第 4 题 Windows PE 文件解析

张旭，软件工程 1805 班，U201817106 组长

蒋熙浩，软件工程 1805 班，U201817097

赵真，软件工程 1805 班，U201817105

张惊天，软件工程 1805 班，U201817108

何旺，软件工程 1805 班，U201817102

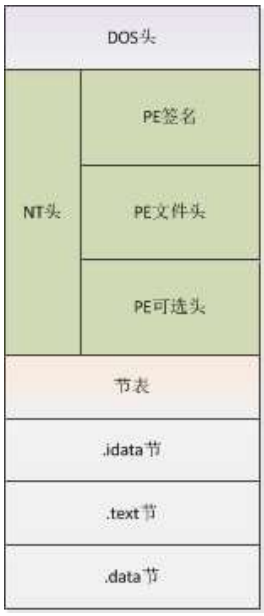
一. 功能需求　（或任务描述）

1.分析 Windows 中 PE 文件的格式和作用

2.结合 C 程序验证, C 程序要求有 2 个全局变量 (1 个已初始化, 1 个未初始化), 2 个自定义函数（一个带参数，1 个不带参数），每个函数都有 2 个局部变量。都有返回值

二. 程序设计思路（或整体结构或整体流程）

1.PE 文件的结构一般如下图所示，DOS 头，NT 头，节表，以及各种具体的节（图中并没有展示所有的节）



PE 文件的执行顺序:

1、PE 文件被执行，PE 装载器为文件在内存分配一个空的位置。创建进程和主线程。

2、PE 装载器检查 DOS MZ header 里的 PE header 偏移量。如果找到，则跳转到 PE header。

3、PE 装载器检查 PE header 的有效性。如果有效, 就跳转到 PE header 的尾部。

4、紧跟 PE header 的是节表。PE 装载器读取其中的节信息，并采用文件映射方

法将这些节映射到内存，同时赋上节表里指定的节属性。

5、PE 文件映射入内存后，PE 装载器将处理 PE 文件中类似 import table（引入表）逻辑部分。

三. 开发环境和配置（或主要模块和函数分析）

解析工具：010 Editor，配置：安装 EXE.bt 模板

接下来分析 PE 文件的结构。

PE 文件的格式部分在 winnt.h 头文件中，打开 winnt.h，搜索 Image Format 即可到达

PE 文件结构说明:

1、DOS 头用来兼容 MS-DOS，目的是当这个文件在 MS-DOS 上运行时提示一段文字，大部分情况下是：This program cannot be run in DOS mode. 还有一个目的，就是指明 NT 头在文件中的位置。

2、NT 头 包含 windows PE 文件的主要信息，其中包括一个 'PE' 字样的签名，PE 文件头(IMAGE_FILE_HEADER)和 PE 可选头(IMAGE_OPTIONAL_HEADER32)。

3、节表：是 PE 文件后续节的描述，windows 根据节表的描述加载每个节。

4、节：每个节实际上是一个容器，可以包含 代码、数据 等等，每个节可以有独立的内存权限，比如代码节默认有读/执行权限，节的名字和数量可以自己定义，未必是上图中的三个。

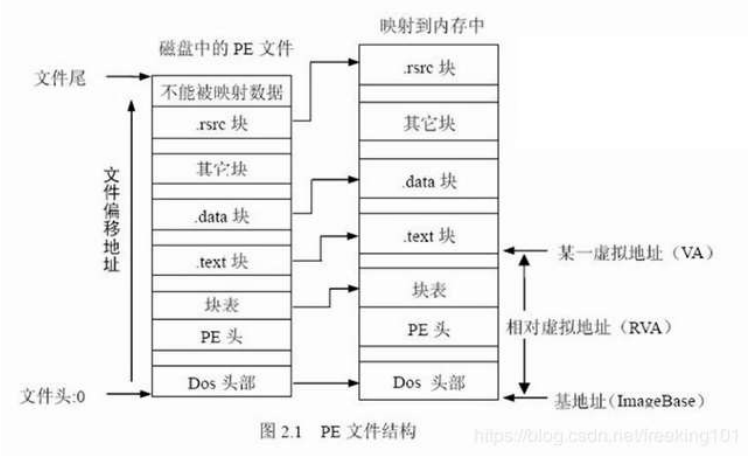


图 2.1 PE 文件结构

由于进程分页加载，所以存在页面对齐，在内存中中间会有间隙，以 0 填充

由于存在不同的对齐方式，所以在 PE 文件内部对地址的描述也采用了两种方式，针对在硬盘上存储文件中的地址，称为原始存储地址或物理地址表示距离文件头的偏移；另外一种是针对加载到内存以后映象中的地址，称为相对虚拟地址（RVA），表示相对内存映象头的偏移。在内存中使用 VA， VA 与 RVA 满足下面的换算关系： **RVA + ImageBase = VA**

具体分析各个部分格式：

1、DOS 头

PE 文件 DOS 头由两部分组成 DosHeader 和 DosStub，都是结构体

（1）DosHeader：

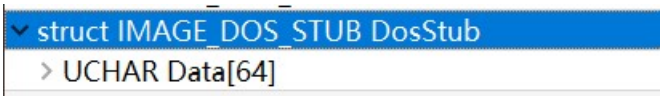
```
typedef struct _IMAGE_DOS_HEADER {      // DOS .EXE header
    WORD   e_magic;                     // Magic number
    WORD   e_cblp;                      // Bytes on last page of file
    WORD   e_cp;                        // Pages in file
    WORD   e_crlc;                      // Relocations
    WORD   e_cparhdr;                   // Size of header in paragraphs
    WORD   e_minalloc;                  // Minimum extra paragraphs needed
    WORD   e_maxalloc;                  // Maximum extra paragraphs needed
    WORD   e_ss;                        // Initial (relative) SS value
    WORD   e_sp;                        // Initial SP value
    WORD   e_csum;                      // Checksum
    WORD   e_ip;                        // Initial IP value
    WORD   e_cs;                        // Initial (relative) CS value
    WORD   e_lfarlc;                    // File address of relocation table
    WORD   e_ovno;                      // Overlay number
    WORD   e_res[4];                    // Reserved words
    WORD   e_oemid;                     // OEM identifier (for e_oeminfo)
    WORD   e_oeminfo;                   // OEM information; e_oemid specific
    WORD   e_res2[10];                  // Reserved words
    LONG   e_lfanew;                    // File address of new exe header
  } IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

DOS 头中主要的两个成员：

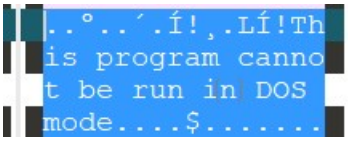
- e_magic：一个 WORD 类型，值是一个常数 0x4D5A，用文本编辑器查看该值位'MZ'，可执行文件必须都是'MZ'开头。
- e_lfanew：为 32 位可执行文件扩展的域，用来表示 DOS 头之后的 NT 头相对文件起始地址的偏移。

  在示例程序中可以看到　e_lfanew = 80h

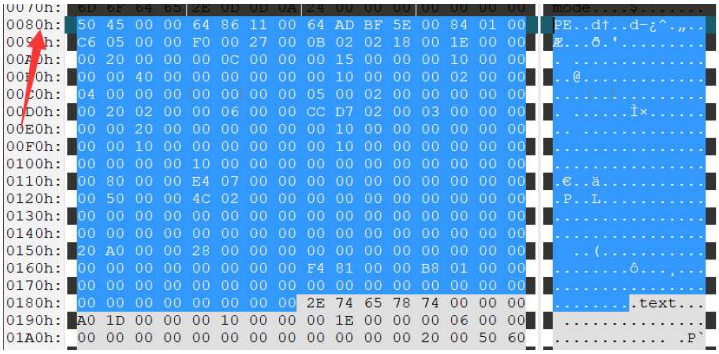
（2）DosStub：



是一个字符串，一串提示信息



2、NT 头（80h）



NT 头中包含 Windows PE 文件的主要信息，其中包括 PE 签名、PE 头文件（IMAGE_FILE_HEADER）和 PE 可选头（IMAGE_OPTIONAL_HEADER32）三个部分。

结构体描述为：

