### PE文件整体结构

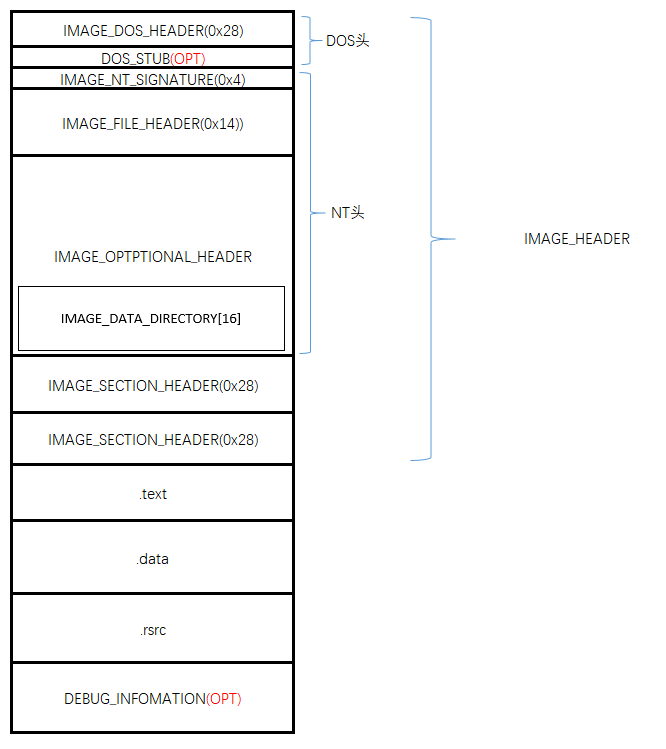


图 1 - 1

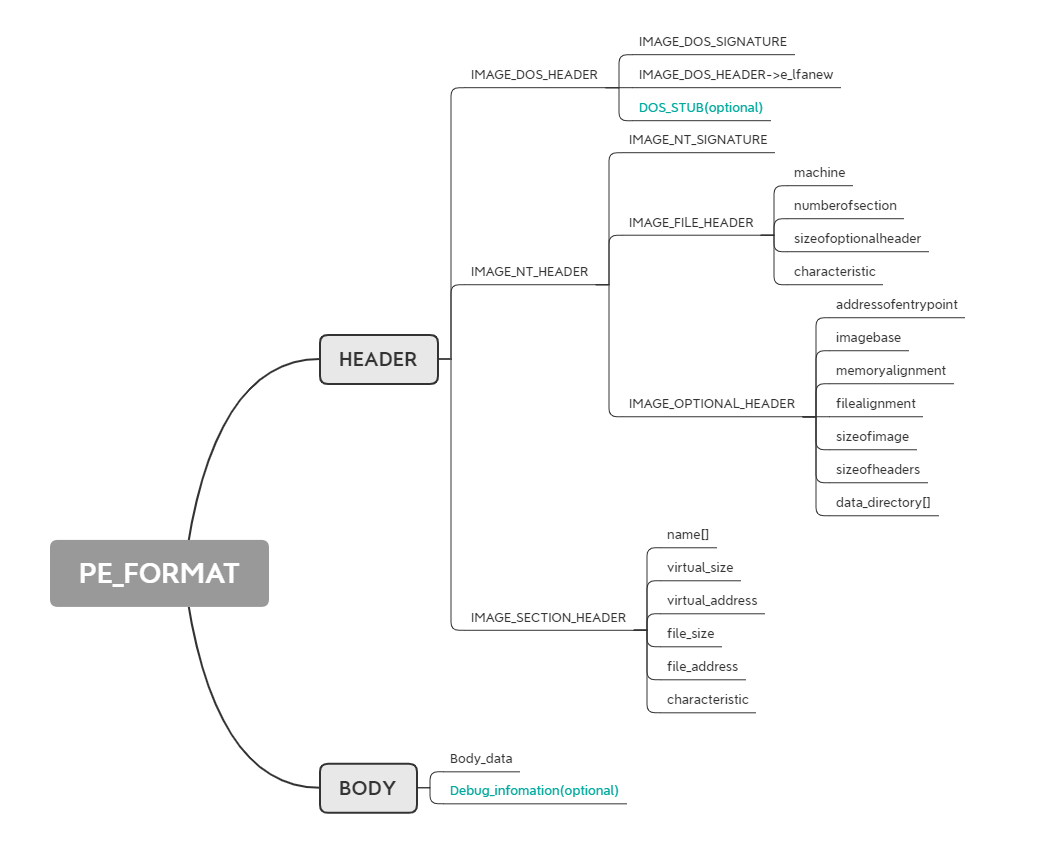


图 1 - 2

以上图 1 - 1和图 1 - 2描述了PE文件格式的整体轮廓，其中图 1 - 1中从文件布局的角度反应了各个结构的相对位置，在PE文件结构的组织上，头部结构大多有固定大小，在图中括号注解标注出。图 1 - 2描述了PE文件头部中关键的结构以及重要的字段，后续解析过程中会详细解析各个结构及其字段的含义。

### PE文件加载映射过程

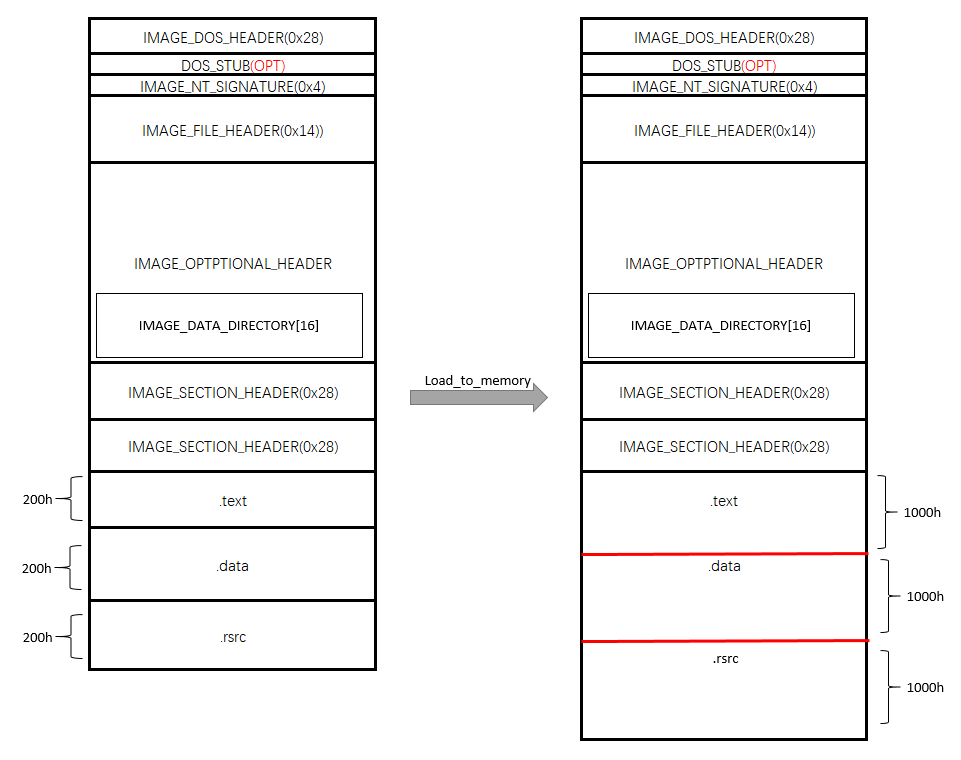


图 2 - 1

加载器在将PE文件从磁盘加载到内存中执行时，直接将IMAGE\_DOS\_HEADER、IMAGE\_NT\_HEADER、IMAGE\_SECTION\_HEADERS等文件头部信息拷贝到ImageBase对应的地址处，然后根据IMAGE\_OPTIONAL\_HEADER中定义的内存和文件对其粒度对节区对其到粒度边界。

