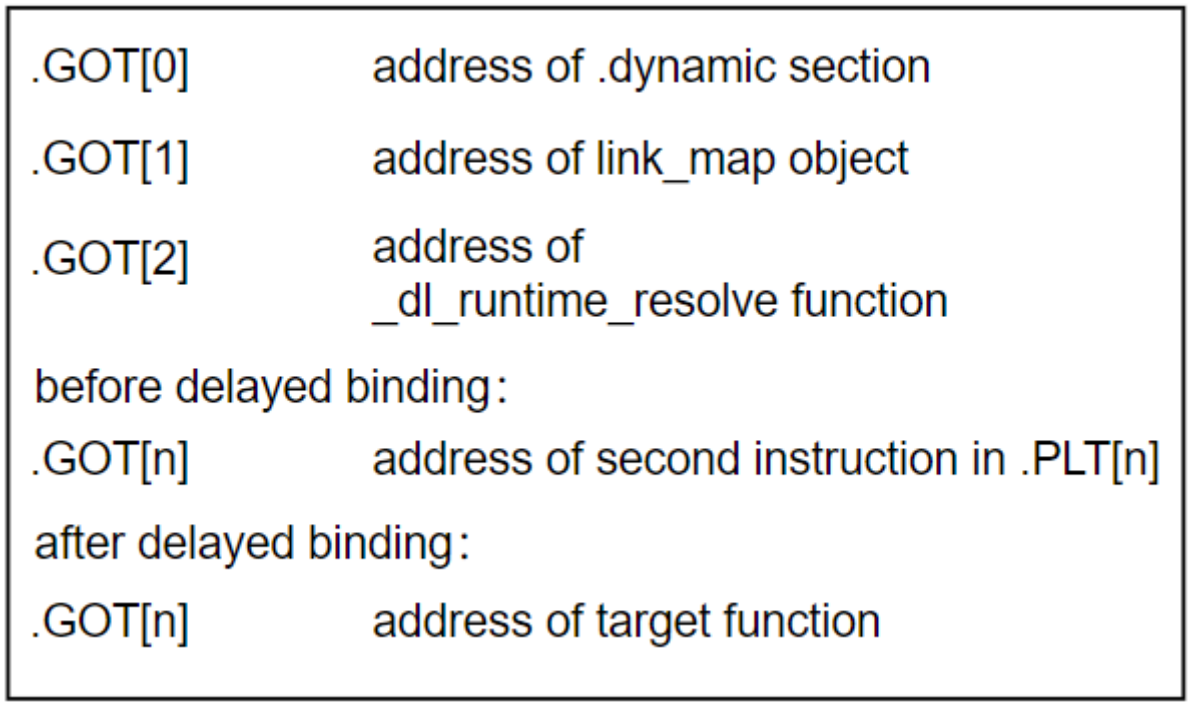

图 1-4 got表结构

还记得plt[n]中push的数字吗？目标函数对应的plt表项中的第二条指令入栈的数字表示该函数在.rel.plt节区的偏移，而.rel.plt节区中，每个项的offset成员则是其got表项的地址，_dl_runtime_resolve()函数则可以通过该信息回写got表。

综上，可以得到 图1-5 延迟绑定过程。当程序调用外部导入函数时，程序会进入该函数对应的plt表项执行指令。指令的执行首先会让程序跳转到对应的got表项保存的地址，若此时是第一次调用则会跳转到对应的plt表项的第二条指令，否则将直接跳转到目标函数起始地址处。Plt表项的第二条指令将该函数所对应的位于.rel.plt节区的项的偏移入栈，然后跳转到plt[0]项中将link_map地址入栈，再跳转到_dl_runtime_resolve函数起始地址开始执行该函数，该函数会调用目标函数，并将目标函数的真实地址写入目标函数对应的got表中(此步骤会依赖.rel.plt节区中的数据)。最后执行完目标函数后返回到调用目标函数指令的下条指令上。