```
typedef struct _IMAGE_NT_HEADERS {
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

Name	Value	Start	Size
struct IMAGE_DOS_HEADER DosHeader		0h	40h
struct IMAGE_DOS_STUB DosStub		40h	40h
UCHAR Data[64]		40h	40h
struct IMAGE_NT_HEADERS NtHeader		80h	108h
DWORD Signature	4550h	80h	4h
struct IMAGE_FILE_HEADER FileHeader		84h	14h
struct IMAGE_OPTIONAL_HEADER64 OptionalHeader		98h	F0h

如图，在示例 exe 程序中，该 NT 头起始位置为 0080h,在其最开头即为 NT 头的

（1）第一部分内容：PE 签名，类似于 DOS 头中的 e_magic，取值为 4550h 占用 4 个字节，在右边可看到翻译出的文本为“PE..”四个字符

（2）NT 头的第二部分是 PE 头文件,PE 文件头中定义了 PE 文件的基本信息和属性，这些属性可以被加载器利用，加载器在加载时会检查 PE 文件头中定义的属性，若其不满足当前的运行环境，将会终止加载该 PE 文件.

```
typedef struct _IMAGE_FILE_HEADER {
    WORD    Machine;
    WORD    NumberOfSections;
    DWORD   TimeDateStamp;
    DWORD   PointerToSymbolTable;
    DWORD   NumberOfSymbols;
    WORD    SizeOfOptionalHeader;
    WORD    Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

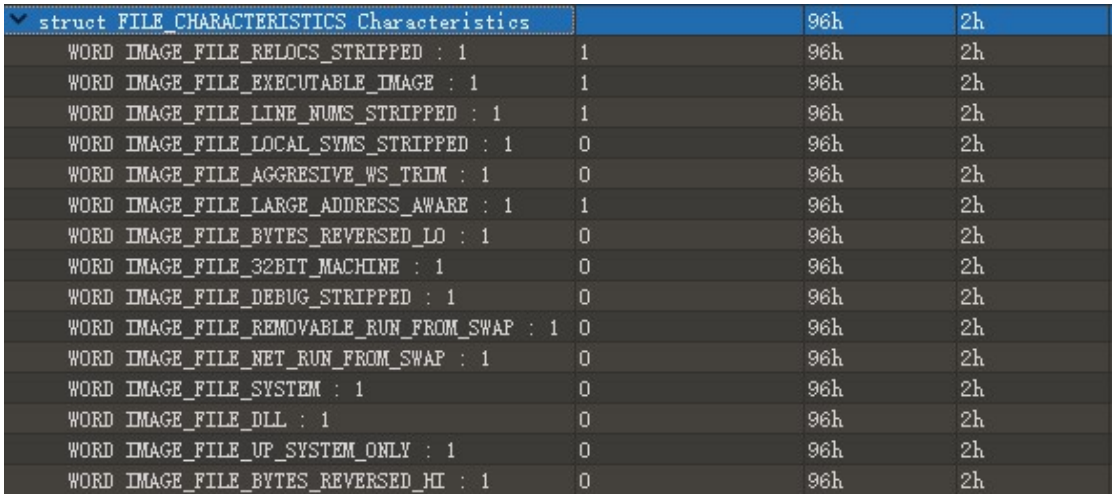
如下图所示：

struct IMAGE_NT_HEADERS NtHeader		80h	108h
DWORD Signature	4550h	80h	4h
struct IMAGE_FILE_HEADER FileHeader		84h	14h
enum IMAGE_MACHINE Machine	AMD64 (8664h)	84h	2h
WORD NumberOfSections	17	86h	2h
time_t TimeDateStamp	05/15/2020 14:53:41	88h	4h
DWORD PointerToSymbolTable	98816	8Ch	4h
DWORD NumberOfSymbols	1477	90h	4h
WORD SizeOfOptionalHeader	240	94h	2h
struct FILE_CHARACTERISTICS Characteristics		96h	2h

- Machine 描述了该文件的运行平台，如 x86、x64 等等，上图示例中为

ADM64(8664h)

- NumberOfSections 描述了该 PE 文件中有多少个节，即节表中的项数，17 个节
- TimeDateStamp 描述了 PE 文件的创建时间，该文件创建时间为 2020/05/15 14:53:41
- PointerToSymbolTable 指出了 COFF 文件符号表在文件中的偏移。
- NumberOfSymbols 表明出了符号表的数量。
- SizeOfOptionalHeader 标明了紧随其后的 PE 可选头的总大小。
- Characteristics 结构体标出了可执行文件的各种属性，其结构如下图：



struct FILE_CHARACTERISTICS Characteristics		96h	2h
WORD IMAGE_FILE_RELOCS_STRIPPED : 1	1	96h	2h
WORD IMAGE_FILE_EXECUTABLE_IMAGE : 1	1	96h	2h
WORD IMAGE_FILE_LINE_NUMS_STRIPPED : 1	1	96h	2h
WORD IMAGE_FILE_LOCAL_SYMS_STRIPPED : 1	0	96h	2h
WORD IMAGE_FILE_AGGRESIVE_WS_TRIM : 1	0	96h	2h
WORD IMAGE_FILE_LARGE_ADDRESS_AWARE : 1	1	96h	2h
WORD IMAGE_FILE_BYTES_REVERSED_LO : 1	0	96h	2h
WORD IMAGE_FILE_32BIT_MACHINE : 1	0	96h	2h
WORD IMAGE_FILE_DEBUG_STRIPPED : 1	0	96h	2h
WORD IMAGE_FILE_REMOVABLE_RUN_FROM_SWAP : 1	0	96h	2h
WORD IMAGE_FILE_NET_RUN_FROM_SWAP : 1	0	96h	2h
WORD IMAGE_FILE_SYSTEM : 1	0	96h	2h
WORD IMAGE_FILE_DLL : 1	0	96h	2h
WORD IMAGE_FILE_UP_SYSTEM_ONLY : 1	0	96h	2h
WORD IMAGE_FILE_BYTES_REVERSED_HI : 1	0	96h	2h

可见该 EXE 满足多个属性，通过异或连接。

