第3章 PE文件结构分析

3.1 实验概述

本章实验旨在让学生深入了解Windows操作系统下的可执行文件格式(PE/PE+),并通过学习各种PE编辑查看工具,详细了解PE文件的结构和组成部分。通过重点分析PE文件头、引入表、引出表以及资源表等关键部分,学生将深入了解PE文件的内部结构和功能。

在本章中,学生将熟悉各种PE编辑查看工具,从而能够对PE文件进行查看和分析。然后,本章重点分析了PE文件的各部分,包括文件头、节表、导入表、导出表以及资源表等,通过分析这些部分,学生将了解PE文件的组织结构和功能。此外,学生将通过自己动手打造一个尽可能小的PE文件的实践,加深对PE文件格式的理解,并学会如何通过编辑工具来操作和修改PE文件。通过本章实验,学生将能够全面掌握PE文件的结构和格式,为进一步学习逆向工程和漏洞分析奠定坚实的基础。

3.2 实验预备知识与基础

3.2.1 PE查看、编辑与调试工具介绍

本次实验将使用相关工具对PE文件进行查看、编辑与调试。

1. PEview

PEview能够快速简便地查看可移植可执行文件(PE),以及组件对象文件格式(COFF)文件。该工具支持的文件类型包括EXE、DLL、OBJ、LIB、DBG等,可以以树状目录的方式显示文件的头部、节、引入表、引出表和资源等信息,以及更具体的各字段的含义。

2. StudyPE+

StudyPE+是一款国产的PE查看/分析集成工具,支持PE32与PE32+,能够显示PE文件的重要字段(但不像PEview一样显示所有字段)。StudyPE+提供了许多实用的功能,例如丰富的PE编辑功能、RVA FOA互相转换功能、PE反汇编及反汇编编辑功能、PE内多种数据搜索功能、有限的查壳功能等。

3. 010Editor

010Editor 是一款专业的文本和十六进制编辑器，能够快速地编辑计算机上任何文件的内容。该软件可以编辑文本文件，包括 Unicode 文件、批处理文件、C/C++ 、XML 等；而在编辑二进制文件时，010Editor 不仅可以查看和编辑二进制文件的单个字节，还可以基于官网提供的模板对各种类型的文件格式化显示，例如 PE 文件；此外，010Editor 还能对内存中的数据进行编辑。

4. OllyDbg

OllyDbg(www.ollydbg.de)是 Windows 系统下的可视化的用户模式调试器，能够调试 32 位程序。OllyDbg 结合了动态调试与静态分析，它的反汇编能力很强，能够自动分析函数、循环语句、字符串等，可以识别数千个 API，并注释其参数。它具有用户友好的界面，其功能可以由第三方插件扩展。其版本 1.10 为 1.x 系列的最终发布版本。2.0 版本于 2010 年 6 月发布。

3.2.2　函数引入机制

1. 引入函数节

代码复用是程序的重要特性，PE 文件也是如此，PE 文件中使用的函数可能来自其他库，例如 ExitProcess。这种被某模块调用但又不在调用者模块中的函数称为引入函数。PE 文件通过引入函数机制从其他(系统或第三方自定义的)DLL 中引入函数，例如 user32.dll、kernel32.dll 等，存储这种引入函数机制的节称为引入函数节，节名一般为.rdata。图 3-1 展示了引入函数节的结构。

图 3-1　引入函数节

2. 引入机制

函数引入机制主要由 3 个重要的数据结构完成，如图 3-1 所示，分别是 IMPORT Directory Table(IDT)、IMPORT Name Table(INT)、IMPORT Address Table(IAT)。

IDT 是一个 IMAGE_IMPORT_DESCRIPORTs 数组，如图 3-2 所示，每个数组元素对应一个 DLL，以全零的数组元素结束数组。每个数组元素有 5 个 DWORD 大小的项，第 1

pFile	Data	Description	Value
00000614	00002000	Import Name Table RVA	
00000618	00000000	Time Date Stamp	
0000061C	00000000	Forwarder Chain	
00000620	00002072	Name RVA	kernel32.dll
00000624	00002000	Import Address Table RVA	
00000628	00002058	Import Name Table RVA	
0000062C	00000000	Time Date Stamp	
00000630	00000000	Forwarder Chain	
00000634	0000209A	Name RVA	user32.dll
00000638	00002008	Import Address Table RVA	
0000063C	00000000		
00000640	00000000		
00000644	00000000		
00000648	00000000		
0000064C	00000000		

图 3-2　引入函数表

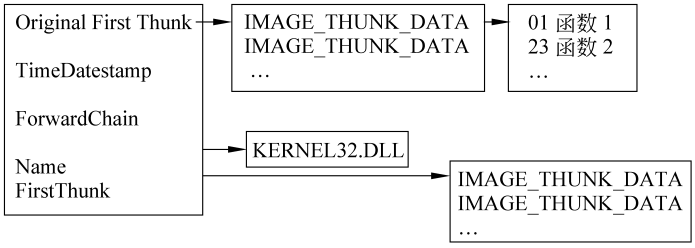
项为 INT 的 RVA(虚拟地址相对偏移),第 5 项为 IAT 的 RVA,第 4 项对应 DLL 名称字符串,告诉加载器这个结构对应的 DLL。

INT 与 IAT 在文件上是相同的,都是一系列的元素大小为 DWORD 的数组,如图 3-3 所示,每个 DWORD 通常是指向引入函数的 Hint 及名称字符串的 RVA(如最高位为 1,即第一字节为 80,则指引入函数的序号),以全零结束。

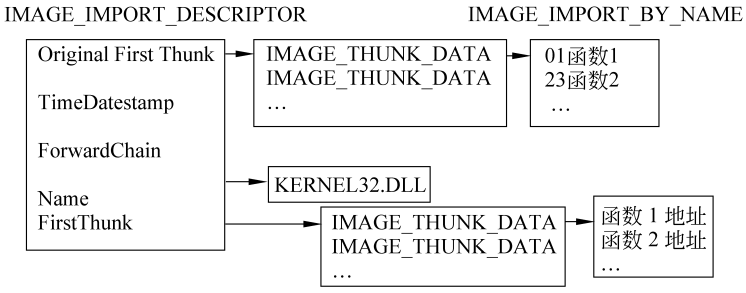
pFile	Data	Description	Value
00000600	00002064	Hint/Name RVA	0000 ExitProcess
00000604	00000000	End of Imports	kernel32.dll
00000658	0000208C	Hint/Name RVA	019D MessageBoxA
0000065C	00002080	Hint/Name RVA	0262 wsprintfA
00000660	00000000	End of Imports	user32.dll

图 3-3　INT/IAT 数据结构

IAT 与 INT 的区别在于,如图 3-4 所示,当 PE 文件加载到内存中时,IAT 中原本存放引入函数名称的 DWORD 会替换为该函数的内存地址,这样 PE 文件运行时可以通过 IAT 在内存中找到相应函数。



(a) 文件中的引入函数表



(b) 内存中的引入函数表

图 3-4　文件与内存中的引入表

3.2.3　函数引出机制

1. 引出函数节

引出函数节的节名一般为.edata,用来描述本文件引出函数的列表等信息及各函数具体代码位置。引出函数节的具体结构如图 3-5 所示。

2. 引出函数机制

引出函数机制主要由 3 个数据结构完成。如图 3-6 所示,AddressOfFunctions(对应 PEview 中的 EXPORT Address Table)存放函数地址,AddressOfNames(对应 PEview 中