### 解析术语

在解析过程会用到以下术语描述PE文件的不同状态。

**VA**：Virtual Address，经过加载器加载到内存后PE文件所处的地址位置。例如，PE文件直接使用VA地址寻址字符串。

**RVA**：Relative Virtual Address，相对虚拟地址

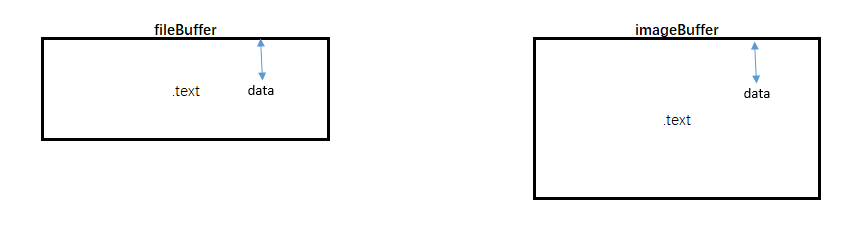


图 3 - 1

由于节区在加载后经过对其，要跟根据imageBuffer中的data地址找到fileBuffer中对应的data，就需要根据图中蓝色距离相等即(data–sec\_base\_address)->fileBuffer==(data–sec\_base\_address)->imageBuffer来定位。而imageBuffer中data的地址即RVA，相对于其节区基址的地址。

**FOA：**File Offset Address上述根据imageBuffer中data定位的描述即可得到fileBuffer中data的FOA地址，相对于fileBuffer的偏移。

### 文件头部解析(IMAGE\_HEADER)

#### 4.1. DOS文件头部

如图 1 - 2描述，已经将PE文件头部重要的字段描述出来，将文件读入fileBuffer之后，即可定位到IMAGE\_DOS\_HEADER,此结构固定大小为0x28字节，重要的字段只有头尾两个e\_magic一个两字节的魔术字，纪念PE文件格式的创建者名字的首字母”MZ”->0x4d5a,加载器会校验此特征，篡改之后PE文件无法正常加载，如图 4 - 3所示，最后一个四字节e\_lfanew指向IMAGE\_NT\_HEADER的偏移。

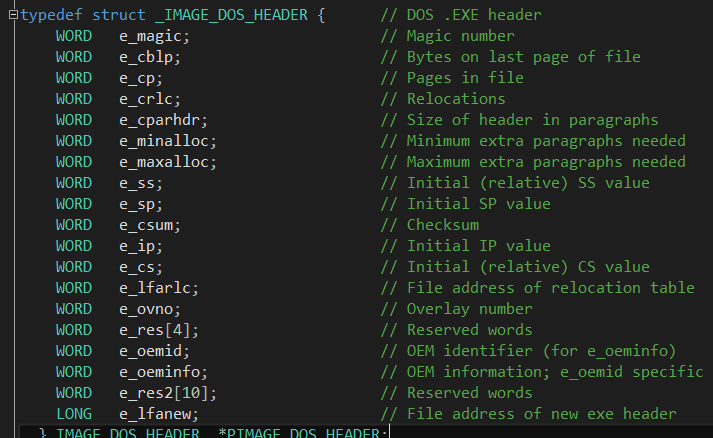


图 4 - 1

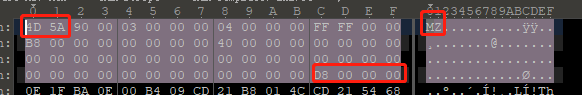


图 4 - 2

篡改文件的DOS\_SIGNATURE之后无法正常加载。

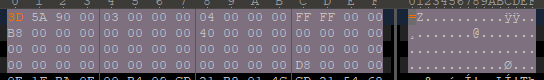




图 4 - 3

#### 4.2 NT文件头部

根据DOS头部最后一个字段偏移寻址到IMAGE\_NT\_HEADER,此结构由三部分组成，首先是一个四字节PE标记IMAGE\_NT\_SIGNATURE，然后是一个20字节的IMAGE\_FILE\_HEADER和一个IMAGE\_OPTIONAL\_HEADER.

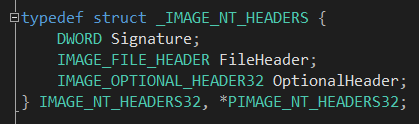


图 4 - 4

同样，Signature字段也是固定标记字，不能篡改。IMAGE\_FILE\_HEADER中定义了PE文件编译目录平台信息Machine字段，在winnt.h中定义了完整的平台特征码。查用如下：

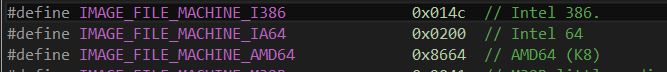


图 4 - 5

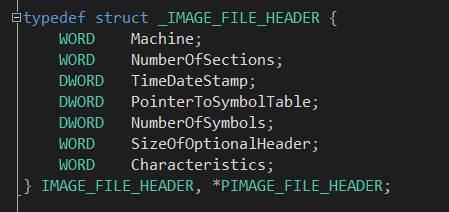


图 4 - 6

NumberOfSections定义了节区的数量，也即IMAGE\_SECTION\_HEADER的数量，节区表和节区一一对应。SizeOfOptionalHeader表明紧挨着的IMAGE\_OPTIONAL\_HEADER的大小，在i386平台上为0xE0, x64平台为0xF0字节大小。最后Characteristic定义了PE文件格式的属性信息，如下是对应的属性码：