```
#define IMAGE_FILE_RELOCS_STRIPPED           0x0001  // Relocation info stripped from file.
#define IMAGE_FILE_EXECUTABLE_IMAGE          0x0002  // File is executable  (i.e. no unresolved externel references).
#define IMAGE_FILE_LINE_NUMS_STRIPPED        0x0004  // Line nunbers stripped from file.
#define IMAGE_FILE_LOCAL_SYMS_STRIPPED       0x0008  // Local symbols stripped from file.
#define IMAGE_FILE_AGGRESIVE_WS_TRIM         0x0010  // Agressively trim working set
#define IMAGE_FILE_LARGE_ADDRESS_AWARE       0x0020  // App can handle >2gb addresses
#define IMAGE_FILE_BYTES_REVERSED_LO         0x0080  // Bytes of machine word are reversed.
#define IMAGE_FILE_32BIT_MACHINE             0x0100  // 32 bit word machine.
#define IMAGE_FILE_DEBUG_STRIPPED            0x0200  // Debugging info stripped from file in .DBG file
#define IMAGE_FILE_REMOVABLE_RUN_FROM_SWAP   0x0400  // If Image is on removable media, copy and run from the swap fi
#define IMAGE_FILE_NET_RUN_FROM_SWAP         0x0800  // If Image is on Net, copy and run from the swap file.
#define IMAGE_FILE_SYSTEM                    0x1000  // System File.
#define IMAGE_FILE_DLL                       0x2000  // File is a DLL.
#define IMAGE_FILE_UP_SYSTEM_ONLY            0x4000  // File should only be run on a UP machine
#define IMAGE_FILE_BYTES_REVERSED_HI         0x8000  // Bytes of machine word are reversed.
```

（3）PE 可选头：

NT 头的第三部分是 PE 可选头，其结构如下：

```
typedef struct _IMAGE_OPTIONAL_HEADER {
    WORD    Magic;
    BYTE    MajorLinkerVersion;
    BYTE    MinorLinkerVersion;
    DWORD   SizeOfCode;
    DWORD   SizeOfInitializedData;
    DWORD   SizeOfUninitializedData;
    DWORD   AddressOfEntryPoint;
    DWORD   BaseOfCode;
    DWORD   BaseOfData;
    DWORD   ImageBase;
    DWORD   SectionAlignment;
    DWORD   FileAlignment;
    WORD    MajorOperatingSystemVersion;
    WORD    MinorOperatingSystemVersion;
    WORD    MajorImageVersion;
    WORD    MinorImageVersion;
    WORD    MajorSubsystemVersion;
    WORD    MinorSubsystemVersion;
    DWORD   Win32VersionValue;
    DWORD   SizeOfImage;
    DWORD   SizeOfHeaders;
    DWORD   CheckSum;
    WORD    Subsystem;
    WORD    DllCharacteristics;
    DWORD   SizeOfStackReserve;
    DWORD   SizeOfStackCommit;
    DWORD   SizeOfHeapReserve;
    DWORD   SizeOfHeapCommit;
    DWORD   LoaderFlags;
    DWORD   NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

- Magic 表示可选头的类型；上图取值为 PE64，表示该可选头为 64 位可选头
- MajorLinkerVersion 和 MinorLinkerVersion 的值指出了链接器的版本号
- SizeOfCode：代码段的长度，如果有多个代码段，则是代码段长度的总和。上图取值表明示例程序中代码段总长度为 7680
- SizeOfInitializedData：初始化的数据长度，取值为 7680
- SizeOfUninitializedData：未初始化的数据长度，取值为 3072
- AddressOfEntryPoint：程序入口的相对虚拟地址 RVA，对于该 exe 文件来说这个地址可以理解为代码中 main 函数的 RVA。
- BaseOfCode：代码段起始地址的 RVA。

- BaseOfData：数据段起始地址的 RVA。
- ImageBase：PE 文件的映像被加载到内存中，而 ImageBase 则指出了映象建议加载的基地址，如果无法加载到这个地址，系统会为其选择其他地址。
- SectionAlignment：节对齐，PE 中的节被加载到内存时会按照这个域指定的值来对齐，图中该值为 4096，即 4k，1000h
- FileAlignment：取值为 512，节在文件中按此值对齐，该值应当小于等于 SectionAlignment，即 200h
- MajorOperatingSystemVersion、MinorOperatingSystemVersion：指出该文件所需操作系统的版本号
- MajorImageVersion、MinorImageVersion：映象的版本号，该条目的值是由开发者自己指定的，由连接器填写。
- MajorSubsystemVersion、MinorSubsystemVersion：所需子系统版本号。
- Win32VersionValue：保留，取值为 0。
- SizeOfImage：映象的大小，PE 文件加载到内存中空间是连续的，这个值指定占用虚拟空间的大小。
- SizeOfHeaders：所有文件头（包括节表）的大小，这个值是以 FileAlignment 对齐的。
- CheckSum：映象文件的校验和。
- Subsystem：运行该 PE 文件所需的子系统，示例中取值为 WINDOWS_CUI（3），表示映像需要 windows 字符子系统
- SizeOfStackReserve：运行时为每个线程栈保留内存的大小。

- SizeOfStackCommit：运行时每个线程栈初始占用内存大小。
- SizeOfHeapReserve：运行时为进程堆保留内存大小。
- SizeOfHeapCommit：运行时进程堆初始占用内存大小。
- LoaderFlags：保留，必须为 0。
- NumberOfRvaAndSizes：数据目录的项数，
- DataDirectory：数据目录，这是一个数组，数组中的每一项对应一个特定的数据结构，包括导入表、导出表等等，其内容如下：

```
#define IMAGE_DIRECTORY_ENTRY_EXPORT          0   // Export Directory
#define IMAGE_DIRECTORY_ENTRY_IMPORT          1   // Import Directory
#define IMAGE_DIRECTORY_ENTRY_RESOURCE        2   // Resource Directory
#define IMAGE_DIRECTORY_ENTRY_EXCEPTION       3   // Exception Directory
#define IMAGE_DIRECTORY_ENTRY_SECURITY        4   // Security Directory
#define IMAGE_DIRECTORY_ENTRY_BASERELOC       5   // Base Relocation Table
#define IMAGE_DIRECTORY_ENTRY_DEBUG           6   // Debug Directory
//      IMAGE_DIRECTORY_ENTRY_COPYRIGHT       7   // (X86 usage)
#define IMAGE_DIRECTORY_ENTRY_ARCHITECTURE    7   // Architecture Specific Data
#define IMAGE_DIRECTORY_ENTRY_GLOBALPTR       8   // RVA of GP
#define IMAGE_DIRECTORY_ENTRY_TLS             9   // TLS Directory
#define IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG    10   // Load Configuration Directory
#define IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT   11   // Bound Import Directory in headers
#define IMAGE_DIRECTORY_ENTRY_IAT            12   // Import Address Table
#define IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT   13   // Delay Load Import Descriptors
#define IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR 14   // COM Runtime descriptor
```

其中每一项的定义如下：

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD   VirtualAddress;
    DWORD   Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

VirtualAddress 定义了该项的相对虚拟地址

Size 定义了该项的大小

在这些成员中比较重要的就是前两个，导出表和导入表

（1）PE 导出表：

Windows 在加载一个程序后会在内存中为该程序开辟一个单独的虚拟地址空间，程序用到的函数会加载到其地址空间运行。而有一些函数会有很多程序都能用到，因此采用将一些函数封装成动态链接库，程序在需要时加载动态链接库的方式可

