第05讲 PE病毒实现原理

系统安全与可信计算研究所 陈泽茂

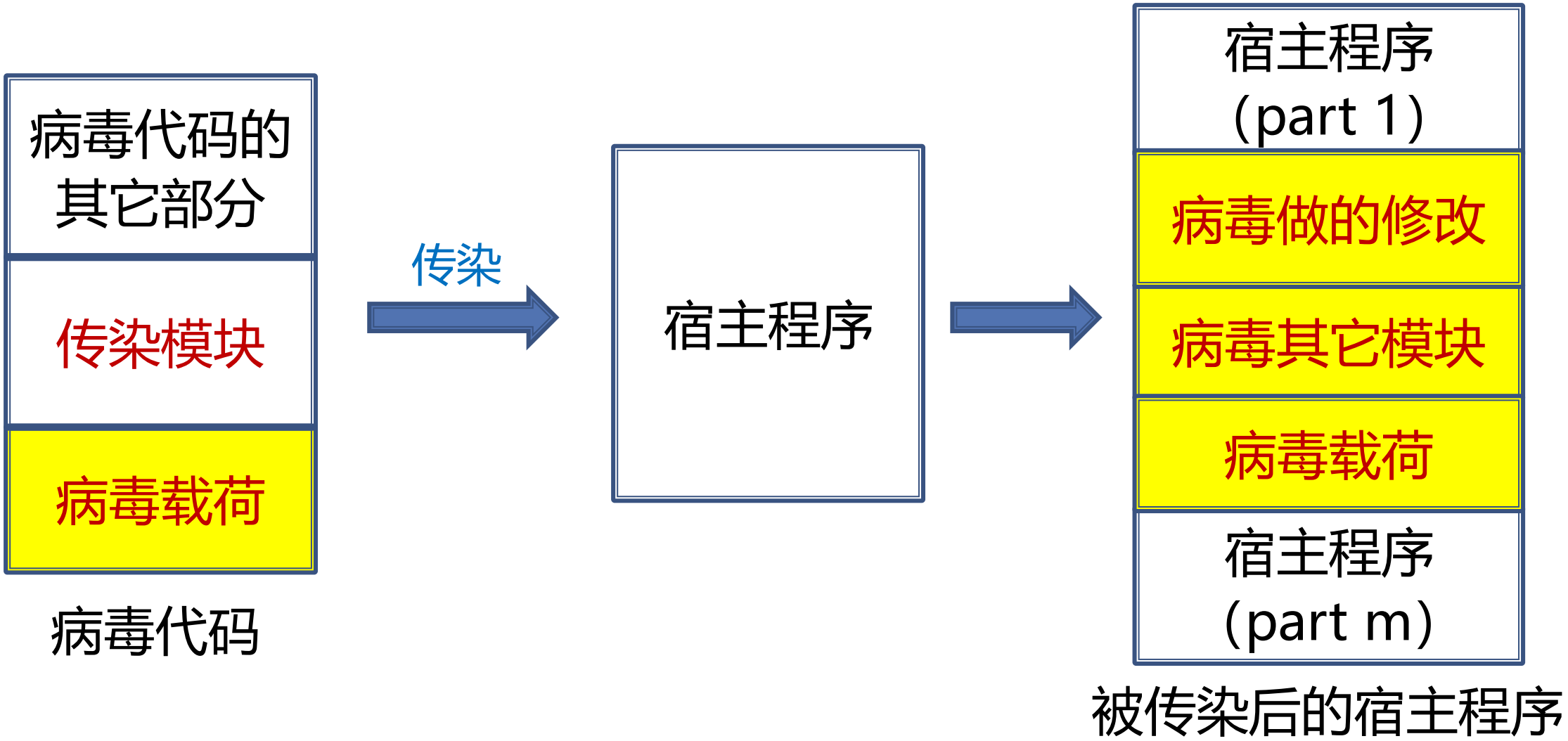
chenzemao@whu.edu.cn

目 录

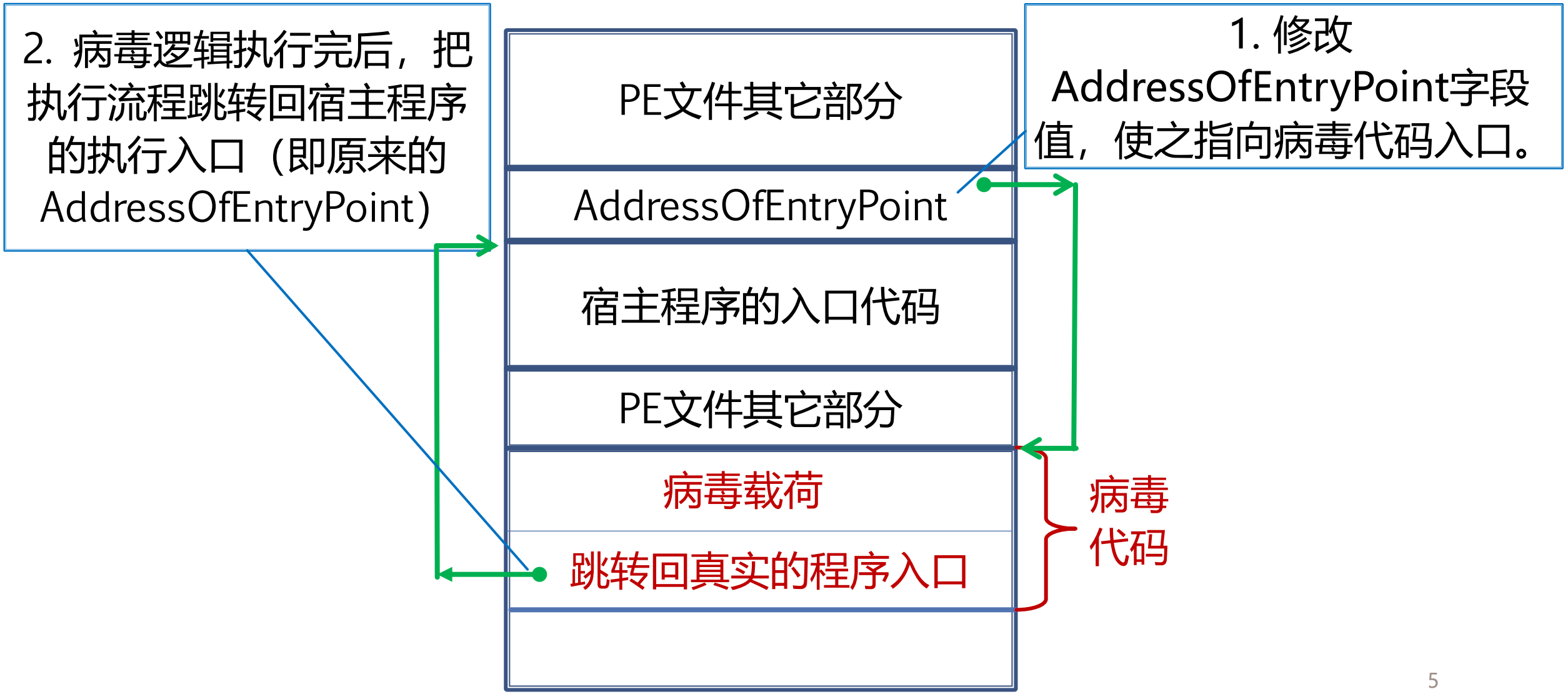
- PE病毒逻辑中的几个问题
- 传染部位的选择
- 宿主程序执行流程的接管与恢复
- 病毒代码的自我重定位
- 系统API函数地址的获取
- 搜索传染目标

一、PE病毒逻辑的几个问题

1.1 文件型病毒基本传染逻辑



1.2 PE病毒基本传染逻辑



1.3 基本PE病毒要解决的几个问题

1. 病毒传染逻辑
   - ① 传染部位怎么选择?
   - ② 怎么传染才能不影响宿主程序的正常运行?
2. 宿主程序执行流程的接管与恢复
   - ① 病毒代码怎么接管宿主执行流程?
   - ② 怎么恢复宿主程序的执行流程?

1.3 基本PE病毒要解决的几个问题

3. 确保植入到宿主程序中的病毒代码能够运行

   ① 病毒代码怎么实现重定位?

   ② 病毒代码怎么调用库函数?

4. 传染目标的搜索

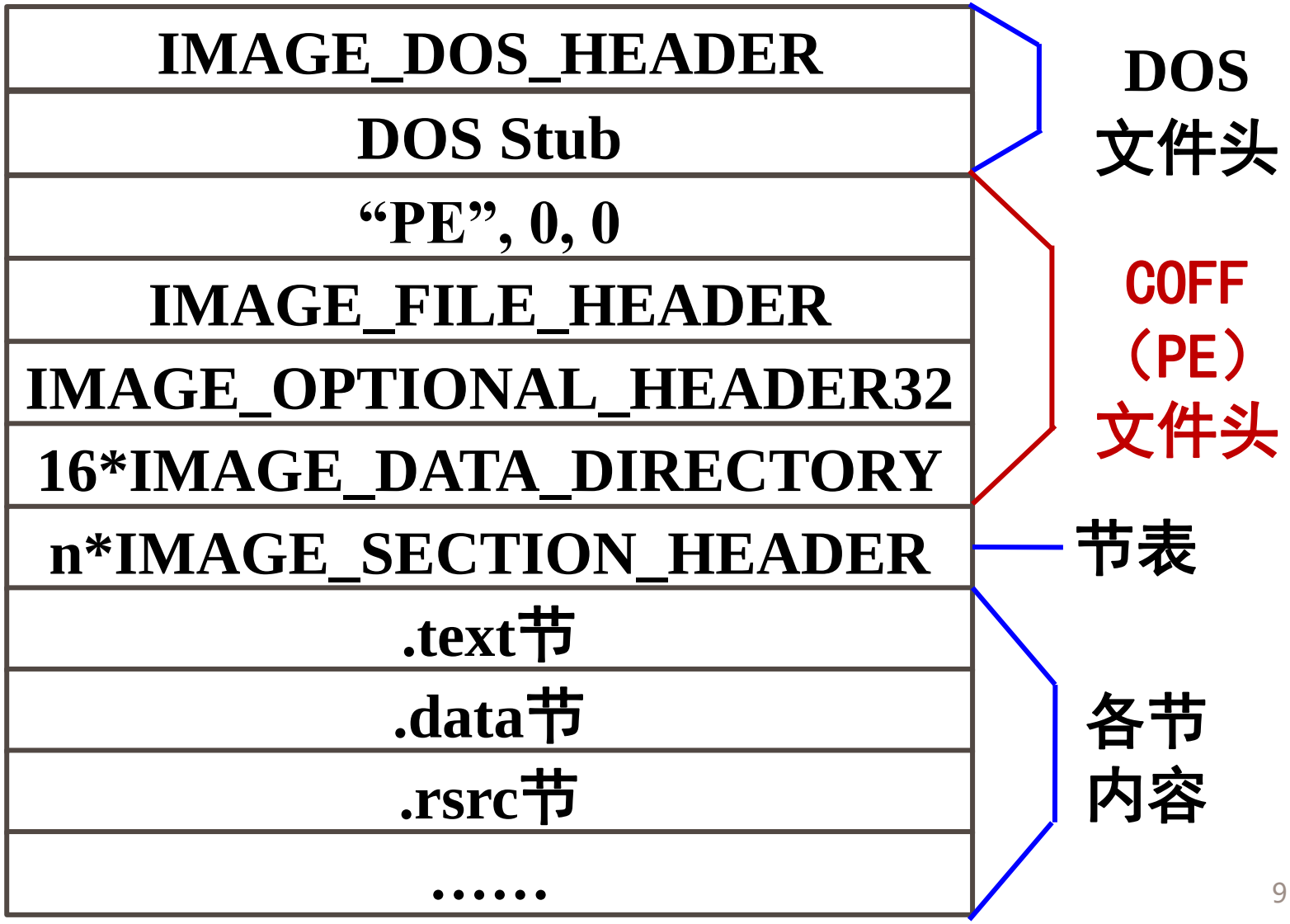
   ① 怎么找到目标系统上可传染的对象?