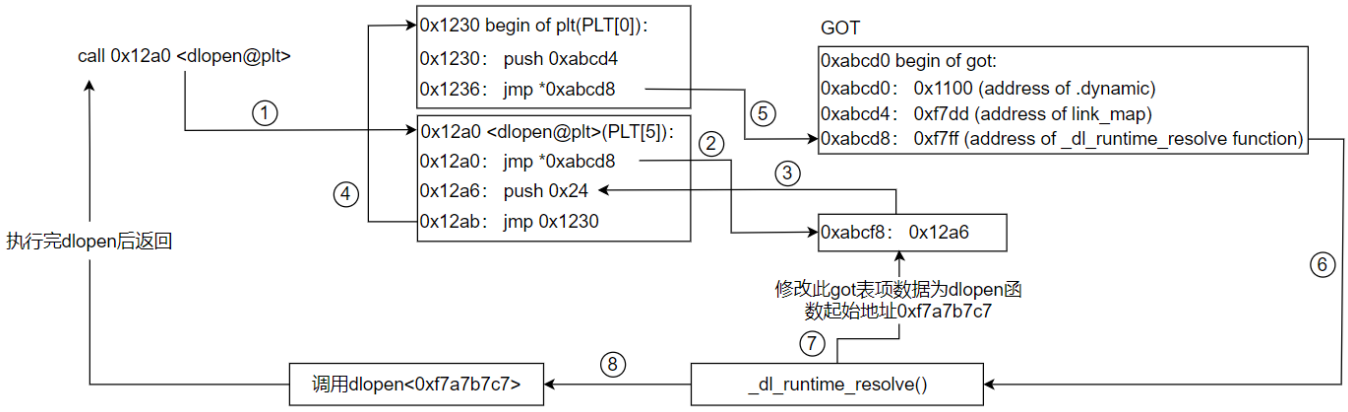

图 1-5 延迟绑定过程

0x13 重定位表

在ELF文件中，以".rel"开头的都属于重定位段，他是一个Elf32_Rel结构数组，每个项对应一个重定位入口。.rel.plt是对函数引用进行修正，在该博客中则需要使用.rel.plt段来获取外部函数地址。

如 图1-6 Elf32_Rel结构所示，重定位表的每个项都有两个成员，分别为offset和info。对于offset成员来说，其是重定位入口的偏移地址，即该符号对应的got表项地址，因此当重定位后将外部函数的真实地址回写到对应的got表项则是依赖于该数据，info成员则是表示该重定位入口的类型，该成员大小为4个字节，高24位表示该符号在符号表中的下标，低8位为info数据。