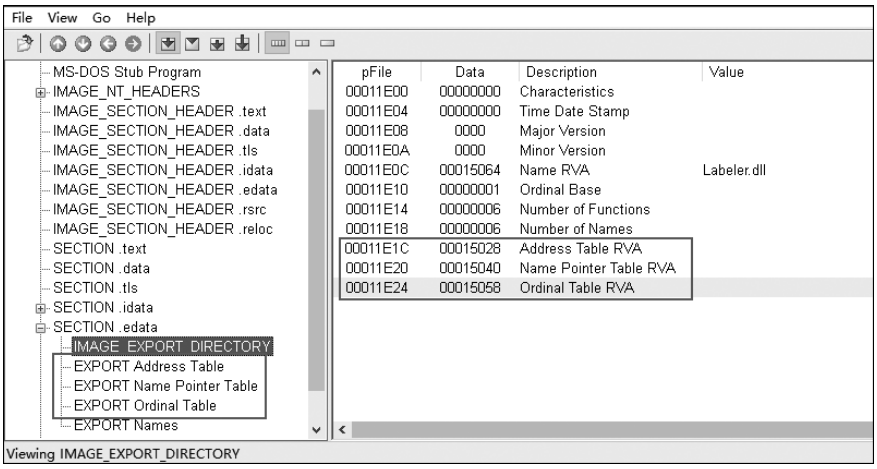


图 3-5　引出函数节具体结构

的 EXPORT Name Pointer Table)存放函数名所在地址，AddressOfNameOrdinals(对应 PEview 中的 EXPORT Ordinal Table)存放每个函数地址在函数地址表中对应的序号。

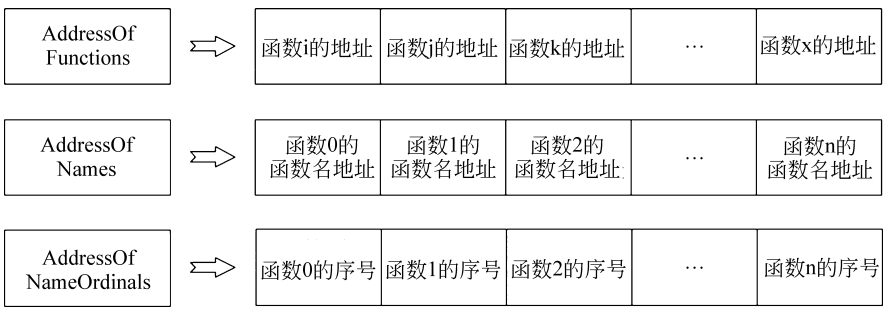


图 3-6　引出函数机制

如果希望通过函数名获取引出函数的地址，需要经过以下流程：

(1) 在 AddressOfNames 找到目标函数的函数名地址，并记下该数组序号 X；

(2) 定位 AddressOfNameOrdinals 的第 X 项，得到序号 Y；

(3) 定位 AddressOfFunctions 的第 Y 项，获得该函数的 RVA 函数地址。

3.2.4　资源节机制

1. 资源节

资源节的节名一般为.rsrc，放有图标、对话框等程序需要的资源。资源节以树状结构组织，它有一个主目录，主目录下又有子目录，子目录下可以是子目录或数据。通常有 3 层目录(资源类型、资源标识符、资源语言 ID)，第 4 层是具体的资源。图 3-7 展示了资源节的树状结构，即资源树。

2. 资源定位机制

资源一般使用树来保存，通常包含 3 层，最高层是类型，其次是名字，最后是语言。资源的定位遵循以下步骤。

(1) 定位资源节开始的位置，首先是一个 IMAGE_RESOURCE_DIRECTORY 结构，

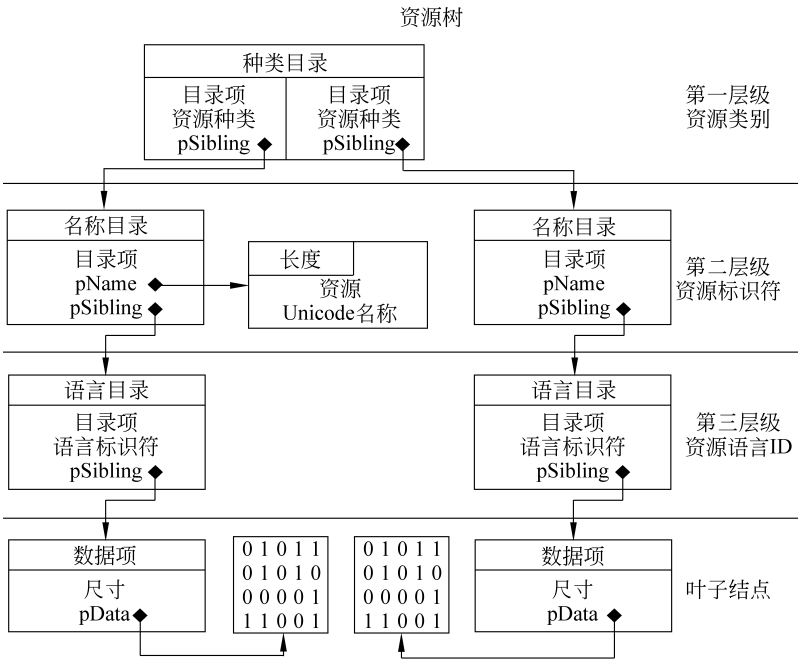


图 3-7　资源节的树状结构

后面紧跟着 IMAGE_RESOURCE_DIRECTORY_ENTRY 数组，这个数组的每个元素代表的资源类型不同。

(2) 通过每个元素，可以找到第二层 IMAGE_RESOURCE_DIRECTORY，后面紧跟着 IMAGE_RESOURCE_DIRECTORY_ENTRY 数组，这个数组的每个元素代表的资源名字不同。

(3) 然后可以找到第三层 IMAGE_RESOURCE_DIRECTORY，后面同样紧跟着 IMAGE_RESOURCE_DIRECTORY_ENTRY 数组，这个数组的每个元素代表的资源语言不同，且直接指向最后的资源(IMAGE_RESOURCE_DATA_ENTRY)。

以上三类 IMAGE_RESOURCE_DIRECTORY 在 PEview 中分别带有 Type、NameID、Language 的后缀，如图 3-8 左侧边栏所示。

(4) 最后通过每个 IMAGE_RESOURCE_DIRECTORY_ENTRY 找到每个 IMAGE_RESOURCE_DATA_ENTRY，从而找到每个真正的资源。