二、传染部位的选择

2.1 PE文件格式概览

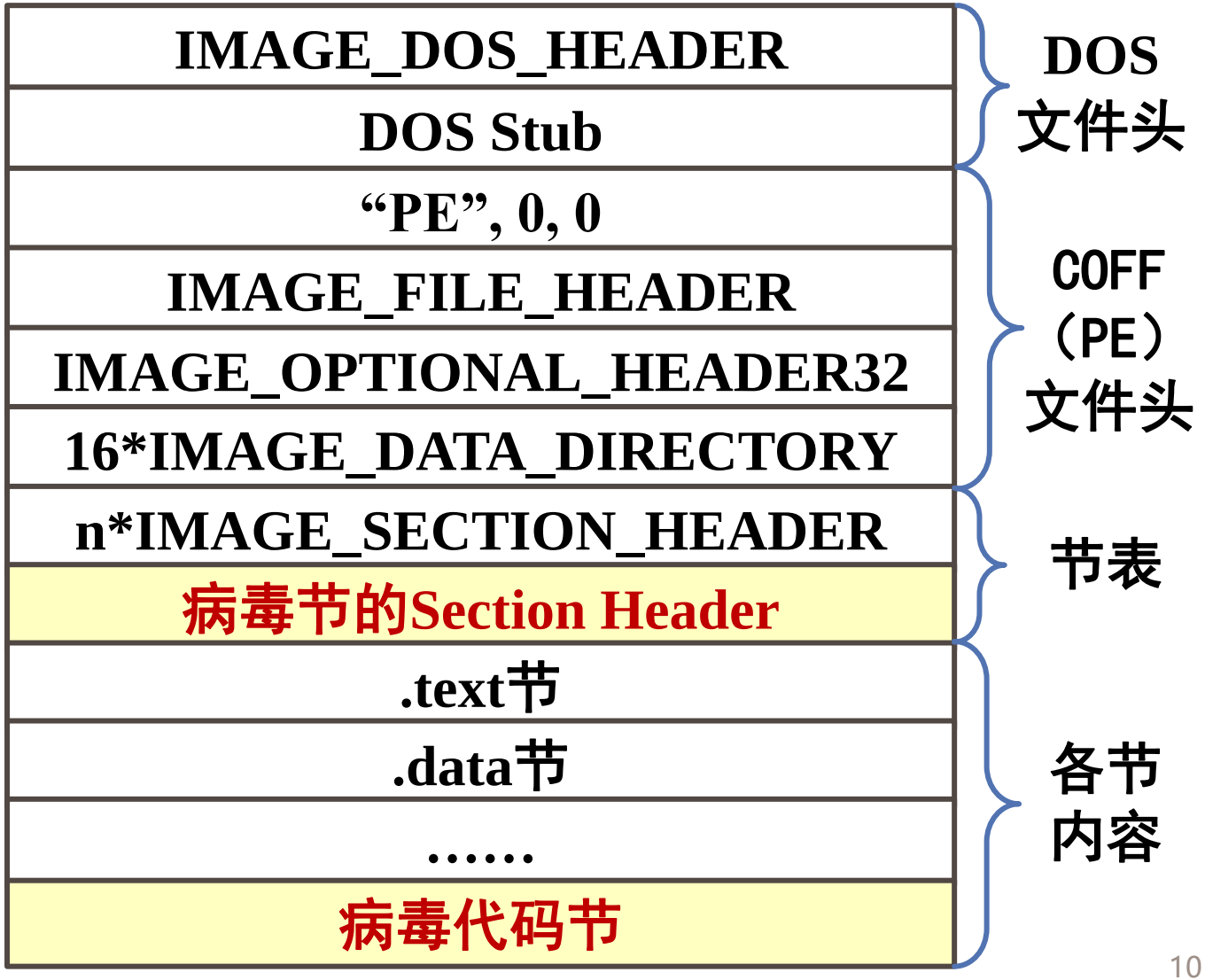
对传染部位的两个要求：

1. 空间够
2. 能运行



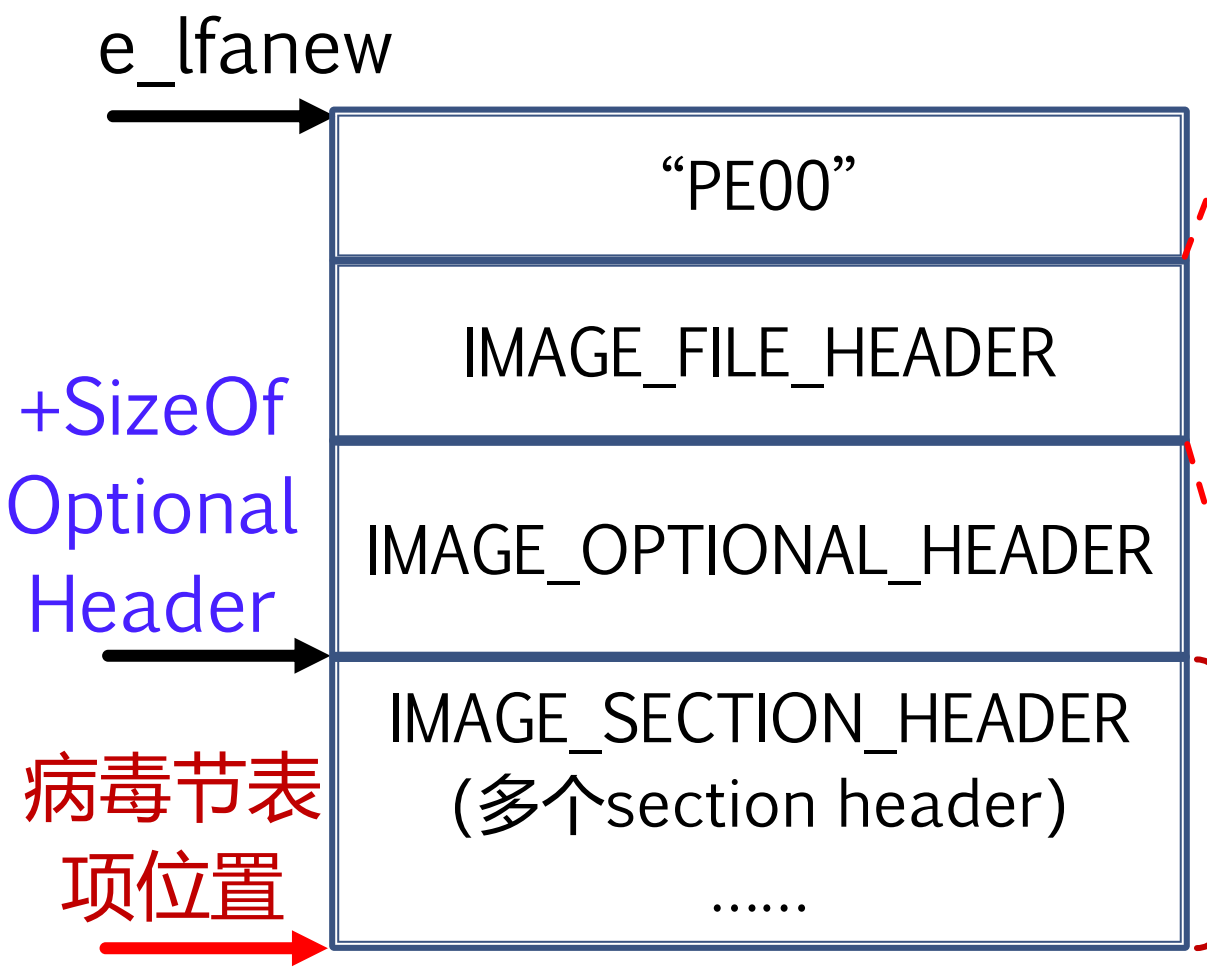
2.2 添加新节

注意：须有足够空间存放新插入的节表项，不能破坏紧邻的section内容。



2.2 添加新节

修改节表，增加一项用以指示病毒节




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

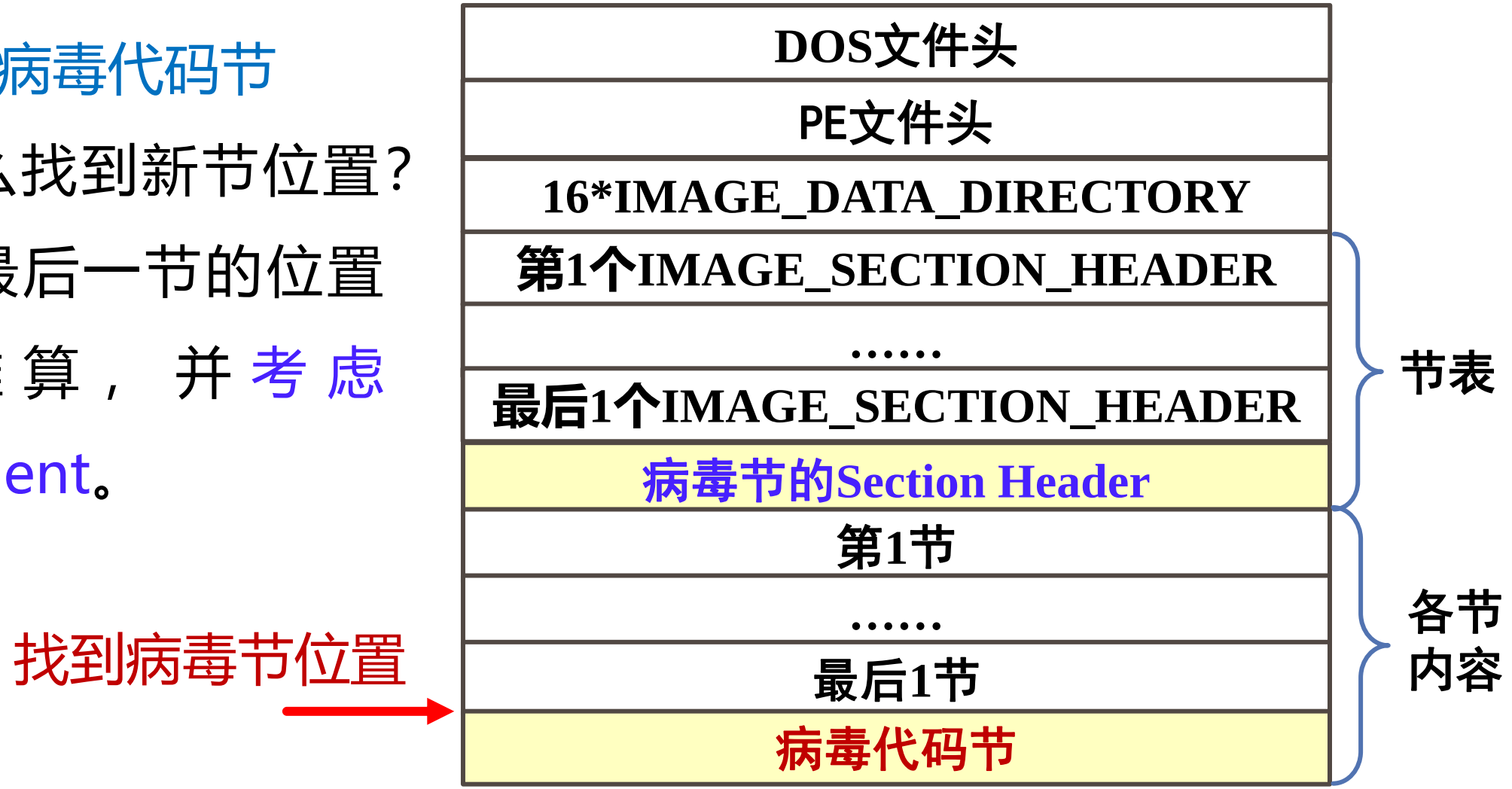
sizeof(IMAGE_SECTION_HEADER) * NumberOfSections

