

电子科技大学
计算机科学与工程学院

标准实验报告

(实验) 课程名称 信息对抗综合设计实验 II

电子科技大学教务处制表

电子科技大学

实验报告

学生姓名：刘芷溢

学号：2020080907009

指

导教师：李忻洋

一、实验项目名称：Windows 病毒寄生实验

二、实验目的：

熟练掌握 Windows 下 x64dbg 汇编指令级调试器的基本操作和使用，熟悉 Windows 下 PE 可执行程序的基本结构，理解病毒感染 PE 文件的原理（修改入口点，在文件空洞处寄生，在非代码区寄生并跳回原入口点以及尾区段寄生扩展区段大小）。同时为实验四做准备。

三、实验原理：

x64dbg 的使用：

x64dbg 是一款免费开源的 x86/x64 汇编指令级动态调试器，软件原生支持中文界面和插件，其界面及操作方法与 OllyDbg 调试工具类似，支持类似 C 的表达式解析器、全功能的 DLL 和 EXE 文件调试、IDA 般的侧边栏与跳跃箭头、动态识别模块和串、快反汇编、可调试的脚本语言自动化等多项实用分析功能。

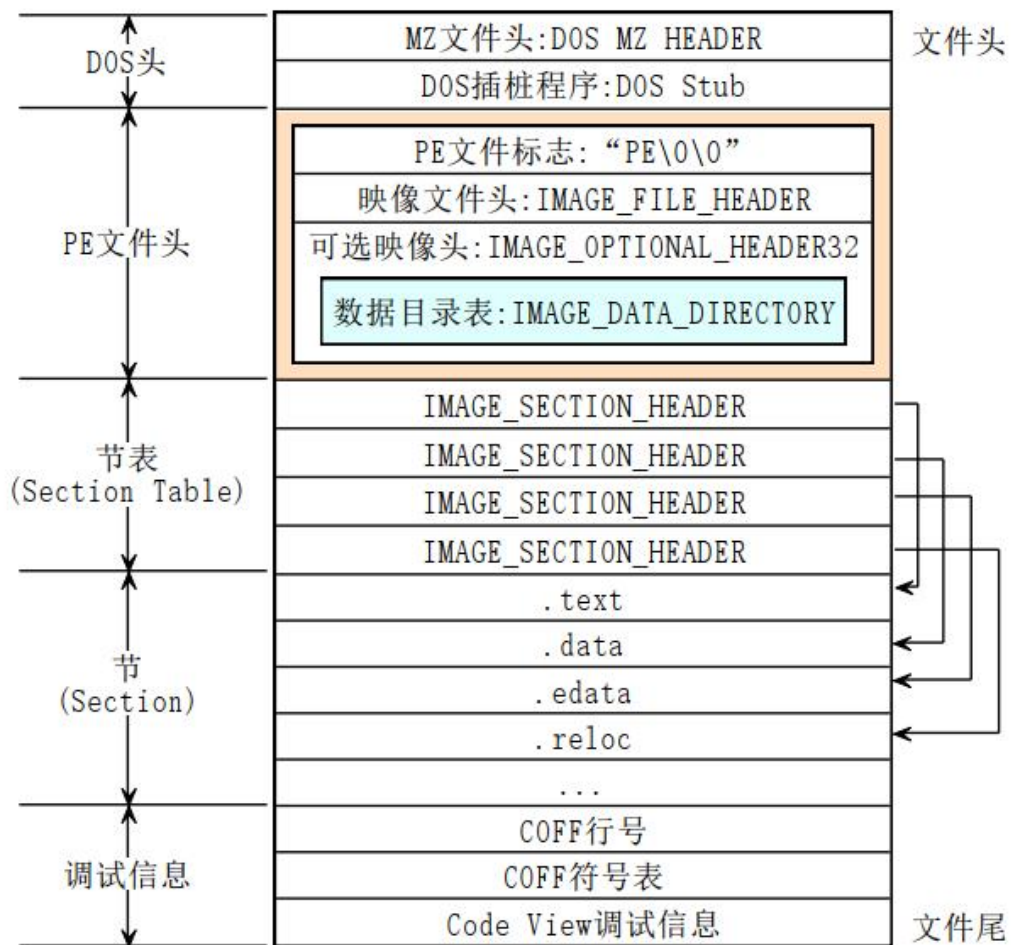
x64dbg 中包含有针对 x86 和 x64 应用程序的两个不同的调试

器，在调试不同类型程序的使用需要使用对应的调试器。可以通过默认的 x96dbg 启动对应的调试器；本次实验所使用到的 x64dbg 基本功能如下：启动一个程序调试、Attach 到一个已经运行的程序调试、单步，step into and step over、断点、继续运行、查看内存、修改内存、查看寄存器、修改寄存器、代码窗体跳到指定地址、修改指令、查看一个进程加载的 dll、查看 dll 中有哪些函数。针对已经启动的目标进程，或目标进程存在“强壳”保护的情况下，也可使用 x64dbg 在中途“附加”到该进程上，进行调试。

查看 dll 中的导出函数：在 Dll 列表窗口，选中对应的 Dll，x64dbg 会自动在右侧展示该 Dll 的所有导入与导出函数：

地址	类型	序号	符号
00007FFA49741100	导出	1	_Aligned_get_default_resource
00007FFA49741120	导出	2	_Aligned_new_delete_resource
00007FFA49741130	导出	3	_Aligned_set_default_resource
00007FFA49741150	导出	4	_Unaligned_get_default_resource
00007FFA49741170	导出	5	_Unaligned_new_delete_resource
00007FFA49741180	导出	6	_Unaligned_set_default_resource
00007FFA497411A0	导出	7	null_memory_resource
00007FFA497415C0	导出	0	OptionalHeader.AddressOfEntryPoint
00007FFA49743070	导出		msvcpl40.7.Xbad_alloc@std@VAXZ
00007FFA49743080	导入		vcruntime140.__std_type_info_destroy_list
00007FFA49743088	导入		vcruntime140._CxxThrowException
00007FFA49743090	导入		vcruntime140.memset
00007FFA49743098	导入		vcruntime140.__std_exception_copy
00007FFA497430A0	导入		vcruntime140.memcpy
00007FFA497430A8	导入		vcruntime140._C_specific_handler
00007FFA497430B0	导入		vcruntime140.__std_exception_destroy
00007FFA497430C0	导入		ucrtbase._aligned_malloc
00007FFA497430C8	导入		ucrtbase._callnewh
00007FFA497430D0	导入		ucrtbase.free
00007FFA497430D8	导入		ucrtbase._aligned_free
00007FFA497430E0	导入		ucrtbase.malloc
00007FFA497430F0	导入		ucrtbase._initialize_onexit_table
00007FFA497430F8	导入		ucrtbase._configure_narrow_argv
00007FFA49743100	导入		ucrtbase._seh_filter_dll
00007FFA49743108	导入		ucrtbase._cexit
00007FFA49743110	导入		ucrtbase._initterm_e
00007FFA49743118	导入		ucrtbase._initterm
00007FFA49743120	导入		ucrtbase._initialize_narrow_environment
00007FFA49743128	导入		ucrtbase._execute_onexit_table
00007FFA49743000	导入		kernel32.QueryPerformanceCounter
00007FFA49743008	导入		kernel32.GetCurrentProcessId
00007FFA49743010	导入		kernel32.GetCurrentThreadId
00007FFA49743018	导入		kernel32.GetSystemTimeAsFileTime
00007FFA49743020	导入		ntdll.InitializeSLISTHead
00007FFA49743028	导入		kernel32.RtlCaptureContext
00007FFA49743030	导入		kernel32.RtlLookupFunctionEntry
00007FFA49743038	导入		kernel32.RtlVirtualUnwind
00007FFA49743040	导入		kernel32.IsDebuggerPresent
00007FFA49743048	导入		kernel32.UnhandledExceptionFilter