3.2.5　重定位机制

重定位节存放了一个重定位表，定位了代码中使用了绝对地址的地方。若装载器在程序默认的基地址加载映像文件，就不需要重定位，否则需要通过重定位表做一些调整，步骤如下。

(1) 计算地址差异 delta。操作系统加载程序会计算默认的基地址(PE 头的 ImageBase 字段)与实际加载的映像文件的基地址的差异(delta)。

(2) 根据重定位的类型，将这个 delta 应用到重定位表指向的需要修改的地方。

重定位节是一个 IMAGE_BASE_RELOCATION 结构，该结构的每一项如表 3-1

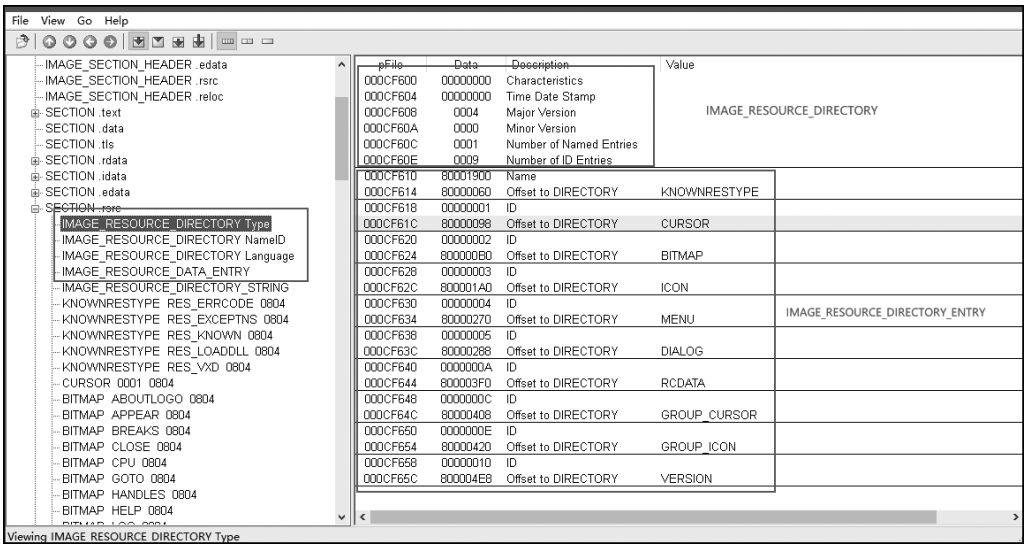


图 3-8　资源节示例

所示。

表 3-1　重定位节的数据结构

顺　序	名　字	大小(字节)	描　述
1	VirtualAddress	4	重定位数据开始的 RVA 地址
2	SizeOfBlock	4	本结构大小
3	TypeOffset[]	不定	重定项数组,每个元素占 2 字节

IMAGE_BASE_RELOCATION 的每项都代表了一个 4K(一页)大小的内存区域中需要重定位的地址。图 3-9 是 kernel32.dll 的重定位表,其中定位项的个数的计算方式为,SizeOfBlock 的大小减去前两项的字节数 8 得到 TypeOffset 数组的大小,再除以 2 得到定位项的个数;定位项每项为 16 位(4 字节),高 4 位代表重定位的类型,剩下的 12 位代表页内的偏移量,加上页地址 VirtualAddress 就得到了具体的内存地址。

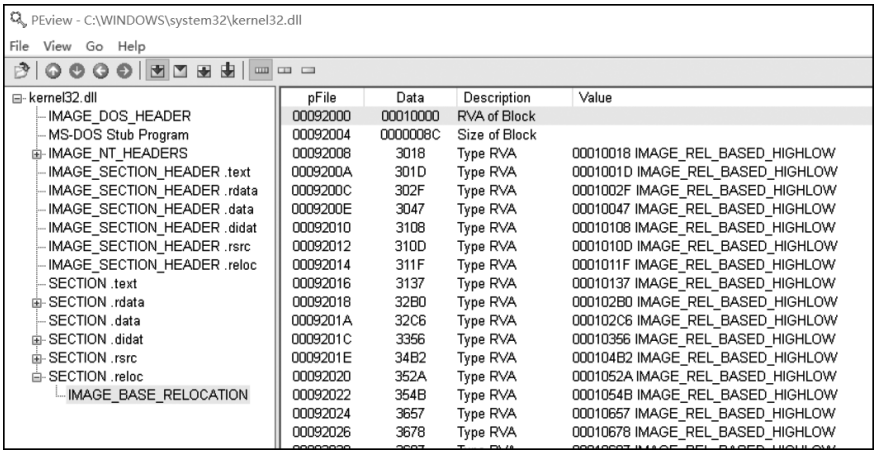


图 3-9　重定位表示例

3.3 PE 查看、编辑与调试工具的用法

3.3.1 实验目的

了解 PE 编辑查看与调试工具的用法，了解 PE 文件在磁盘上的结构与在内存上的结构。

通过本实验，学生能够查看 PE 文件的各部分，包括文件头、节表、导入表、导出表以及资源表等，并可以进行调试和分析，这有助于深入理解 Windows 可执行文件的工作原理和运行机制。

3.3.2 实验内容及实验环境

1. 实验内容

(1) 使用二进制查看工具 PEview 观察 PE 文件例子程序 hello25.exe 的十六进制数据，并定位其中重要数据结构。

(2) 使用 StudyPE+观察 PE32+格式的目标程序 pe32+.exe，了解 32 位 PE 程序与 64 位 PE 程序的差异。

(3) 使用 OllyDbg 对 hello25.exe 进行初步调试，初步了解 OllyDbg 的用法，理解该程序功能结构，在内存中观察该程序的完整结构。

(4) 使用 PE 编辑工具 010Editor 修改 hello25.exe，使得该程序仅弹出第二个对话框。

2. 实验环境

(1) 系统：Windows XP 版本及以上的操作系统，实机、虚拟机均可。

(2) 工具：OllyDbg1.10、010Editor、PEview、StudyPE+。

3.3.3 实验步骤

1. 观察 PE 文件示例程序 hello25.exe 的十六进制数据

(1) 用 PEview 打开示例程序。

使用 PEview 打开示例程序 hello25.exe(通过菜单栏的 File→Open 打开示例程序或者直接将示例程序拖入打开的 OllyDbg 中)，如图 3-10 所示，左侧的目录中显示了 hello25.exe 的结构，单击即可查看；右侧是对应的十六进制数据。

可以看到，该 PE 文件由 MZ 头部(DOS_HEADER)、DOS Stub、PE 文件头(NT_HEADER)、可选文件头、节表、节组成，其中节分为代码节(.text)，引入函数节(.rdata)与数据节(.data)。

(2) 观察 PE 文件头。

PEview 左侧目录显示，PE 文件头由签名(0x4550，即 PE)、文件头与可选文件头组成。在目录中单击各组成部分可查看详细信息，例如单击可选文件头，如图 3-11 所示，右侧显示了可选文件头的各个字段，包括入口点、映像基址、对齐粒度等重要的文件信息。