2.2 添加新节

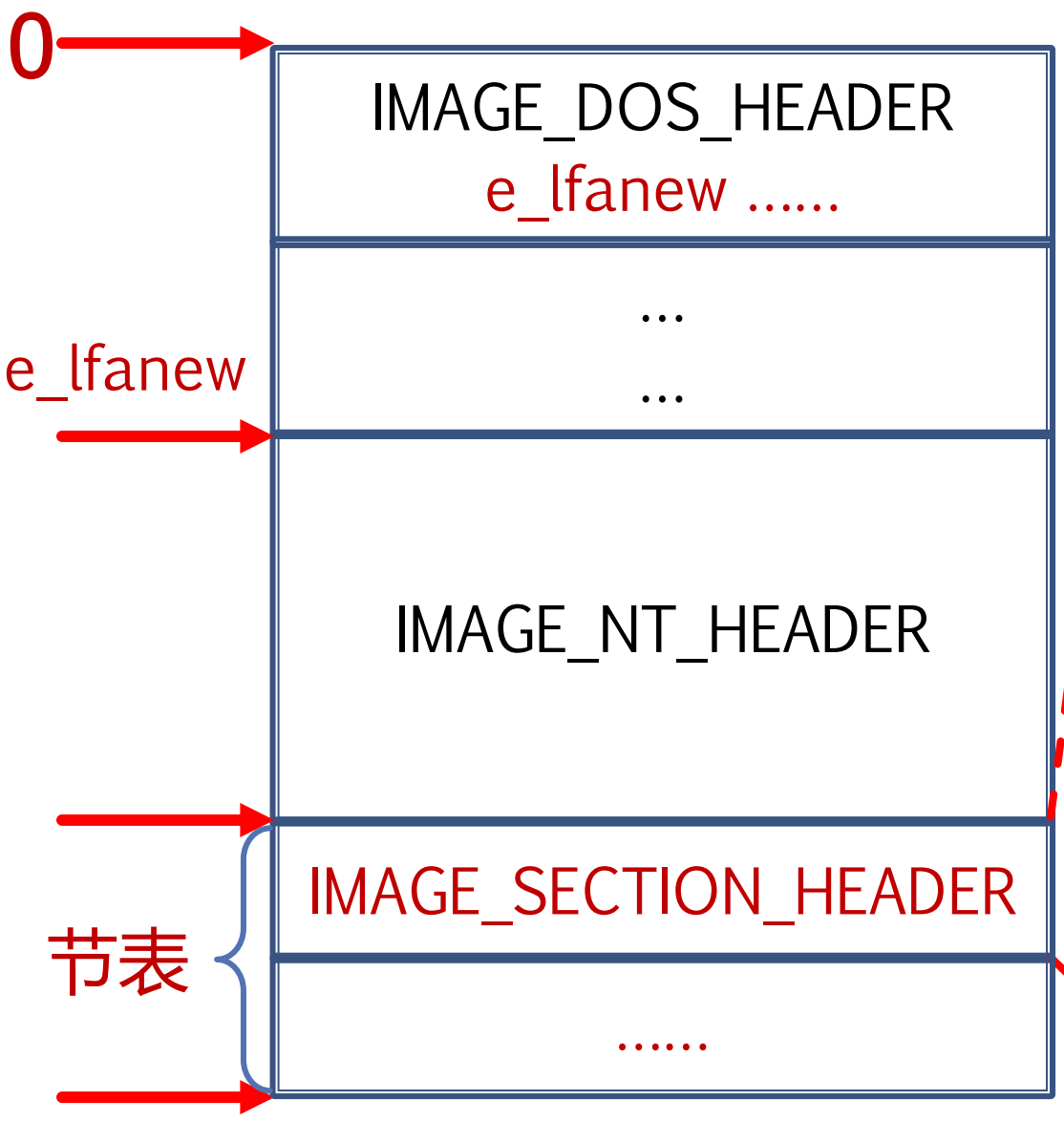
新增一病毒代码节

问题：怎么找到新节位置？

从原有的最后一节的位置和大小推算，并考虑FileAlignment。



IMAGE_SECTION_HEADER




```
typedef struct _IMAGE_SECTION_HEADER {
    BYTE    Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
            DWORD   PhysicalAddress;
            DWORD   VirtualSize;
    } Misc;
    DWORD   VirtualAddress;
    DWORD   SizeOfRawData;
    DWORD   PointerToRawData;
    DWORD   PointerToRelocations;
    DWORD   PointerToLinenumbers;
    WORD    NumberOfRelocations;
    WORD    NumberOfLinenumbers;
    DWORD   Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

DWORD	SizeOfRawData	The size (in bytes) of data stored for the section in the executable or OBJ. For executables, this must be a multiple of the file alignment given in the PE header. If set to 0, the section is uninitialized data.

- `DWORD SizeOfRawData`

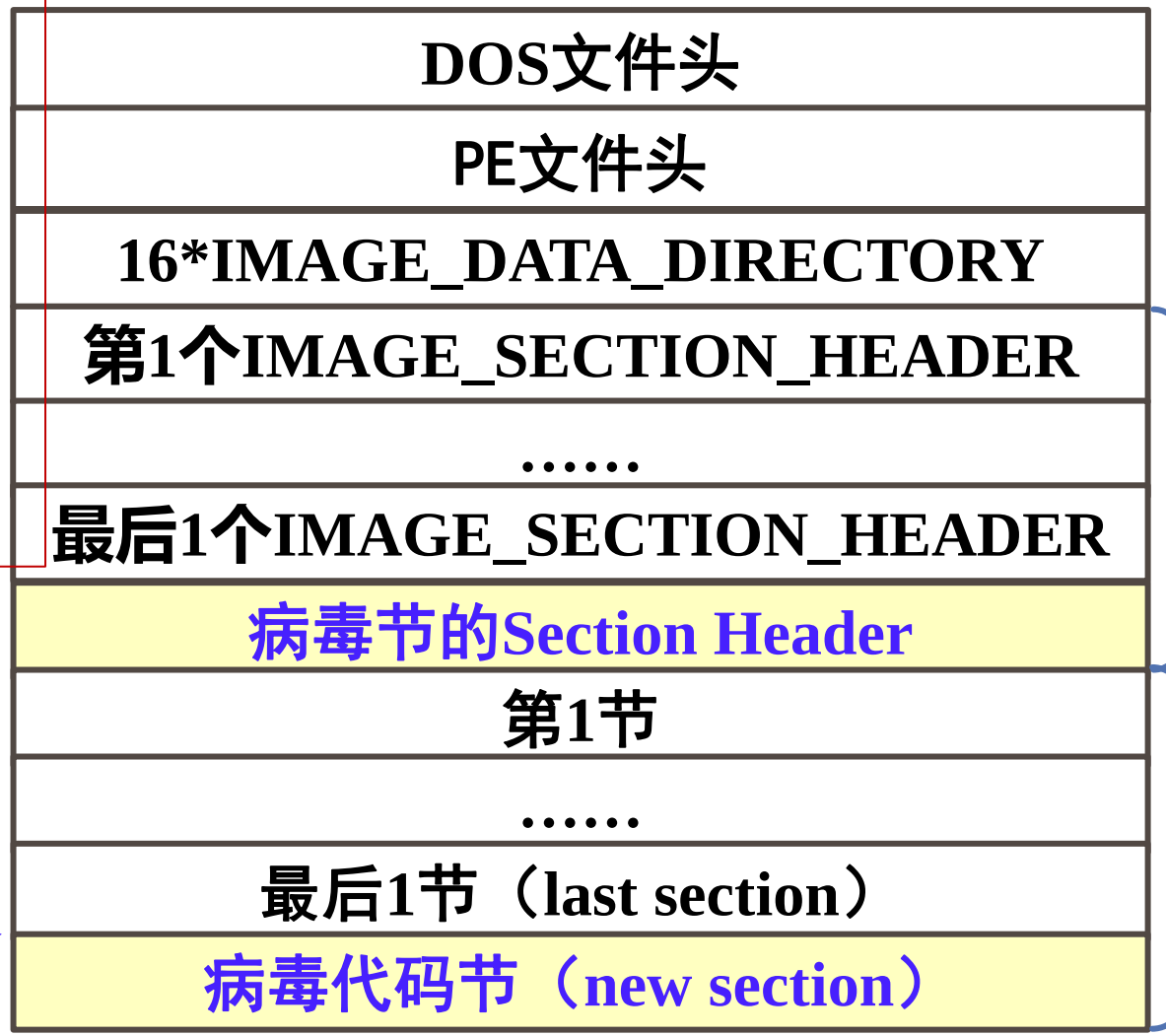