PE 文件结构：



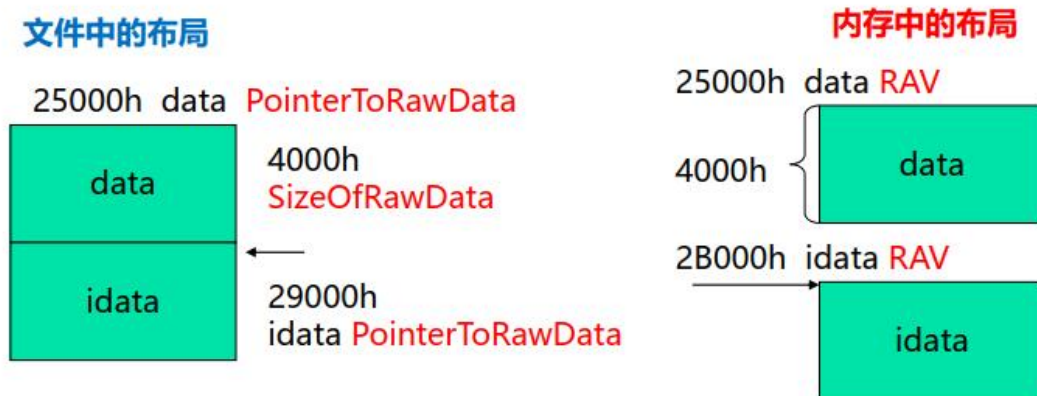
FOA: 文件偏移地址，相对于文件头部的偏移地址

VA: PE 加载到内存中，在进程虚拟地址空间中的地址

RVA (相对虚拟地址空间): PE 加载到内存后， 内存映像中某个位置相对于其 PE 头部的偏移地址。

FOA & RVA: 在 .data 节中， SizeOfRawData 表明 data 节在文件对齐后的大小是 4000h， PointerToRawData 表明该节在文件中的偏移是 25000。把该节加载到内存后，如 果内存对齐方式和文件对齐方式一致，下一个 idata 节的 RVA 应该是 25000+4000=29000; 但 idata 节的 RVA 却是 2B000，在内存中往后移动了，说明内存对齐和文件对

齐方式是不同的，进一步观察 idata 节的 PointerToRawData，可以看见其依然是 29000h

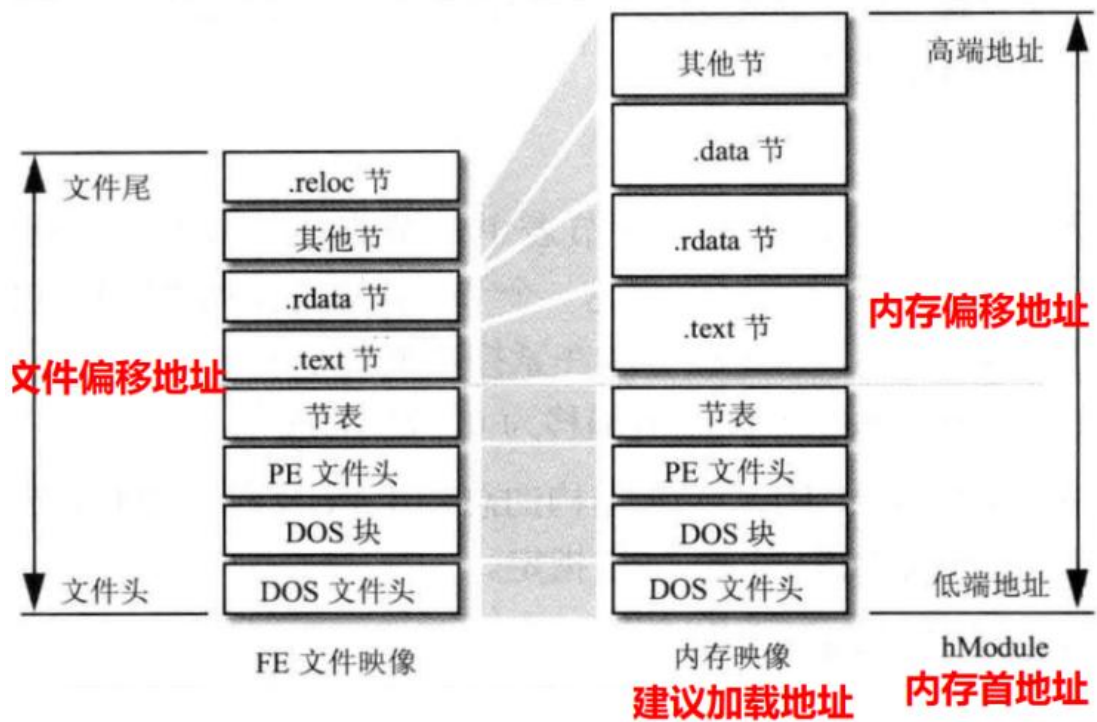


FOA 与 RVA 转换：

FOA 转 RVA： 循环遍历各区段头，找出其文件偏移起始位置 **PointerToRawData** 以及结束位置 **PointerToRawData+SizeOfRawData**；若 FOA 落在某个区段的区间范围内，则用 $FOA - \text{PointerToRawData} + \text{区段起始位置 RVA}$ ，得到其 RVA；

RVA 转 FOA： 循环遍历各区段头，找出该区段起始位置 RVA 以及结束位置 **RVA+SizeOfRawData(或 VirtualSize)**，若 RVA 落在某个区段的区间范围内，则用 $RVA - \text{该区段起始位置 RVA} + \text{区段起始文件偏移位置 PointerToRawData}$ 得到其 FOA

四种地址



PE 格式的三个头:

DOS 头: 该头部的第一个字段 `e_magic` 就是 MZ; 最后一个字段 `e_lfanew` 是偏移量, 就是从文件开始到 PE 文件头 (NT 头) 的偏移量; NT 头处于 DOS 头的后面。虽然 DOS 头长度确定, 但因为 DOS 头后有一小段代码, 由编译器生成, 长度不一, 所以需要 `e_lfanew` 指示 NT 头的起始偏移;


```

TImage_Dos_Header = record
    e_magic: Word;           // Magic number
    e_cblp: Word;            // Bytes on last page of file
    e_cp: Word;              // Pages in file
    e_crlc: Word;            // Relocations
    e_cparhdr: Word;         // Size of header in paragraphs
    e_minalloc: Word;        // Minimum extra paragraphs needed
    e_maxalloc: Word;        // Maximum extra paragraphs needed
    e_ss: Word;              // Initial (relative) SS value
    e_sp: Word;              // Initial SP value
    e_csum: Word;            // Checksum
    e_ip: Word;              // Initial IP value
    e_cs: Word;              // Initial (relative) CS value
    e_lfarlc: Word;          // File address of relocation table
    e_ovno: Word;            // Overlay number
    e_res: array[0..4-1] of Word; // Reserved words
    e_oemid: Word;           // OEM identifier (for e_oeminfo)
    e_oeminfo: Word;         // OEM information; e_oemid specific
    e_res2: array[0..10 - 1] of Word; // Reserved words
    e_lfanew: Cardinal;      // File address of new exe header
end;

```

NT 头:

PE 文件头是 PE 相关结构 IMAGE_NT_HEADERS 的简称，即 NT 映像头，存放 PE 整个文件信息分布的重要字段。 它包含 3 部分：PE 文件标志（Signature）、映像文件头（IMAGE_FILE_HEADER）、可选映像文件头（IMAGE_OPTIONAL_HEADER32）

IMAGE_NT_HEADERS STRUCT

```

Signature dd ?
FileHeader IMAGE_FILE_HEADER <>
OptionalHeader IMAGE_OPTIONAL_HEADER32 <>
IMAGE_NT_HEADERS ENDS

```

判断一个文件是否是 PE 格式:

- 1.先判断文件头 2 字节是否为“MZ”
- 2.判断 NT 头的 Signature 是否为“PE”

NT 头的文件头：利用 `SizeOfOptionalHeader`，就可以知道节表的起始位置；利用 `NumberOfSections`，就可以知道最后一个节表的末尾地址；

```
TImage_File_Header = record
    Machine: Word;                //执行环境及平台
    NumberOfSections: Word;      //文件中节的个数
    TimeDateStamp: Cardinal;      //文件建立时间
    PointerToSymbolTable: Cardinal; //符号表偏移
    NumberOfSymbols: Cardinal;    //符号数目
    SizeOfOptionalHeader: Word;  //可选头长度
    Characteristics: Word;       //标志集合
end;
```

包含PE文件的基本信息

NT 头的可选头：

Option Header

```
TImage_Optional_Header32 = record
    //
    // Standard fields.
    //
    Magic: Word;
    MajorLinkerVersion: Byte;
    MinorLinkerVersion: Byte;
    SizeOfCode: Cardinal;
    SizeOfInitializedData: Cardinal;
    SizeOfUninitializedData: Cardinal;
    AddressOfEntryPoint: Cardinal; //代码入口RVA，第一条指令的RAV
    BaseOfCode: Cardinal;
    BaseOfData: Cardinal;

    //
    // NT additional fields.
    //
```



```

ImageBase: Cardinal;           //载入程序首选的RAV
SectionAlignment: Cardinal;  //节在内存中对齐方式
FileAlignment: Cardinal;     //节在文件中对齐方式
MajorOperatingSystemVersion: Word;
MinorOperatingSystemVersion: Word;
MajorImageVersion: Word;
MinorImageVersion: Word;
MajorSubsystemVersion: Word;
MinorSubsystemVersion: Word;
Win32VersionValue: Cardinal;
SizeOfImage: Cardinal;
SizeOfHeaders: Cardinal;        //所有头加节表的大小，可以作为第一节的文件偏移
Checksum: Cardinal;
Subsystem: Word;                00000088    00001000    Address of Entry Point
DllCharacteristics: Word;       0000008C    00001000    Base of Code
SizeOfStackReserve: Cardinal;   00000090    0000A000    Base of Data
SizeOfStackCommit: Cardinal;    00000094    00400000    Image Base
SizeOfHeapReserve: Cardinal;    00000098    00001000    Section Alignment
SizeOfHeapCommit: Cardinal;     0000009C    00000200    File Alignment
LoaderFlags: Cardinal;
NumberOfRvaAndSizes: Cardinal;  //数据目录的项数
DataDirectory: array[0..IMAGE_NUMBEROF_DIRECTORY_ENTRIES - 1] of
TImage_Data_Directory;         //数据目录表
end;

```

病毒寄生: 将病毒代码写入文件空洞处，为执行病毒有两种方式：
 第一种直接修改入口点；第二种入口点字节码改为跳转指令，跳转到病毒，病毒最后跳转回原入口点。

四、实验环境（设备、元器件）：

个人 PC 机； Windows10 系统； Control.exe

五、实验步骤：

实验一：修改入口点

1. 找到 AddressOfEntryPoint，第一条指令的 RVA ，程序从这里开始执行；在 NT 头--可选头中--AddressOfEntryPoint

RVA	Data	Description	Value
000000E8	010B	Magic	IMAGE_NT_OPTIONAL_HDR32_MAGIC
000000EA	0A	Major Linker Version	
000000EB	00	Minor Linker Version	
000000EC	00000600	Size of Code	
000000F0	0000FA00	Size of Initialized Data	
000000F4	00000000	Size of Uninitialized Data	
000000F8	00001404	Address of Entry Point	
000000FC	00001000	Base of Code	
00000100	00002000	Base of Data	
00000104	00400000	Image Base	
00000108	00001000	Section Alignment	

2. 找到入口 RVA 所在的区段；RVA: 0x1404; $1000 < 1404 < 2000$;

在 .text 节

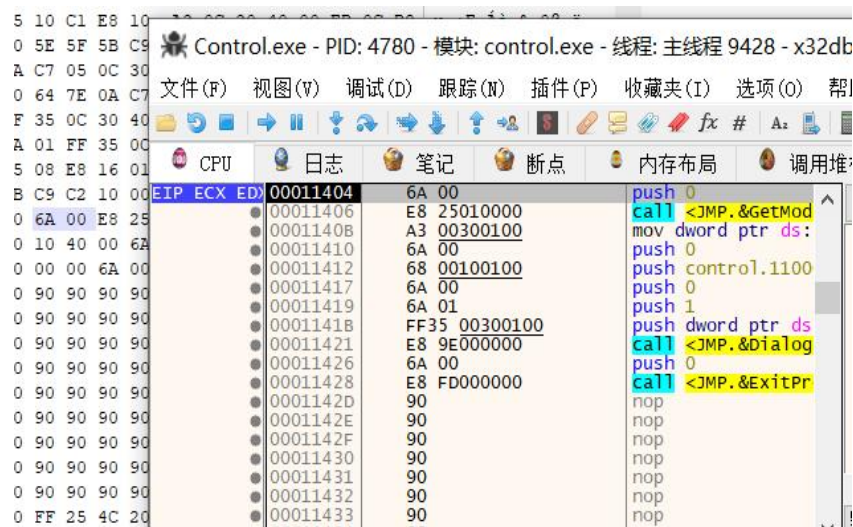
RVA	Data	Description	Value
000001C8	2E 74 65 78	Name	.text
000001CC	74 00 00 00		
000001D0	0000053C	Virtual Size	
000001D4	00001000	RVA	
000001D8	00000600	Size of Raw Data	
000001DC	00000400	Pointer to Raw Data	
000001E0	00000000	Pointer to Relocations	
000001E4	00000000	Pointer to Line Numbers	
000001E8	0000	Number of Relocations	
000001EA	0000	Number of Line Numbers	
000001EC	60000020	Characteristics	
	00000020		IMAGE_SCN_CNT_CODE
	20000000		IMAGE_SCN_MEM_EXECUTE
	40000000		IMAGE_SCN_MEM_READ