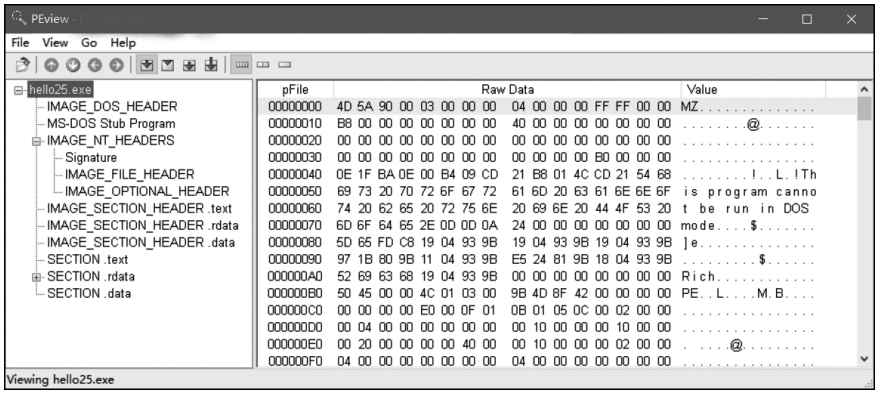


图 3-10　PEview 查看示例程序 hello25.exe

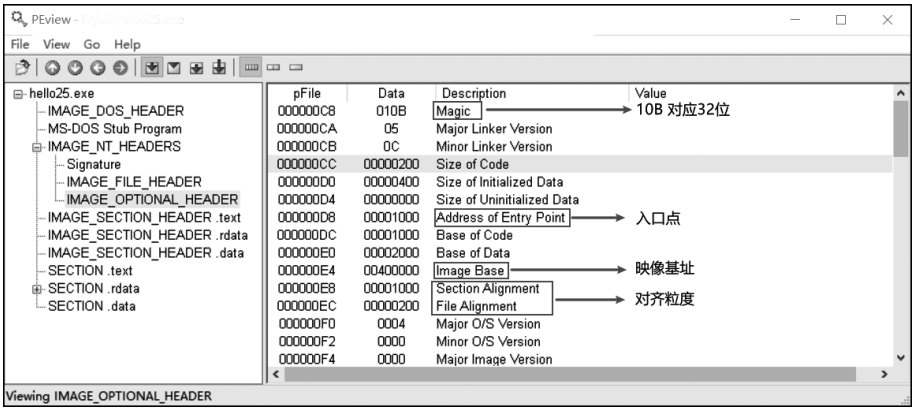


图 3-11　hello25.exe 的 PE 文件头

(3) 观察引入函数节。

引入函数节(.rdata)是 PE 文件的重要数据结构。展开查看该节,如图 3-12 所示,引入函数节包含引入地址表、引入目录表、引入名字表等内容。

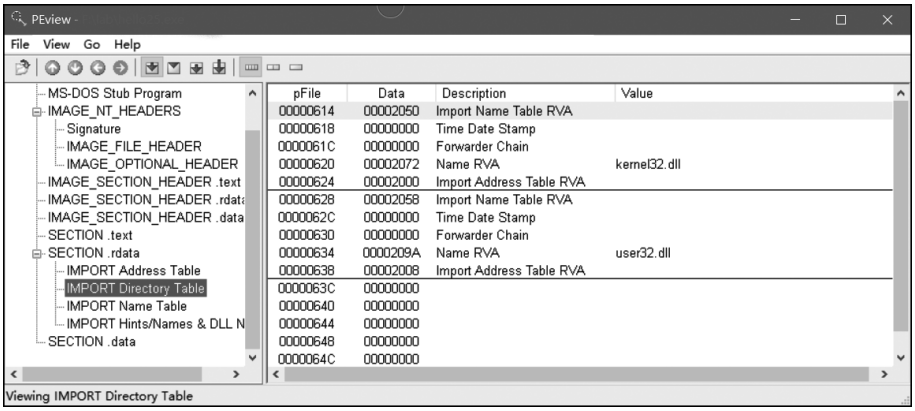


图 3-12　hello25.exe 的引入函数节

2. 查看 PE32+文件结构与 PE32 的差异

PE32＋是 64 位 Windows 所使用的文件格式,在 PE 格式的基础上做了一些简单的修

改。虽然 PEview 能够很直观地展示各字段的意义，但这个工具不支持 PE32＋格式，所以接下来使用 StudyPE＋观察 PE32＋格式。将程序 pe32＋.exe 拖到打开的 StudyPE＋中，即可观察该程序的重要信息，如图 3-13 所示，单击“PE 头”标签，可以查看 PE 头的重要字段。

图 3-13　StudyPE＋打开 pe32＋.exe

PE32＋文件结构与 PE32 的差异主要有 3 方面。

（1）Magic。PE32＋中的 Magic 值为 020B，PE32 中的 Magic 值为 010B，PE 装载器通过检查该字段值来判断文件是 64 位还是 32 位。

（2）BaseOfData。图 3-13 中可以看到，PE32＋中删除了该字段，在 32 位 PE 文件格式中该字段表示指向数据段开头位置（RVA）。

（3）字段大小变化。PE32＋中共 5 个字段由 4 字节拓展为 8 字节，来表达更大的内存范围。例如图 3-13 中的 ImageBase 字段，除此之外还有堆栈相关的 4 个字段：SizeOfStackReverse、SizeOfStackCommit、SizeOfHeapReverse、SizeOfHeapCommit。

3. 初步调试该程序

（1）用 OD 打开程序。

使用 PE 调试工具 OllyDbg 打开示例程序 hello25.exe（通过菜单栏的 File→Open 打开示例程序或者直接将示例程序拖入打开的 OllyDbg 中），图 3-14 展示了程序加载后的 OllyDbg 界面，主要分为 4 个区域，左上角是反汇编界面，显示了程序的反汇编代码；右上角是寄存器界面，显示各寄存器的值；左下角显示程序内存的值；右下角显示栈的内容。

（2）单步步过到第一个弹框。

接下来让程序单步运行到第一个弹框，快捷键 F8 代表单步步过（每次执行一条指令，不跟踪到调用内部），这个过程中观察程序的行为：首先压栈 4 个参数，第一个参数代表弹框类型，第二、三个参数分别为对话框的标题字符串地址与内容字符串地址；再调用 user32.dll 中的弹框函数 MessageBoxA。运行到这里就会出现图 3-15 所示的弹框。

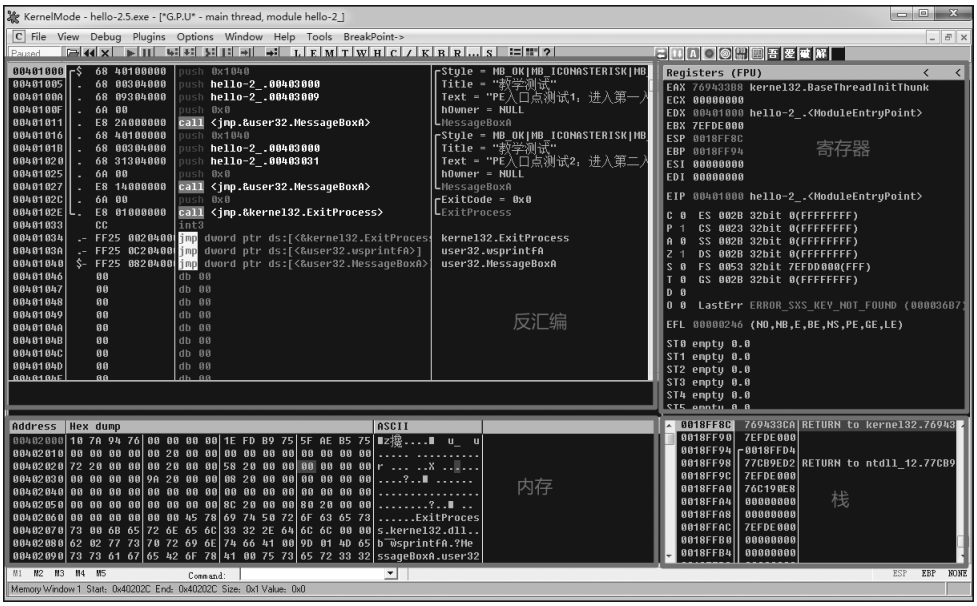


图 3-14　OllyDbg 打开 hello25.exe

(3) 单步运行到程序结束。

继续单步运行,接下来的代码是第二个 MessageBoxA 弹框,如图 3-16 所示,与第一个弹框标题相同但内容不同。反汇编代码中也体现了这一点,第一个弹框压入的文本参数为 0x403009,第二个弹框压入的文本参数则是 0x403031(如果在调用 MessageBoxA 函数的位置按快捷键 F7 来单步步入,跟踪到函数内部,会发现程序停在了连续的 JMP 指令中,这些 JMP 指令指向了函数的 IAT 表,来获得函数的内存地址)。