  In EXEs, this field contains the size of the section after it's been rounded up to the file alignment size. For example, assume a file alignment size of 0x200. If the VirtualSize field from above says that the section is 0x35A bytes in length, this field will say that the section is 0x400 bytes long. In OBJs, this field contains the exact size of the section emitted by the compiler or assembler. In other words, for OBJs, it's equivalent to the VirtualSize field in EXEs.

2.2 添加新节

```
new_section->PointerToRawData =
align_to_boundary (
    last_section->PointerToRawData +
            last_section->SizeOfRawData,
    nt_headers-> OptionalHeader.FileAlignment );
```

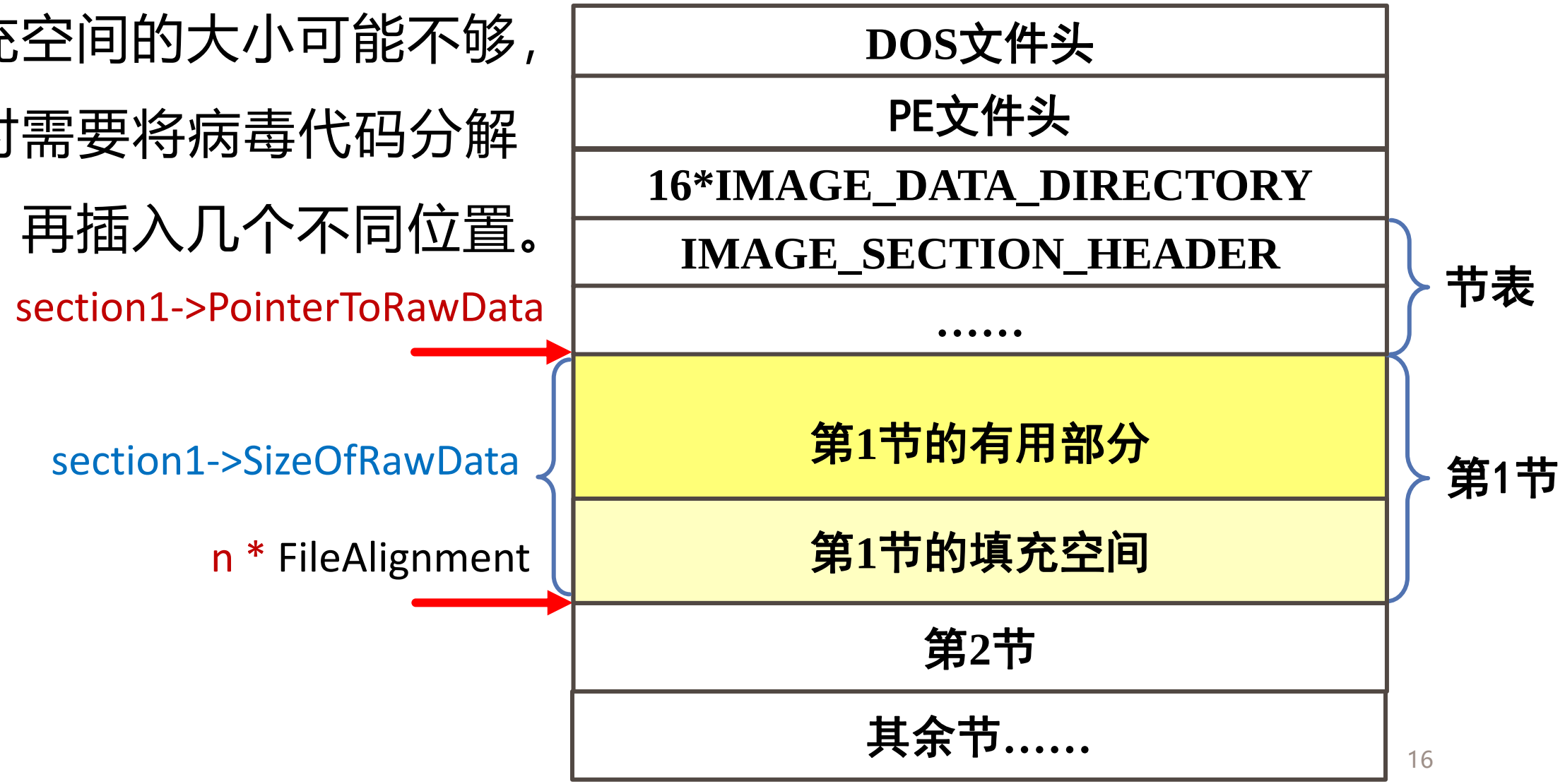
```
new_section->PointerToRawData =
    last_section->PointerToRawData +
    last_section->SizeOfRawData
```

new_section->PointerToRawData →



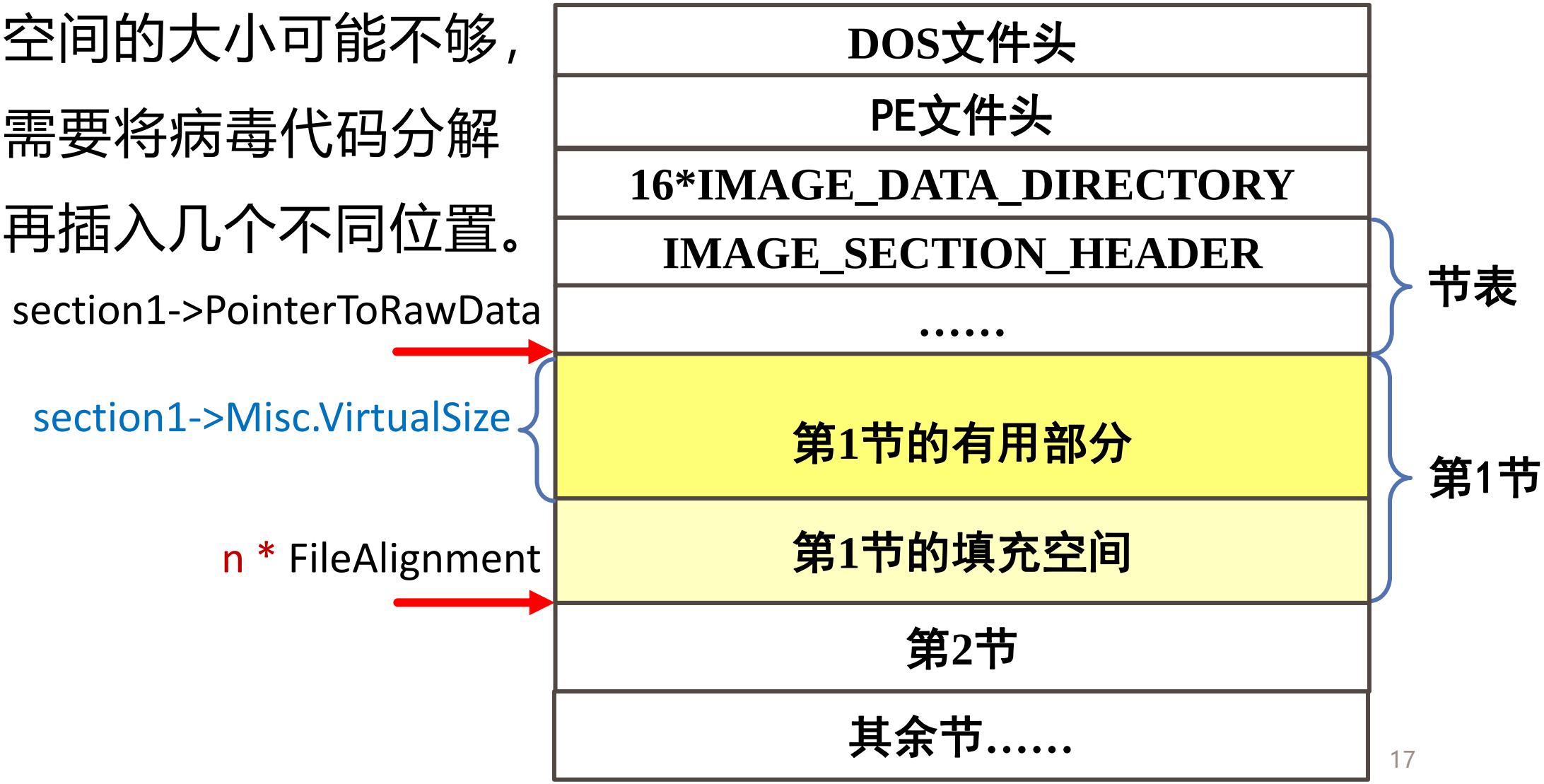
2.3 嵌入到节尾的未用空间

填充空间的大小可能不够，此时需要将病毒代码分解后，再插入几个不同位置。



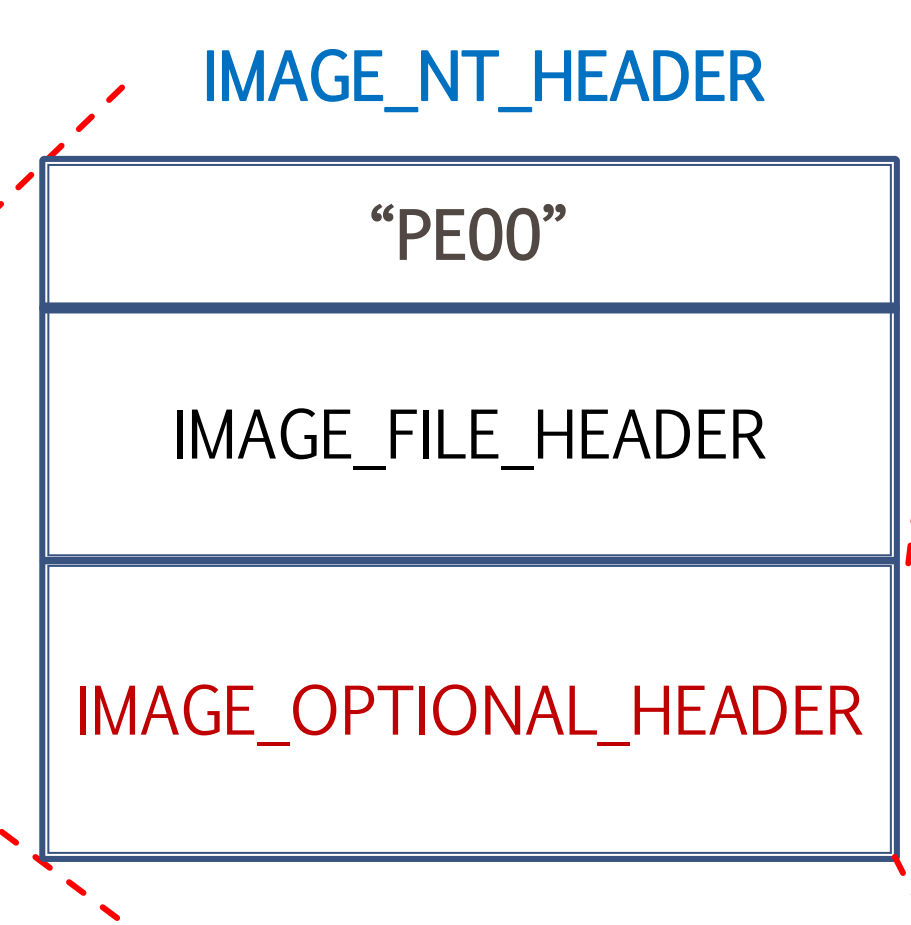
Misc.VirtualSize vs SizeOfRawData

填充空间的大小可能不够，此时需要将病毒代码分解后，再插入几个不同位置。



三、宿主程序执行流程的接管与恢复

3.1 程序执行入口

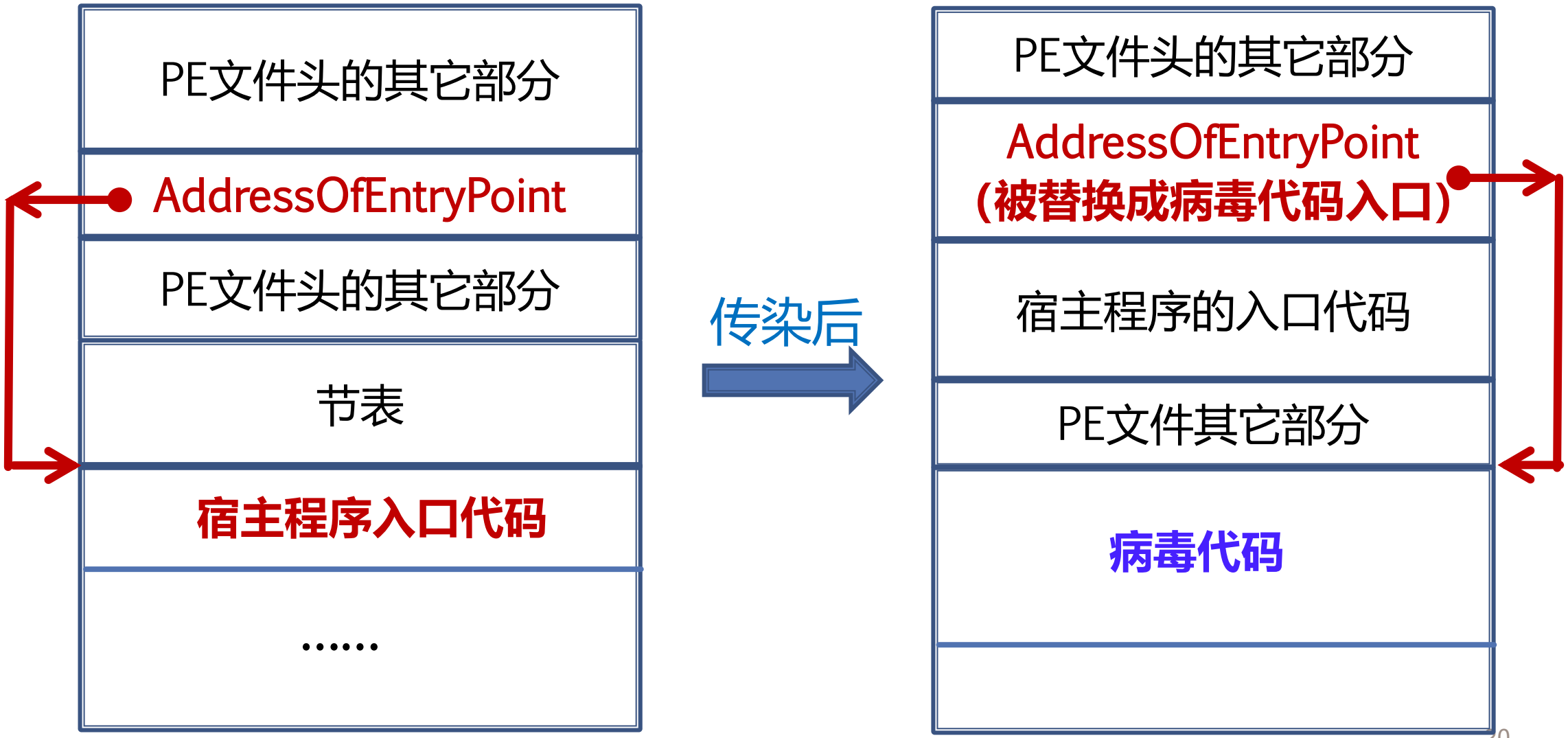