以大大节约内存空间。

导出表是用来记载动态链接库的一些导出信息的结构，其主要成分是一个表格，内含函数名称、输出序数等等。结构体定义如下：

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD   Characteristics;
    DWORD   TimeDateStamp;
    WORD    MajorVersion;
    WORD    MinorVersion;
    DWORD   Name;
    DWORD   Base;
    DWORD   NumberOfFunctions;
    DWORD   NumberOfNames;
    DWORD   AddressOfFunctions;     // RVA from base of image
    DWORD   AddressOfNames;         // RVA from base of image
    DWORD   AddressOfNameOrdinals;  // RVA from base of image
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

其中主要的一些有意义字段如下：

- TimeDateStamp 记录导出表生成的时间戳，由连接器生成。
- Name：模块的名字，指向一个定义模块名称的字符串
- Base：导出函数序号的起始值，按序号导出函数的序号值从 Base 开始递增。
- NumberOfFunctions：文件中所有导出函数的总数
- NumberOfNames：可以通过函数名的方式导出的函数的总数。
- AddressOfFunctions：一个 RVA，指向一个 DWORD 数组，数组中的每一项是一个导出函数的 RVA，顺序与导出序号相同。
- AddressOfNames：一个 RVA，依然指向一个 DWORD 数组，数组中的每一项仍然是一个 RVA，指向函数名字的字符串。
- AddressOfNameOrdinals：一个 RVA，还是指向一个 WORD 数组，数组中的每一项与 AddressOfNames 中的每一项对应，表示该名字的函数在 AddressOfFunctions 中的序号。

通过上面导出表的结构可以看出，函数导出的方式有按名字导出和按序号导出两种，每种导出方式在导出表中的描述方式也是不同的。

导出表一般存在于.dll 文件中，而.exe 文件中一般不存在导出表。

但是拥有导入表。

（2）导入表

IMAGE_DIRECTORY_ENTRY_IMPORT，即导入表

struct IMAGE_DATA_DIRECTORY Import		110h	8h
DWORD VirtualAddress	8000h	110h	4h
DWORD Size	2020	114h	4h

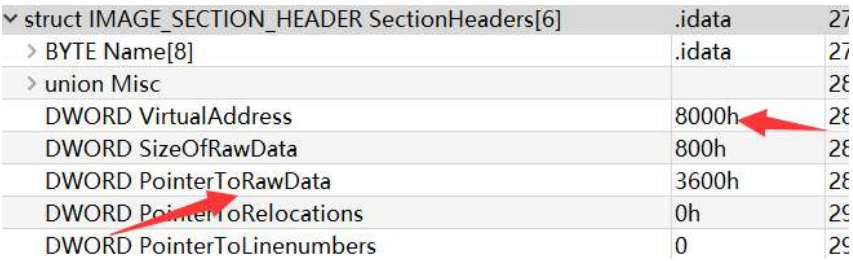
在 PE 文件加载时，会根据这个表里的内容加载依赖的 DLL，并填充所需函数的地址。

- IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT：绑定导入表，在第一种导入表导入地址的修正是在 PE 加载时完成，如果一个 PE 文件导入的 DLL 或者函数多那么加载起来就会略显的慢一些，所以出现了绑定导入，在加载以前就修正了导入表，这样就会快一些。
- IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT：延迟导入表，一个 PE 文件也许提供了很多功能，也导入了很多其他 DLL，但是并非每次加载都会用到它提供的所有功能，也不一定会用到它需要导入的所有 DLL，因此延迟导入就出现了，只有在一个 PE 文件真正用到需要的 DLL，这个 DLL 才会被加载，甚至于只有真正使用某个导入函数，这个函数地址才会被修正。
- IMAGE_DIRECTORY_ENTRY_IAT：导入函数地址表，前面的三个表其实是导入函数的描述，真正的函数地址是被填充在导入函数地址表中的。

这些结构的成员均为 VA 和 size。

将 VA 转化为文件偏移地址可以得到相应的导入表中的信息。

VA=8000h，对照节表，



struct IMAGE_SECTION_HEADER SectionHeaders[6]	.idata	27
BYTE Name[8]	.idata	27
union Misc		28
DWORD VirtualAddress	8000h	28
DWORD SizeOfRawData	800h	28
DWORD PointerToRawData	3600h	28
DWORD PointerToRelocations	0h	29
DWORD PointerToLinenumbers	0	29

可以得到偏移为 0，

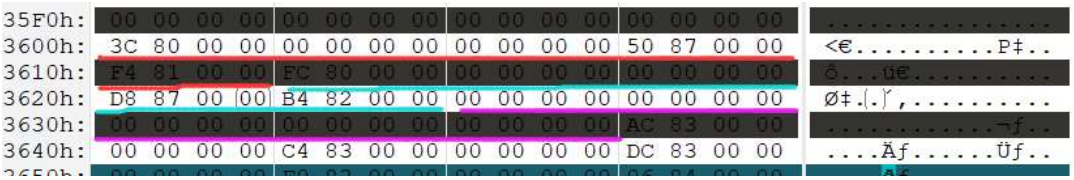
文件偏移地址为 3600h+0 = 3600h，可以找到导入表对应的数组，简称 IID。在这个数组中并没有指出有多少项，但是他最后以一个全为 NULL (0) 的 IID 结尾。

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD   Characteristics;            // 0 for terminating null import descriptor
        DWORD   OriginalFirstThunk;         // RVA to original unbound IAT (PIMAGE_THUNK_DATA)
    } DUMMYUNIONNAME;
    DWORD   TimeDateStamp;                  // 0 if not bound,
                                            // -1 if bound, and real date\time stamp
                                            //     in IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT (new BIND)
                                            // O.W. date/time stamp of DLL bound to (Old BIND)

    DWORD   ForwarderChain;                 // -1 if no forwarders
    DWORD   Name;
    DWORD   FirstThunk;                     // RVA to IAT (if bound this IAT has actual addresses)
} IMAGE_IMPORT_DESCRIPTOR;

typedef IMAGE_IMPORT_DESCRIPTOR UNALIGNED *PIMAGE_IMPORT_DESCRIPTOR;
```

结构的大小为 5*4=20 字节。



```
35F0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
3600h: 3C 80 00 00 00 00 00 00 00 00 00 00 50 87 00 00 <€..........P‡..
3610h: F4 81 00 00 FC 80 00 00 00 00 00 00 00 00 00 00 ô...ü€..........
3620h: D8 87 00 00 B4 82 00 00 00 00 00 00 00 00 00 00 Ø‡..´‚..........
3630h: 00 00 00 00 00 00 00 00 00 00 00 00 AC 83 00 00 ............¬ƒ..
3640h: 00 00 00 00 C4 83 00 00 00 00 00 00 DC 83 00 00 ....Äƒ......Üƒ..
```

可以看到第三组为 0，所以只有两个_IMAGE_IMPORT_DESCRIPTOR。每个导入的 DLL 都会对应一个，也就是说示例 EXE 存在两个导入 DLL。

- Characteristics 和 OriginalFirstThunk：一个联合体，如果是数组的最后一项 Characteristics 为 0，否则 OriginalFirstThunk 保存一个 RVA，指向一个 IMAGE_THUNK_DATA 的数组，这个数组中的每一项表示一个导入函数。
- TimeDateStamp：映象绑定前，这个值是 0，绑定后是导入模块的时间戳。

- ForwarderChain：转发链，如果没有转发器，这个值是-1。
- Name：一个 RVA，指向导入模块的名字，所以一个 IMAGE_IMPORT_DESCRIPTOR 描述一个导入的 DLL。
- FirstThunk：也是一个 RVA，也指向一个 IMAGE_THUNK_DATA 数组。

其中 OriginalFirstThunk 指向一个 IMAGE_THUNK_DATA 的数组 INT，FirstThunk 指向一个 IMAGE_THUNK_DATA 数组 IAT。（都是以 0 标志结束）