3. 确定入口 RVA 的文件位置：RVA 与 FOA 偏移不同；入口 FOA = 入口

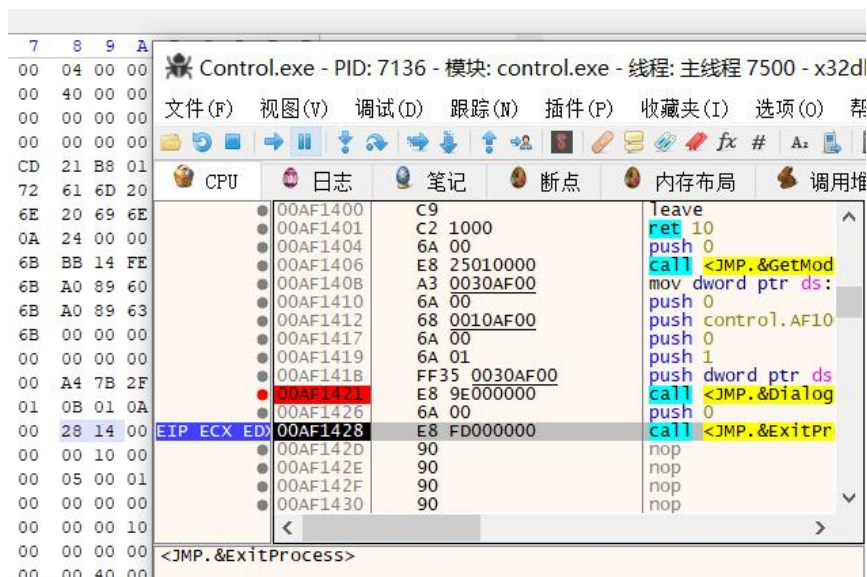
$RVA - RVA + \text{Point to Raw Data} = 0x804$

00000780	75 0D 8B 45 10 C1 E8 1
00000790	01 00 00 00 5E 5F 5B C
000007A0	00 00 7D 0A C7 05 0C 3
000007B0	0C 30 40 00 64 7E 0A C
000007C0	00 6A 00 FF 35 0C 30 4
000007D0	01 00 00 6A 01 FF 35 C
000007E0	6A 6D FF 75 08 E8 16 C
000007F0	00 5E 5F 5B C9 C2 10 C
00000800	C9 C2 10 00 6A 00 E8 2
00000810	6A 00 68 00 10 40 00 6
00000820	00 E8 9E 00 00 00 6A C
00000830	90 90 90 90 90 90 90 9
00000840	90 90 90 90 90 90 90 9
-----	-- -- -- -- -- -- --

4. 查看文件中入口点指令用 x64dbg 启动程序进行验证



5. 入口地址后移 9 条指令；RVA 起始地址变为：1428H；将 F8 所记录的入口点地址改为 28 14；执行的指令字节码为：EB FD 00 00 00；修改入口点成功。



实验二：寄生在 PE 文件最后节的空洞，寄生后文件长度不变

选择的文件是 control.exe

1. 观察最后一个节.reloc：实际大小：122h；对齐大小：200h；说明 reloc 节存在空洞 DEh

Control_old.exe	RVA	Data	Description	Value
IMAGE_DOS_HEADER	00000268	2E 72 65 6C	Name	.reloc
MS-DOS Stub Program	0000026C	6F 63 00 00		
IMAGE_NT_HEADERS	00000270	00000122	Virtual Size	
Signature	00000274	00014000	RVA	
IMAGE_FILE_HEADER	00000278	00000200	Size of Raw Data	
IMAGE_OPTIONAL_HEADER	0000027C	00010000	Pointer to Raw Data	
IMAGE_SECTION_HEADER .text	00000280	00000000	Pointer to Relocations	
IMAGE_SECTION_HEADER .rdata	00000284	00000000	Pointer to Line Numbers	
IMAGE_SECTION_HEADER .data	00000288	0000	Number of Relocations	
IMAGE_SECTION_HEADER .rsrc	0000028A	0000	Number of Line Numbers	
IMAGE_SECTION_HEADER .reloc	0000028C	42000040	Characteristics	
SECTION .text		00000040	IMAGE_SCN_CNT_INITIALIZED_DATA	
SECTION .rdata		02000000	IMAGE_SCN_MEM_DISCARDABLE	
SECTION .rsrc		40000000	IMAGE_SCN_MEM_READ	
SECTION .reloc				

2. 在文件中定位寄生位置：10000 (PointerToRawData) + 122 (VirtualSize) = 10122

3. 在文件中填入寄生代码 EB 02 90 90 90 90 90 90 90 90

00010070	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00010080	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00010090	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
000100A0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
000100B0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
000100C0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
000100D0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
000100E0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
000100F0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00010100	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00010110	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00010120	00 00 EB 02 90 90 90 90	90 90 90 90 90 90 00 00
00010130	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00010140	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00010150	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00010160	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00010170	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00010180	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00010190	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
000101A0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
000101B0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00

4. 修改 PE 文件相关字段

修改 PE 文件的实际大小 (VirtualSize) (地址 270)；由于没有增加新节，所以我们先不用修改 SizeOfRawData 以及 SizeOfImage 字段

00000250	00 F2 00 00 00 0E 00 00	00 00 00 00 00 00 00 00	0	
00000260	00 00 00 00 40 00 00 40	2E 72 65 6C 6F 63 00 00	@	@.reloc
00000270	2D 01 00 00 00 40 01 00	00 02 00 00 00 00 01 00	-	@
00000280	00 00 00 00 00 00 00 00	00 00 00 00 40 00 00 42		@ B
00000290	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00		

5. 修改入口点 RVA (地址 f8)