```
typedef struct _IMAGE_OPTIONAL_HEADER {
    WORD    Magic;
    BYTE    MajorLinkerVersion;
    BYTE    MinorLinkerVersion;
    DWORD   SizeOfCode;
    DWORD   SizeOfInitializedData;
    DWORD   SizeOfUninitializedData;
    DWORD   AddressOfEntryPoint;
    DWORD   BaseOfCode;
    DWORD   BaseOfData;
```

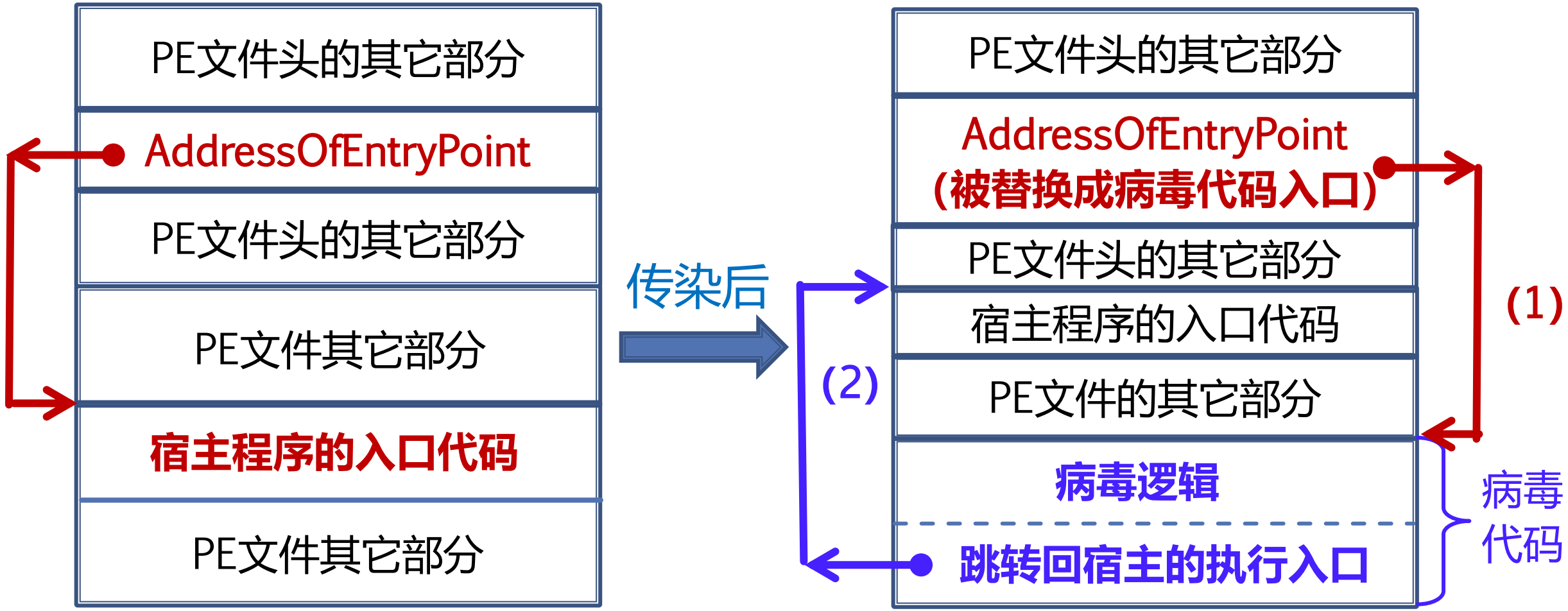
The RVA of the first code byte that will be executed.

……

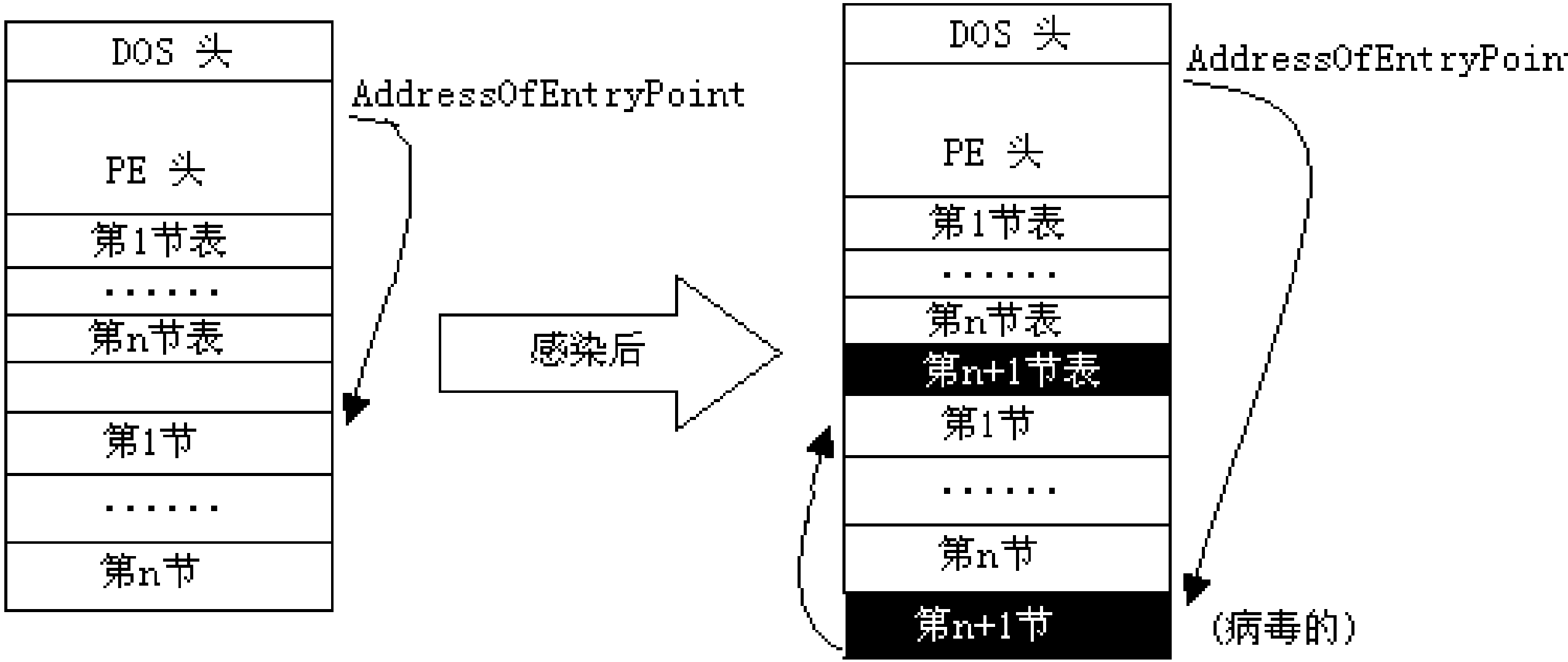
3.2 接管宿主的执行入口



3.3 恢复宿主的执行


PE文件头的其它部分
AddressOfEntryPoint
PE文件头的其它部分
PE文件其它部分
宿主程序的入口代码
PE文件其它部分
传染后
PE文件头的其它部分
AddressOfEntryPoint
(被替换成病毒代码入口)
PE文件头的其它部分
宿主程序的入口代码
PE文件的其它部分
病毒逻辑
跳转回宿主的执行入口
(1)
(2)
病毒
代码

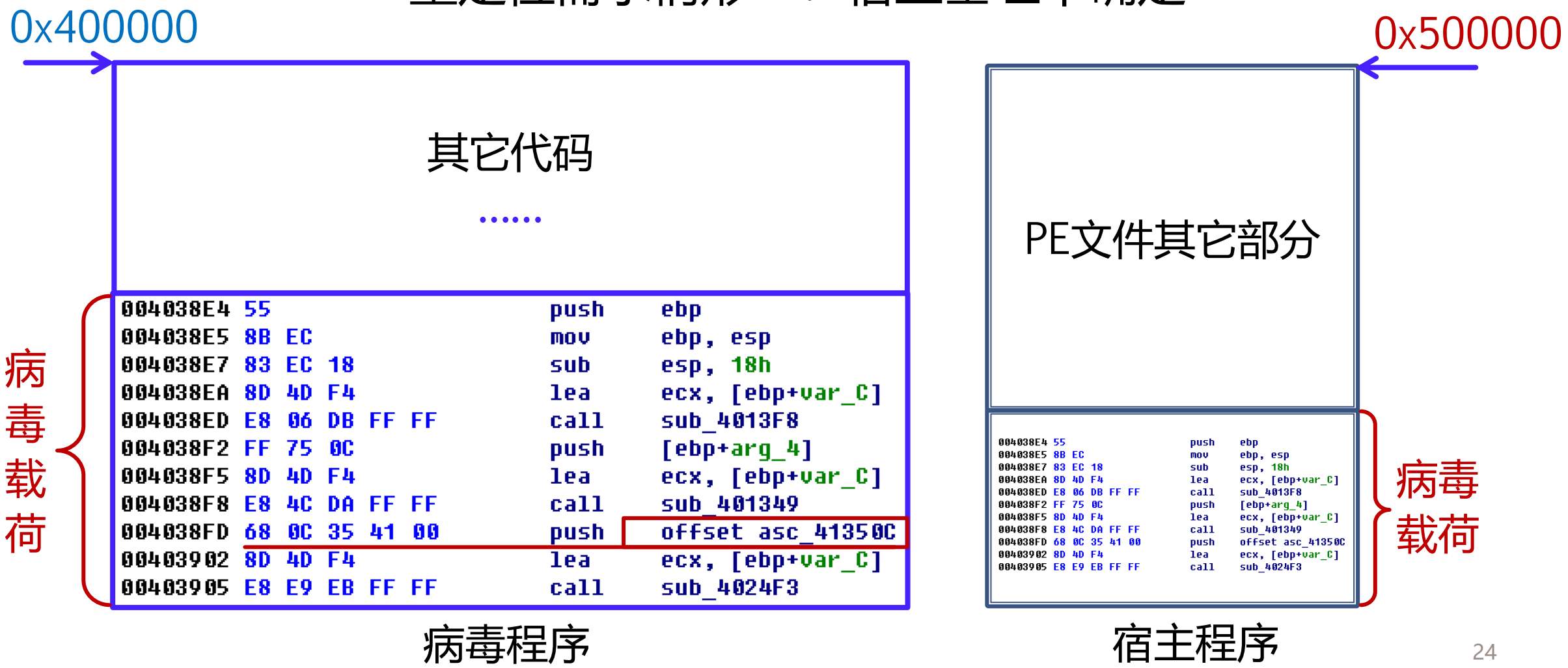
3.4 以添加新节为例


DOS 头
PE 头
第1节表
.....
第n节表
第1节
.....
第n节
AddressOfEntryPoint
感染后
DOS 头
PE 头
第1节表
.....
第n节表
第n+1节表
第1节
.....
第n节
第n+1节
AddressOfEntryPoin
(病毒的)

四、病毒代码自我重定位

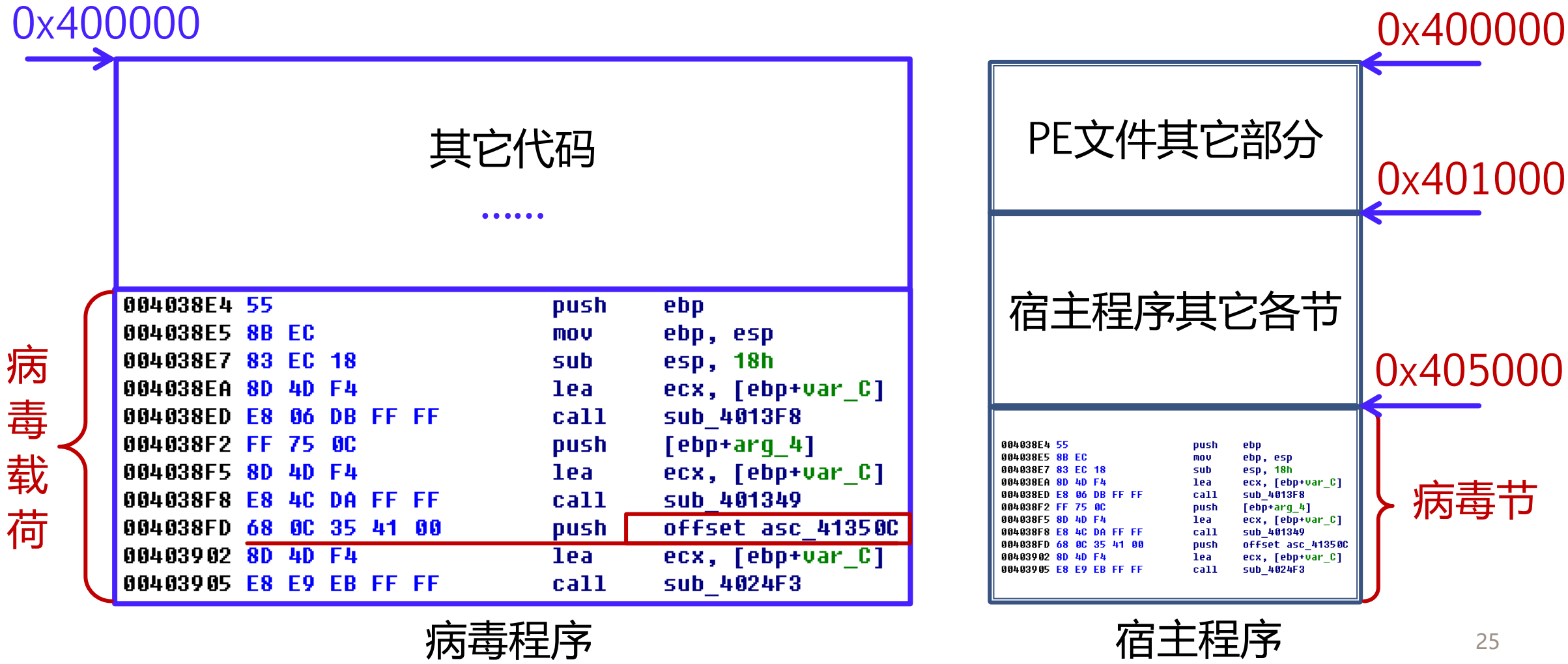
4.1 为什么需要重定位

重定位需求情形一：宿主基址不确定



4.1 为什么需要重定位

重定位需求情形之二：感染位置不确定



4.2 重定位的实现

- 重定位的目的

  修正程序中的代码或全局变量地址，使程序能够正常运行。

- 重定位方法

  由病毒代码运行过程中进行自我重定位