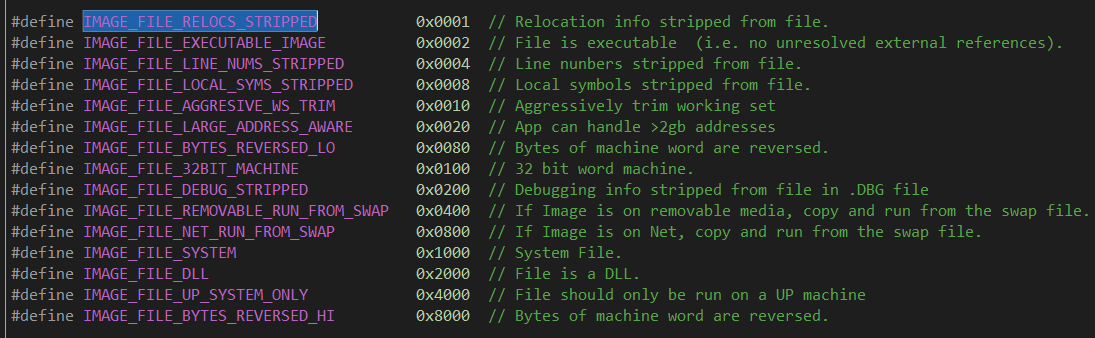


图 4 - 7

之后便是IMAGE\_OPTIONAL\_HEADER，此结构较大，重要的字段有相对偏移0x10的AddressOfEntryPoint，其定义了PE文件加载到内存中执行的第一条指令的偏移。然后是偏移0x1c的ImageBase，加载器会将PE文件加载此虚拟地址处，紧接着两个DWORD分别代表内存映像对齐粒度SectionAlignment和磁盘文件对齐粒度FileAlignment，加载过程中会根据内存对齐粒度”拉伸”对应的节区。后面是0x38偏移处的SizeOfImage,文件加载到内存中的大小，和SizeOfHeader，文件头部的总大小(经过文件对齐)，包含DOS、NT、节表。最后最重要的就是0x60偏移处的数据目录表了，包含十六项，每一个8字节，分别代表对应数据目录表对应的RVA和大小。后续解析对应数据目录的时候会用到此目录。

以上偏移地址对应于32位PE文件，在64位PE文件格式中，头部仅仅增大了IMAGE\_OPTIONAL\_HEADER，由0xE0->0xF0,具体有：

1. 将原本偏移0x18和0x1c的SizeOfData和ImageBase合并为一个8字节的ImageBase。
2. 将原本偏移0x48处的四个4字节字段扩展为8字节。

所以整个结构就增大了16字节，数据目录表就位于0x70处了。

#### 4.3 节表

解析完IMAGE\_OPTIONAL\_HEADER后紧接着就是IMAGE\_SECTION\_HEADER，每一个节表对应描述一个节区的属性，如Name指向节区的名称，以’\0’为结束符。VirtualSize,该节区在内存中的大小(未对齐),VirtualAddress, 节区加载到内存中相对于基址的偏移。同理，SizeOfRawData和PointerToRawData代表文件中对应的大小和偏移，Characteristics指示对应节区的属性，有如下预定义的特征码：

如是否为代码节、是否可读、可写、可执行等等。

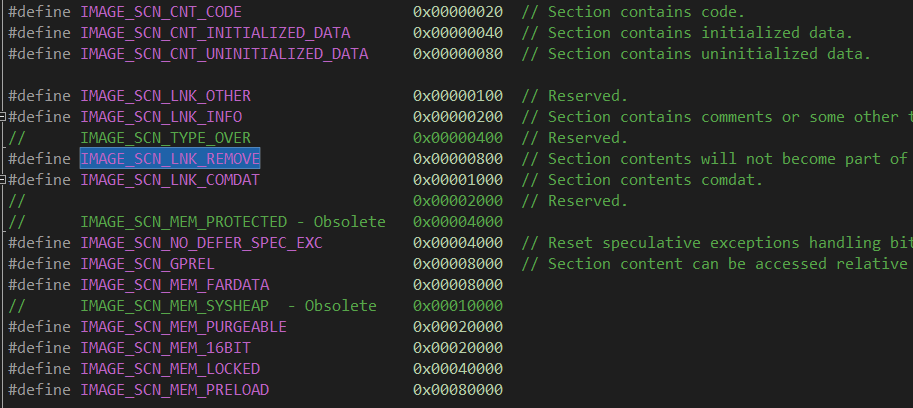


图 4 - 8

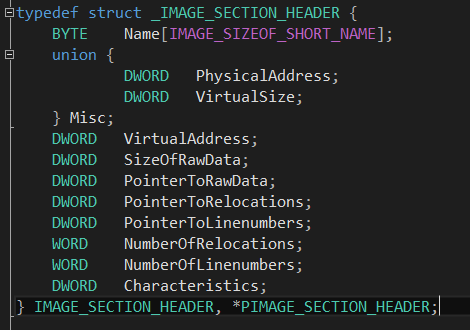


图 4 - 9

### 数据目录表解析(IMAGE\_DATA\_DIRECTORY)

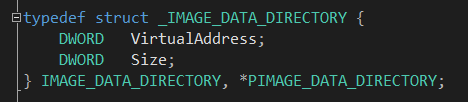


图 5 - 1

数据目录结构中只有两个成员，VirtualAddress指定了数据目录的RVA，Size则表示该块的大小。数据目录表根据编译的目标平台不同可能位于相对IMAGE\_OPTIONAL\_HEADER头偏移0x60或0x70处，其大小为0x80字节。包含16个数据项，每一项8字节，分别指向对应数据目录项的RVA和其大小。其中比较重要的有第0项导出表、第1项导入表、第2项资源表、第4项证书表(主要用于PE校验和检测)、第5项重定位表.

### 导出表解析(IMAGE\_EXPORT\_DIRECTORY)

首先根据数据目录表中对应的项RVA转换FOA后定位到IMAGE\_EXPORT\_DIRECTORY，

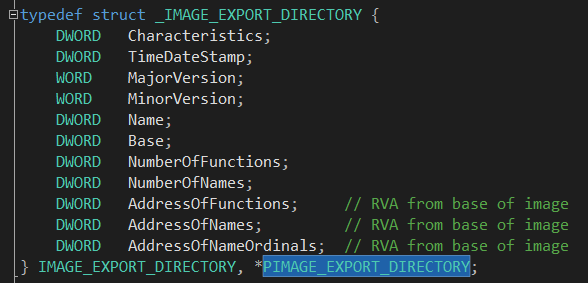


图 6 - 1

导出表用于导出函数，导出函数供外部使用，结构中前四个字段基本没什么用，除此之外的字段都是重要字段。可以看到一个函数的导出可以以名称或者直接以ID进行导出，于是便可看到最后三个字段分别指向导出函数地址表，导出函数名称表，导出函数序号表。

AddressOfFunction, RVA地址，用于索引到存储导出函数地址的表。NumberOfFunctions限定表长。

AddressOfNames，RVA地址，用于索引到所有以名称导出的函数的名称表所在，表长为NumberOfNames，索引到的结构也是一个RVA地址组成的指针表。

AddressOfOrdinals，RVA地址，用于索引到序号表，该表元素为2字节的序号组成的数组，长度为NumberOfNames.