寄生代码的 RVA = 14000 (reloc 节起始 RVA) + 122 = 14122

原来的入口点 RVA:

```

00000000: 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00
00000010: B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
00000020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000030: 00 00 00 00 00 00 00 00 00 00 00 00 00 D0 00 00 00
00000040: 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68
00000050: 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F
00000060: 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20
00000070: 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00
00000080: FF 75 90 38 BB 14 FE 6B BB 14 FE 6B BB 14 FE 6B
00000090: 35 0B ED 6B A7 14 FE 6B A0 89 60 6B BA 14 FE 6B
000000A0: A0 89 64 6B BA 14 FE 6B A0 89 63 6B BA 14 FE 6B
000000B0: 52 69 63 68 BB 14 FE 6B 00 00 00 00 00 00 00 00
000000C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000D0: 50 45 00 00 4C 01 05 00 A4 7B 2F 64 00 00 00 00
000000E0: 00 00 00 00 E0 00 02 01 0B 01 0A 00 00 06 00 00
000000F0: 00 FA 00 00 00 00 00 00 04 14 00 00 00 10 00 00
00000100: 00 20 00 00 00 00 40 00 00 10 00 00 00 02 00 00
00000110: 05 00 01 00 00 00 00 00 05 00 01 00 00 00 00 00
00000120: 00 50 01 00 00 04 00 00 00 00 00 00 02 00 40 81
00000130: 00 00 10 00 00 10 00 00 00 00 10 00 00 10 00 00

```

修改后的入口点 RVA:

```

00000000: 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00
00000010: B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
00000020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000030: 00 00 00 00 00 00 00 00 00 00 00 00 00 D0 00 00 00
00000040: 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68
00000050: 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F
00000060: 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20
00000070: 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00
00000080: FF 75 90 38 BB 14 FE 6B BB 14 FE 6B BB 14 FE 6B
00000090: 35 0B ED 6B A7 14 FE 6B A0 89 60 6B BA 14 FE 6B
000000A0: A0 89 64 6B BA 14 FE 6B A0 89 63 6B BA 14 FE 6B
000000B0: 52 69 63 68 BB 14 FE 6B 00 00 00 00 00 00 00 00
000000C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000D0: 50 45 00 00 4C 01 05 00 A4 7B 2F 64 00 00 00 00
000000E0: 00 00 00 00 E0 00 02 01 0B 01 0A 00 00 06 00 00
000000F0: 00 FA 00 00 00 00 00 00 22 41 01 00 00 10 00 00
00000100: 00 20 00 00 00 00 40 00 00 10 00 00 00 02 00 00
00000110: 05 00 01 00 00 00 00 00 05 00 01 00 00 00 00 00
00000120: 00 50 01 00 00 04 00 00 00 00 00 00 02 00 40 81
00000130: 00 00 10 00 00 10 00 00 00 00 10 00 00 10 00 00

```

结果:

```

00010060 1A 35 20 35 26 35 2C 35 32 35 38 35 00 00 00 00 5 5&5,52585
00010070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00010080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00010090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000100A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000100B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000100C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000100D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000100E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000100F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00010100 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00010110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00010120 00 00 EB 02 90 90 90 90 90 90 90 90 90 90 90 90
00010130 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00010140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00010150 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00010160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00010170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00010180 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

```

实验三：寄生到非代码区段末尾跳回原入口点

1. 在.rdata 处进行病毒寄生（有空洞，VirtualSize != Size of Raw Data）

	pFile	Data	Description	Value
Control.exe				
IMAGE_DOS_HEADER	000001F0	2E 72 64 61	Name	.rdata
MS-DOS Stub Program	000001F4	74 61 00 00		
IMAGE_NT_HEADERS	000001F8	000002F6	Virtual Size	
Signature	000001FC	00002000	RVA	
IMAGE_FILE_HEADER	00000200	00000400	Size of Raw Data	
IMAGE_OPTIONAL_HEADER	00000204	00000A00	Pointer to Raw Data	
IMAGE_SECTION_HEADER .text	00000208	00000000	Pointer to Relocations	
IMAGE_SECTION_HEADER .rdata	0000020C	00000000	Pointer to Line Numbers	
IMAGE_SECTION_HEADER .data	00000210	0000	Number of Relocations	
IMAGE_SECTION_HEADER .rsrc	00000212	0000	Number of Line Numbers	
IMAGE_SECTION_HEADER .reloc	00000214	40000040	Characteristics	
SECTION .text			00000040	IMAGE_SCN_CNT_INITIALIZED_DATA
SECTION .rdata			40000000	IMAGE_SCN_MEM_READ
SECTION .rsrc				
SECTION .reloc				

2. 调试器调试发现跳转到不知名指令，该段缺少内存可执行属性；

77BC4FDB	EB 03	jmp <ntdll.KiUserExceptionDispatcher>	
77BC4FDD	CC	int3	
77BC4FDE	CC	int3	
77BC4FDF	CC	int3	
77BC4FE0	83D CC69C777 00	cmp dword ptr ds:[77C769CC],0	KiUserExceptionDispatcher
77BC4FE7	74 0E	je ntdll.77BC4FF7	
77BC4FE9	8B0D CC69C777	mov ecx,dword ptr ds:[77C769CC]	ecx:EntryPoint
77BC4FEF	FF15 E091C777	call dword ptr ds:[<RtlDebugPrintTimes>]	ecx:EntryPoint
77BC4FF5	FFE1	jmp ecx	
77BC4FF7	FC	cld	

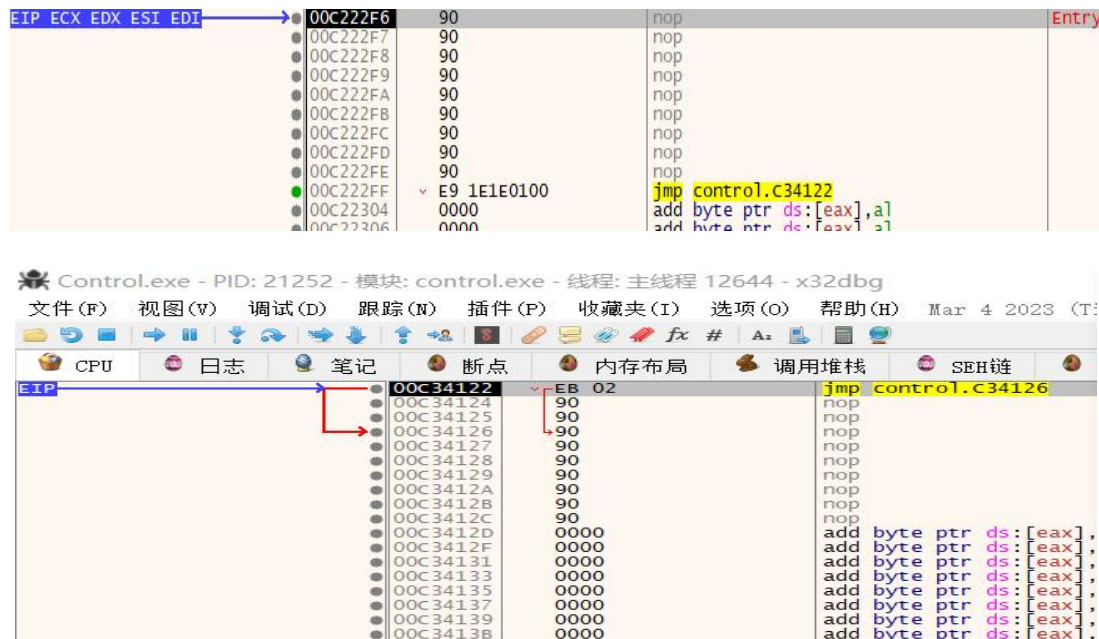
00000190:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000001A0:	00 00 00 00 00 00 00 00 00 20 00 00 60 00 00 00
000001B0:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000001C0:	00 00 00 00 00 00 00 00 00 2E 74 65 78 74 00 00
000001D0:	3C 05 00 00 00 10 00 00 00 06 00 00 00 04 00 00
000001E0:	00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 60
000001F0:	2E 72 64 61 74 61 00 00 F6 02 00 00 00 20 00 00
00000200:	00 04 00 00 00 0A 00 00 00 00 00 00 00 00 00 00
00000210:	00 00 00 00 40 00 00 60 2E 64 61 74 61 00 00 00