```
typedef struct _IMAGE_THUNK_DATA32 {
    union {
        DWORD ForwarderString;      // PBYTE
        DWORD Function;             // PDWORD
        DWORD Ordinal;
        DWORD AddressOfData;        // PIMAGE_IMPORT_BY_NAME
    } u1;
} IMAGE_THUNK_DATA32;
typedef IMAGE_THUNK_DATA32 * PIMAGE_THUNK_DATA32;
```

4 个字节大小

- ForwarderString 是转发用的
- Function 表示函数地址，如果函数是按序号导入 Ordinal 就有用了，
- AddressOfData 指向名字信息，如果函数按名字导入。

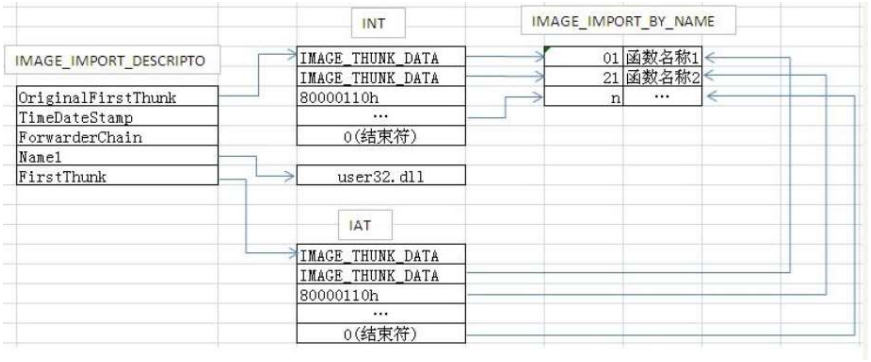
可以看出这个结构体就是一个大的 union，但是在不同时刻代表不同的意义那到底应该是名字还是序号，该如何区分呢？可以通过 Ordinal 判断，如果 Ordinal 的最高位是 1，就是按序号导入的，这时候，低 16 位就是导入序号，如果最高位是 0，则 AddressOfData 是一个 RVA，指向一个 IMAGE_IMPORT_BY_NAME 结构，用来保存名字信息，由于 Ordinal 和 AddressOfData 实际上是同一个内存空间，所以 AddressOfData 其实只有低 31 位可以表示 RVA，但是一个 PE 文件不可能超过 2G，所以最高位永远为 0，这样设计很合理的利用了空间。实际编写代码的时候微软提供两个宏定义处理序号导入：IMAGE_SNAP_BY_ORDINAL 判断是

否按序号导入，IMAGE_ORDINAL 用来获取导入序号。

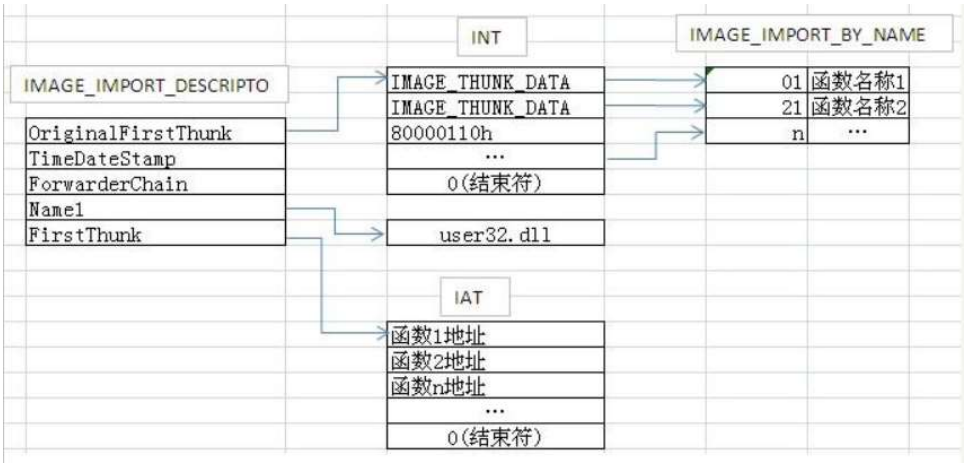
FirstThunk 和 OriginalFirstThunk 都指向 IMAGE_THUNK_DATA 数组，但是却是不同的。OriginalFirstThunk 指向的是单独的一项，而且不能被修改，成为 INT。FirstThunk 指向的事实上是由 PE 加载器重写的。

在 PE 文件加载以前或者说在导入表未处理以前，FirstThunk 所指向的数组与 OriginalFirstThunk 中的数组虽不是同一个，但是内容却是相同的，都包含了导入信息，而在加载之后，FirstThunk 中的 Function 开始生效，他指向实际的函数地址，因为 FirstThunk 实际上指向 IAT 中的一个位置，IAT 就充当了 IMAGE_THUNK_DATA 数组，加载完成后，这些 IAT 项就变成了实际的函数地址，即 Function 的意义。

PE 加载前：

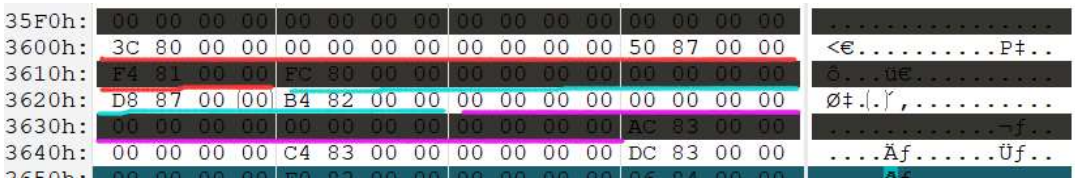


PE 加载后：

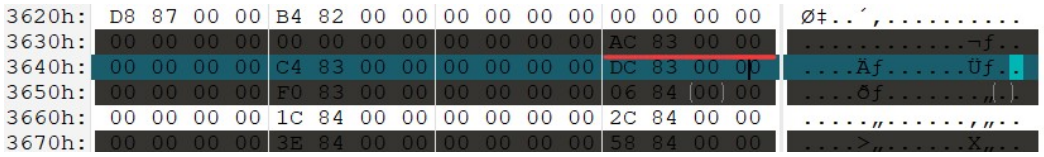


PE 加载器首先搜索 OriginalFirstThunk，找到之后加载程序迭代搜索数组中的每个指针，找到每个 IMAGE_IMPORT_BY_NAME 结构所指向的输入函数的地址，然后加载器用函数真正入口地址来代替由 FirstThunk 数组中的一个入口，因此称为输入地址表 IAT。

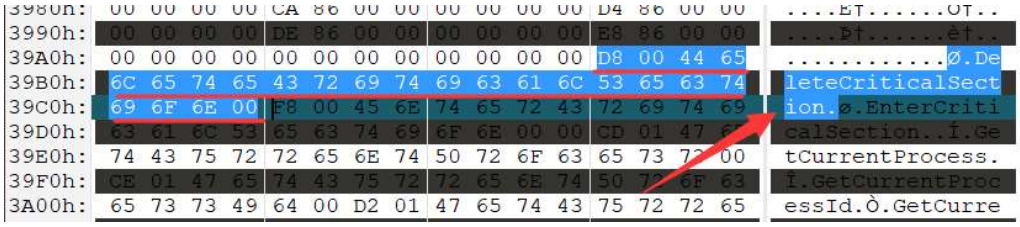
结合示例程序分析:



首先看第一个 OriginalFirstThunk 为 00 00 80 3C，所以偏移为 3C，文件偏移地址为 3600h+3ch = 363ch

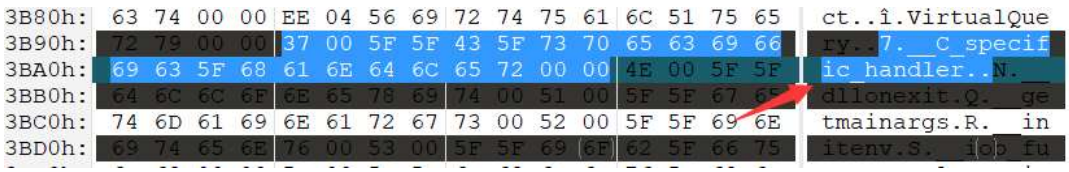


找到了 INT，最高位为 0，所以 RVA=83ACh，文件偏移地址为 39Ach



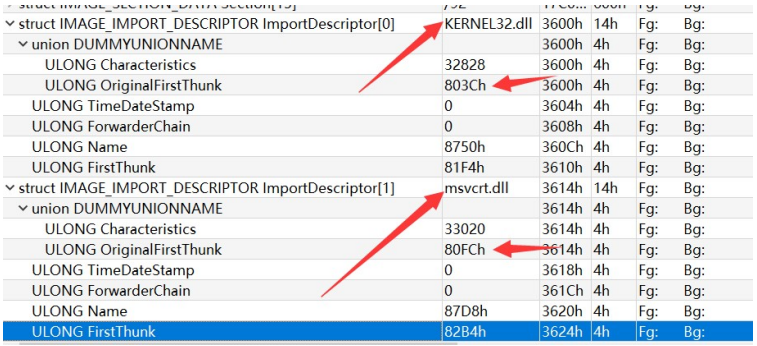
可以得到导入的函数名称

同理可以得到第二个对应文件偏移地址 3B94h



得到第二个名称

与模板中的对照



struct IMAGE_IMPORT_DESCRIPTOR ImportDescriptor[0]	KERNEL32.dll	3600h	14h	Fg:	Bg:
union DUMMYUNIONNAME		3600h	4h	Fg:	Bg:
ULONG Characteristics	32828	3600h	4h	Fg:	Bg:
ULONG OriginalFirstThunk	803Ch	3600h	4h	Fg:	Bg:
ULONG TimeDateStamp	0	3604h	4h	Fg:	Bg:
ULONG ForwarderChain	0	3608h	4h	Fg:	Bg:
ULONG Name	8750h	360Ch	4h	Fg:	Bg:
ULONG FirstThunk	81F4h	3610h	4h	Fg:	Bg:
struct IMAGE_IMPORT_DESCRIPTOR ImportDescriptor[1]	msvcrt.dll	3614h	14h	Fg:	Bg:
union DUMMYUNIONNAME		3614h	4h	Fg:	Bg:
ULONG Characteristics	33020	3614h	4h	Fg:	Bg:
ULONG OriginalFirstThunk	80FCh	3614h	4h	Fg:	Bg:
ULONG TimeDateStamp	0	3618h	4h	Fg:	Bg:
ULONG ForwarderChain	0	361Ch	4h	Fg:	Bg:
ULONG Name	87D8h	3620h	4h	Fg:	Bg:
ULONG FirstThunk	82B4h	3624h	4h	Fg:	Bg:

可以看出计算正确，找到了函数名称。

3、节表

节表由一系列的 IMAGE_SECTION_HEADER 结构排列而成，每个结构用来描述一个节。其排列顺序和节在文件中的排列顺序一致。节表以一个空的 IMAGE_SECTION_HEADER 结构作为结尾，因此节表中 IMAGE_SECTION_HEADER 结构的总数等于节的数量+1。

Name	Value	Start	Size
struct IMAGE_SECTION_HEADER SectionHeaders[17]		188h	2A8h
struct IMAGE_SECTION_HEADER SectionHeaders[0]	.text	188h	28h
struct IMAGE_SECTION_HEADER SectionHeaders[1]	.data	1B0h	28h
struct IMAGE_SECTION_HEADER SectionHeaders[2]	.rdata	1D8h	28h
struct IMAGE_SECTION_HEADER SectionHeaders[3]	.pdata	200h	28h
struct IMAGE_SECTION_HEADER SectionHeaders[4]	.xdata	228h	28h
struct IMAGE_SECTION_HEADER SectionHeaders[5]	.bss	250h	28h
struct IMAGE_SECTION_HEADER SectionHeaders[6]	.idata	278h	28h
struct IMAGE_SECTION_HEADER SectionHeaders[7]	.CRT	2A0h	28h
struct IMAGE_SECTION_HEADER SectionHeaders[8]	.tls	2C8h	28h
struct IMAGE_SECTION_HEADER SectionHeaders[9]	/4	2F0h	28h
struct IMAGE_SECTION_HEADER SectionHeaders[10]	/19	318h	28h
struct IMAGE_SECTION_HEADER SectionHeaders[11]	/31	340h	28h
struct IMAGE_SECTION_HEADER SectionHeaders[12]	/45	368h	28h
struct IMAGE_SECTION_HEADER SectionHeaders[13]	/57	390h	28h
struct IMAGE_SECTION_HEADER SectionHeaders[14]	/70	3B8h	28h
struct IMAGE_SECTION_HEADER SectionHeaders[15]	/81	3E0h	28h