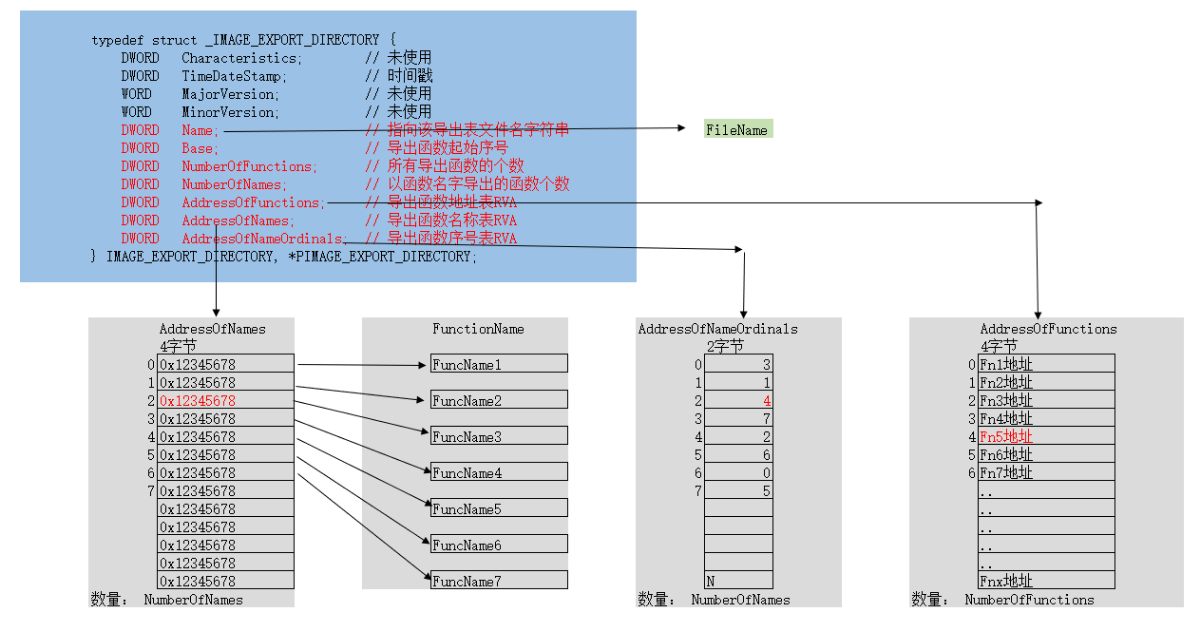


图 6 - 2

通过导出表查找函数地址的方式：

1. 通过函数名查找、

(1). 首先定位函数名称表，遍历函数名称与所查找名称比较，取得函数名称在表中的索引；

(2). 使用索引取得序号表中对于的序号；

(3). 使用取得的序号作为索引在函数地址表中取得函数地址。

1. 通过序号查找

(1). 直接计算函数地址表的索引，使用待查序号-导出表base值即可

(2). 使用序号作为索引在函数地址表中取得函数地址。

### 七．导入表解析(IMAGE\_IMPORT\_DESCRIPTOR)

导入表是当PE需要使用其他DLL中的函数时所需的。加载器在将PE文件装载到内存中后，根据导入表将待导入的DLL加载到相应基址后，解析导入表中导入的函数地址到IAT表。

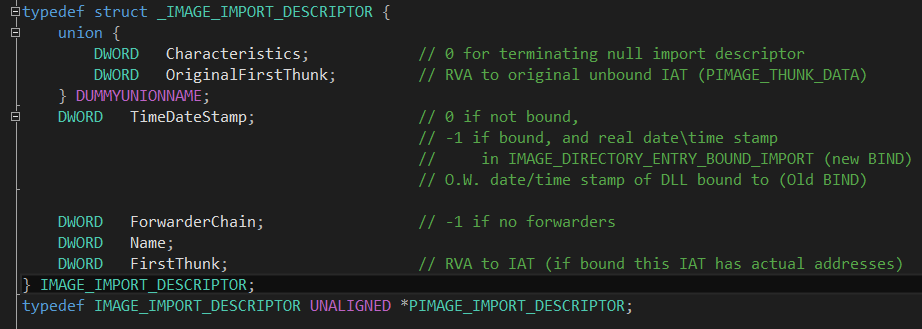


图 7 - 1

OriginalFirstThunk, RVA地址，指向IMAGE\_TRUNK\_DATA结构数组，数组以一个全0的元素作为结束，每一个IMAGE\_TRUNK\_DATA元素四字节大小，若元素最高位为1，则表示待导入的函数是以序号导入。剩余位标识序号。

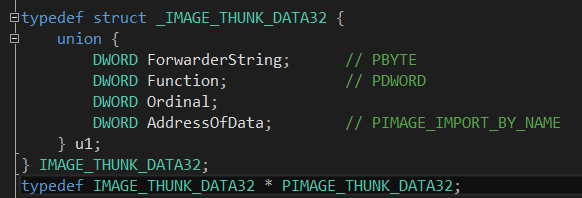


图 7 - 2

若最高位不为1，则此元素是一个RVA地址，指向IMAGE\_IMPORT\_BY\_NAME结构，表示以名称导入函数，结构由一个两字节的序号和名称字符串组成。

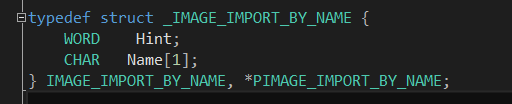
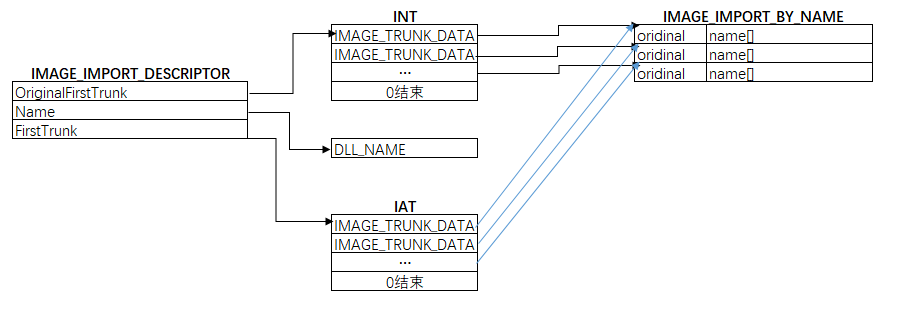


图 7 - 3

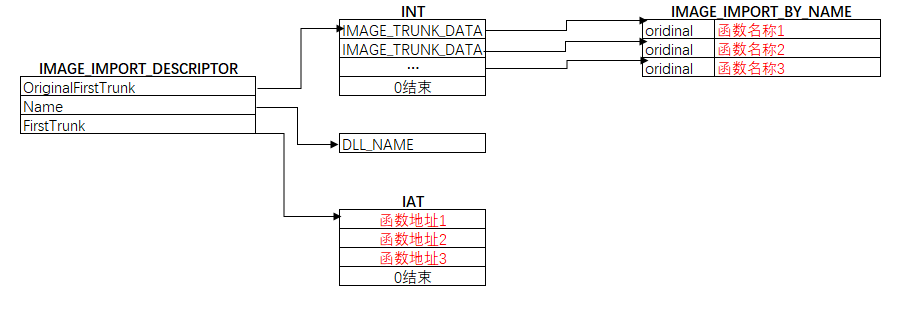
Name, RVA地址，指向待导入的文件名称字符串。

FirstTrunk, RVA地址，指向IAT表，即函数地址表，在加载器未解析函数地址之前，此地址指向的内容也跟OriginalFirstTrunk一样。



解析前(静态)

加载器解析IAT表之后会将解析到的函数地址填充到FirstTrunk指向的IAT表中。



解析后

### 资源表解析(IMAGE\_RESOURCE\_DIRECTORY)

资源表的解析最为复杂，用于存储PE文件中的资源信息，包括图标、鼠标指针、字体、菜单等等。资源表的组织有三层结构组成，前两层的结构又类似。都是由IMAGE\_RESOURCE\_DIRECTORY和IMAGE\_RESOURCE\_DIRECTORY\_ENTRY组成，IMAGE\_RESOUCE\_DIRECTORY定义了资源目录项的信息，结构中前几个字段不重要。