3. 插入病毒地址：A00 + 2F6 = CF6；插入指令：9 条 nop + Jmp 回原地；跳回原来的入口点（14122），相对偏移：14122 - （2000 + 2F6 + 9 + 5（jmp 指令本身）） = 11E1E

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000CE0	6C	65	74	65	4F	62	6A	65	63	74	00	00	67	64	69	33
00000CF0	32	2E	64	6C	6C	00	90	90	90	90	90	90	90	90	90	E9
00000D00	1E	1E	01	00	00	00	00	00	00	00	00	00	00	00	00	00
00000D10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000D20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000D30	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

4. 修改入口点（F8 处修改）：2000 + 2F6 = 22F6

结果:



实验四：尾区段寄生之扩展区段大小

1. 在 reloc 节原 VirtualSize 后添加 JMP xx xx xx xx, 跳到 reloc 节后面: JMP 指令的跳转位移量 = 原 SizeOfRawData - (原 Virtualsize + JMP 指令长 (5 字节));

reloc 节偏移: $200 - (122 + 5) = D9$; 在 10122 处写入 e9 d9 00 00 00



2. 在 reloc 节后添加 9 个的 NOP 指令

3. 修改 reloc 节头的 SizeOfRawData, 加一个 FileAlignment

文件粒度:200, 新 sizeofrawdata = old sizeofrawdata + 200 = 200

+ 200 = 400

00000104	00400000	Image Base
00000108	00001000	Section Alignment
0000010C	00000200	File Alignment
00000110	0005	Major O/S Version
00000112	0001	Minor O/S Version
00000114	0000	Major Image Version

00000210:	00 00 00 00 40 00 00 40 2E 64 61 74 61 00 00 00
00000220:	10 00 00 00 00 30 00 00 00 00 00 00 00 00 00 00
00000230:	00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 C0
00000240:	2E 72 73 72 63 00 00 00 28 F0 00 00 00 40 00 00
00000250:	00 F2 00 00 00 0E 00 00 00 00 00 00 00 00 00 00
00000260:	00 00 00 00 40 00 00 40 2E 72 65 6C 6F 63 00 00
00000270:	0E 02 00 00 00 40 01 00 00 04 00 00 00 00 01 00
00000280:	00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 42
00000290:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000002A0:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000002B0:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000002C0:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

4. 修改 reloc 节头的 VirtualSize，新 virtualsize = old sizeofrawdata + 0e (9 个 nop 和 JMP 指令) = 200 + 0e = 20e

00000210:	00 00 00 00 40 00 00 40 2E 64 61 74 61 00 00 00
00000220:	10 00 00 00 00 30 00 00 00 00 00 00 00 00 00 00
00000230:	00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 C0
00000240:	2E 72 73 72 63 00 00 00 28 F0 00 00 00 40 00 00
00000250:	00 F2 00 00 00 0E 00 00 00 00 00 00 00 00 00 00
00000260:	00 00 00 00 40 00 00 40 2E 72 65 6C 6F 63 00 00
00000270:	0E 02 00 00 00 40 01 00 00 04 00 00 00 00 01 00
00000280:	00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 42
00000290:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000002A0:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000002B0:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000002C0:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000002D0:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

5. 修改可选映像头的 SizeOfImage = (relocRVA+新 VirtualSize)除以 SectionAlign 取上整；新 sizeofimage = ceil[(新 virtualsize + reloc 节起始 RVA) / sectionAlignment] = ceil[(20e + 14000) / 1000] = 15000, sizeofimage 大小未改变

00000112	0001	Minor O/S Version
00000114	0000	Major Image Version
00000116	0000	Minor Image Version
00000118	0005	Major Subsystem Version
0000011A	0001	Minor Subsystem Version
0000011C	00000000	Win32 Version Value
00000120	00015000	Size of Image
00000124	00000400	Size of Headers
00000128	00000000	Checksum

6. 在 NOP 后手动增加 FileAlignment-14 个字节，内容不论，为对齐后填充内容

7. 修改入口点 RVA(AddressOfEntryPoint)为寄生代码开始处(.reloc RVA + 原来的 virtualsize = 14000 + 200 = 14200)

RVA	Data	Description	Value
00000268	2E 72 65 6C	Name	.rel
0000026C	6F 63 00 00		
00000270	0000020E	Virtual Size	
00000274	00014000	RVA	
00000278	00000400	Size of Raw Data	
0000027C	00010000	Pointer to Raw Data	

8. 最后加上跳转指令跳转回原来入口点（1404H）

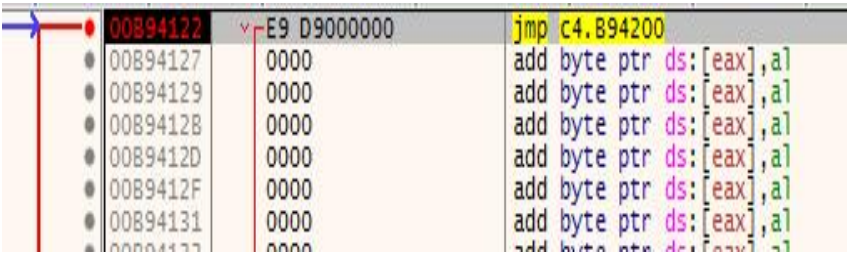
跳转偏移: 0x1404 - (0x14200 + 9 (nop 指令) + 5 (JMP 指令))
= 0xffffed1f6

RVA	Data	Description	Value
000000EC	00000600	Size of Code	
000000F0	0000FA00	Size of Initialized Data	
000000F4	00000000	Size of Uninitialized Data	
000000F8	00001404	Address of Entry Point	
000000FC	00001000	Base of Code	
00000100	00002000	Base of Data	
-----	-----	-----	-----

Enjoy C32asm (HEX)c4.exe	
00010120:	00 00 E9 D9 00 00 00 00 00 00 00 00 00 00 00 00
00010130:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00010140:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00010150:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00010160:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00010170:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00010180:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00010190:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000101A0:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000101B0:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000101C0:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000101D0:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000101E0:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000101F0:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00010200:	90 90 90 90 90 90 90 90 90 90 E9 F6 D1 FE FF 00 00
00010210:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00010220:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00010230:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00010240:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00010250:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