```
// 实现代码重定位
#define RECODE(A) { \
        _asm call A     \
        _asm A:         \
            _asm pop ebx    \
            _asm lea eax, A \
            _asm sub ebx,eax    \
    }

#define RecodeVarOffset(dwGlovalVar,dwLocalVar)  {  \
        _asm mov eax,[ebx+dwGlovalVar]  \
        _asm mov dwLocalVar,eax         \
    }
```

RECODE(A)

RecodeVarOffset(全局变量，局部变量)

全局变量重定位

段地址重定位原理

```
// call的返回地址是标号A的地址，
// call指令将此地址入栈
    call A ;
A:
    // pop取出的正好是标号A的地址。
    // 此地址是进程空间中的绝对地址
    pop ebx ;

    // 把绝对地址和相对偏移相减就可以
    // 获得段的起始地址
    sub ebx, offset A ;
```

seg_addr：段的初始地址

offset取的是标号相对seg_addr的偏移量

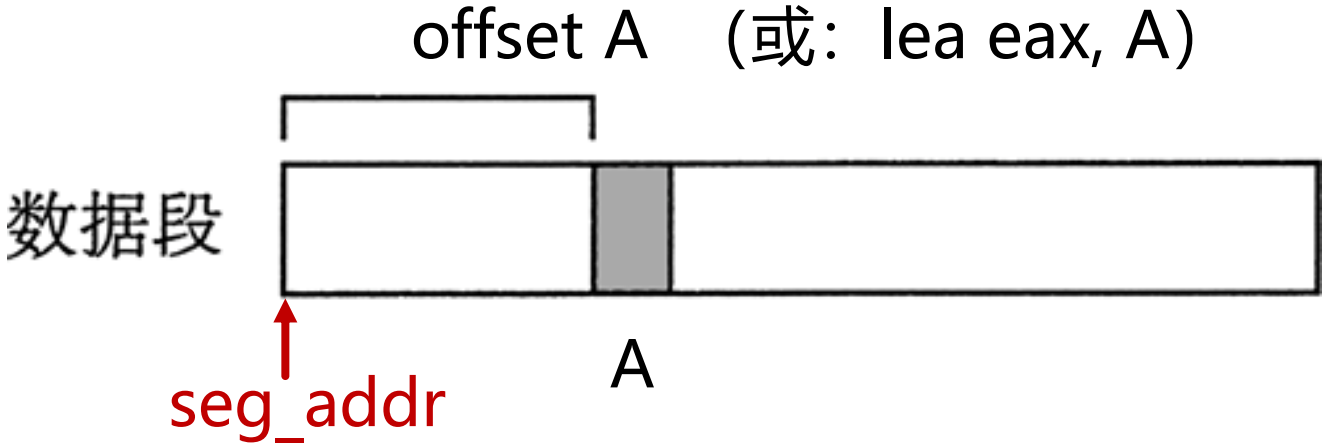
故：

[ebx] = seg_addr+offset A

seg_addr = [ebx] - offset A

ebx ← seg_addr

offset运算符返回数据标号的偏移量，该偏移量按字节计算，表示的是该数据标号距离数据段起始地址的距离。



五、获取系统API函数地址

5.1 PE文件导出函数的常规调用方法

- LoadLibrary
- GetProcAddress

HMODULE **LoadLibraryA**(

LPCSTR lpLibFileName);

lpLibFileName

The name of the module. This can be either a .dll file or an .exe file.

FARPROC **GetProcAddress**(

HMODULE hModule,

LPCSTR lpProcName);

hModule

A handle to the DLL module that contains the function or variable

lpProcName

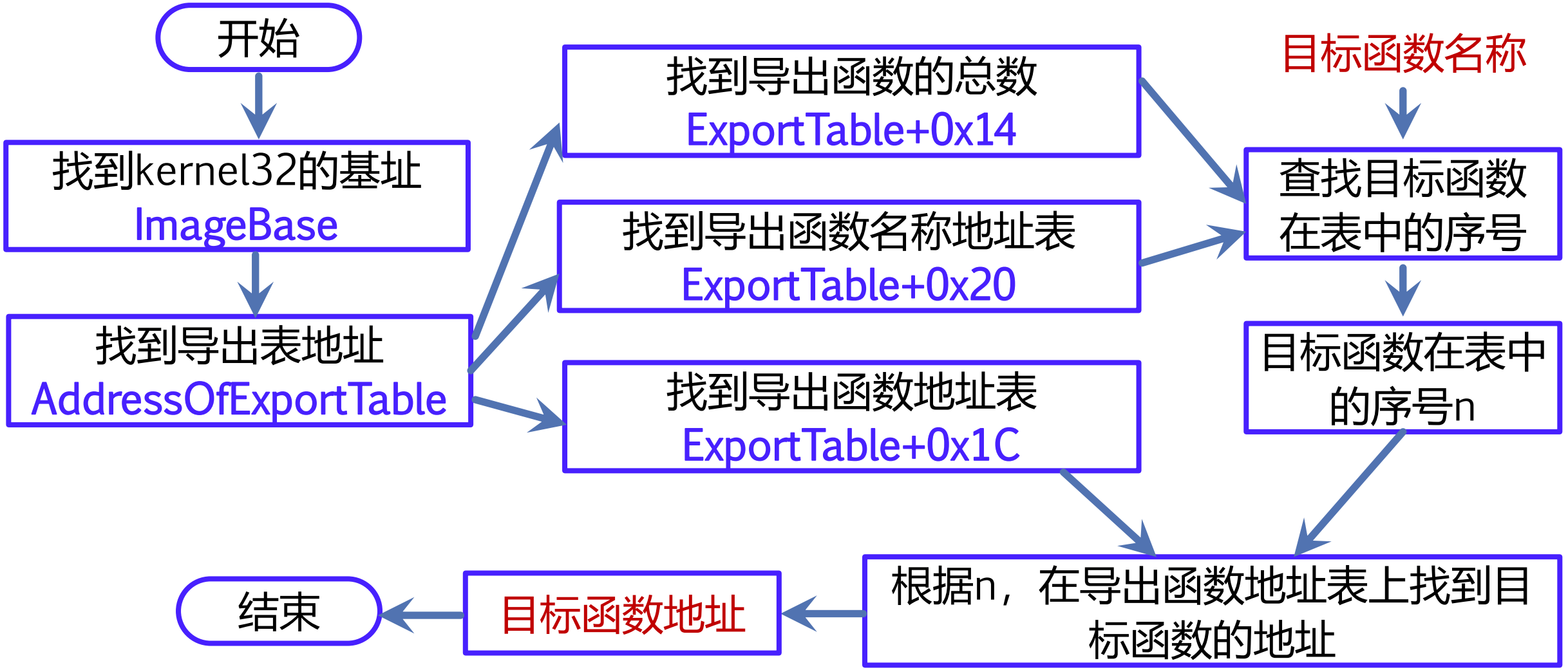
The function or variable name, or the function's ordinal value.

5.1 PE文件导出函数的常规调用方法

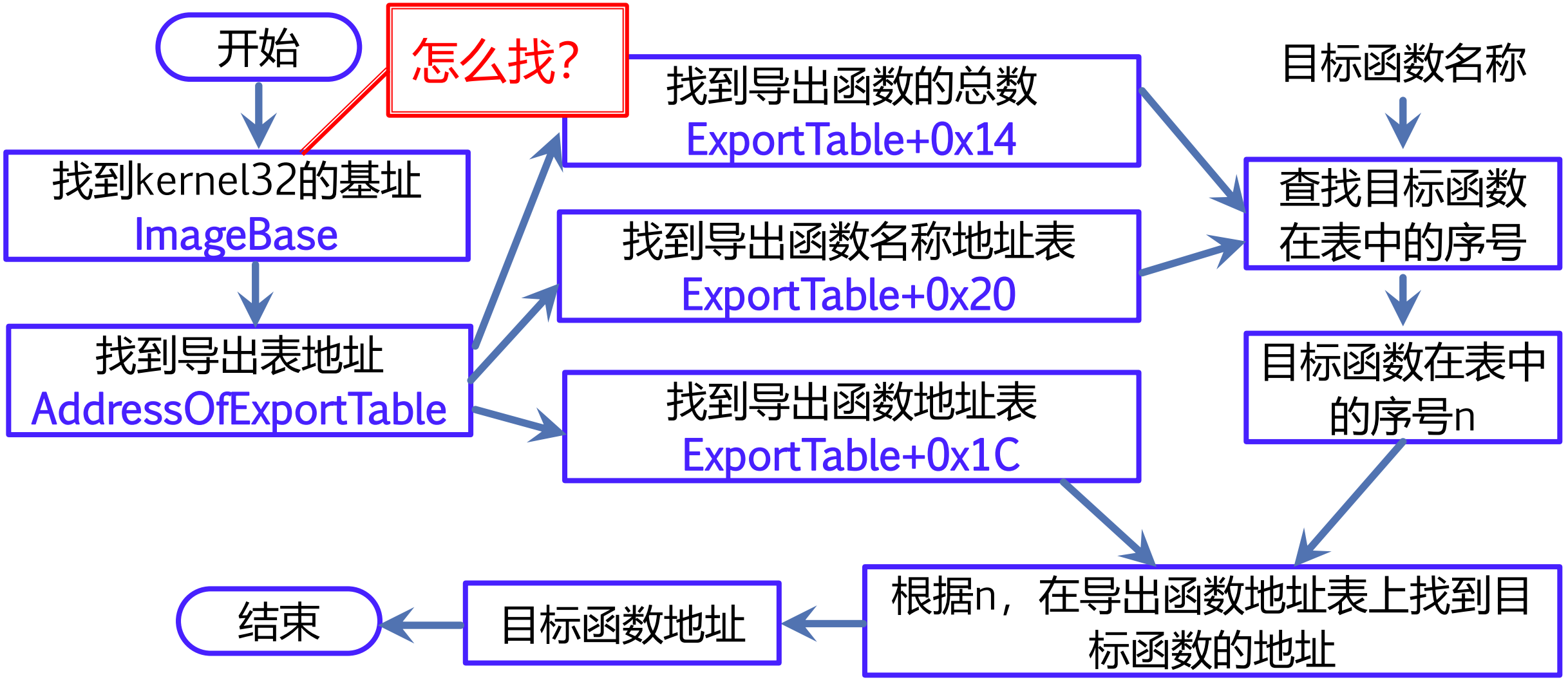
```
typedef int(WINAPI *ShellAboutProc)(HWND, LPCSTR, LPCSTR, HICON);

int main() {
  HMODULE hModule = LoadLibrary(TEXT("Shell32.dll"));

  ShellAboutProc shellAbout =
      (ShellAboutProc)GetProcAddress(hModule, "ShellAboutA");

  shellAbout(NULL, "hello", "world", NULL);

  FreeLibrary(hModule);
}
```

LoadLibrary/GetProcAddress用法示例

5.2 动态获取kernel32.dll的导出函数地址



5.3 获取kernel32模块基址

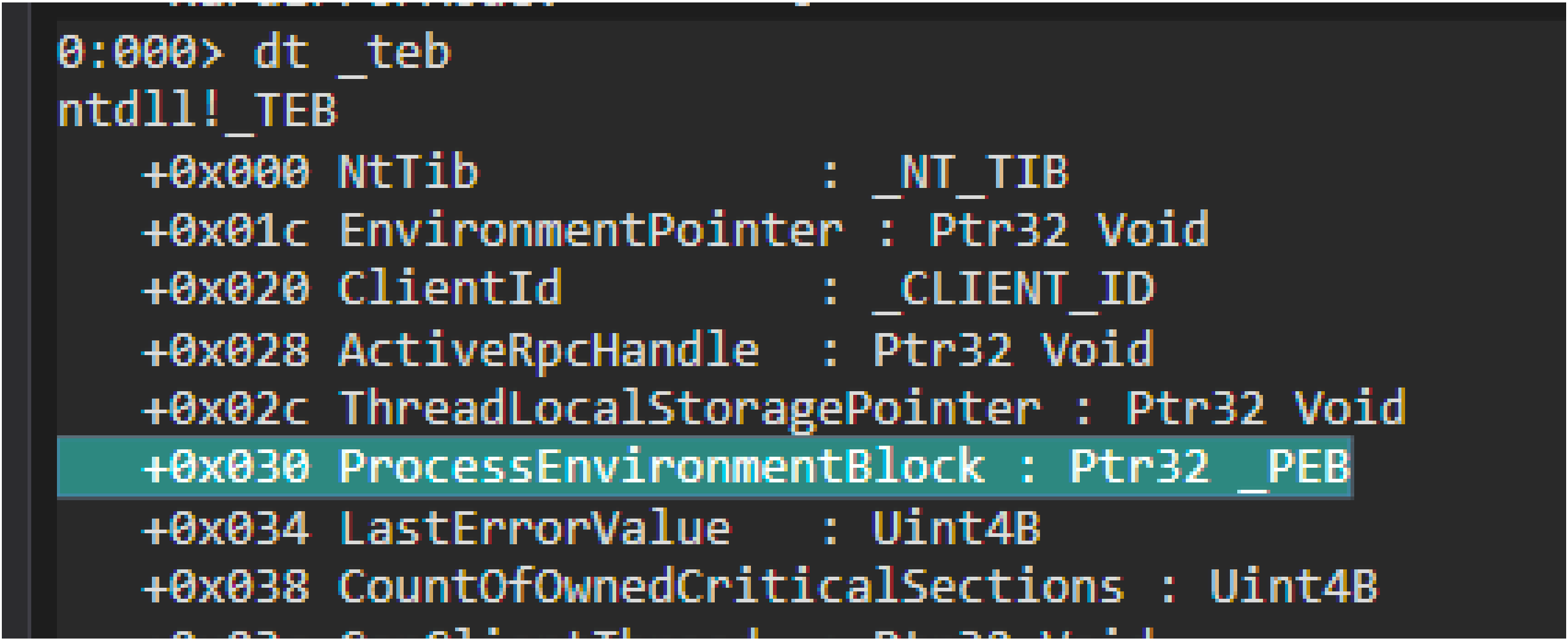


5.3 获取kernel32模块的基地址

- 基本查找路径

  TEB→PEB→ Ldr→ InMemoryOrderModuleList

  已加载模块信息在InMemoryOrderModuleList链表上
- fs:[0]指向TEB结构，PEB地址存放在fs:[30h]处
- 遍历InMemoryOrderModuleList，找到与kernel32模块对应的节点。



PEB ← TEB + 0x30 = fs:[30h]