NumberOfNameEntries, 表示接下来有名称的目录项数量。

NumberOfIdEntries，表示接下来有ID的目录项数量，两者之和表示这一层目录中总共的目录项数。

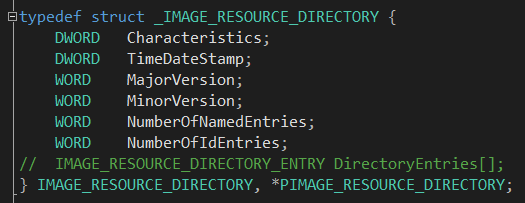


图 8 - 1

紧接在IMAGE\_RESOURCE\_DIRECTORY目录后面的是目录项结构IMAGE\_RESOURCE\_DIRECTORY\_ENTRY,整个结构由两个union组成，前一个DWORD用于指示资源是以名称还是ID存储的，若最高位置位，则剩余31位，作为偏移（注意，整个资源表中的偏移既不是RVA也不是FOA，而是相对于资源表基址的偏移，即第一层目录的基址。）

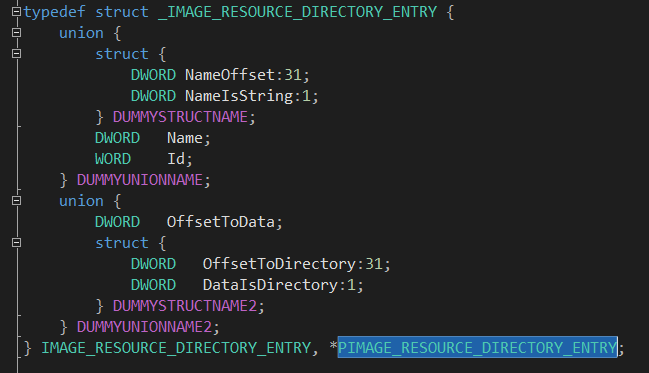
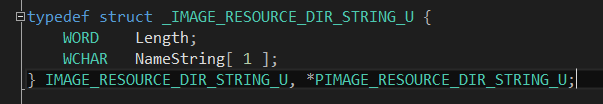


图 8 - 2

索引到IMAGE\_RESOURCE\_DIR\_STRING\_U结构，指示名称字符串的长度Length和基址NameString，此处为unicode编码字符。



否则，低WORD表示资源的ID，Windows为一级目录预定义了14种资源特征码。



图 8 - 3

后一个DWORD同理，若最高位置位，则表示此目录项指向一个数据项，并根据低位的数据偏移索引到数据项对应结构IMAGE\_RESOURCE\_DATA\_ENTRY。

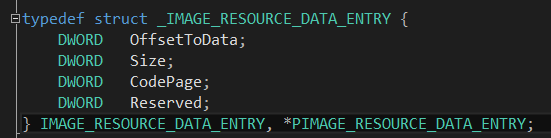


图 8 - 4

整个资源表的组织框图如下：

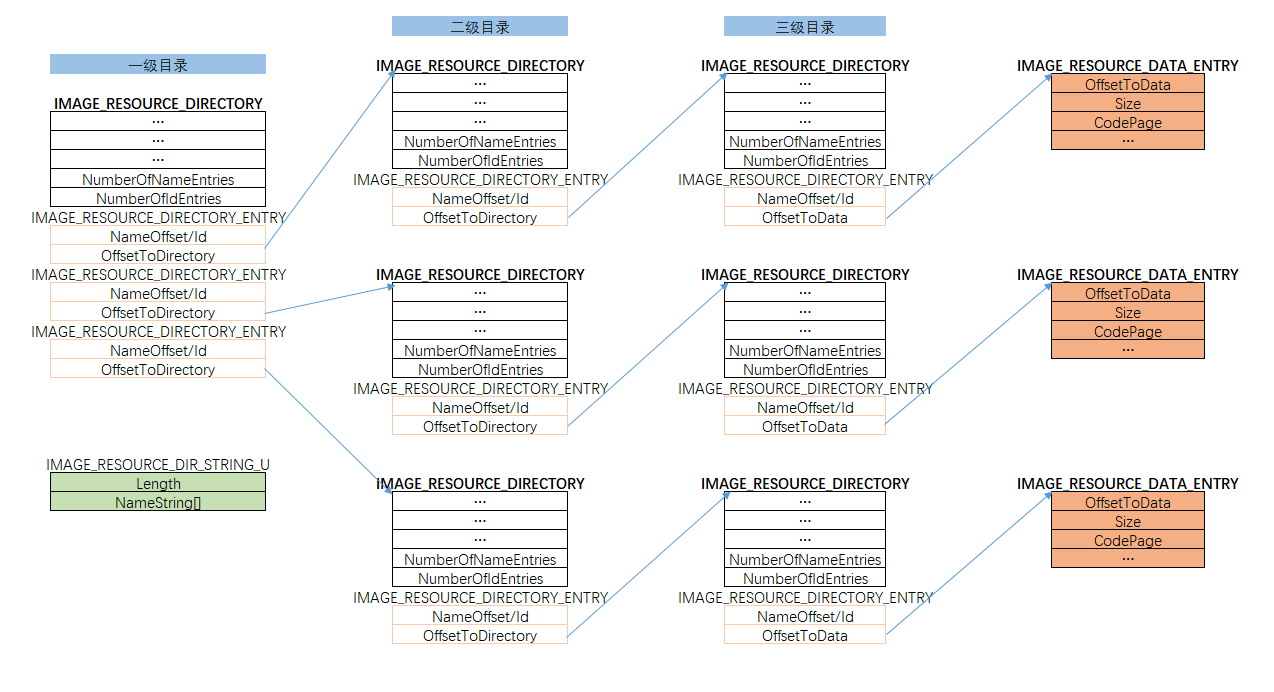


图 8 - 5

### 九．重定位表解析(IMAGE\_BASE\_RELOCATION)

程序加载执行时会导入多个dll文件，每个dll文件也是PE结构，也有着对于的加载基址，当多个dll加载的基址冲突时，就需要将后加载的dll文件加载到内存的其他位置，而PE文件中有一些地址是以VA地址的形式出现在文件中的，一旦该表其加载基址，这些VA地址都将失效，为了修复这些地址引入重定位表。修复的方式为：需要重定位的地址 – PE文件中定义的基址 + 当前加载的真实基址。