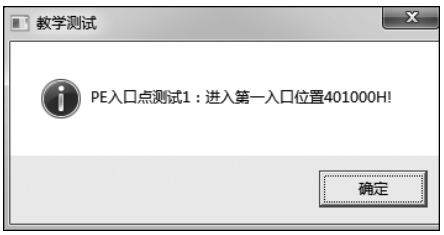


图 3-15　hello25.exe 的第一个弹框

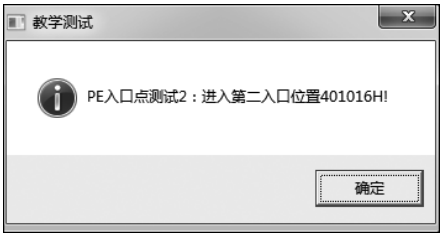


图 3-16　hello25.exe 的第二个弹框

继续单步,调用 ExitProcess 函数结束运行。按快捷键 Ctrl+F2 可以重新运行程序,帮助再次分析。

4. 修改该程序,使该程序仅弹出第二个对话框

使程序仅弹出第二个程序可以有许多方法,可以修改程序的入口点(Address of Entry Point),使程序加载后从第二个弹框处开始运行。

(1) 定位入口点。

首先需要知道入口点字段在什么位置。010Editor 官网提供了 PE 模板,可以辅助分析 PE 文件字段;也可以用 PEview 打开一份程序副本,查看入口点位置,如图 3-17 所示,可见入口点在 D8 位置。

pFile	Data	Description
000000C8	010B	Magic
000000CA	05	Major Linker Version
000000CB	0C	Minor Linker Version
000000CC	00000200	Size of Code
000000D0	00000400	Size of Initialized Data
000000D4	00000000	Size of Uninitialized Data
000000D8	00001000	Address of Entry Point
000000DC	00001000	Base of Code
000000E0	00002000	Base of Data

图 3-17　PEview 查看 hello25.exe 的入口点

(2) 修改入口点。

在前面的调试过程中(图 3-14),我们知道了第二个弹框的开始内存地址为 0x00401016,所以使用 010Editor 打开程序,将入口点(D8 处)的值修改为第二个弹框的 RVA,即 0x00001016,如图 3-18 所示。

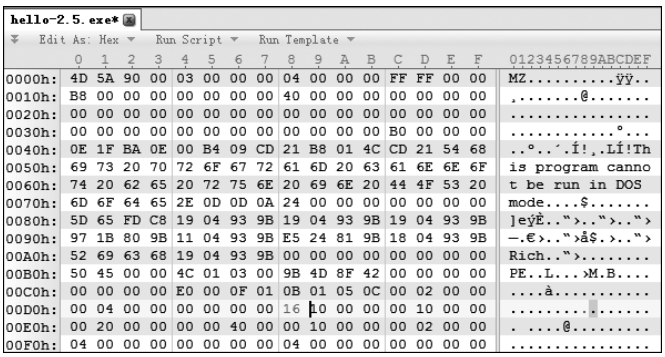


图 3-18　修改 hello25.exe 的入口点为 0x00001016(D8 处的 4 字节)

(3) 确认修改效果。

保存文件之后重新运行程序,可见只弹出了第二个对话框。再次用 OllyDbg 打开程序,如图 3-19 所示,程序代码从第二个弹框开始。

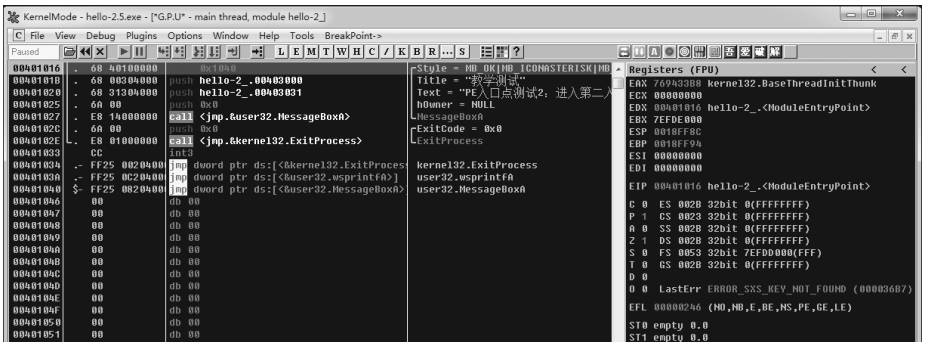


图 3-19　OD 调试修改入口点的 hello25.exe

除了快捷键调试程序，OllyDbg 还在菜单栏下方提供了图形按钮进行调试，如图 3-20 所示，标注的 4 个按钮的功能分别为重新运行程序、关闭程序、单步步入、单步步过。

图 3-20　OD 中常用的图形调试按钮

3.4　函数的引入/引出机制分析与修改

3.4.1　实验目的

熟悉 PE 文件头部、引入表的结构，了解与使用引入表引入函数，了解与使用 user32 的函数手动引入函数。本实验可以让学生深入了解 Windows 可执行文件的内部机制，并为软件开发和逆向工程提供重要的基础，同时有利于学生理解恶意软件脱壳及进行 API 函数定位的机制。

3.4.2　实验内容及实验环境

1. 实验内容

（1）使用 PE 查看工具 PEview 与调试工具 OllyDbg，结合预备知识与示例程序 hello25.exe，熟悉 PE 文件引入表结构，熟悉函数导入的基本原理。

（2）从 hello25.exe 的内存空间找到模块 user32，进一步查找函数 MessageBoxA 的地址，并验证该地址是否正确。通过实例了解 PE 文件如何引入外部模块的函数。

（3）用二进制编辑工具修改 hello25.exe 程序的引入表，使该程序仅可以从 kernel32.dll 中引入 LoadLibrary 和 GetProcAddress 函数，而不从 user32.dll 导入任何函数，在代码节中写入部分代码利用这两个函数获取 MessageBoxA 的函数地址，使 hello25.exe 程序原有功能正常。

2. 实验环境

（1）系统：Windows XP 以上的操作系统，实机、虚拟机均可。

（2）工具：OllyDbg1.10、010Editor、PEview。

3.4.3　实验步骤

1. 观察示例程序 hello25.exe 的函数导入机制

（1）文件中的 IAT。

程序通过 IAT 定位函数的地址，PE 文件的 IAT 在文件中与内存中含义有区别。hello25.exe 示例程序在文件中的 IAT 如图 3-21 所示，通过函数名称引入了函数 ExitProcess、MessageBoxA 与 wsprintfA（也可以通过序号引入函数）。

（2）内存中的 IAT。

而 PE 文件运行过程中，如何在内存中找到函数地址呢？我们通过 OllyDbg 来观察。用 OD 打开 hello25.exe，如图 3-22 所示，在代码节尾部存放着通过 IAT 表间接跳转到引入

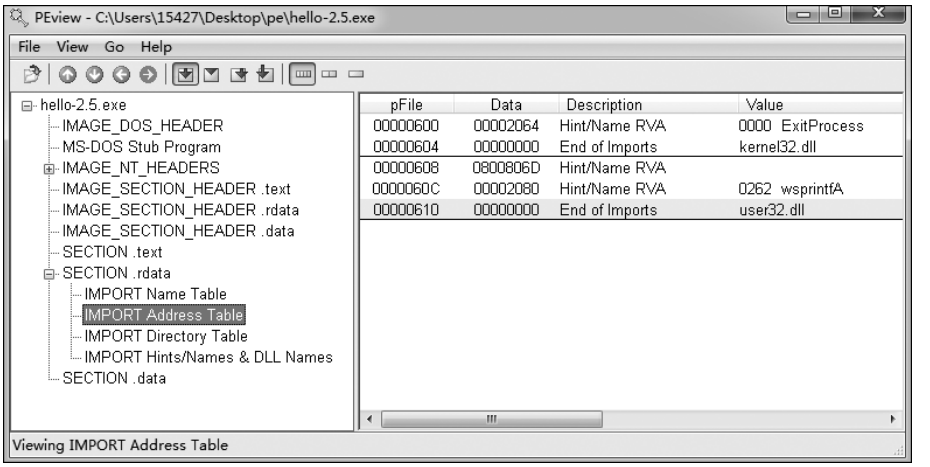


图 3-21　hello25.exe 的 IAT 表

函数的指令，这就是跳转表。图 3-22 中用方框圈出了跳转表跳转的地址，分别是 0x402000、0x40200c、0x402008，换算为文件偏移 FA 就是 0x600、0x60c、0x608，正好是图 3-21 显示的 IAT 的函数地址。