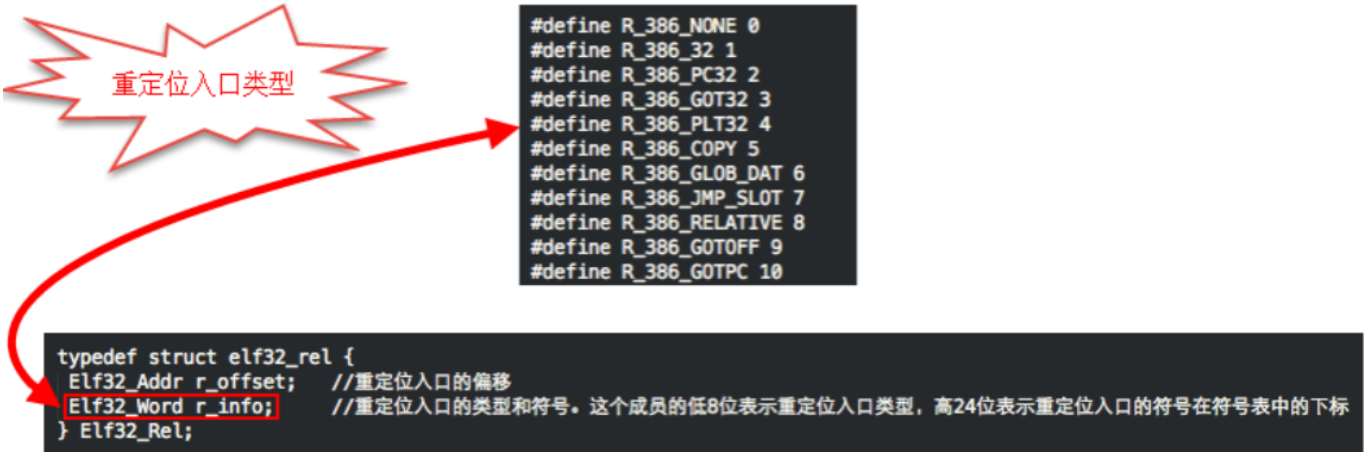

图 1-6 elf32_rel结构.png

那么以上就是要弄清楚如何从ELF文件中获取外部动态链接库函数地址所需的前提知识了。

0x2 获取ELF文件内部函数地址

ELF文件内部地址很容易获取，因此简略的说一下。

如 图2-1 内部函数在符号表中保存的地址数据所示，对于存在符号表的ELF文件，那么内部函数可以在符号表(symtab)中获取，动态符号例如导出函数还可以在动态符号表(dynsym)获取其地址。