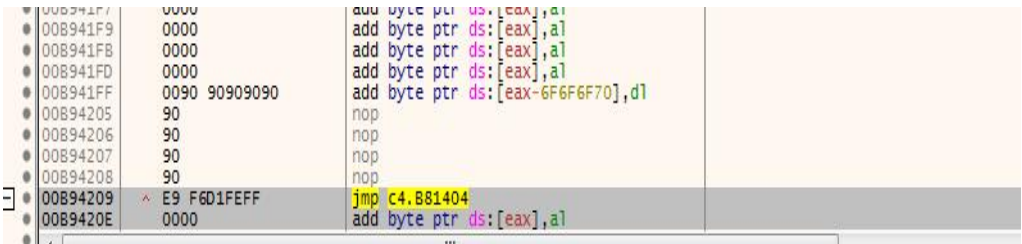
结果：

跳转到文件尾部




Address	Disassembly
00894122	jmp c4.B94200
00894127	add byte ptr ds:[eax], al
00894129	add byte ptr ds:[eax], al
0089412B	add byte ptr ds:[eax], al
0089412D	add byte ptr ds:[eax], al
0089412F	add byte ptr ds:[eax], al
00894131	add byte ptr ds:[eax], al

病毒逻辑执行



Address	Disassembly
008941F9	add byte ptr ds:[eax], al
008941FB	add byte ptr ds:[eax], al
008941FD	add byte ptr ds:[eax], al
008941FF	add byte ptr ds:[eax-6F6F6F70], dl
00894205	nop
00894206	nop
00894207	nop
00894208	nop
00894209	jmp c4.B81404
0089420E	add byte ptr ds:[eax], al

跳转回原来的入口点



Address	Disassembly
00881404	push 0
00881406	call <JMP.&GetModuleHandleA>
00881408	mov dword ptr ds:[B83000], eax
00881410	push 0
00881412	push c4.B81000
00881417	push 0
00881419	push 1
0088141B	push dword ptr ds:[B83000]
00881421	call <JMP.&DialogBoxParamA>
00881426	push 0
00881428	call <JMP.&ExitProcess>
0088142D	nop
0088142E	nop

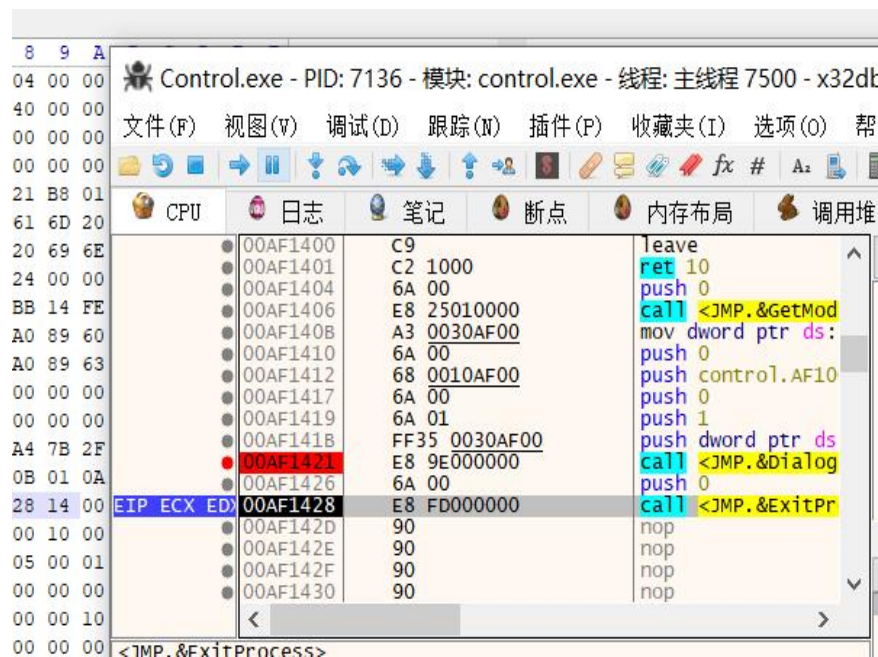
六、实验结论：

实验一：入口点修改

通过修改 RVA 对应的 FOA，改变 PE 文件的入口点，确实可以让程序执行的第一条指令发生变化。

入口地址后移 9 条指令；RVA 起始地址变为：1428H；将 F8 所记录的入口点地址改为 28 14；执行的指令字节码为：EB FD 00 00 00；

修改入口点成功。

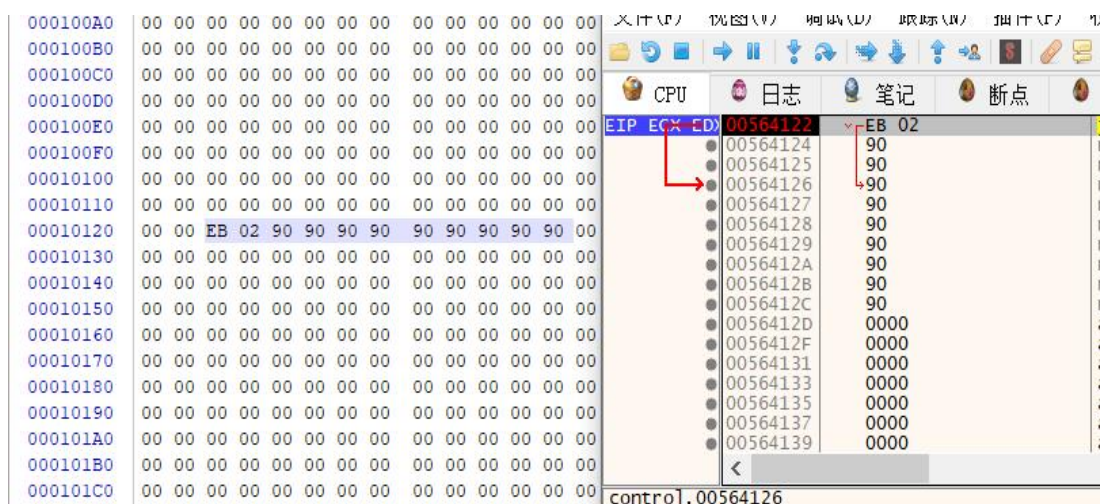


实验二：病毒尾区段寄生实验

病毒在.text 段空洞处寄生，寄生后 VirtualSize 字段改变；对齐后的文件大小 SizeOfRawData 保持不变；修改入口点 RVA 为病毒寄生处的 RVA；在文件中定位寄生位置： $10000 \text{ (PointerToRawData)} + 122 \text{ (VirtualSize)} = 10122$ ；在文件中填入寄生代码 EB 02 90 90 90 90 90 90 90 90 90；修改文件大小为 012D；寄生代码的 RVA = $14000 \text{ (reloc 节起始 RVA)} + 122 = 14122$

00010070	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00010080	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00010090	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
000100A0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
000100B0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
000100C0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
000100D0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
000100E0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
000100F0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00010100	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00010110	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00010120	00 00 EB 02 90 90 90 90	90 90 90 90 90 90 90 90
00010130	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00010140	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00010150	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00010160	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00010170	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00010180	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
00010190	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
000101A0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
000101B0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00