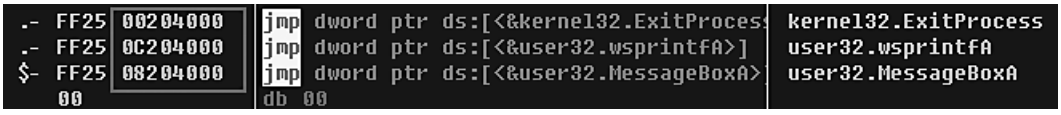


图 3-22　hello25.exe 的跳转表

(3) IAT 的函数地址变化。

下面转到 IAT 的内存处查看，如图 3-23 所示，此时内存中 IAT 的元素与文件 IAT 的元素不同，内存中 IAT 的元素是函数的内存地址，而不是指向函数名的 RVA。这样跳转表可以根据 IAT 找到函数在内存中的真实地址。

图 3-23　内存中的 IAT 表

2. 从该程序的内存空间定位查找 user32.dll 的函数 MessageBoxA 的地址

(1) 定位模块 user32.dll。

要从程序的内存空间找到 user32 再找到 MessageBoxA 地址，首先要找到 user32 模块的内存地址。如图 3-24 所示，单击 M(即 Memory)方块查看内存中的模块，然后找到 user32 模块，基址为 0x75b30000，双击即可进入该地址。

(2) user32 的引出表。

弹出的 Dump 窗口显示了 user32 的头部信息。向下翻可以获取数据目录项中的引出表的地址，如图 3-25 所示，引出表 RVA 为 0x10548，加上基址，引出表的 VA 就是 0x75B40548。

图 3-25 的 Dump 不适合查看内存，因此在 OllyDbg 的内存窗口中继续分析。单击上方的 C(即 CPU)方块切换到之前的窗口，在内存窗口中按快捷键 Ctrl+G 跳转到 user32 的引

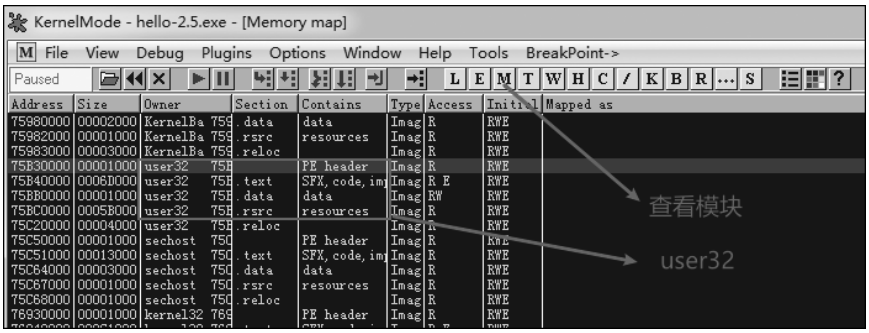


图 3-24　OD 查看程序的模块

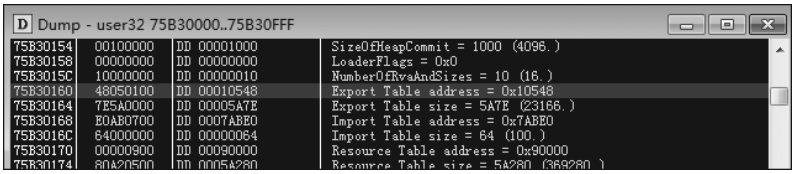


图 3-25　user32 的 Dump 界面

出表，地址为 0x75B40548，如图 3-26 所示。

引出表的结构见图 3-26。图 3-26 中地址 0x75B40548 处是 user32 的引出表目录，图中数据从 0 开始计数，偏移 0C 处的 2 个 DWORD 分别为函数名的起始 RVA（所有函数的名称字符串），起始函数序号 0x5dc；偏移 1C 处的 3 个 DWORD 分别是函数地址表、函数名指针表、函数序号表的 RVA。

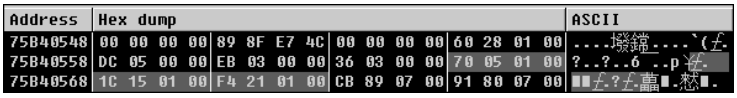


图 3-26　内存中查看 user32 的引出表

（3）定位 MessageBoxA 的字符串。

首先找到 MessageBoxA 字符串的地址。根据所有名称字符串的 RVA（0x12860），跳转到相应的 VA（0x75b42860），按快捷键 Ctrl＋B 搜索字符串 MessageBoxA，得知 MessageBoxA 字符串的地址为 0x75B44AF8，如图 3-27 所示。

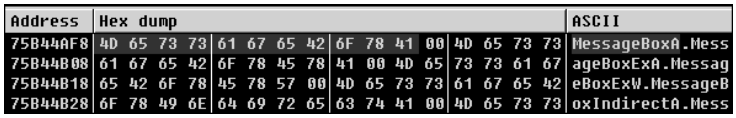


图 3-27　内存中查找 user32 的 MessageBoxA 字符串

（4）计算 MessageBoxA 的项数。

然后计算 MessageBoxA 在函数名指针表中的表项位置。根据函数名指针表 RVA（0x1151c），跳转到 VA（0x75b4151c）。此时需要搜索的是 MessageBoxA 字符串的 RVA，即 0x14af8，因此搜索 HEX 值“F8 4A 01 00”，得到相应地址 0x75b41d54，如图 3-28 所示。因为（0x75b41d54－0x75b4151c）/4 为 526，所以 MessageBoxA 是第 526 项。

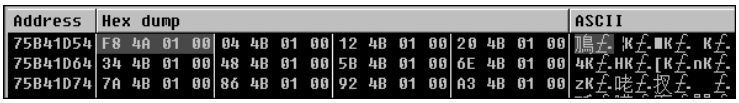


图 3-28　函数名指针表中查找对应 MessageBoxA 的项

(5) 计算 MessageBoxA 的函数序号。

接着,找到 MessageBoxA 的函数序号。根据函数序号表 RVA(0x121f4),跳转到序号表的第 526 项(0x75b42610＝0x75b421f4＋2×526,序号表元素大小为 2 字节),得到 MessageBoxA 的函数序号为 0x21b,如图 3-29 所示。

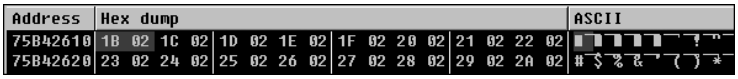


图 3-29　函数序号表中查找 MessageBoxA 对应序号

(6) 获取 MessageBoxA 的 RVA。

最后,函数地址表第 0x21b 项就是 MessageBoxA 的 RVA。根据函数地址表 RVA(0x10570),跳转到目的 VA(0x75b40ddc＝0x75b40570＋4×0x21b),得到 MessageBoxA 函数的 RVA 为 0x6fd1e,如图 3-30 所示。

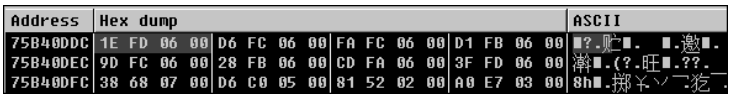


图 3-30　MessageBoxA 函数的 RVA

(7) 验证函数地址。

现在验证寻找的函数地址是否正确。手动寻找的 MessageBoxA 的 RVA 为 0x6fd1e,则 VA 为 0x75b9fd1e;在反汇编窗口中单击跳转表中的 MessageBoxA 函数,图 3-31 中的方框提示了函数地址也是 0x75b9fd1e,两者一致,说明地址正确。