struct IMAGE_SECTION_HEADER SectionHeaders[16]	/92	408h	28h

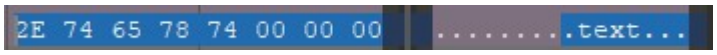
示例程序有 16 个节，故节表中有 17 个 IMAGE_SECTION_HEADER 结构。这个数字保存在 Nt 头的 NumberOfSections 中。

struct IMAGE_NT_HEADERS NtHeader		80h	108h	Fg:	Bg:
DWORD Signature	4550h	80h	4h	Fg:	Bg: IMAGE_NT_SIGNATURE = 0x00004550
struct IMAGE_FILE_HEADER FileHeader		84h	14h	Fg:	Bg:
enum IMAGE_MACHINE Machine	AMD64 (8664h)	84h	2h	Fg:	Bg: WORD
WORD NumberOfSections	17	86h	2h	Fg:	Bg: Section num

所有 IMAGE_SECTION_HEADER 结构具有相同的结构，以示例程序节表中第一个 IMAGE_SECTION_HEADER 结构为例，这个结构描述了程序中的.text 节：

struct IMAGE_SECTION_HEADER SectionHeaders[0]	.text
BYTE Name[8]	.text
union Misc	
DWORD PhysicalAddress	7648
DWORD VirtualSize	7648
DWORD VirtualAddress	1000h
DWORD SizeOfRawData	1E00h
DWORD PointerToRawData	600h
DWORD PointerToRelocations	0h
DWORD PointerToLinenumbers	0
WORD NumberOfRelocations	0
WORD NumberOfLinenumbers	0
struct SECTION_CHARACTERISTICS Characteristics	

BYTE Name：8 个字节，记录节的名称的 ASC II 码。



VirtualSize：节在没有进行对齐处理前的实际大小。例如该节的实际大小为 7648 字节。

VirtualAddress：该节装载到内存中的 RVA 地址，这个地址是按照内存页来对齐的。

SizeOfRawData：该节在磁盘中所占的大小。数值等于节的实际大小按照 Nt 头中记录的 FileAlignment 的值对齐后的大小。例如，该文件的 FileAlignment 的值为 512 字节，

Name	Value	Start	Size	Color
DWORD FileAlignment	512	BCh	4h	Fg: Bg:

该节的实际大小为 7648 字节，故该节的 SizeOfRawData 值=7648/512 的值向上取整再乘以 512，得 15*512=7680，即 16 进制下的 1E00h。

PointerToRawData：节在磁盘文件中所处的位置，数值等于从文件头开始算起的偏移量。

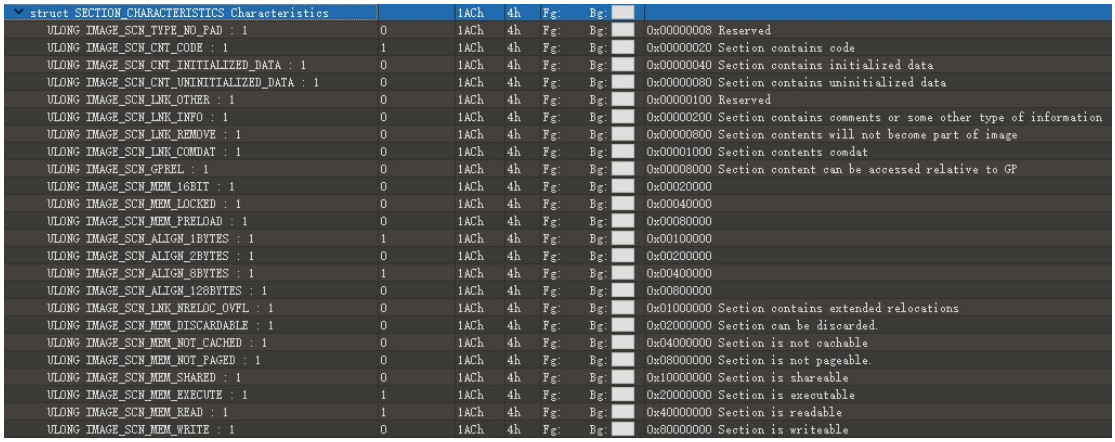
PointerToRelocations：在 obj 文件中使用，重定位的偏移。

PointerToLinenumbers：行号表的偏移(供调试使用地)。

NumberOfRelocations：在 obj 文件中使用，重定位项数目。

NumberOfLinenumbers：行号表中行号的数目。

Characteristics：



struct SECTION_CHARACTERISTICS Characteristics		1ACh	4h	Fg:	Bg:	
ULONG IMAGE_SCN_TYPE_NO_PAD : 1	0	1ACh	4h	Fg:	Bg:	0x00000008 Reserved
ULONG IMAGE_SCN_CNT_CODE : 1	1	1ACh	4h	Fg:	Bg:	0x00000020 Section contains code
ULONG IMAGE_SCN_CNT_INITIALIZED_DATA : 1	0	1ACh	4h	Fg:	Bg:	0x00000040 Section contains initialized data
ULONG IMAGE_SCN_CNT_UNINITIALIZED_DATA : 1	0	1ACh	4h	Fg:	Bg:	0x00000080 Section contains uninitialized data
ULONG IMAGE_SCN_LNK_OTHER : 1	0	1ACh	4h	Fg:	Bg:	0x00000100 Reserved
ULONG IMAGE_SCN_LNK_INFO : 1	0	1ACh	4h	Fg:	Bg:	0x00000200 Section contains comments or some other type of information
ULONG IMAGE_SCN_LNK_REMOVE : 1	0	1ACh	4h	Fg:	Bg:	0x00000800 Section contents will not become part of image
ULONG IMAGE_SCN_LNK_COMDAT : 1	0	1ACh	4h	Fg:	Bg:	0x00001000 Section contents comdat
ULONG IMAGE_SCN_GPREL : 1	0	1ACh	4h	Fg:	Bg:	0x00008000 Section content can be accessed relative to GP
ULONG IMAGE_SCN_MEM_16BIT : 1	0	1ACh	4h	Fg:	Bg:	0x00020000
ULONG IMAGE_SCN_MEM_LOCKED : 1	0	1ACh	4h	Fg:	Bg:	0x00040000
ULONG IMAGE_SCN_MEM_PRELOAD : 1	0	1ACh	4h	Fg:	Bg:	0x00080000
ULONG IMAGE_SCN_ALIGN_1BYTES : 1	1	1ACh	4h	Fg:	Bg:	0x00100000
ULONG IMAGE_SCN_ALIGN_2BYTES : 1	0	1ACh	4h	Fg:	Bg:	0x00200000
ULONG IMAGE_SCN_ALIGN_8BYTES : 1	1	1ACh	4h	Fg:	Bg:	0x00400000
ULONG IMAGE_SCN_ALIGN_128BYTES : 1	0	1ACh	4h	Fg:	Bg:	0x00800000
ULONG IMAGE_SCN_LNK_NRELOC_OVFL : 1	0	1ACh	4h	Fg:	Bg:	0x01000000 Section contains extended relocations
ULONG IMAGE_SCN_MEM_DISCARDABLE : 1	0	1ACh	4h	Fg:	Bg:	0x02000000 Section can be discarded.
ULONG IMAGE_SCN_MEM_NOT_CACHED : 1	0	1ACh	4h	Fg:	Bg:	0x04000000 Section is not cachable
ULONG IMAGE_SCN_MEM_NOT_PAGED : 1	0	1ACh	4h	Fg:	Bg:	0x08000000 Section is not pageable.
ULONG IMAGE_SCN_MEM_SHARED : 1	0	1ACh	4h	Fg:	Bg:	0x10000000 Section is shareable
ULONG IMAGE_SCN_MEM_EXECUTE : 1	1	1ACh	4h	Fg:	Bg:	0x20000000 Section is executable
ULONG IMAGE_SCN_MEM_READ : 1	1	1ACh	4h	Fg:	Bg:	0x40000000 Section is readable
ULONG IMAGE_SCN_MEM_WRITE : 1	0	1ACh	4h	Fg:	Bg:	0x80000000 Section is writeable

节的属性，每一位存储一项，包括该节是否包含代码、是否包含数据、是否可读、是否可写、是否可执行等。

4、具体的节

.idata 节

导入段。包含程序需要的所有 DLL 文件信息。

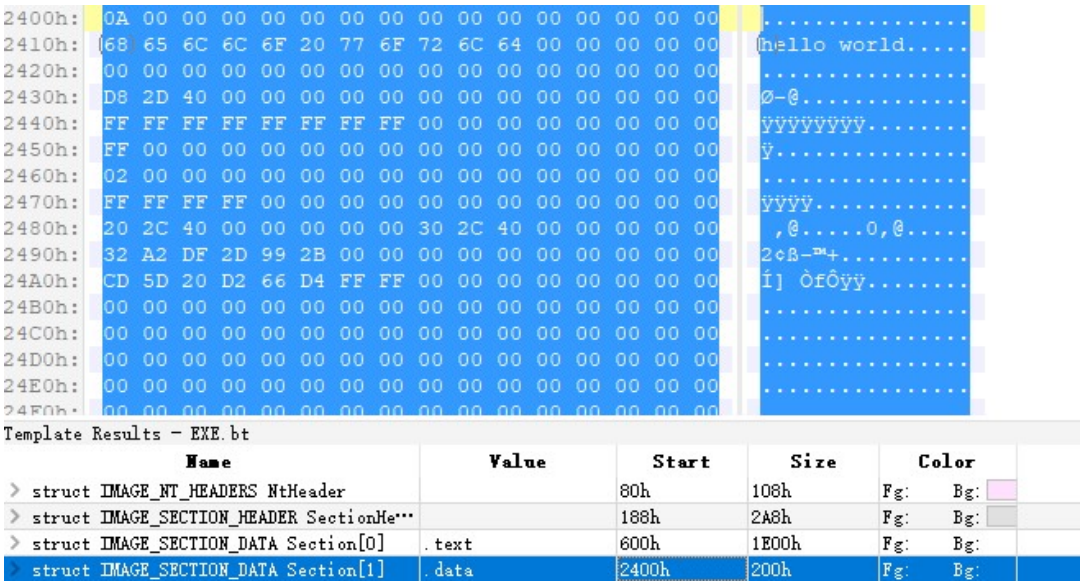
.data 节

数据段（data segment）通常是指用来存放程序中已初始化的全局变量的一块内存区域。数据段属于静态内存分配。

根据示例程序中