重定位表结构如下所示：

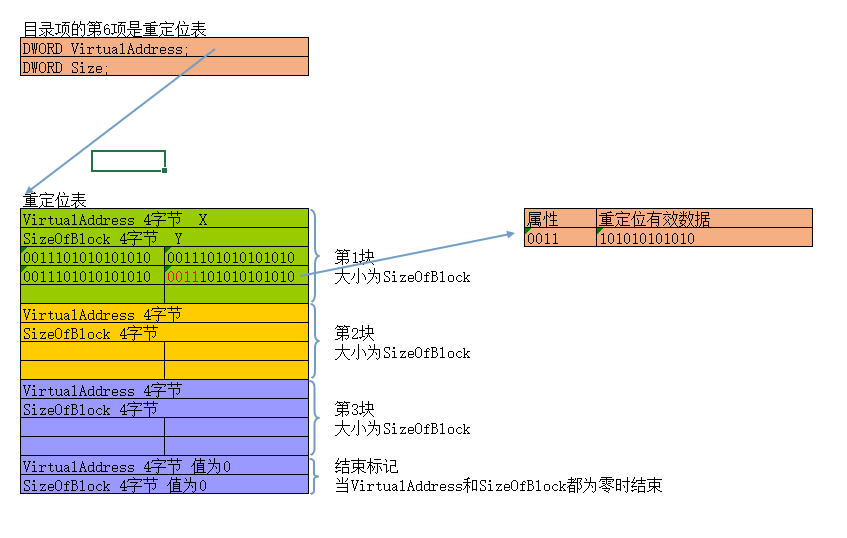


图 9 - 1

数据区以高4位作为属性位，低12位作为有效数据位，此数据加上VirtualAddress才是真正的待重定位数据的RVA地址，转换FOA之后即可定位到对应的重定位数据。

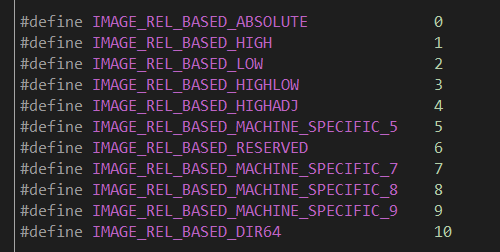
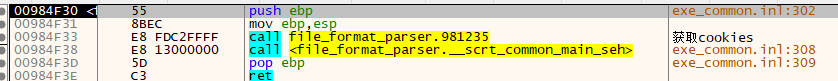


图 9 - 2

### 二进制分析

#### 10.1 CRTMainStartUp入口

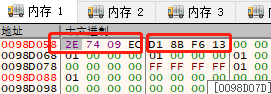


进入crt入口后只有两个call，由于编译时开启了GS检查，首先在第一个call中获取cookie的值。跟入查看cookie计算方法。



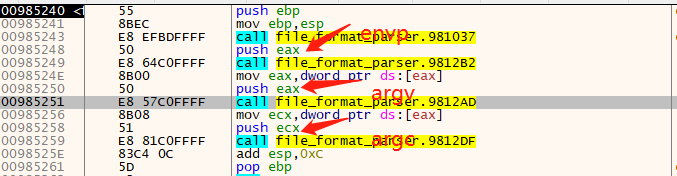
根据右边分析注释，获取cookie主要有三个分支组成，首先与一个特征值比较，若相等，则跳到标签3，逐次执行图中注释操作后，再次与特征值比较，若不等，则进一步判断高两个字节是否为0；最后函数出口结果如图中标有[ret]的注释给处：

1. 在cookie不等于特征值和高两个字节不为0的情况下，直接取反并返回；
2. 在1不满足的情况下，逐次计算中间过程，再次与特征值比较，若相等，则cookie赋值为特征值，并取反返回。
3. 若2中与特征值比较不相等，则再判断高两个字节是否为0，若不为0，则cookie取反存入[\_\_security\_cookie\_complement],返回
4. 若3中判断成立，则取出cookie临时值，经过最后一个[ret]后续操作后，取反，返回。

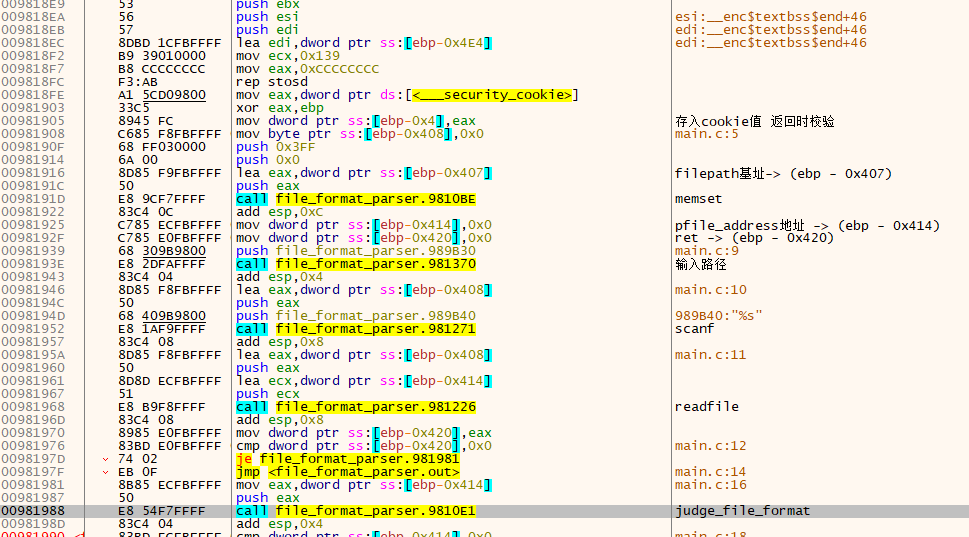


#### 10.2 main函数

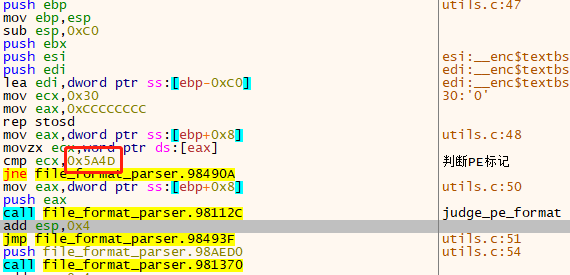
继续跟踪，根据main函数入口标记的三个参数，可定位到main函数：



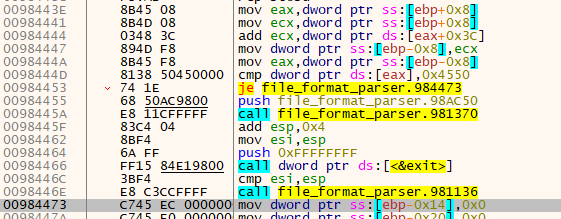
跟入main函数，可看到函数逻辑如下注释



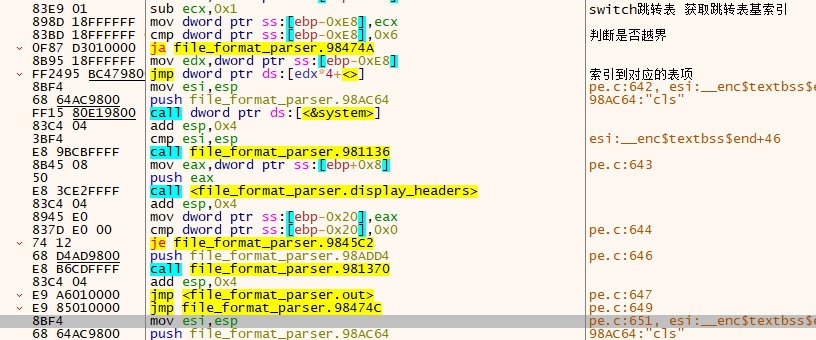
首先根据之前获取的cookie，再栈帧底部存入cookie，便于返回时校验，之后便初始化相应的局部变量：filepath路径名地址->(ebp-0x407)、pfile\_address地址->(ebp-0x414)、ret地址->(ebp-0x420)，完成初始化后，等待用户输入文件路径。之后就调用readfile读入文件到内存pfile\_address处，进入文件格式判断。