结果

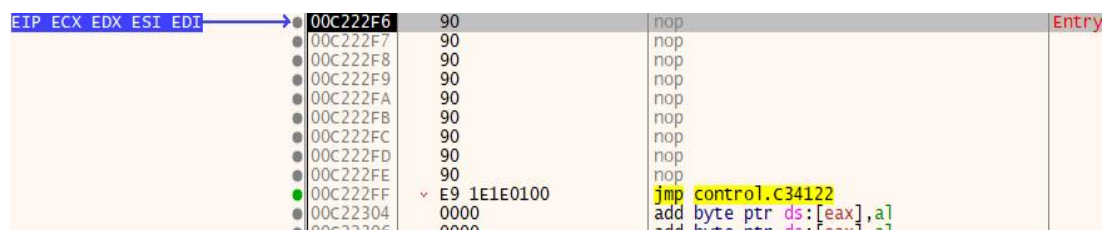


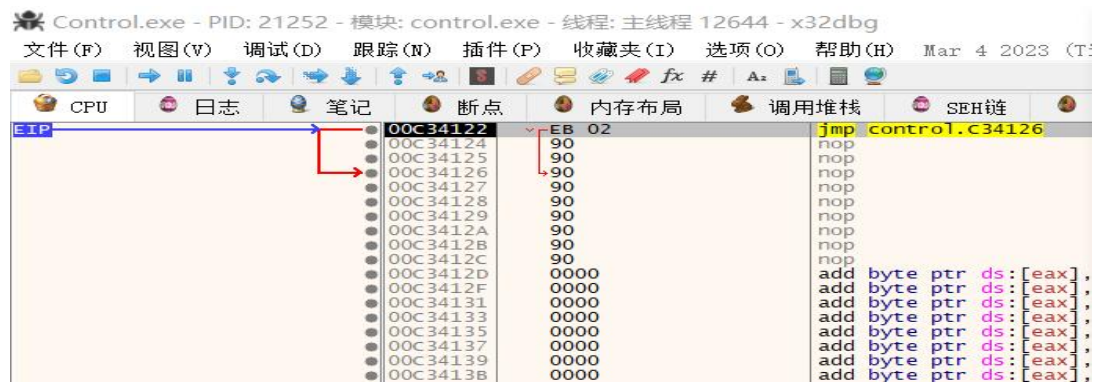
实验三：寄生到非代码区段末尾跳回原入口点

在.rdata 段进行寄生；需要对该段添加内存可执行属性；入口点修改为病毒体 RVA；在寄生位置最后插入 JMP 指令，跳回到原来的入口点；

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000CE0	6C	65	74	65	4F	62	6A	65	63	74	00	00	67	64	69	33
00000CF0	32	2E	64	6C	6C	00	90	90	90	90	90	90	90	90	90	E9
00000D00	1E	1E	01	00	00	00	00	00	00	00	00	00	00	00	00	00
00000D10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000D20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000D30	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

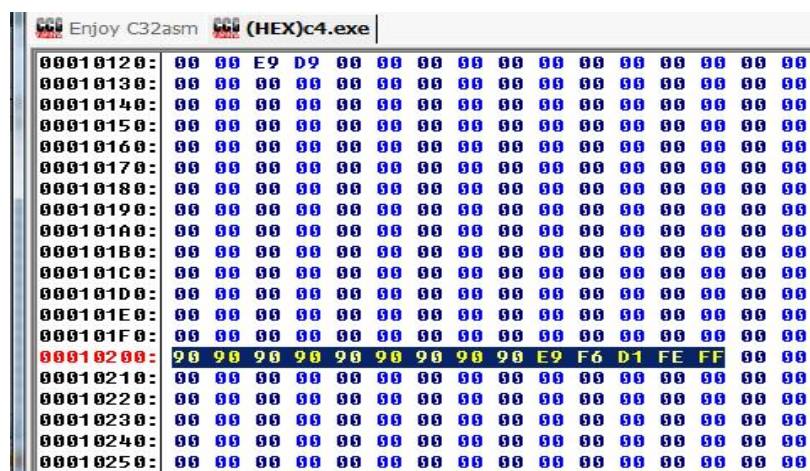
结果：





实验四：尾区段寄生之扩展区段大小

模拟寄生代码大于末节（.reloc）的空洞；需要修改可选映像头的 SizeOfImage，原来的病毒文件增加一个 FileAlignment 大小；VirtualSize 大小改变为原来的 SizeOfRawData + nop 指令 + JMP 指令大小；最后一条指令需要跳回原来的入口点（1404H），跳转偏移：
 $0x1404 - (0x14200 + 9(\text{nop 指令}) + 5(\text{JMP 指令})) = 0xffffed1f6$



结果：

跳转到文件尾部

00894122	E9 D9000000	jmp c4.B94200
00894127	0000	add byte ptr ds:[eax],al
00894129	0000	add byte ptr ds:[eax],al
0089412B	0000	add byte ptr ds:[eax],al
0089412D	0000	add byte ptr ds:[eax],al
0089412F	0000	add byte ptr ds:[eax],al
00894131	0000	add byte ptr ds:[eax],al
00894133	0000	add byte ptr ds:[eax],al

病毒逻辑执行

008941F7	0000	add byte ptr ds:[eax],al
008941F9	0000	add byte ptr ds:[eax],al
008941FB	0000	add byte ptr ds:[eax],al
008941FD	0000	add byte ptr ds:[eax],al
008941FF	0090 90909090	add byte ptr ds:[eax-6F6F6F70],dl
00894205	90	nop
00894206	90	nop
00894207	90	nop
00894208	90	nop
00894209	E9 F6D1FEFF	jmp c4.B81404
0089420E	0000	add byte ptr ds:[eax],al

跳转回原来的入口点

笔记	断点	内存布局	调用堆栈	链	脚本	符号	源代码	引用	线程
00881404	6A 00	push 0							
00881406	E8 25010000	call <JMP.&GetModuleHandleA>							
00881408	A3 00308800	mov dword ptr ds:[883000],eax							
00881410	6A 00	push 0							
00881412	68 00108800	push c4.B81000							
00881417	6A 00	push 0							
00881419	6A 01	push 1							
0088141B	FF35 00308800	push dword ptr ds:[883000]							
00881421	E8 9E000000	call <JMP.&DialogBoxParamA>							
00881426	6A 00	push 0							
00881428	E8 FD000000	call <JMP.&ExitProcess>							
0088142D	90	nop							
0088142E	90	nop							

七、总结及心得体会：

通过这次实验，我对 PE 结构有了深刻的了解，对 NT 可选头有了更加清晰的认识；其中 PE 结构的每个节都有对应的属性，病毒想要执行必须要有内存可执行权限；大部分病毒会寄生在节与节间的空洞处，很难发现；如果寄生病毒过大会进行尾区段的扩展，警示我们如果文件突然间增大需要警惕病毒侵入；虽然这次实验只是对病毒的原理进行模拟，但也让我对病毒的加载和执行过程有了清晰的认识。

八、对本实验过程及方法、手段的改进建议：

希望可以有更多的现实例子和高级语言编程的示例。

报告评分：

指导教师签字：