```
char str1[20] = "hello world";
char str2[20];
```

通过 010editor，我们可以在.data 段找到如下信息

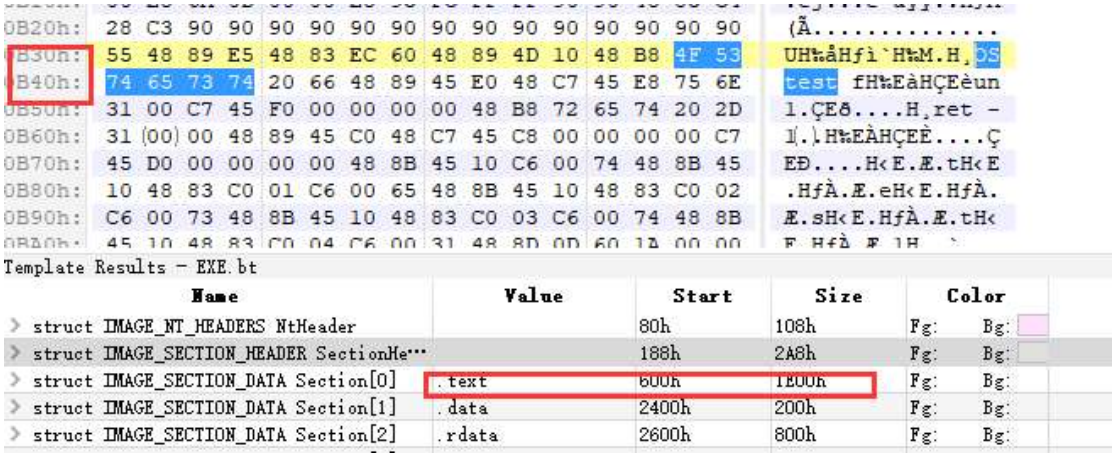


.text/CODE 节

代码段（text segment）通常是指用来存放程序执行代码的一块内存区域。这部分区域的大小在程序运行前就已经确定，并且内存区域属于只读。在代码段中，也有可能包含一些只读的常数变量，例如字符串常量等。需要注意的是，borland 这里叫做 code，而不是 text 。