首先取出pfile\_address处的两字节与MZ标记比较，相等则进入PE标记判断。



首先取出[ebp+0x8]处的参数即pfile\_address，然后根据DOS头最后一个DWORD指向NT头偏移索引到NT头结构，之后判断NT头标记，若相等则进入后续流程，否则调用exit(-1)跳出main函数。

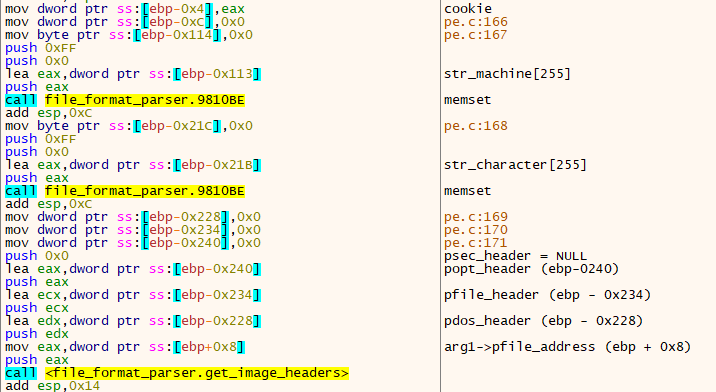


首先解析开头使用的switch语法的case项为连续标识，经过编译器优化后形成跳转表，每一个表项存储对应case项的起始地址，如下图所示：

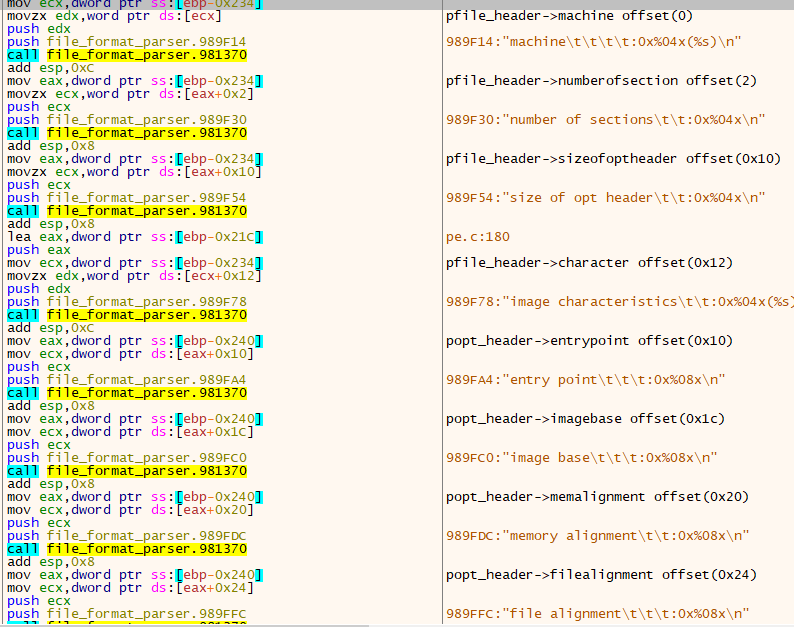


经过switch跳转表之后便进入到对应的解析函数中。

#### 10.3 文件头部分



如图中右边注释所示，函数开头先初始化相应局部变量：如str\_machine字符数组地址(ebp-0x113)、popt\_header地址(ebp-0x240)、pfile\_address地址[ebp+0x8]等等。根据结构偏移获取到各个头部结构后就是打印字段偏移处的字段了。

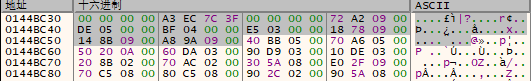


#### 10.4 导出表部分

根据数据目录表索引到导出表开始结构



查看对应内存数据

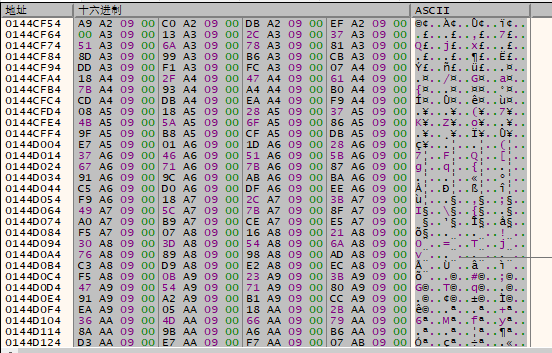


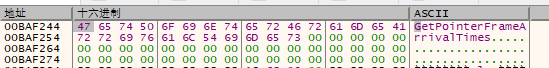
如图看到对应的结构如阴影区域，导出dll名称RVA(0x09A272)，索引到该地址