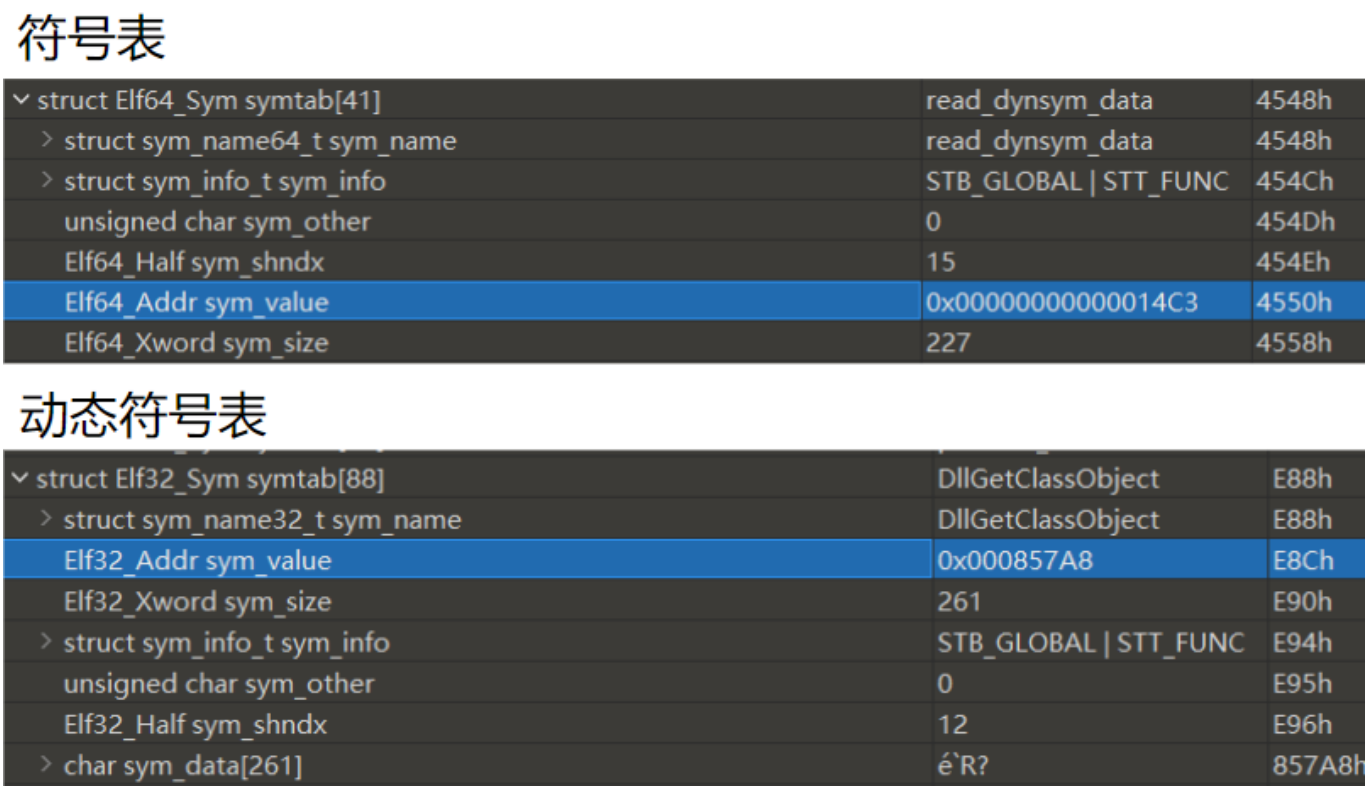

图 2-1 内部函数在符号表中保存的地址数据

如 图2-2 存储在.data节区中的内部函数地址数据所示，如果ELF文件中没有符号表，那么除了导出函数的函数地址可以从动态符号表获取，其余内部函数只能从.data节区的数据获取内部函数地址，例如图中的 dd offset sub_5F488的数据就是sub_5F488的函数地址。

```
.data:00476500                                  _data segment align_32 public 'CODE' use32
.data:00476500                                  assume cs:_data
.data:00476500                                  ;org 476500h
.data:00476500                                  assume es:nothing, ss:nothing, ds:_data, f
.data:00476500 00 65 47 00                      off_476500 dd offset off_476500         ; 
.data:00476500                                                                          ;
.data:00476500                                                                          ;
.data:00476500                                                                          ;
.data:00476504 70 64 4B 00                      off_476504 dd offset dword_4B6470       ; 
.data:00476504                                                                          ;
.data:00476504                                                                          ;
.data:00476508 00 00 00 00                      dword_476508 dd 0                       ; 
.data:00476508                                                                          ;
.data:0047650C 00 00 00 00 00 00 00 00 00 00+align 20h
.data:00476520 9C F3 05 00                      off_476520 dd offset sub_5F39C          ; 
.data:00476524 88 F4 05 00                      dd offset sub_5F488
.data:00476528 A8 F4 05 00                      dd offset sub_5F4A8
.data:0047652C 00 00 00 00 00 00 00 00 00 00+align 20h
.data:00476540 04 F8 05 00                      off_476540 dd offset sub_5F804          ; 
.data:00476544 90 F8 05 00                      dd offset sub_5F890
.data:00476548 B0 F8 05 00                      dd offset sub_5F8B0
.data:0047654C D4 F5 05 00                      dd offset sub_5F5D4
.data:00476550 54 F6 05 00                      dd offset sub_5F654
.data:00476554 24 F9 05 00                      dd offset sub_5F924
.data:00476558 84 F9 05 00                      dd offset sub_5F984
.data:0047655C E4 F9 05 00                      dd offset sub_5F9E4
```

图 2-2 存储在.data节区中的内部函数地址数据

0x3 获取ELF文件外部动态链接库函数地址

在0x12的延迟绑定分析中，程序调用外部函数则是通过call该函数对应的plt表项地址来实现的。因此关键问题是如何获取外部动态链接库函数对应plt表项的地址。

在分析plt表的过程中，注意到push的数字与plt表项的索引有着对应关系，从plt[1]开始，该项的push的数字则为0， plt[2]中push的为8，依次push的就是(n-1)*8了。前面也提到了，push的数字为该函数在.rel.plt重定位表中的偏移，因此只要定位到了外部导入函数对应的重定位表数据，就可以倒推出对应的plt表的位置。如 图3-1 plt表与重定位表对应示例所示，readlink对应的plt[1] push了0，而该函数在.rel.plt表的第一项；sem_trywait对应plt[2] push了8，该函数在.rel.plt表的第二项。