```
char tmp1[20] = "OStest fun1", t
```

```
char tmp1[20] = "OStest fun2",
```



.rdata 节

.rdata 节一般最少两种用途。通常.data 节的开头是程序的 Debug 目录，该目录仅存在于.EXE 文件当中。Debug 目录是一个 IMAGE_DEBUG_DIRECTORY 结构体

的数组，这些结构体存储了文件各种不同的 Debug 信息。

同时.rdata 节有一部分是用来描述字符串信息。如果程序的 DEF 文件中存在特别指定的 DESCRIPTION entry，那么这个 DESCRIPTION entry 就会出现在.rdata 节中。

.CRT 节

.CRT 节是另一个 Microsoft C/C++ run-time libraries 的初始化数据段。

.tls 节

.tls=thread local storage，当用户使用 compiler directive _ _declspec 线程时，定义的数据会出现在当前段。当处理.tls 节时，内存管理器会建立一个页表来保证进程切换线程时，一组新的物理存储页会映射到.tls 节的地址空间。

四．关键技术和难点分析

关键点：RVA 地址

RVA 地址（Relative Virtual Address）是相对虚拟地址的缩写，PE 文件中各种数据结构中涉及地址的字段大部分是以 RVA 地址表示的。RVA 表示一个 PE 文件被装载到内存后，某个数据位置相对于文件头的偏移量。例如，Windows 装载器将一个 PE 文件装入 00400000h 处的内存中，某个数据的 RVA 为 1000h，则该数据在内存中的实际地址为 00400000h+1000h=00401000h。

任何 RVA 都要经过到文件偏移的换算才能用于定位和访问文件中的数据，换算需要三步：

1. 扫描节表，根据每个 IMAGE_SECTION_HEADER 结构中的 VirtualAddress 字段（描述了该节的起始地址）和 SizeOfRawData 字段（描述了该节的实际大小）可以知道每个节的起始和结束地址，依次判断所要查找的目标 RVA 是否在某个

节内；

2. 确定了目标所在的节后，用目标 RVA 减去目标所在的节的 VirtualAddress，得到目标 RVA 相对于节的起始地址的偏移量；

3. 用该偏移量加上目标所在的节的 PointerToRawData（描述了节在文件中的偏移地址）,得到目标 RVA 在文件中的偏移地址。

五. 运行和测试过程（或结论或总结）

PE 文件是 Windows 操作系统上的程序文件，可移植的可执行文件。一个操作系统的可执行文件在很多方面是这个操作系统的一面镜子。通过对 PE 文件结构，内部原理，作用的研究，让我们了解了 Windows 操作系统下，可执行文件是如何加载的，同时对操作系统的理解也更加深入了。同时也对软件安全，软件解密有了一定的了解。通过大作业的学习，也加深了对《操作系统原理》这门课程的理解。

六. 参考网址

https://baike.baidu.com/item/pe%E6%96%87%E4%BB%B6/6488140?fr=aladdin

https://blog.csdn.net/freeking101/article/details/102752048?utm_medium=distribute.pc_relevant.none-task-blog-baidujs-2

https://blog.csdn.net/adam001521/article/details/84658708

https://www.cnblogs.com/guanlaiy/archive/2012/04/28/2474504.html

https://docs.microsoft.com/en-us/previous-versions/ms809762(v=msdn.10)?redirectedfrom=MSDN