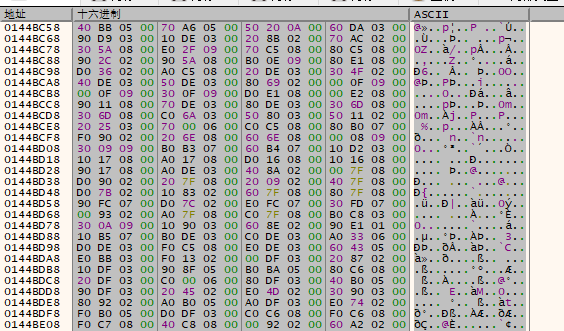
看到导出dll名称为user32.dll

其他结构如导出名称表：

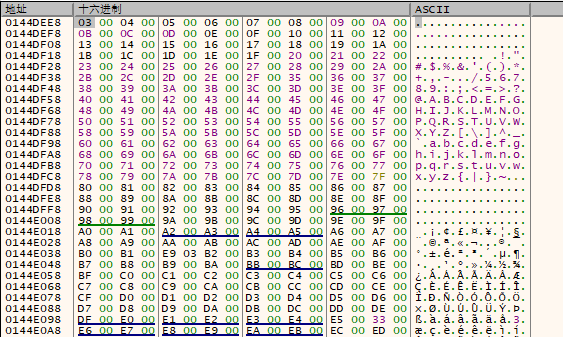




导出地址表：

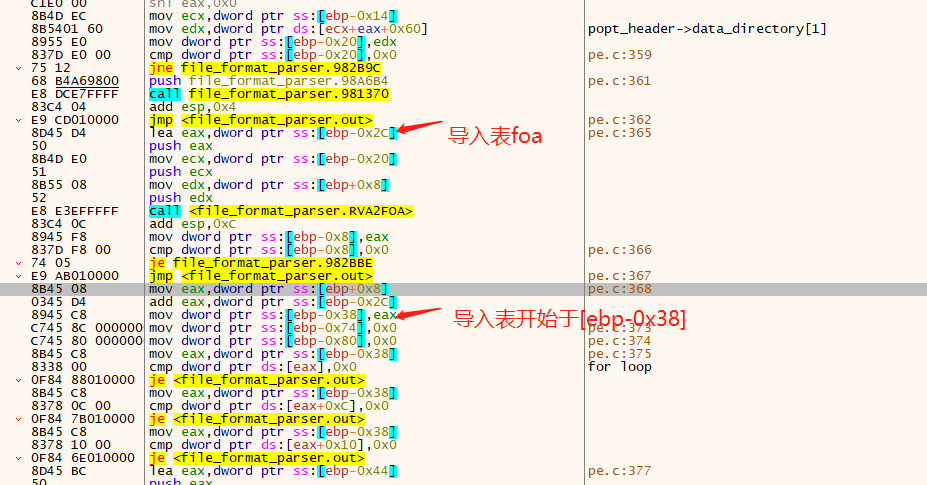


导出序号表：

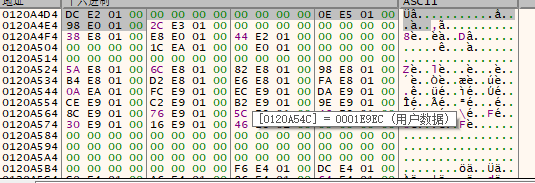


#### 10.5 导入表部分

根据数据目录表索引到导入表开始结构

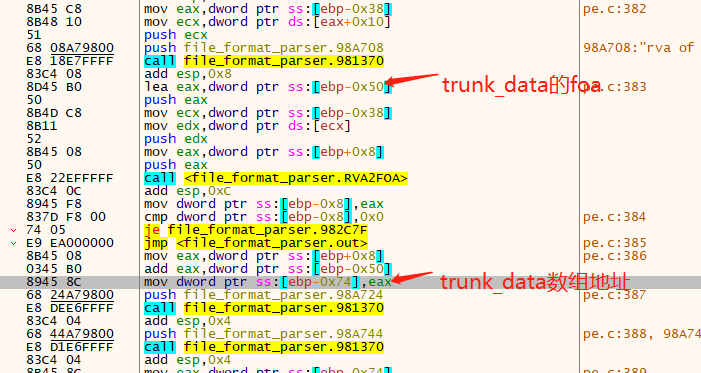


查看对应地址处结构如下：

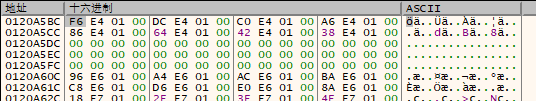


导入表结构是一个0x14字节的结构，如阴影区域即第一个导入表结构，整个结构以全0结束。

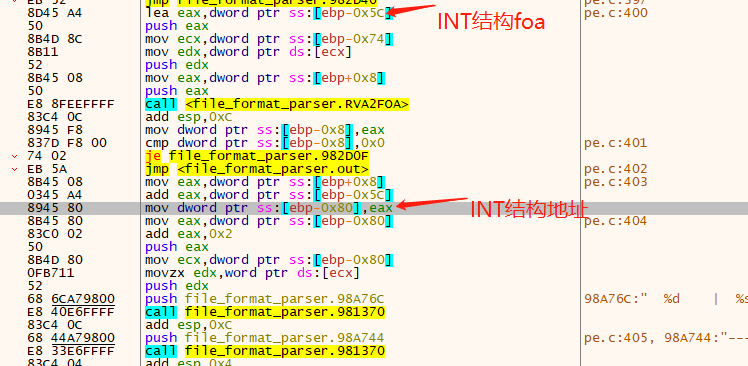
INT结构区域：便利上述导入表头数组，第一个字段指向trunk\_data结构，转换RVA地址后可索引到对应的结构



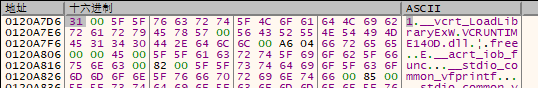
查看地址处内存：



可看到一共导入了八个符号，其中最高位为1的表示以序号导入，转换这些rva地址即可索引到最终导入的符号名称和序号值。

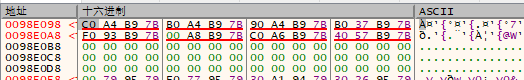


查看内存处地址：



即看到导入一个序号0x31的\_\_vcrt\_LoadLibraryExw符号。其他的符号索引类似。

IAT区域，在PE加载到内存中后，经过符号解析，IAT地址表就形成了

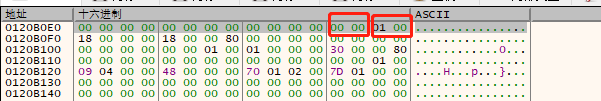


#### 10.6 资源表部分

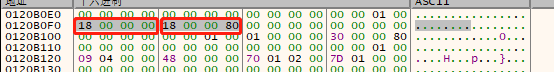
首先跟据数据目录表索引到resource\_dir结构的地址：



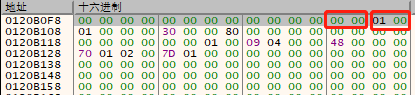
查看内存该地址内容：



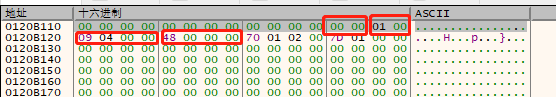
可看到此PE没有以name命名的目录项，有一个以ID命令的目录项。紧挨着就是一个八字节的目录项结构，



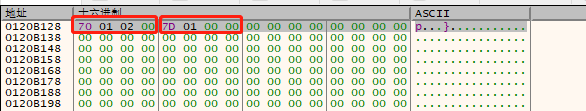
由于第一个DWORD最高位没有置位，表示该资源的ID为0x18，第二个DWORD最高位置位，表示还有第二级目录，索引到第二级目录为：

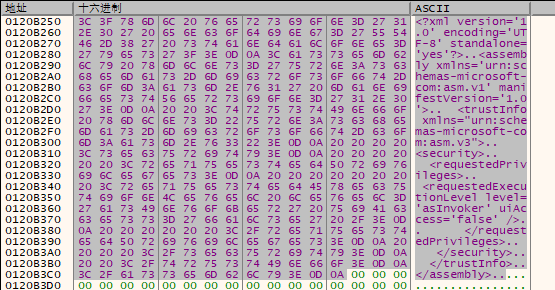


如图，第二级资源目录也仅有一个ID命令的资源项，紧接之后一个八字节结构表示此资源ID为1，并且还有第三级目录，相对于资源表基址的偏移为0x30:



与第二级目录类似，第三级目录仅有一个ID为0x409的资源，资源相对于资源表基址的偏移为0x48

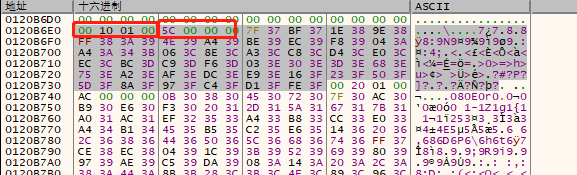




即可定位到最终资源的内容。

#### 10.7 重定位表部分

根据数据目录表定位到重定位结构：



可看到第一个物理页的重定项如阴影区域，起始基址为0x011000，整个结构大小为0x5c，后续每一个WORD表示一个重定位项，去掉高4四位后就可以索引对应的地址。