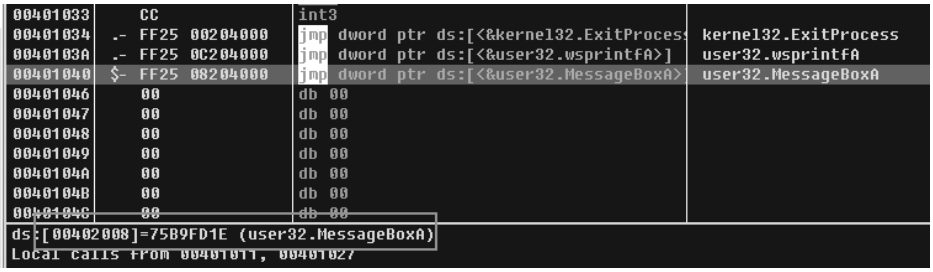


图 3-31　跳转表中的 MessageBoxA 的地址

3. 修改该程序,使该程序使用 LoadLibrary 和 GetProcAddress 导入函数 MessageboxA

(1) 修改引入表。

程序的引入表中没有 LoadLibrary、GetProcAddress 这两个函数,因此先使用 010Editor 修改 PE 文件引入表,才能在代码中使用这两个函数。

① 修改字符串。添加两个函数的字符串,如图 3-32(注意函数 LoadLibrary 的字符串是 LoadLibraryA)。

② 修改 IDT。函数 LoadLibrary 和 GetProcAddress 都在 kernel32.dll 中，程序需要的另一个函数 ExitProcess 也在 kernel32 中，所以程序只需要从 kernel32 中引入所需函数，引入表原来的 user32 部分可以去掉，改为全零表示 IDT 结束。

③ 修改 IAT 与 INT。IAT 与 INT 在文件中完全一致，这里以 IAT 举例。IAT 在 0x600 的位置开始，0x604 的 DWORD 引入 LoadLibrary。LoadLibrary 字符串的地址是 0x682，但字符串前两字节表示 hit(hit 的值不影响引入)，所以引入的起始地址应为 0x680，对应 RVA2080。对 GetProcAddress 以同样的方式引入，然后以全零 DWORD 结束 IAT。

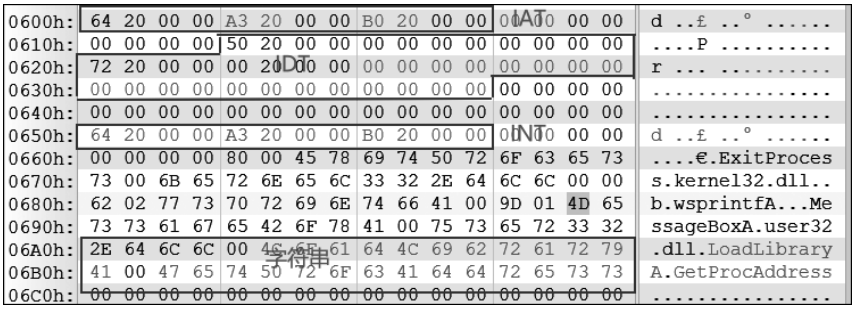


图 3-32　修改 hello25.exe 的引入表

(2) 修改程序，手动加载函数 MessageBoxA。

成功引入函数之后，接下来使用 OD 修改代码，使程序通过 LoadLibrary 与 GetProcAddress 引入函数 MessageBoxA。这里相当灵活，同学们可以用各种方法实现实验目的，例如修改入口点在原代码尾部，引入 MessageBoxA 后再跳转到原代码开头。本书采用另一种方法，完全重写代码，先引入函数 MessageBoxA，再调用该函数弹框，最后结束程序。使用 OD 打开修改好引入表的程序，如图 3-33 所示。

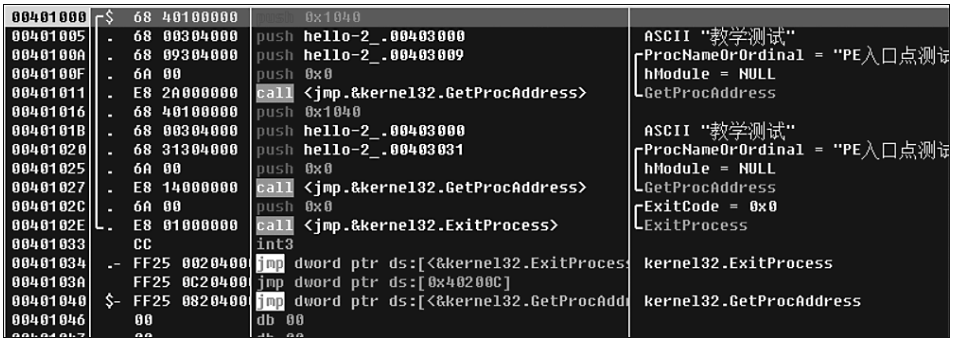


图 3-33　OD 打开修改了引入表的 hello25.exe

双击汇编代码可以对代码进行修改，首先压栈字符串 user32.dll，然后调用 LoadLibrary (这里直接 call IAT 中的函数地址)；接着调用 GetProcAddress 加载 MessageBoxA 的地址，用寄存器 ebx 保存；然后是两次弹框代码与结束进程。可以反复单步调试与重新运行来检查代码的正确性，如图 3-34 所示。

(3) 运行程序。

反汇编窗口右击，选择“Copy to Executable”(复制到可执行文件)→“All Modifications”(所有修改)，可以在修改后保存为可执行程序。保存后运行，确认功能正常。

图 3-34　在 OD 中修改代码，手动加载 MessageBoxA

3.5　资源节资源操作

3.5.1　实验目的

了解 PE 文件资源节结构，能够修改 PE 文件图标，汉化 PE 文件。

通过本实验，学生将深入了解 PE 文件的资源节结构，包括资源的组织方式和存储位置。这不仅有助于学生理解 PE 文件中资源的存储和调用方式，同时也能帮助学生理解病毒经常采用的资源寄生与图标替换等机制。

3.5.2　实验内容及实验环境

1. 实验内容

（1）用二进制编辑工具修改 PEview.exe，使得该文件的图标变成 csWhu.ico（图标见光盘，也可以是任意 ICO 图标文件）。

（2）熟悉 eXeScope 工具的使用，并利用该工具汉化程序 PEview.exe。

2. 实验环境

（1）系统：Windows XP 及以上的操作系统，实机、虚拟机均可。

（2）工具：OllyDbg1.10、010Editor、PEview、eXeScope。

3.5.3　实验步骤

1. 修改程序 PEview.exe 的图标

修改程序的图标就是将程序资源节中的图标数据进行修改，因此需要了解资源节的结构和图标的结构，然后进行修改。

（1）查看图标。

用 010Editor 打开要替换的图标 csWhu.ico，如图 3-35 所示。前面 6 字节是图标的头部，3 个 WORD 分别是保留部分、类型、数量。接着是图标的目录项，可知图标大小为 32×32，由于只有一个图标，0x16 开始到文件末尾是图标数据。

（2）查看 PEview 的图标资源。

图 3-35　图标 csWhu.ico 的十六进制数据

PE 文件中的图标保存格式与 ICO 文件中图标的保存格式略有不同。PE 文件中，把 ICON 目录项和图标资源作为两种资源类型分别保存，前者是 GROUP_ICON 类型，后者是 ICON 类型。用 PEview 打开要修改的 PEview，GROUP_ICON 如图 3-36 所示，可见该文件包含 3 个图标数据(对应文件夹的不同大小的视图)，其中 20×20 的图标刚好与 csWhu.ico 大小一致，所以可以对第二个图标进行替换。