```
0:000> dt _peb
ntdll!_PEB
   +0x000 InheritedAddressSpace : UChar
   +0x001 ReadImageFileExecOptions : UChar
   +0x002 BeingDebugged     : UChar
   +0x003 BitField          : UChar
   +0x003 ImageUsesLargePages : Pos 0, 1 Bit
   +0x003 IsProtectedProcess : Pos 1, 1 Bit
   +0x003 IsImageDynamicallyRelocated : Pos 2, 1 Bit
   +0x003 SkipPatchingUser32Forwarders : Pos 3, 1 Bit
   +0x003 IsPackagedProcess : Pos 4, 1 Bit
   +0x003 IsAppContainer    : Pos 5, 1 Bit
   +0x003 IsProtectedProcessLight : Pos 6, 1 Bit
   +0x003 IsLongPathAwareProcess : Pos 7, 1 Bit
   +0x004 Mutant            : Ptr32 Void
   +0x008 ImageBaseAddress  : Ptr32 Void
   +0x00c Ldr               : Ptr32 _PEB_LDR_DATA
   +0x010 ProcessParameters : Ptr32 _RTL_USER_PROCESS_PARAMETERS
   +0x014 SubSystemData     : Ptr32 Void
```

Ldr ← PEB + 0x0C

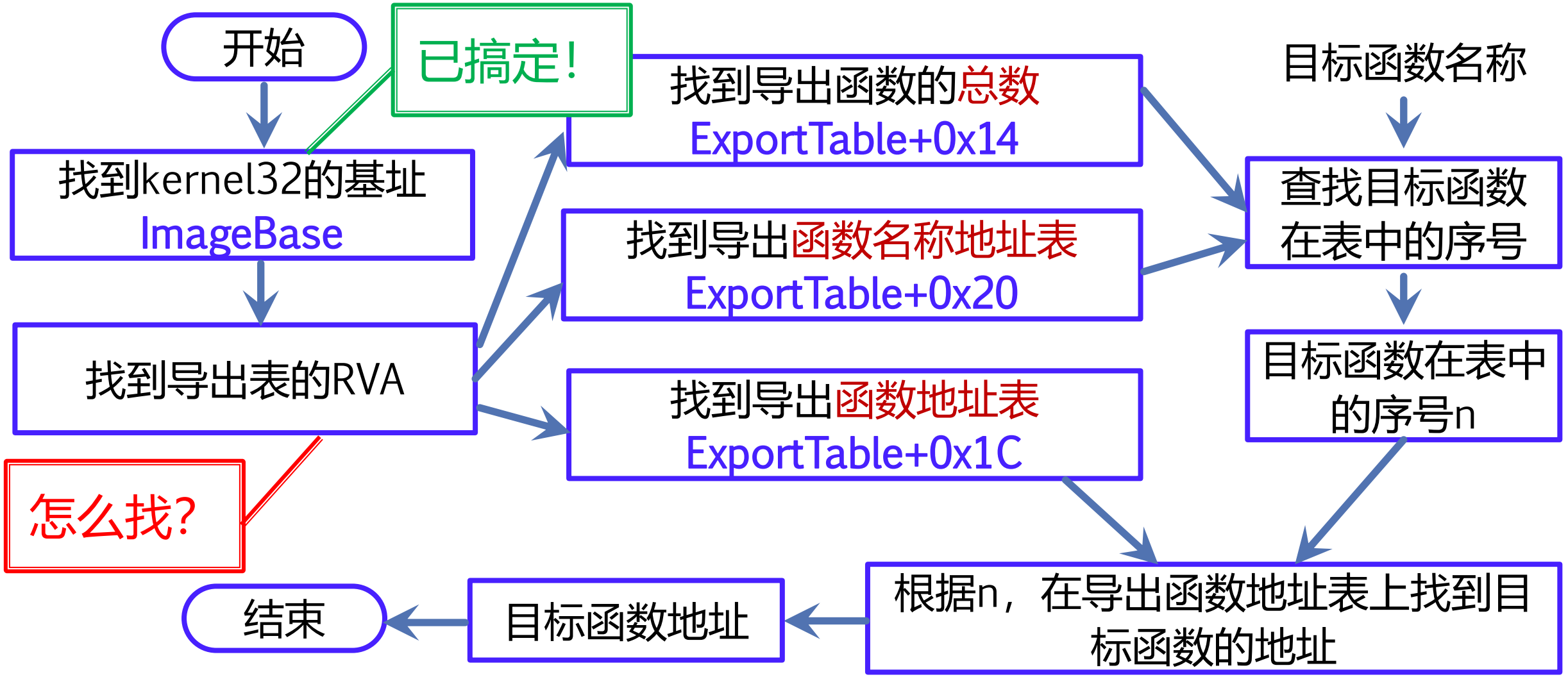
```
0:000> dt _PEB_LDR_DATA
ntdll!_PEB_LDR_DATA
   +0x000 Length                  : Uint4B
   +0x004 Initialized             : UChar
   +0x008 SsHandle                : Ptr32 Void
   +0x00c InLoadOrderModuleList   : _LIST_ENTRY
   +0x014 InMemoryOrderModuleList : _LIST_ENTRY
   +0x01c InInitializationOrderModuleList : _LIST_ENTRY
   +0x024 EntryInProgress         : Ptr32 Void
   +0x028 ShutdownInProgress      : UChar
   +0x02c ShutdownThreadId        : Ptr32 Void
```

- InMemoryOrderModuleList ← Ldr + 0x14
- InMemoryOrderModuleList指向一个链表，表上存放装载到进程空间的各个程序模块信息。

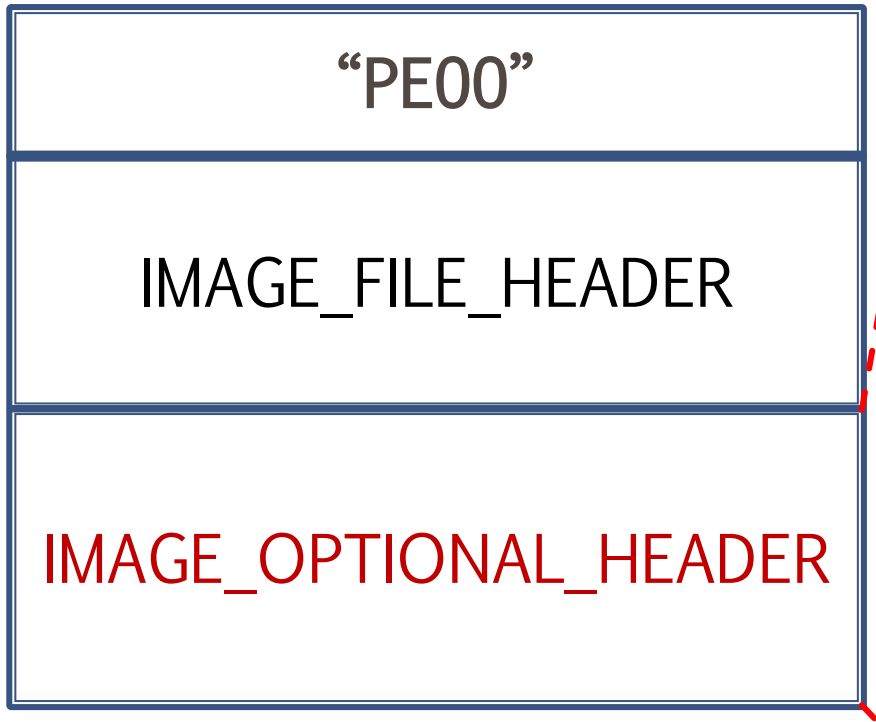
```
0:000> dt _LDR_DATA_TABLE_ENTRY
ntdll!_LDR_DATA_TABLE_ENTRY
   +0x000 InLoadOrderLinks : _LIST_ENTRY
   +0x008 InMemoryOrderLinks : _LIST_ENTRY
   +0x010 InInitializationOrderLinks : _LIST_ENTRY
   +0x018 DllBase          : Ptr32 Void
   +0x01c EntryPoint       : Ptr32 Void
   +0x020 SizeOfImage      : Uint4B
   +0x024 FullDllName      : _UNICODE_STRING
   +0x02c BaseDllName      : _UNICODE_STRING
   +0x034 FlagGroup        : [4] UChar
   +0x034 Flags            : Uint4B
   +0x034 PackagedBinary   : Pos 0, 1 Bit
   +0x034 MarkedForRemoval : Pos 1, 1 Bit
   +0x034 ImageDll         : Pos 2, 1 Bit
   +0x034 LoadNotificationsSent : Pos 3, 1 Bit
```

- InMemoryOrderModuleList指向的类型实为LDR_DATA_TABLE_ENTRY。
- InMemoryOrderModuleList链表每个LDR_DATA_TABLE_ENTRY节点，存放一个模块的信息。节点第0x18偏移处存放模块的基址，第0x2C位置存放模块名称。

5.4 获取kernel32模块导出表的地址



PE文件头中的DataDirectory

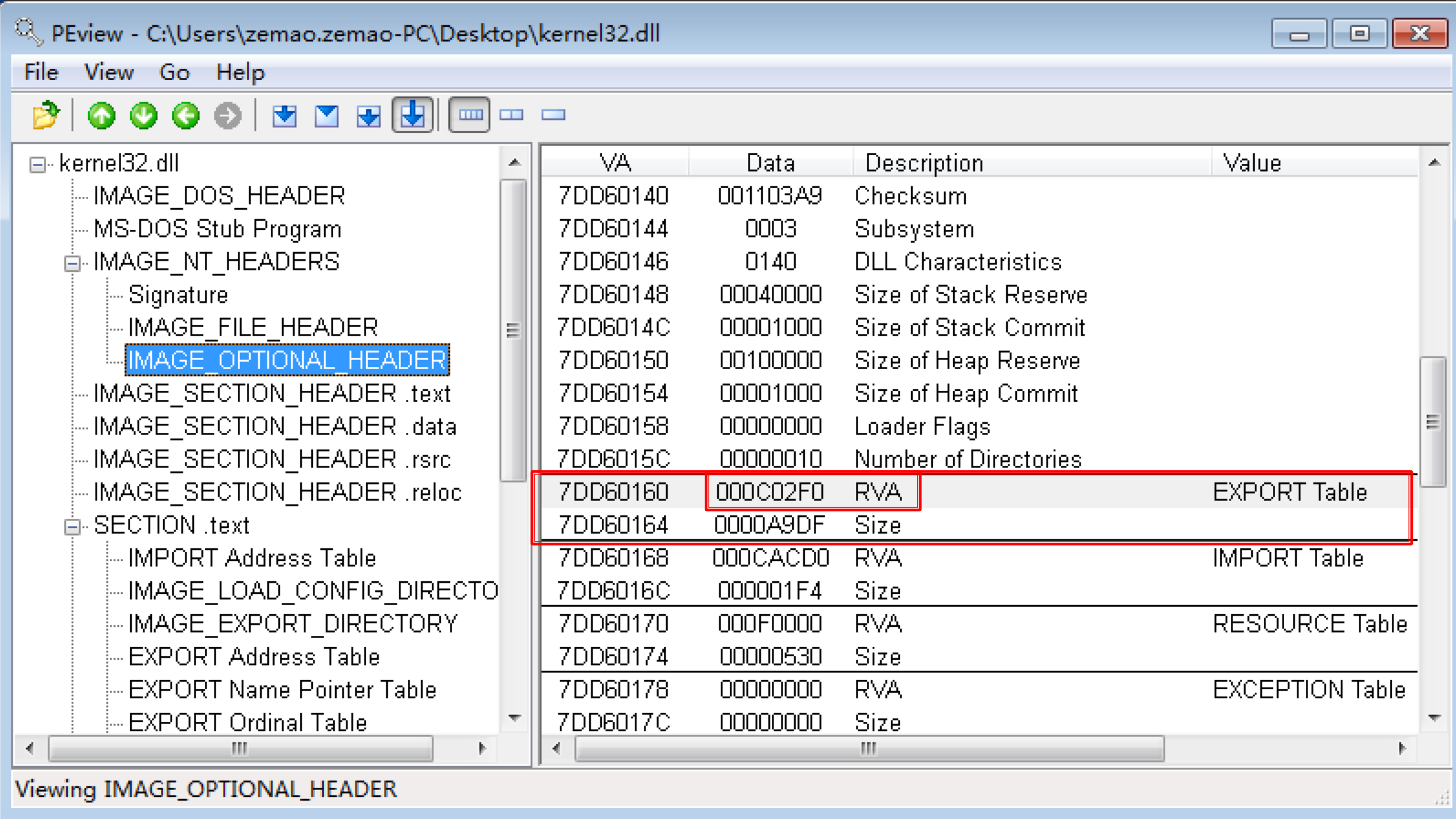