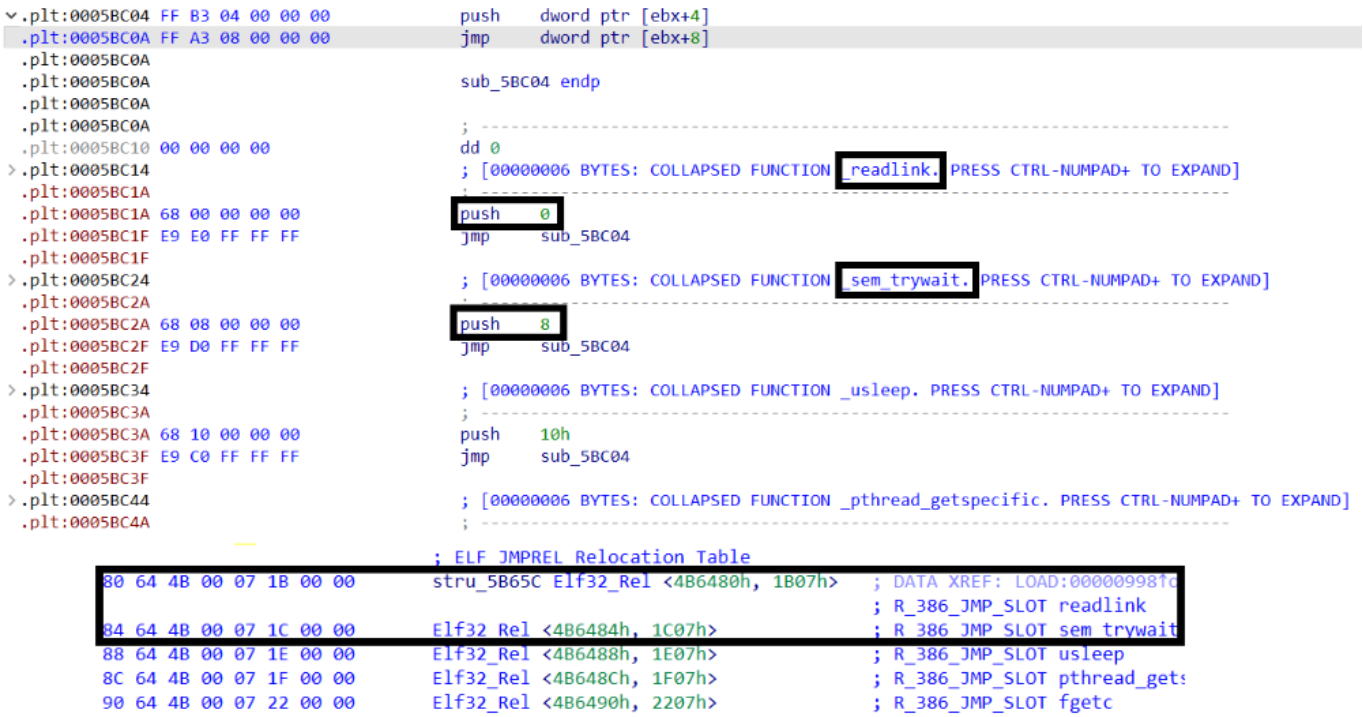

图 3-1 plt表与重定位表对应示例

那么如何定位该外部函数位于重定位表的哪个位置呢？

在前面我们提到重定位表中的Elf32_Rel结构的info成员的高24位代表该符号在符号表的下标，那么通过该下标就可以获取该符号的名称，那么与函数名进行对比即可定位到目标外部导入函数了。实现的方法如 图3-2 通过重定位数据对比符号名称所示，首先获取info成员高24位数据，通过它获取符号表的名称，在与外部导入函数名称对比即可判断该重定位项是否是目标外部导入函数的。

```
idx = rel_table[c].r_info>>8; // get index in .dynsym table
fseek(fd,str_off+sym_table[idx].st_name,SEEK_SET);
if(!strcmp(fd->_IO_read_ptr,"dlopen"))
```

图 3-2 通过重定位数据对比符号名称

当确定了该重定位项就是目标外部导入函数的之后，就可以计算该项在重定位表的偏移了，然后根据重定位表偏移 = (n-1)*8算出n，在用plt表起始地址 + n*0x10即可获得到对应的plt表项的地址了。

如 图3-3 计算对应的plt表项地址所示，首先获取重定位表第一项的offset数据，然后在获取目标外部导入函数的offset数据，然后作差除以4，那么结果就是该重定位项在重定位表的偏移了。因为重定位向的offset数据为目标外部导入函数在got表的地址，而got表与plt表有对应关系，自然也与重定位表也有对应关系。got表与重定位表的对应关系则是got[n] <--> .rel.plt[n-3]， n>3。got表项大小为4，因此 n-3 = (address of got[n] - address of got[4])/4 刚好为该重定位项在重定位表的偏移。那么最后就通过plt表起始地址 + n*0x10算出目标外部导入函数所在plt表项的地址了。

```
uint32_t rel_begin = rel_table[0].r_offset;
dop_addr = rel_table[c].r_offset;
plt_idx_dop = (dop_addr - rel_begin)/4;
dop_addr = plt_addr + (plt_idx_dop+1)*0x10;
printf("dlopen@plt address is 0x%x\n",dop_addr);
```

图 3-3 计算对应的plt表项地址

实现代码

实现代码