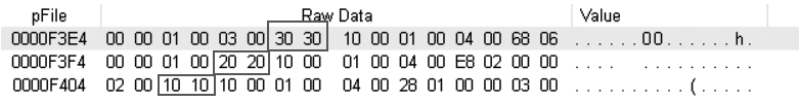


图 3-36　PEview 的 GROUP_ICON

找到二号图标 ICON 0002，如图 3-37 所示，该图标从 0xDB28 开始，到 0xDE0F 结束(图 3-37 中未显示)。

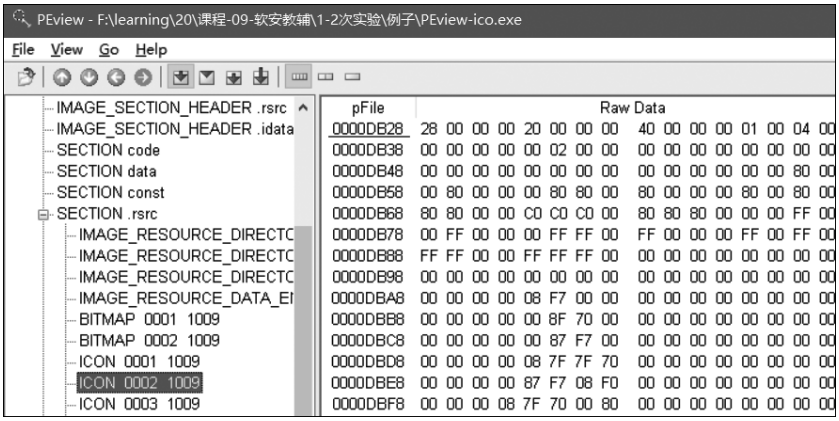


图 3-37　PEview 的图标资源

(3) 替换图标。

现在被替换的数据与替换的数据都已确认，用 010Editor 打开图标和程序，将图标数据复制到程序中(使用快捷键 Ctrl＋Shift＋C、Ctrl＋Shift＋V)并保存。如图 3-38 所示，切换到列表视图就可以看到被修改的图标。

图 3-38　替换后的 PEview

2. 汉化该程序

汉化程序就是将对应的英文字符串改为中文。实验使用的 eXeScope 可以很方便地辅助汉化。

用 eXeScope 打开 PEview，如图 3-39 所示，左边树状结构中，Resource→Menu→1 是程

序目录的字符串。

把“&”后的英文修改为中文(双击即可修改),如图 3-40 所示。

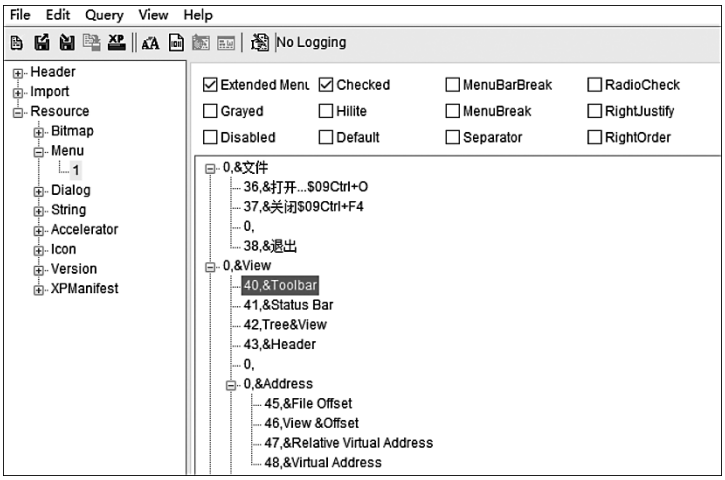


图 3-39　exscope 打开 PEview

图 3-40　汉化后的 PEview

3.6　手工重定位

3.6.1　实验目的

通过手工对代码节进行重定位修复,理解绝对地址为什么需要重定位,明白重定位的机制与原理。

在这个实验中,学生将深入研究可执行文件的代码节,了解其中绝对地址的特点和作用。通过手工进行重定位修复,学生将亲身体验在程序加载和执行过程中,绝对地址需要进行重定位的原因和必要性。通过实践操作,学生将理解重定位的需求、机制和原理,包括重定位表的结构和使用方法。这将有助于学生在逆向工程、病毒与漏洞分析中更好地理解程序的运行机制,提高其实践能力和分析水平。

3.6.2　实验内容及实验环境

1. 实验内容

(1) 修改 hello25.exe 的加载基址为 0x600000,破坏程序的运行机制。

(2) 在代码节中手工修正数据,使程序能够正常运行,理解重定位必要性。

2. 实验环境

(1) 系统：Windows XP 以上的操作系统,实机、虚拟机均可。

(2) 工具：OllyDbg1.10、010Editor。

3.6.3　实验步骤

1. 修改 ImageBase 字段

通过 PEview(或下载 010Editor 的 PE 模板)获取 ImageBase 字段的偏移为 0xE4,使用 010Editor 打开 hello25.exe,修改 ImageBase 字段为 0x600000,保存本次修改,如图 3-41 所示。

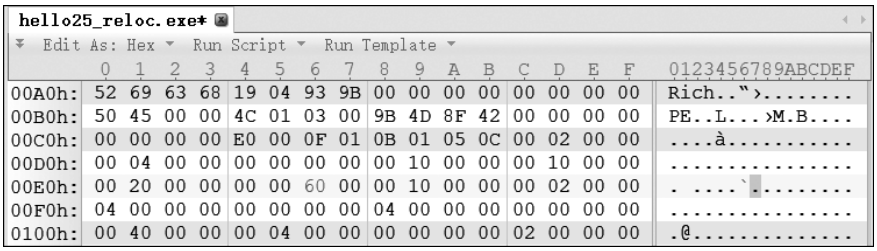


图 3-41　修改 ImageBase

2. 手工重定位

(1) 观察程序异常。

使用 OllyDbg 打开程序,观察代码节的异常与需要修改的地方。图 3-42 中用方框标明了需要修改的地方。首先,弹框函数中压栈字符串的地方直接使用了绝对地址,需要修改(例如 push 0x403000,字符串地址已变为了 0x603000);然后可以看到 call 指令的目标地址没有解析为对应 API,但这不是 call 指令的问题,而是 call 指令的目标对象——跳转表需要修改(例如 jmp dword ptr[0x402000])。这些需要修改的地方都有同样的特征:使用了绝对地址,这正是重定位表的任务,即修正代码中的绝对地址。

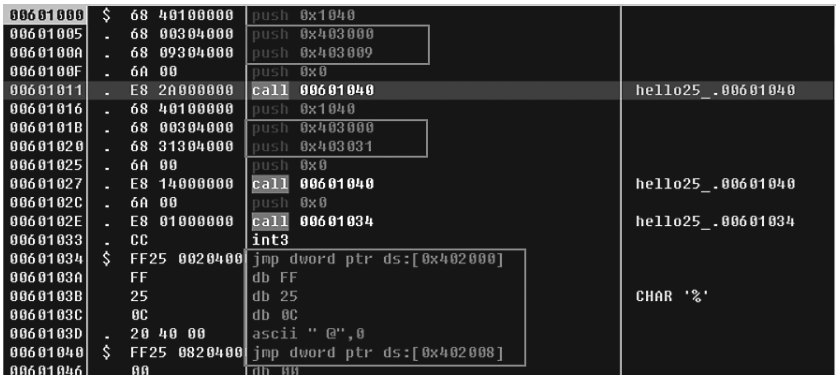


图 3-42　程序中的绝对地址

(2) 手工修复。

计算加载的基址差 delta(0x600000－0x400000＝0x200000),然后对所有绝对地址加上 delta 即可,如图 3-43 所示(跳转表的第二项没有正确解析,但没有影响,直接双击修改即可)。

修改完毕后保存为新文件(反汇编窗口右击→复制到可执行文件→所有修改→全部复制),运行查看是否功能正常。