```
    DWORD   Win32VersionValue;
    DWORD   SizeOfImage;
    DWORD   SizeOfHeaders;
    DWORD   CheckSum;
    WORD    Subsystem;
    WORD    DllCharacteristics;
    DWORD   SizeOfStackReserve;
    DWORD   SizeOfStackCommit;
    DWORD   SizeOfHeapReserve;
    DWORD   SizeOfHeapCommit;
    DWORD   LoaderFlags;
    DWORD   NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NU
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEA
```

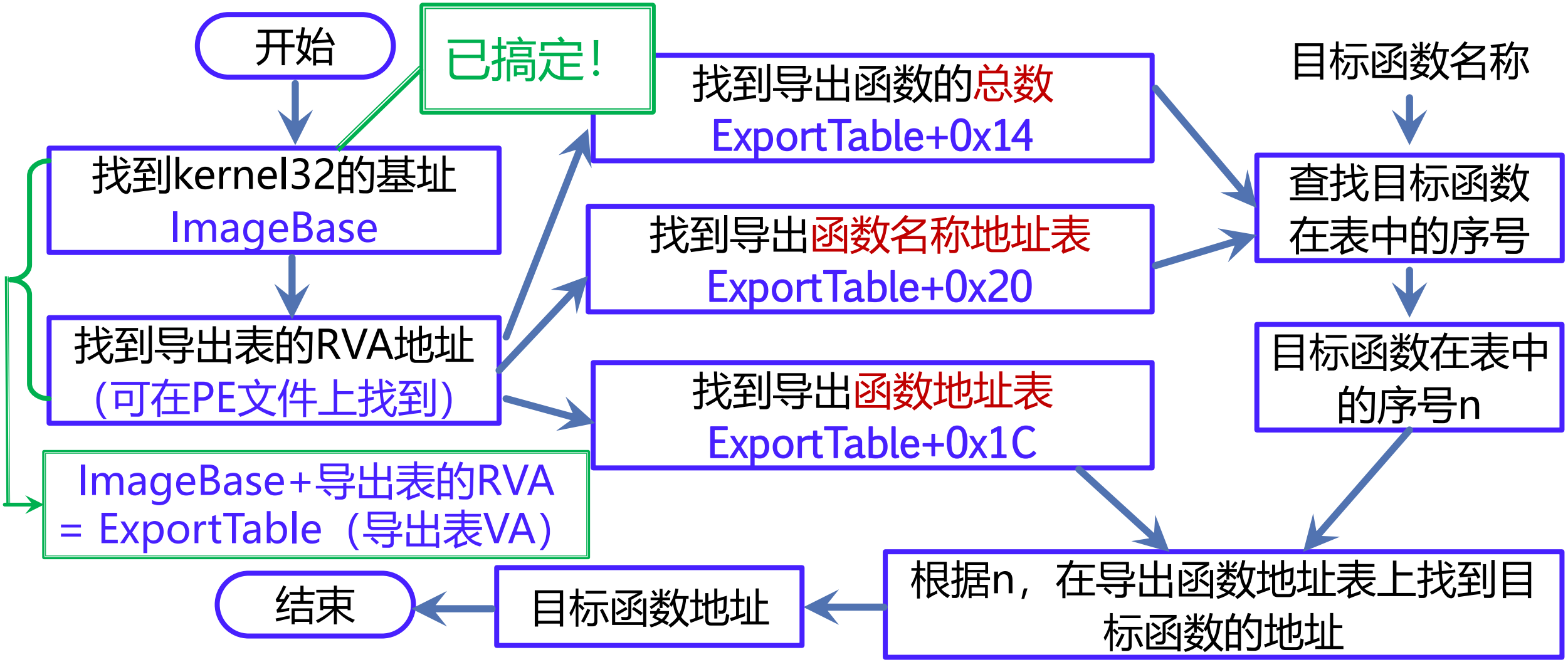
固定值：16

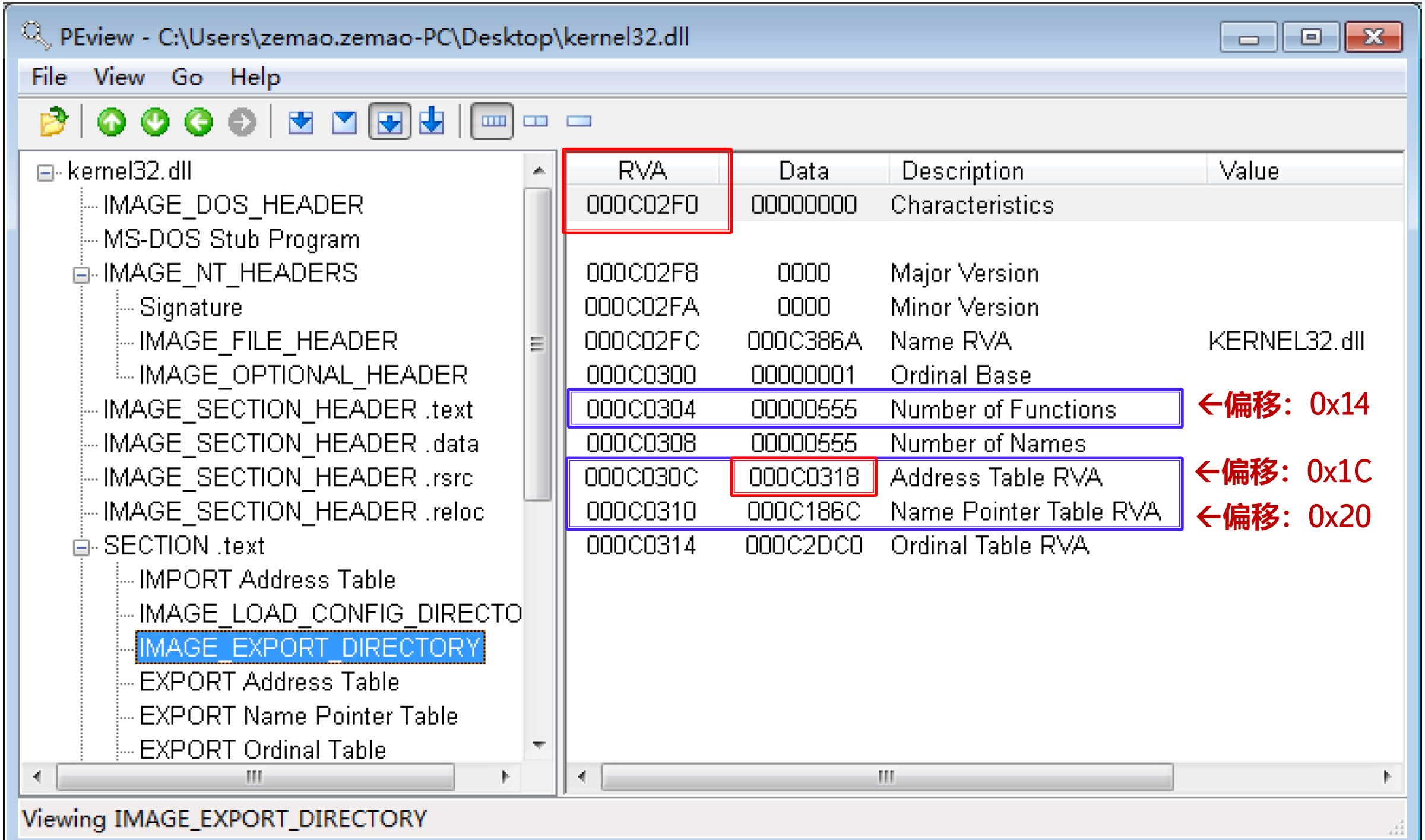
PE文件头中的DataDirectory

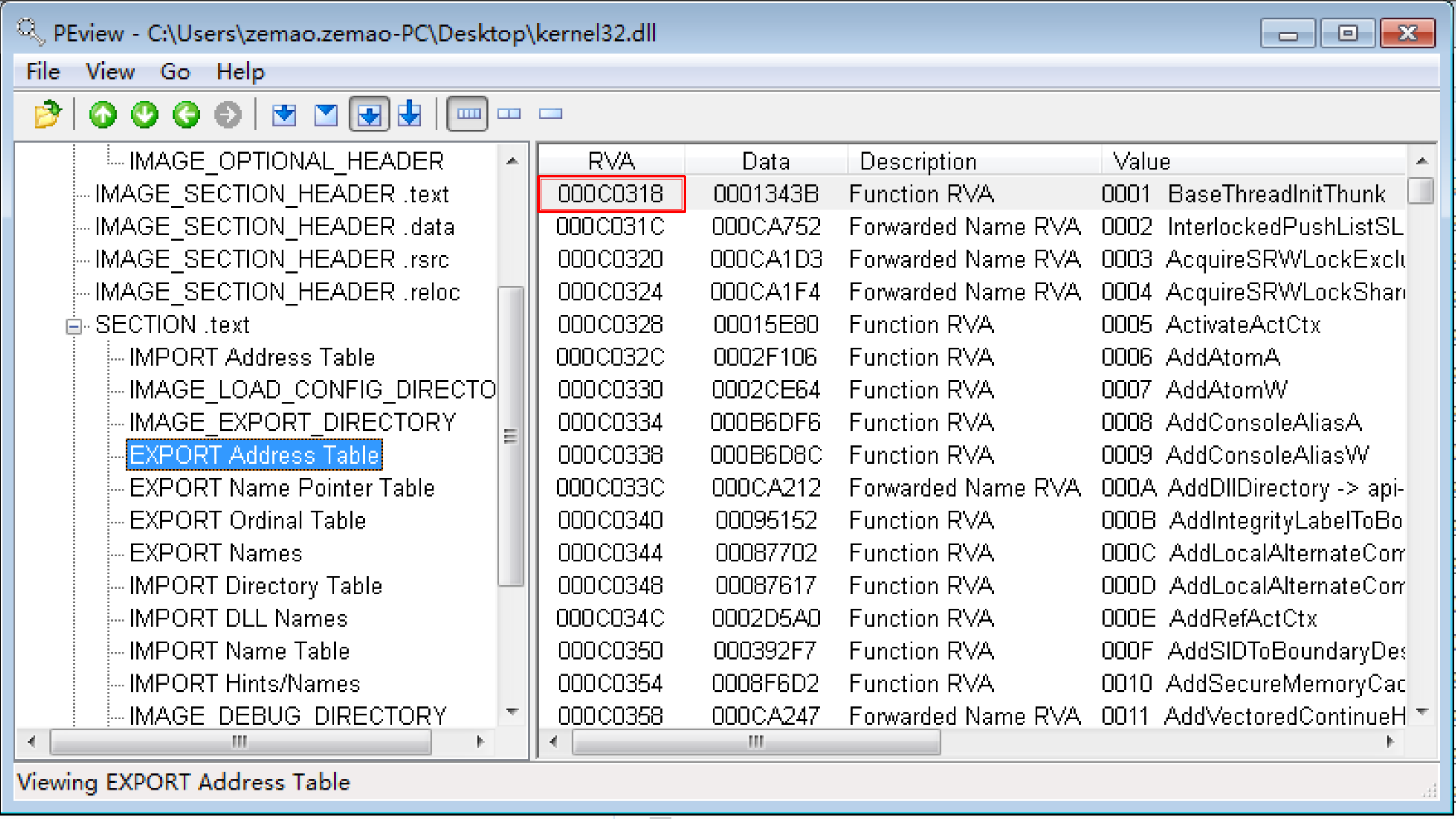
```
#define IMAGE_DIRECTORY_ENTRY_EXPORT          0   // Export Directory
#define IMAGE_DIRECTORY_ENTRY_IMPORT          1   // Import Directory
#define IMAGE_DIRECTORY_ENTRY_RESOURCE        2   // Resource Directory
#define IMAGE_DIRECTORY_ENTRY_EXCEPTION       3   // Exception Directory
#define IMAGE_DIRECTORY_ENTRY_SECURITY        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
```


PEview - C:\Users\zemao.zemao-PC\Desktop\kernel32.dll
File View Go Help
kernel32.dll
IMAGE_DOS_HEADER
MS-DOS Stub Program
IMAGE_NT_HEADERS
Signature
IMAGE_FILE_HEADER
IMAGE_OPTIONAL_HEADER
IMAGE_SECTION_HEADER .text
IMAGE_SECTION_HEADER .data
IMAGE_SECTION_HEADER .rsrc
IMAGE_SECTION_HEADER .reloc
SECTION .text
IMPORT Address Table
IMAGE_LOAD_CONFIG_DIRECTO
IMAGE_EXPORT_DIRECTORY
EXPORT Address Table
EXPORT Name Pointer Table
EXPORT Ordinal Table
VA Data Description Value
7DD60140 001103A9 Checksum
7DD60144 0003 Subsystem
7DD60146 0140 DLL Characteristics
7DD60148 00040000 Size of Stack Reserve
7DD6014C 00001000 Size of Stack Commit
7DD60150 00100000 Size of Heap Reserve
7DD60154 00001000 Size of Heap Commit
7DD60158 00000000 Loader Flags
7DD6015C 00000010 Number of Directories
7DD60160 000C02F0 RVA EXPORT Table
7DD60164 0000A9DF Size
7DD60168 000CACD0 RVA IMPORT Table
7DD6016C 000001F4 Size
7DD60170 000F0000 RVA RESOURCE Table
7DD60174 00000530 Size
7DD60178 00000000 RVA EXCEPTION Table
7DD6017C 00000000 Size
Viewing IMAGE_OPTIONAL_HEADER

5.4 获取kernel32模块导出函数的地址




PEview - C:\Users\zemao.zemao-PC\Desktop\kernel32.dll
File View Go Help
kernel32.dll
IMAGE_DOS_HEADER
MS-DOS Stub Program
IMAGE_NT_HEADERS
Signature
IMAGE_FILE_HEADER
IMAGE_OPTIONAL_HEADER
IMAGE_SECTION_HEADER .text
IMAGE_SECTION_HEADER .data
IMAGE_SECTION_HEADER .rsrc
IMAGE_SECTION_HEADER .reloc
SECTION .text
IMPORT Address Table
IMAGE_LOAD_CONFIG_DIRECTO
IMAGE_EXPORT_DIRECTORY
EXPORT Address Table
EXPORT Name Pointer Table
EXPORT Ordinal Table
RVA Data Description Value
000C02F0 00000000 Characteristics
000C02F8 0000 Major Version
000C02FA 0000 Minor Version
000C02FC 000C386A Name RVA KERNEL32.dll
000C0300 00000001 Ordinal Base
000C0304 00000555 Number of Functions
000C0308 00000555 Number of Names
000C030C 000C0318 Address Table RVA
000C0310 000C186C Name Pointer Table RVA
000C0314 000C2DC0 Ordinal Table RVA
←偏移：0x14
←偏移：0x1C
←偏移：0x20
Viewing IMAGE_EXPORT_DIRECTORY


PEview - C:\Users\zemao.zemao-PC\Desktop\kernel32.dll
File View Go Help
IMAGE_OPTIONAL_HEADER
IMAGE_SECTION_HEADER .text
IMAGE_SECTION_HEADER .data
IMAGE_SECTION_HEADER .rsrc
IMAGE_SECTION_HEADER .reloc
SECTION .text
IMPORT Address Table
IMAGE_LOAD_CONFIG_DIRECTO
IMAGE_EXPORT_DIRECTORY
EXPORT Address Table
EXPORT Name Pointer Table
EXPORT Ordinal Table
EXPORT Names
IMPORT Directory Table
IMPORT DLL Names
IMPORT Name Table
IMPORT Hints/Names
IMAGE_DEBUG_DIRECTORY
RVA Data Description Value
000C0318 0001343B Function RVA 0001 BaseThreadInitThunk
000C031C 000CA752 Forwarded Name RVA 0002 InterlockedPushListSL
000C0320 000CA1D3 Forwarded Name RVA 0003 AcquireSRWLockExclu
000C0324 000CA1F4 Forwarded Name RVA 0004 AcquireSRWLockShar
000C0328 00015E80 Function RVA 0005 ActivateActCtx
000C032C 0002F106 Function RVA 0006 AddAtomA
000C0330 0002CE64 Function RVA 0007 AddAtomW
000C0334 000B6DF6 Function RVA 0008 AddConsoleAliasA
000C0338 000B6D8C Function RVA 0009 AddConsoleAliasW
000C033C 000CA212 Forwarded Name RVA 000A AddDllDirectory -> api-
000C0340 00095152 Function RVA 000B AddIntegrityLabelToBo
000C0344 00087702 Function RVA 000C AddLocalAlternateCom
000C0348 00087617 Function RVA 000D AddLocalAlternateCom
000C034C 0002D5A0 Function RVA 000E AddRefActCtx
000C0350 000392F7 Function RVA 000F AddSIDToBoundaryDes
000C0354 0008F6D2 Function RVA 0010 AddSecureMemoryCac
000C0358 000CA247 Forwarded Name RVA 0011 AddVectoredContinueH
Viewing EXPORT Address Table

六、搜索传染目标

6.1 目标程序遍历搜索

- 以PE格式的文件（如EXE、SCR、DLL等）为感染目标。
- 用到的kernel32 API函数
  - FindFirstFile
  - FindNextFile
  - FindClose

6.2 目标搜索

Input: targetPath：要搜索的目标路径

Output: 无

FindAndInfect(targetPath):

1. 调用**FindFirstFile**开始搜索
2. 已搜索完毕？是，执行**FindClose**后返回；否，继续下步.
3. 找到的是文件还是目录？是目录，则调用**FindAndInfect**；
   否，继续下步.
4. 是文件，如符合传染条件，则感染之.
5. 调用**FindNextFile**搜索下一个目标，转到第3步继续.

FindFirstFileA/FindNextFile/ FindClose

```
HANDLE FindFirstFile(
  LPCSTR             lpFileName,
  LPWIN32_FIND_DATAA lpFindFileData
);

BOOL FindNextFile(
  HANDLE             hFindFile,
  LPWIN32_FIND_DATAA lpFindFileData
);

BOOL FindClose(
  HANDLE hFindFile
);
```

FindFirstFileA/FindNextFile/ FindClose用法示例

```
WIN32_FIND_DATA pNextInfo;
hFile = FindFirstFile(lpFileName, &pNextInfo);
if (hFile != INVALID_HANDLE_VALUE) {
    while(FindNextFile(hFile, &pNextInfo)) {
        // ……
    }
}
FindClose(hFile);
```

小结

- 传染部位的选择
- 宿主程序执行流程的接管与恢复
- 病毒代码的自我重定位
- 系统API函数地址的获取
- 搜索传染目标